Building Skills in Python

A Programmer's Introduction to Python

Steven F. Lott

5/29/2009

Table of Contents

List of Figures

List of Tables

List of Examples

List of Equations

Preface

Table of Contents

The Zen Of Python.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

--Tim Peters

Why Read This Book?

You need this book because you need to learn Python. There are lots of reasons why you need to learn Python. Here are a few.

- You need a programming language which is easy to read and has a vast library of modules focused on solving the problems you're faced with.
- You saw an article about Python specifically, or dynamic languages in general, and want to learn more.
- You're starting a project where Python will be used or is in use.
- A colleague has suggested that you look into Python.
- You've run across a Python code sample on the web and need to learn more.

Python reflects a number of growing trends in software development, putting it at or near the leading edge of good programming languages. It is a very simple language

surrounded by a vast library of add-on modules. It is an open source project, supported by many individuals. It is an object-oriented language, binding data and processing into class definitions. It is a platform-independent, scripted language, with complete access to operating system API's. It supports integration of complex solutions from pre-built components. It is a dynamic language, which avoids many of the complexities and overheads of compiled languages.

This book is a complete presentation of the Python language. It is oriented toward learning, which involves accumulating many closely intertwined concepts. In our experience teaching, coaching and doing programming, there is an upper limit on the "clue absorption rate". In order to keep within this limit, we've found that it helps to present a language as ever-expanding layers. We'll lead you from a very tiny, easy to understand subset of statements to the entire Python language and all of the built-in data structures. We've also found that doing a number of exercises helps internalize each language concept.

Three Faces of a Language. There are three facets to a programming language: how you write it, what it means, and the additional practical considerations that make a program useful. While many books cover the syntax and semantics of Python, in this book we'll also cover the pragmatic considerations. Our core objective is to build enough language skills that good object-oriented design will be an easy next step.

The *syntax* of a language is often covered in the language reference manuals. In the case of relatively simple languages, like Python, the syntax is simple, and is covered in the Python Language tutorial that is part of the basic installation kit. We'll provide additional examples of language syntax. For people new to programming, we'll provide additional tips focused on the newbie.

The *semantics* of the language can be a bit more slippery than the syntax. Some languages involve obscure or unique concepts that make it difficult to see what a statement really means. In the case of languages like Python, which have extensive additional *libraries*, the burden is doubled. First, one has to learn the language, then one has to learn the libraries. The number of open source packages made available by the Python community can increase the effort required to understand an entire architecture. The reward, however, is high-quality software based on high-quality components, with a minimum of development and integration effort.

Many languages offer a number of tools that can accomplish the same basic task. Python is no exception. It is often difficult to know which of many alternatives performs better or is easier to adapt. We'll try to focus on showing the most helpful approach, emphasizing techniques that apply for larger development efforts. We'll try to avoid quick and dirty solutions that are only appropriate when learning the language.

Audience

Professional programmers who need to learn Python are our primary audience. We provide specific help for you in a number of ways.

- Since Python is simple, we can address *newbie* programmers who don't have deep experience in a number of other languages. We will call out some details in specific newbie sections. Experienced programmers can skip these sections.
- Since Python has a large number of sophisticated built-in data structures, we address these separately and fully. An understanding of these structures can simplify complex programs.
- The object-orientation of Python provides tremendous flexibility and power. This is a deep subject, and we will provide an introduction to object-oriented programming in this book. More advanced design techniques are addressed in *Building Skills in Object-Oriented Design*, [Lott05].
- The accompanying libraries make it inexpensive to develop complex and complete solutions with minimal effort. This, however, requires some time to understand

the packaged components that are available, and how they can be integrated to create useful software. We cover some of the most important modules to specifically prevent programmers from reinventing the wheel with each project.

Instructors are a secondary audience. If you are looking for classroom projects that are engaging, comprehensible, and focus on perfecting language skills, this book can help. Each chapter in this book contains exercises that help students master the concepts presented in the chapter.

This book assumes an basic level of skill with any of the commonly-available computer systems. The following skills will be required.

- Download and install open-source application software. Principally, this is the Python distribution kit from Python.org. However, we will provide references to additional software components.
- Create text files. We will address doing this in IDLE, the Python Integrated Development Environment (IDE). We will also talk about doing this with a garden-variety text editor like Komodo, VIM, EMACS, TEXTPAD and BBEDIT.
- Run programs from the command-line. This includes the DOS command shell in Microsoft Windows™, or the Terminal tool in Linux or Apple's Macintosh OS X™.
- Be familiar with high-school algebra and some trigonometry. Some of the exercises make heavy use of basic algebra and trigonometry.

When you've finished with this book you should be able to do the following.

- Use of the core procedural programming constructs: variables, statements, exceptions, functions. We will not, for example, spend any time on design of loops that terminate properly.
- Create class definitions and subclasses. This includes managing the basic features of inheritance, as well as overloaded method names.
- Use the Python collection classes appropriately, this includes the various kinds of sequences, and the dictionary.

Organization of This Book

This book falls into five distinct parts. To manage the clue absorption rate, the first three parts are organized in a way that builds up the language in layers from central concepts to more advanced features. Each layer introduces a few new concepts, and is presented in some depth. Programming exercises are provided to encourage further exploration of each layer. The last two parts cover the extension modules and provide specifications for some complex exercises that will help solidify programming skills.

Some of the chapters include digressions on more advanced topics. These can be skipped, as they cover topics related to programming in general, or notes about the implementation of the Python language. These are reference material to help advanced students build skills above and beyond the basic language.

Part I, “Language Basics” introduces the basic feartures of the Python language, covering most of the statements but sticking with basic numeric data types. Chapter 1, *Background and History* provides some history and background on Python. Chapter 3, *Getting Started* covers installation of Python, using the interpreter interactively and creating simple program files. Chapter 4, *Simple Numeric Expressions and Output* covers the basic expressions and core numeric types. Chapter 6, *Variables, Assignment and Input* introduces variables, assignment and some simple input constructs. Chapter 7, *Truth, Comparison and Conditional Processing* adds truth, conditions and loops to the language. Chapter 9, *Functions* introduces the basic function definition and function call constructs; Chapter 10, *Additional Notes On Functions* introduces the call by name and call by value features of Python, as well as advanced function call features.

Part II, "Data Structures" adds a number of data structures to enhance the expressive power of the language. In this part we will use a number of different kinds of objects, prior to designing our own objects. Chapter 11, *Sequences: Strings, Tuples and Lists* extends the data types to include various kinds of sequences. These include Chapter 12, *Strings*, Chapter 13, *Tuples* and Chapter 14, *Lists*. Chapter 15, *Mappings and Dictionaries* describes mappings and dictionaries. Chapter 17, *Exceptions* covers exception objects, and exception creation and handling. Chapter 19, *Files* covers files and several closely related operating system (OS) services. Chapter 20, *Advanced Sequences* describes more advanced sequence techniques, including multi-dimensional matrix processing. This part attempts to describe a reasonably complete set of built-in data types.

Part III, "Data + Processing = Objects" describes the object-oriented programming features of Python. Chapter 21, *Classes* introduces basics of class definitions and introduces simple inheritance. Chapter 22, *Advanced Class Definition* adds some features to basic class definitions. Chapter 23, *Some Design Patterns* extend this discussion further to include several common design patterns that use polymorphism. Chapter 24, *Creating or Extending Data Types* describes the mechanism for adding types to Python that behave like the built-in types.

Part IV, "Components, Modules and Packages" describes modules, which provide a higher-level grouping of class and function definitions. It also summarizes selected extension modules provided with the Python environment. Chapter 28, *Modules* provides basic semantics and syntax for creating modules. We cover the organization of packages of modules in Chapter 29, *Packages*. An overview of the Python library is the subject of Chapter 30, *The Python Library*. Chapter 31, *Complex Strings: the `re` Module* covers string pattern matching and processing with the `re` module. Chapter 32, *Dates and Times: the `time` and `datetime` Modules* covers the `time` and `datetime` module. Chapter 35, *Programs: Standing Alone* covers the creation of main programs. We touch just the tip of the client-server iceberg in Chapter 36, *Programs: Clients, Servers, the Internet and the World Wide Web*.

Some of the commonly-used modules are covered during earlier chapters. In particular the `math` and `random` modules are covered in the section called "The `math` Module" and the `string` module is covered in Chapter 12, *Strings*. Chapter 19, *Files* touches on `fileinput`, `os`, `os.path`, `glob`, and `fnmatch`.

Part V, "Projects" presents several larger and more complex programming problems. These are ranked from relatively simple to quite complex. Chapter 37, *Areas of the Flag* covers computing the area of the symbols on the American flag. Chapter 38, *The Date of Easter* has several algorithms for finding the date for Easter in a given year. Chapter 39, *Musical Pitches* has several algorithms for the exact frequencies of musical pitches. Chapter 40, *Bowling Scores* covers scoring in a game of bowling. Chapter 41, *Mah Jongg Hands* describes algorithms for evaluating hands in the game of Maj Jongg. Chapter 42, *Chess Game Notation* deals with interpreting the log from a game of chess.

Limitations

This book can't cover everything Python. There are a number of things which we will not cover in depth, and some things which we can't even touch on lightly. This list will provide you directions for further study.

- The rest of the Python library. The library is a large, sophisticated, rapidly-evolving collection of software components. We selected a few modules that are widely-used. There are many books which cover the library in general, and books which cover specific modules in depth.

- The subject of Object-Oriented (OO) design is the logical next stepd in learning Python. That topic is covered in *Building Skills in Object-Oriented Design* [Lott05].

- Database design and programming requires a knowledge of Python and a grip on OO design. It requires a digression into the relational model and the SQL language.

- Graphical User Interface (GUI) development requires a knowledge of Python, OO design and database design. There are two commonly-used toolkits: `Tkinter` and `pyGTK`. We'll cover `pyGTK` in a future volume on graphics programming and GUI design with GTK.

- Web application development, likewise, requires a knowledge of Python, OO design and database design. This topic requires digressions into internetworking protocols, specifically HTTP and SOAP, plus HTML, XML and CSS languages. There are numerous web development frameworks for Python.

Programming Style

We have to adopt a *style* for presenting Python. We won't present a complete set of *coding standards*, instead we'll present examples. This section has some justification of the style we use for the examples in this book.

Just to continune this rant, we find that actual examples speak louder than any of the gratuitously detailed coding standards which are so popular in IT shops. We find that many IT organizations waste considerable time trying to write descriptions of a preferred style. A good example, however, trumps any description. As consultants, we are often asked to provide standards to an inexperienced team of programmers. The programmers only look at the examples (often cutting and pasting them). Why spend money on empty verbiage that is peripheral to the useful example?

One important note: we specifically reject using complex prefixes for variable names. Prefixes are little more than visual clutter. In many places, for example, an integer parameter with the amount of a bet might be called `pi_amount` where the prefix indicates the scope (*p* for a parameter) and type (*i* for an integer). We reject the `pi_` as useless and uninformative.

This style of name is only appropriate for primitive types, and doesn't address complex data structures well at all. How does one name a parameter that is a list of dictionaries of class instances? `pldc_`?

In some cases, prefixes are used to denote the scope of an instance variables. Variable names might include a cryptic one-letter prefix like “f” to denote an instance variable; sometimes programmers will use “my” or “the” as an English-like prefix. We prefer to reduce clutter. In Python, instance variables are always qualified by `self.`, making the scope crystal clear.

All of the code samples were tested on Python 2.5 for MacOS, using an iMac running MacOS 10.5. Additional testing of all code was done with Windows 2000 on a Dell Latitude laptop, as well as a Dell Precision running Fedora Core.

Conventions Used in This Book

Here is a typical Code sample.

Example 1. Typical Python Example

```
combo = { } ❶
for i in range(1,7):
    for j in range(1,7):
        roll= i+j
        combo.setdefault( roll, 0 ) ❷
        combo[roll] += 1
for n in range(2,13):
```

```
    print "%d %.2f%%" % ( n, combo[n]/36.0 ) ❸
```

❶ This creates a Python dictionary, a map from key to value. If we initialize it with something like the following: `combo = dict( [ (n,0) for n in range(2,13) ] )`, we don't need the `setdefault` function call below.

❷ This assures that the rolled number exists in the dictionary with a default frequency count of 0.

❸ Print each member of the resulting dictionary. Something more obscure like `[ (n,combo[n]/36.0) for n in range(2,13)]` is certainly possible.

The output from the above program will be shown as follows:

```
2 0.03%
3 0.06%
4 0.08%
5 0.11%
6 0.14%
7 0.17%
8 0.14%
9 0.11%
10 0.08%
11 0.06%
12 0.03%

Tool completed successfully
```

We will use the following type styles for references to a specific `Class`, `method function`, `attribute`, which includes both class variables or instance variables.

Sidebars

When we do have a significant digression, it will appear in a sidebar, like this.

Tip

There will be design tips, and warnings, in the material for each exercise. These reflect considerations and lessons learned that aren't typically clear to starting OO designers.

Acknowledgements

I'd like to thank Carl Frederick for asking me if I was using Python to develop complex applications. At the time, I said I'd have to look into it. This is the result of that investigation.

I am indebted to Thomas Pautler, Jim Bullock, Michaël Van Dorpe, Matthew Curry, Igor Sakovich, Drew, John Larsen and Robert Lucente for supplying much-needed corrections to errors in previous editions.

Language Basics

The Processing View

A programming language involves two closely interleaved topics. On one hand, there are the *statement* constructs that process information inside the computer, with visible effects on the various external devices. On the other hand are the various types of data and relationship structures for organizing the information manipluated by the program.

This part describes the most commonly-used Python statements, sticking with basic numeric data types. Part II, “Data Structures” will present a reasonably complete set of

built-in data types and features for Python. While the two are tightly interwoven, we pick the statements as more fundamental because we we can (and will) add new data types. Indeed, the essential thrust of object-oriented programming (covered in Part III, "Data + Processing = Objects") is the creation of new data types.

Some of the examples in this part refer to the rules of various common casino games. Knowledge of casino gambling is not essential to understanding the language or this part of the book. We don't endorse casino gambling. Indeed, many of the exercises reveal the magnitude of the house edge in most casino games. However, casino games have just the right level of algorithmic complexity to make for excellent programming exercises.

In Chapter 3, *Getting Started* we'll describe the basics of computer programming, installing a Python interpreter, using Python interactively, and writing simple scripts. In Chapter 4, *Simple Numeric Expressions and Output* we'll introduce the **print** statement, and arithmetic expressions including the numeric data types, operators, conversions, and some built-in functions. We'll expand on this in Chapter 5, *Advanced Expressions*. We'll introduce variables, the assignment statement, and input in Chapter 6, *Variables, Assignment and Input*, allowing us to create simple input-process-output programs. When we add truth, comparisons, conditional processing, iterative processing and assertions in Chapter 7, *Truth, Comparison and Conditional Processing*, we'll have all the tools necessary for programming. In Chapter 9, *Functions* and Chapter 10, *Additional Notes On Functions*, we'll show how to define and use functions, the first of many tools for organizing programs to make them understandable.

Table of Contents

Chapter 1. Background and History

History of Python and Comparison with Other Languages

Table of Contents

This chapter describes the history of Python in the section called “History”. After that, the section called “Comparisons” is a subjective comparison between Python and a few other other languages, using some quality criteria harvested from two sources: the *Java Language Environment White Paper* and “On the Design of Programming Languages”. This material can be skipped by newbies: it doesn't help explain Python, it puts it into a context among other programming languages.

History

Python is a relatively simple programming language that includes a rich set of supporting libraries. This approach keeps the language simple and reliable, while providing specialized feature sets as separate extensions.

Python has an easy-to-use syntax, focused on the programmer who must type in the program, read what was typed, and provide formal documentation for the program. Many languages have syntax focused on developing a simple, fast compiler; but those languages may sacrifice readability and writability. Python strikes a good balance between fast compilation, readability and writability.

Python is implemented in C, and relies on the extensive, well understood, portable C libraries. It fits seamlessly with Unix, Linux and POSIX environments. Since these standard C libraries are widely available for the various MS-Windows variants, and other non-POSIX operating systems, Python runs similarly in all environments.

The Python programming language was created in 1991 by Guido van Rossum based on lessons learned doing language and operating system support. Python is built from concepts in the ABC language and Modula-3. For information ABC, see *The ABC Programmer's Handbook* [Geurts91], as well as www.cwi.nl/~steven/abc/. For information on Modula-3, see *Modula-3* [Harbison92], as well as www.research.compaq.com/SRC/modula-3/html/home.html.

The current Python development is centralized in Python.org. See www.python.org for the latest developments.

Features of Python

Python reflects a number of growing trends in software development. It is a very simple language surrounded by a vast library of add-on modules. It is an open source project, supported by dozens of individuals. It is an object-oriented language. It is a platform-independent, scripted language, with complete access to operating system API's. It supports integration of complex solutions from pre-built components. It is a dynamic language, allowing more run-time flexibility than statically compiled languages.

Additionally, Python is a scripting language with full access to Operating System (OS) services. Consequently, Python can create high level solutions built up from other complete programs. This allows someone to integrate applications seamlessly, creating high-powered, highly-focused meta-applications. This kind of very-high-level programming (*programming in the large*) is often attempted with shell scripting tools. However, the programming power in most shell script languages is severely limited. Python is a complete programming language in its own right, allowing a powerful mixture of existing application programs and unique processing to be combined.

Python includes the basic text manipulation facilities of Awk or Perl. It extends these with extensive OS services and other useful packages. It also includes some additional data types and an easier-to-read syntax than either of these languages.

Python has several layers of program organization. The Python package is the broadest organizational unit; it is collection of modules. The Python module, analogous to the Java package, is the next level of grouping. A module may have one or more classes and free functions. A class has a number of static (class-level) variables, instance variables and methods. We'll lookl at these layers in detail in appropriate sections.

Some languages (like COBOL) have features that are folded into the language itself, leading to a complicated mixture of core features, optional extensions, operating-system features and special-purpose data structures or algorithms. These poorly designed languages may have problems with portability. This complexity makes these languages hard to learn. One hint that a language has too many features is that a language subset is available. Python suffers from none of these defects: the language has only 21 statements (of which five are declaratory in nature), the compiler is simple and portable. This makes the the language is easy to learn, with no need to create a simplified language subset.

Comparisons

We'll measure Python with two yardsticks. First, we'll look at a yardstick originally used for Java. Then we'll look at yardstick based on experience designing Modula-2.

The Java Yardstick

The *Java Language Environment White Paper* [Gosling96] lists a number of desirable features of a programming language:

- Simple and Familiar
- Object-Oriented
- Secure
- Interpreted
- Dynamic
- Architecture Neutral
- Portable
- Robust
- Multithreaded
- Garbage Collection
- Exceptions
- High Performance

Python meets and exceeds most of these expectations. We'll look closely at each of these twelve desireable attributes.

Simple and Familiar. By simple, we mean that there is no GOTO statement, we don't need to explicitly manage memory and pointers, there is no confusing preprocessor, we don't have the aliasing problems associated wuth unions. We note that this list summarizes the most confusing and bug-inducing features of the C programming language.

Python is simple. It relies on a few core data structures and statements. The rich set of features is introduced by explicit import of extension modules. Python lacks the problem-plagued GOTO statement, and includes the more reliable **break**, **continue** and exception **raise** statements. Python conceals the mechanics of object references from the programmer, making it impossible to corrupt a pointer. There is no language preprocessor to obscure the syntax of the language. There is no C-style union (or COBOL-style REDEFINES) to create problematic aliases for data in memory.

Python uses an English-like syntax, making it reasonably familiar to people who read and write English or related languages. There are few syntax rules, and ordinary, obvious indentation is used to make the structure of the software very clear.

Object-Oriented. Python is object oriented. Almost all language features are first class objects, and can be used in a variety of contexts. This is distinct from Java and C++ which create confusion by having objects as well as primitive data types that are not objects. The built-in `type(x)` function can interrogate the types of all objects. The language permits creation of new object classes. It supports single and multiple inheritance. Polymorphism is supported via run-time interpretation, leading to some additional implementation freedoms not permitted in Java or C++.

Secure. The Python language environment is reasonably secure from tampering. Pre-compiled python modules can be distributed to prevent altering the source code. Additional security checks can be added by supplementing the built-in `__import__` function.

Many security flaws are problems with operating systems or framework software (for example, database servers or web servers). There is, however, one prominent language-related security problem: the "buffer overflow" problem, where an input buffer, of finite size, is overwritten by input data which is larger than the available buffer. Python doesn't suffer from this problem.

Python is a dynamic language, and abuse of features like the **exec** statement or the `eval` function can introduce security problems. These mechanisms are easy to identify and audit in a large program.

Interpreted. An interpreted language, like Python allows for rapid, flexible, exploratory software development. Compiled languages require a sometimes lengthy edit-compile-link-execute cycle. Interpreted languages permit a simpler edit-execute cycle. Interpreted languages can support a complete debugging and diagnostic environment. The Python interpreter can be run interactively; which can help with program development and testing.

The Python interpreter can be extended with additional high-performance modules. Also, the Python interpreter can be embedded into another application to provide a handy scripting extension to that application.

Dynamic. Python executes dynamically. Python modules can be distributed as source; they are compiled (if necessary) at import time. Object messages are interpreted, and problems are reported at run time, allowing for flexible development of applications.

In C++, any change to centrally used class headers will lead to lengthy recompilation of dependent modules. In Java, a change to the public interface of a class can invalidate a number of other modules, leading to recompilation in the best case, or runtime errors in the worst case.

Portable. Since Python rests squarely on a portable C source, Python programs behave the same on a variety of platforms. Subtle issues like memory management are completely hidden. Operating system inconsistency makes it impossible to provide perfect portability of every feature. Portable GUI's are built using the widely-ported Tk GUI tools `Tkinter`, or the GTK+ tools and the the `pyGTK` bindings.

Robust. Programmers do not directly manipulate memory or pointers, making the language run-time environment very robust. Errors are raised as exceptions, allowing programs to catch and handle a variety of conditions. All Python language mistakes lead to simple, easy-to-interpret error messages from exceptions.

Multithreaded. The Python `threading` module is a Posix-compliant threading library. This is not completely supported on all platforms, but does provide the necessary interfaces. Beyond thread management, OS process management is also available, as are execution of shell scripts and other programs from within a Python program.

Additionally, many of the web frameworks include thread management. In products like TurboGears, individual web requests implicitly spawn new threads.

Garbage Collection. Memory-management can be done with explicit deletes or automated garbage collection. Since Python uses garbage collection, the programmer doesn't have to worry about memory leaks (failure to delete) or dangling references (deleting too early).

The Python run-time environment handles garbage collection of all Python objects. Reference counters are used to assure that no live objects are removed. When objects go out of scope, they are eligible for garbage collection.

Exceptions. Python has exceptions, and a sophisticated **try** statement that handles exceptions. Unlike the standard C library where status codes are returned from some functions, invalid pointers returned from others and a global error number variable used for determining error conditions, Python signals almost all errors with an exception. Even common, generic OS services are *wrapped* so that exceptions are raised in a uniform way.

High Performance. The Python interpreter is quite fast. However, where necessary, a

class or module that is a bottleneck can be rewritten in C or C++, creating an extension to the runtime environment that improves performance.

The Modula-2 Yardstick

One of the languages which strongly influenced the design of Python was Modula-2. In 1974, N. Wirth (creator of Pascal and its successor, Modula-2) wrote an article "On the Design of Programming Languages" [Wirth74], which defined some timeless considerations in designing a programming language. He suggests the following:

- a language be easy to learn and easy to use;
- safe from misinterpretation;
- extensible without changing existing features;
- machine [*platform*] independent;
- the compiler [*interpreter*] must be fast and compact;
- there must be ready access to system services, libraries and extensions written in other languages;
- the whole package must be portable.

Python syntax is designed for readability; the language is quite simple, making it easy to learn and use. The Python community is always alert to ways to simplify Python. The Python 3.0 project is actively working to remove a few poorly-concieved features of Python. This will mean that Python 3.0 will be simpler and easier to use, but incompatible with Python 2.x in a few areas.

Most Python features are brought in via modules, assuring that extensions do not change or break existing features. This allows tremendous flexibility and permits rapid growth in the language libraries.

The Python interpreter is very small. Typically, it is smaller than the Java Virtual Machine. Since Python is (ultimately) written in C, it has the same kind of broad access to external libraries and extensions. Also, this makes Python completely portable.

Chapter 2. Python Installation

Downloading, Installing and Upgrading Python

Table of Contents

This chapter is becoming less and less relevant as Python comes pre-installed with most Linux-based operating systems. Consequently, the most interesting part of this chapter is the Windows Installation, where we describe downloading and installing Python on Windows.

Python runs on a wide, wide variety of platforms. If your particular operating system isn't described here, refer to www.python.org/community for more information.

Mac OS developers will find it simplest to upgrade to Leopard (Max OS 10.5), since it has Python 2.5.1 included. This installation includes the complete suite of tools, but doesn't include a clever Applications icon for launching IDLE. You can make this icon yourself using Applescript. If installiung Leopard is undesirable, for some reason, you'll need to do an install or upgrade of Python in your existing Mac OS. We'll cover this in Macintosh Installation.

For other GNU/Linux developers, you'll find that Python is generally included in most distributions. In the unlikely event that you have an old distribution and can't upgrade, we'll present some alternatives that should cover the most common situations.

Start with GNU/Linux and UNIX Overview. This section has a quick procedure for examining your current configuration to see if you have Python in the first place. If you have Python, and it's version 2.5, you're all done. Otherwise, you'll have to determine what tools you have for doing an installation or upgrade.

- If you have Yellowdog Updater Modified (YUM) see YUM Installation.
- If you have one of the GNU/Linux variants that uses the Red Hat Package Manager (RPM), see RPM Installation.
- The alternative to use the source installation procedure in "Build from Scratch" Installation.

The goal of installation is to get the Python interpreter and associated libraries. Windows users will get a program called `python.exe`. Linux and MacOS users will get the Python interpreter, a program named `python`.

In addition to the libraries and the interpreter, your Python installation comes with a tutorial document on Python that will step you through a number of quick examples. For newbies, this provides an additional point of view that you may find helpful. You may also want to refer to the official Beginner's Guide Wiki.

Windows Installation

In some circumstances, your Windows environment may require administrator privilege. The details are beyond the scope of this book. If you can install software on your PC, then you have administrator privileges. In a corporate or academic environment, someone else may be the administrator for your PC.

The Windows installation of Python has three broad steps.

1. Pre-installation: make backups and download the installation kit.
2. Installation: install Python.
3. Post-installation: check to be sure everything worked.

We'll go through each of these in detail.

Windows Pre-Installation

Backup. Before installing software, back up your computer. I strongly recommend that you get a tool like Norton's Ghost™ . This product will create a CD that you can use to reconstruct the operating system on your PC in case something goes wrong. It is difficult to undo an installation in Windows, and get your computer back the way it was before you started.

I've never had a single problem installing Python. I've worked with a number of people, however, who either have bad luck or don't read carefully and have managed to corrupt their Windows installation by downloading and installing software. While Python is safe, stable, reliable, virus-free, and well-respected, you may be someone with bad luck who has a problem. Often the problem already existed on your PC and installing Python was the straw that broke the camel's back. A backup is cheap insurance.

You should also have a folder for saving your downloads. You can create a folder in `My Documents` called `downloads`. I suggest that you keep all of your various downloaded tools and utilities in this folder for two reasons. If you need to reinstall your software, you know exactly what you downloaded. When you get a new computer (or an additional computer), you know what needs to be installed on that computer.

Download. After making a backup, go to the www.python.org web site and look for the Download area. In here, you're looking for the pre-built Windows installer. This book will emphasize Python 2.5. In that case, the kit is `python-2.5.x.msi`. When you click on the filename, your browser should start downloading the file. Save it in your `downloads` folder.

Backup. Now is a good time to make a second backup. Seriously. This backup will have your untouched Windows system, plus the Python installation kit. It is still cheap insurance.

If you have anti-virus software [*you do, don't you?*] you may need to disable this until you are done installing Python.

At this point, you have everything you need to install Python:

- A backup
- The Python installer

Windows Installation

You'll need two things to install Python. If you don't have both, see the previous section on pre-installation.

- A backup
- The Python installer

Double-click the Python installer (`python-2.5.x.msi`).

The first step is to select a destination directory. The default destination should be `C:\Python25`. Note that Python does not expect to live in the `C:\My Programs` folder. Because the `My Programs` folder has a space in the middle of the name — something that is atypical for all operating systems other than Windows — subtle problems can arise. Consequently, Python folks prefer to put Python into `C:\Python25` on Windows machines. Click Next to continue.

The next step is to confirm that you want to backup replaced files. The option to make backups is already selected and the folder is usually `C:\Python25\BACKUP`. This is the way it should be. Click Next to continue.

The next step is the list of components to install. You have a list of five components.

- Python interpreter and libraries. You want this.
- Tcl/Tk (Tkinter, IDLE, pydoc). You want this, so that you can use IDLE to build programs.
- Python HTML Help file. This is some reference material that you'll probably want to have.
- Python utility scripts (Tools/). We won't be making any use of this; you don't need to install it. It won't hurt anything if you install it.
- Python test suite (Lib/test/). We won't make any use of this, either. It won't hurt anything if you install it.

There is an Advanced Options... button that is necessary if you are using a company-supplied computer for which you are not the administrator. If you are not the administrator, and you have permission to install additional software, you can click on this button to get the Advanced Options panel. There's a button labeled Non-Admin install that you'll need to click in order to install Python on a PC where you don't have administrator privileges.

Click Next to continue.

You can pick a Start Menu Group for the Python program, IDLE and the help files. Usually, it is placed in a menu named `Python 2.5`. I can't see any reason for changing this, since it only seems to make things harder to find. Click Next to continue.

The installer puts files in the selected places. This takes less than a minute.

Click Finish; you have just installed Python on your computer.

> **Debugging Windows Installation**
>
> The only problem you are likely to encounter doing a Windows installation is a lack of administrative privileges on your computer. In this case, you will need help from your support department to either do the installation for you, or give you administrative privileges.

Windows Post-Installation

In your Start... menu, under All Programs, you will now have a Python 2.5 group that lists five things:

- IDLE (Python GUI)
- Module Docs
- Python (command line)
- Python Manuals
- Uninstall Python

GUI is the Graphic User Interface. We'll turn to IDLE in the section called “The IDLE Development Environment”.

> **Testing**
>
> If you select the Python (command line) menu item, you'll see the `Python (command line)` window. This will contain something like the following.

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

If you hit **Ctrl-Z** and then **Enter**, Python will exit. The basic Python program works. You can skip to the next chapter to start using Python.

If you select the Python Manuals menu item, this will open a Microsoft Help reader that will show the complete Python documentation library.

Macintosh Installation

Python is part of the MacOS environment. Tiger (MacOS 10.4) includes Python 2.3.5 and IDLE. Leopard (MacOS 10.5) includes Python 2.5.1. Generally, you don't need to do much to get started. You'll just need to locate the various Python files. Look in `/System/Library/Frameworks/Python.Framework/Versions` for the relevant files.

In order to upgrade software in the Macintosh OS, you must know the administrator, or "owner" password. If you are the person who installed or initially setup the computer, you had to pick an owner password during the installation. If someone else did the installation, you'll need to get the password from them.

The Mac OS installation of Python has three broad steps.

1. Pre-installation: make backups and download the installation kit.
2. Installation: install Python.
3. Post-installation: check to be sure everything worked.

We'll go through each of these in detail.

Macintosh Pre-Installation

Before installing software, back up your computer. While you can't easily burn a DVD of everything on your computer, you can usually burn a DVD of everything in your Mac OS X Home directory.

I've never had a single problem installing Python. I've worked with a number of people, however, who either have bad luck or don't read carefully and have managed to corrupt their Mac OS installation by downloading and installing software. While Python is safe, stable, reliable, virus-free, and well-respected, you may be someone with bad luck who has a problem. A backup is cheap insurance.

You should also have a folder for saving your downloads. You can create a folder in hour `Documents` called `downloads`. I suggest that you keep all of your various downloaded tools and utilities in this folder for two reasons. If you need to reinstall your software, you know exactly what you downloaded. When you get a new computer (or an additional computer), you know what needs to be installed on that computer.

Download. After making a backup, go to the www.python.org web site and look for the Download area. In here, you're looking for the pre-built Mac OS X installer. This book will emphasize Python 2.5. In that case, the kit is `python-2.5.`*`x`*`-macosx`*`date`*`.dmg`. When you click on the filename, your browser should start downloading the file. Save it in your `downloads` folder.

Backup. Now is a good time to make a second backup. Seriously. It is still cheap insurance.

At this point, you have everything you need to install Python:

- A backup
- The Python installer

Macintosh Installation

When you double-click the `python-2.5.x-macosxdate.dmg` file, it will create a disk image named `Universal MacPython 2.5.x`. This disk image has your license, a ReadMe file, and the `MacPython.mpkg`.

When you double-click the `MacPython.mpkg` fie, it will take all the necessary steps to install Python on your computer. The installer will take you through seven steps. Generally, you'll read the messages and

Introduction. Read the message and click Continue.

Read Me. This is the contents of the ReadMe file on the installer disk image. Read the message and click Continue.

License. You can read the history of Python, and the terms and conditions for using it. To install Python, you must agree with the license. When you click Continue, you will get a pop-up window that asks if you agree. Click Agree to install Python.

Select Destination. Generally, your primary disk drive, usually named `Macintosh HD` will be highlighted with a green arrow. Click Continue.

Installation Type. If you've done this before, you'll see that this will be an upgrade. If this is the first time, you'll be doing an install. Click the Install or Upgrade button.

You'll be asked for your password. If, for some reason, you aren't the administrator for this computer, you won't be able to install software. Otherwise, provide your password so that you can install software.

Finish Up. The message is usually "The software was successfully installed". Click Close to finish.

Macintosh Post-Installation

In your Applications folder, you'll find a `MacPython 2.5` folder, which contains a number of applications.

- BuildApplet
- Extras
- IDLE
- PythonLauncher
- Update Shell Profile.command

Look in `/System/Library/Frameworks/Python.Framework/Versions` for the relevant files. In the `bin`, `Extras` and `Resources` directories you'll find the various applications. The `bin/idle` file will launch IDLE for us.

Once you've finished installation, you should check to be sure that everything is working correctly.

Testing

Now you can go to your `Applications` folder, and double click the IDLE application. This will open two windows, the Python Shell window is what we need, but it is buried under a Console window.

Here's what you'll see in the Python Shell window.

```
Python 2.5.1 (r251:54863, Oct  5 2007, 21:08:09)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>>
```

The menu bar will have a menu named Python with the menu item Quit Python. Use this to finish using IDLE for now, and skip to the next chapter.

You may notice a Help menu. This has the Python Docs menu item, which you can access through the menu or by hitting **F1**. This will launch Safari to show you the Python documents available on the Internet.

GNU/Linux and UNIX Overview

In order to install software in GNU/Linux, you must know the administrator, or "root" password. If you are the person who installed the GNU/Linux, you had to pick an administrator password during the installation. If someone else did the installation, you'll need to get the password from them.

Normally, we never log in to GNU/Linux as `root` except when we are installing software. In this case, because we are going to be installing software, we need to log in as `root`, using the administrative password.

If you are a GNU/Linux newbie and are in the habit of logging in as `root`, you're going to have to get a good GNU/Linux book, create another username for yourself, and start using a proper username, not `root`. When you work as `root`, you run a terrible risk of damaging or corrupting something. When you are logged on as anyone other than `root`, you will find that you can't delete or alter important files.

Do You Already Have Python? Many GNU/Linux and Unix systems have Python installed. On some older Linuxes *[Linuxi? Lini? Linen?]* there may be an older version of Python that needs to be upgraded. Here's what you do to find out whether or not you already have Python.

You'll need to run the Terminal tool. The GNOME desktop that comes with Red Hat and Fedora has a `Start Here` icon which displays the applications that are configured into you GNOME environment. The `System Tools` icon includes the Terminal application. Double click Terminal icon, or pick it off the menu, and you'll get a window which prompts you by showing something like `[slott@linux01 slott]$`. In response to this prompt, enter `env python`, and see what happens.

Here's what happens when Python is not installed.

```
slott% env python
tcsh: python: not found
```

Here's what you see when there is a properly installed, but out-of-date Python on your GNU/Linux box.

```
slott% env python
Python 2.3.5 (#1, Mar 20 2005, 20:38:20)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1809)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
```

In this case, the version number is 2.3.5, which is good, but we need to install an upgrade.

Unix is not Linux. For non-Linux commercial Unix installations (Solaris™, AIX™, HP/UX™, etc.), check with your vendor (Sun, IBM, HP, etc.) It is very likely that they have an extensive collection of open source projects like Python pre-built for your UNIX variant. Getting a pre-built kit from your operating system vendor is the best way to install Python.

YUM Installation

If you are a Red Hat or Fedora user, you likely have a program named Yum. If you don't have Yum, you should upgrade to Fedora Core 8, which includes Python 2.5, Yum (plus many other new programs.)

Note that Yum repositories do not cover every combination of operating system and Python distribution. For example, there isn't a handy pre-built setup for Python 2.5 on Fedora Core 6. In many cases, you'll want to do an operating system upgrade in order to introduce a new Python distribution.

If you have an out-of-date Python, you'll have to enter two commands in the Terminal window.

```
yum upgrade python
yum install tkinter
```

The first command will upgrade the Python 2.5 distribution. You can use the command "`install`" instead of "`upgrade`" in the unlikely event that you somehow have Yum, but don't have Python.

The second command will assure that the extension package named `tkinter` is part of your Fedora installation. It is not, typically, provided automatically. You'll need this to make use of the IDLE program used extensively in later chapters.

RPM Installation

Many variants of GNU/Linux use the Red Hat Package Manager (RPM). The rpm tool automates the installation of software and the important dependencies among software components. If you don't know whether on not your GNU/Linux uses the Red Hat Package manager, you'll have to find a GNU/Linux expert to help you make that determination.

Red Hat Linux (and the related Fedora Core distributions) have a version of Python pre-installed. Sometimes, the pre-installed Python is an older release and needs an upgrade.

This book will focus on Fedora Core GNU/Linux because that's what I have running. Specifically, Fedora Core 8. You may have a different GNU/Linux, in which case, this procedure is close, but may not be precisely what you'll have to do.

The Red Hat and Fedora GNU/Linux installation of Python has three broad steps.

1. Pre-installation: make backups.
2. Installation: install Python. We'll focus on the simplest kind of installation.
3. Post-installation: check to be sure everything worked.

We'll go through each of these in detail.

Resolving RPM Dependencies. When doing manual RPM installations, you may have to make several passes. This occurs when the package you're trying to install depends on a package that is missing or out of date. The installation step breaks down into a series of attempts to locate and install the needed packages.

1. First, attempt to install the package. If all of the foundation is in place, this will work, and you'll be finished. If you don't have the complete foundation in place, you'll get messages telling you what's missing or out-of-date.

2. If RPM reports any missing or wrong version packages, you must freshen or install the missing packages. This will build the proper foundation. Note that foundations have sub-foundations, and sub-foundations have sub-sub-foundations. This process can go pretty deep, so keep notes on where you are in the process.

    a. Search for the missing our out of date RPM.
    b. Freshen or Install the missing package. This may lead to a search for additional packages.

3. Finally, finish installing the package you started out trying to install. Ideally, you took a detour and installed everything on which this package depends, so the second time around it should install. If the package doesn't install, you'll be back at step 2 with a different list of foundational components to put in place.

RPM Pre-Installation

Before installing software, back up your computer.

You should also have a directory for saving your downloads. I recommend that you create a `/opt` directory for these kinds of options which are above and beyond the basic Linx installation. You can keep all of your various downloaded tools and utilities in this directory for two reasons. If you need to reinstall your software, you know exactly what you downloaded. When you get a new computer (or an additional computer), you know what needs to be installed on that computer.

RPM Installation

A typical scenario for installing Python is a command like the following. This has specific file names for Fedora Core 9. You'll need to locate appropriate RPM's for your distribution of Linux.

```
rpm -i http://download.fedora.redhat.com/pub/fedora/linux/development\
/i386/os/Packages/python-2.5.1-18.fc9.i386.rpm
```

Often, that's all there is to it. In some cases, you'll get warnings about the DSA signature. These are expected, since we didn't tell RPM the public key that was used to sign the packages.

RPM Post-Installation

Testing

Run the Terminal tool. At the command line prompt, enter `env python`, and see what happens. Generally, we should be able to simply enter **python** and run the python environment.

```
[slott@linux01 ~]$ python
Python 2.5.1 (r251:54863, Oct 30 2007, 13:54:11)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you hit Ctrl-D (the GNU/Linux end-of-file character), Python will exit. The basic Python program works.

"Build from Scratch" Installation

There are many GNU/Linux variants, and we can't even begin to cover each variant. You can use a similar installation on Windows or the Mac OS, if you have the GCC compiler installed. Here's an overview of how to install using a largely manual sequence of steps.

1. **Pre-Installation.** Make backups and download the installation kit. You're looking for the a file named `python-2.5.x.tgz`.

2. **Installation.** The installation involves a fairly common set of commands. If you are an experienced system administrator, but a novice programmer, you may recognize these.

   Change to the /opt/python directory with the following command.

   ```
   cd /opt/python
   ```

   Unpack the archive file with the following command.

   ```
   tar -zxvf Python-2.5.x.tgz
   ```

   Do the following four commands to configure the installation scripts, make the Python package and then install Python on your computer.

   ```
   cd Python-2.5
   ./configure
   make
   make install
   ```

3. Post-installation: check to be sure everything worked.

Testing

Run the Terminal tool. At the command line prompt, enter `env python`, and see what happens.

```
[slott@linux01 ~]$ python
Python 2.5.1 (r251:54863, Oct 30 2007, 13:54:11)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you hit Ctrl-D (the GNU/Linux end-of-file character), Python will exit. The basic Python program works.

Debugging Other Unix Installation

The most likely problem you'll encounter in doing a generic installation is not having the appropriate GNU GCC compiler. In this case, you will see error messages from **configure** which identifies the list of missing packages. Installing the GNU GCC can become a complex undertaking.

Chapter 3. Getting Started

Interacting with Python

Table of Contents

Python is an interpreted, dynamic language. The Python interpreter can be used in two modes: interactive and scripted. In *interactive* mode, Python responds to each statement while we type. In *script* mode, we give Python a file of statements and turn it loose to interpret all of the statements in that script. Both modes produce identical results. When we're producing a finished application program, we set it up to run as a script. When we're experimenting or exploring, however, we may use Python interactively.

We'll describe the interactive command-line mode for entering simple Python statements in Command-Line Interaction. In The IDLE Development Environment we'll cover the basics of interactive Python in the IDLE environment. We'll describes the script mode for running Python program files in Script Mode.

Command-Line Interaction

We'll look at interaction on the command line first, because it is the simplest way to interact with Python. It parallels scripted execution, and helps us visualize how Python application programs work. This is the heart of IDLE as well as the foundation for any application programs we build.

Starting and Stopping Command-Line Python

Starting and stopping Python varies with your operating system. Generally, all of the

variations are nearly identical, and differ only in minor details.

Windows. There are two ways to start interactive Python under Windows. You can run the command tool (`cmd.exe`) or you can run the Python (Command Line) program. If you run `cmd.exe`, enter the command **python** to start interactive Python. If you run Python (Command Line) from the Start menu, the result will be essentially the same.

To exit from Python, enter the end-of-file character sequence, **Control-Z** and **Return**.

Mac OS, GNU/Linux and Unix. You will run the Terminal tool. You can enter the command **python** to start interactive Python.

To exit from Python, enter the end-of-file character, **Control-D**.

Entering Python Statements

When we run the Python interpreter (called `python`, or `Python.exe` in Windows), we see a greeting like the following:

```
52:~ slott$ env python
Python 2.5.1 (r251:54863, Oct  5 2007, 21:08:09)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When we get the >>> prompt, the Python interpreter is looking for input. We can type any Python statements we want. Each complete statement is executed when it is entered. Some statements are called *compound statements*; a compound statement has a suite of statements indented inside of it. Python will wait until the entire compound statement is entered before it does does the evaluation.

In this section, we'll emphasize the prompts from Python. This can help newbies see the complete cycle of interaction between themselves and the Python interpreter. In the long run we'll be writing scipts and won't see this level of interaction.

The Return Key. For this section only, we'll also emphasize the usually invisible **Return** (↵) key. When we start using compound statements in Chapter 7, *Truth, Comparison and Conditional Processing*, we'll add some additional syntax rules. For now, however, statements can't be indented; they must begin without any leading spaces or tabs.

```
>>> 2 + 3 ↵
5
>>>
```

This shows Python doing simple integer arithmetic. When you entered **`2 + 3`** and then hit **Return**, the Python interpreter evaluated this statement. Since the statement was only an expression, Python printed the results.

We'll dig into to the various kinds of numbers in Simple Numeric Expressions and Output. For now, it's enough to know that you have integers and floating-point numbers that look much like other programming languages. As a side note, integers have two slightly different flavors -- fast (but small) and long (but slow). Python prefers will use the fast integers (called `int`) until your numbers get so huge that it has to switch to `long`.

Arithmetic operators include the usual culprits: `+`, `-`, `*`, `/`, `%` and `**` standing for addition, subtraction, multiplication, division, modulo (remainder after division) and raising to a power. The usual mathematical rules of operator precedence (multiplys and divides done before adds and subtracts) are in full force, and ()'s are used to group terms against precedence rules.

For example, converting 65° Fahrenheit to Celsius is done as follows:

```
>>> (65 - 32) * 5 / 9 ↵
18
>>> (65.-32)*5/9 ↵
18.333333333333332
>>>
```

Note that the first example used all integer values, and the result was an integer result. In the second example, the presence of a float caused all the values to be coerced to float.

End-Of-Statement Rules. What happens when the expression is obviously incomplete?

```
>>> ( 65 - 32 ) * 5 / ↵

  File "<stdin>", line 1
    (65-32)*5 /
              ^
SyntaxError: invalid syntax

>>>
```

One basic syntax rule is that statements must be complete on a single line. There are few formal lexical rules for the structure of statements; we'll present them more formally after we've experienced them.

There is an escape clause in the basic rule of "one statement one line". When the parenthesis are incomplete, Python will allow the statement to run on to multiple lines.

```
>>> ( 65 - 32 ↵
... ) * 5 / 9 ↵
18
>>>
```

It is also possible to continue a long statement using a \ escape just before hitting **Return** at the end of the line.

```
>>> 5 + 6 *\ ↵
... 7↵
47
>>>
```

This *escape* allows you to break up an extremely long statement for easy reading. It creates an escape from the usual meaning of the end-of-line character; the end-of-line is demoted to just another whitespace character, and loses it's meaning of "end-of-statement, commence execution now".

Indentation. Python relies heavily on indendentation to make a program readable. When interacting with Python, we are often typing simple expression statements, which are not indented. Later, when we start typing compound statements, indentation will begin to matter.

Here's what happens if we try to indent a simple expression statement.

```
>>>        5+6

SyntaxError: invalid syntax
```

Command History. When we type a complete expression, Python evaluates it and displays the result. When we type a complete statement, Python executes it silently. We'll see more of this, starting in Chapter 6, *Variables, Assignment and Input*. We'll make heavy use of this feature in Chapter 4, *Simple Numeric Expressions and Output*.

Small mistakes can be frustrating when typing a long or complex statement. Python has a reasonable command history capability, so you can use the **up-arrow** key to recover a previous statement. Generally, you'll prefer to create script files and run the scripts. When debugging a problem, however, interactive mode can be handy for experimenting.

One of the desirable features of well-written Python is that most things can be tested and demonstrated in small code fragments. Often a single line of easy-to-enter code is the desired style for interesting programming features. Many examples in reference manuals and unit test scripts are simply captures of interactive Python sessions.

The IDLE Development Environment

There are a number of possible integrated development environments (IDE) for Python. Python includes the IDLE tool, which we'll emphasize. Additionally, you can purchase a number of IDE's that support Python. In Other Tools we'll look at tools that you can use to create a IDE-like toolset.

Starting and Stopping

Starting and stopping IDLE varies with your operating system. Generally, all of the variations are nearly identical, and differ only in minor details.

Windows. There are several ways to start IDLE in Windows. You can use IDLE (Python GUI) from the Python2.5 menu on the Start menu.

You can also run IDLE from the command prompt. This requires two configuration settings in Windows.

- Assure that `C:\Python25\Lib\idlelib` on your system `PATH`. This directory contains `IDLE.BAT`.
- Assure that .pyw files are associated with `C:\Python25\pythonw.exe`. In order suppress creation of a console window for a GUI application, Windows offers `pythonw.exe`.

You can quit IDLE by using the Quit menu item under the File menu.

Mac OS. In the Mac OS, if you've done an upgrade, you may find the IDLE program in the `Python 2.5` folder in your `Applications` folder. You can double-click this icon to run IDLE.

You can always find IDLE in the directory `/System/Library/Frameworks/Python.framework/Versions/Current/bin`. Generally, this is on your PATH, and you can type the command **idle** in a Terminal window to start IDLE. The `idle2.5` icon is a document, not an applicaction, and can't easily be put into your Dock.

You can create your own Applescript icon to run the IDLE shell script. This requires a single line of Applescript:

```
do shell script "/System/Library/Frameworks/Python.framework/Versions/Current/bin/idle"
```

When you run IDLE from the icon, you'll notice that two windows are opened: a Python Shell window and a Console window. The Console window isn't used for much.

When you run IDLE from the Terminal window, no console window is opened. The Terminal window is the Python console.

You can quit IDLE by using the Quit menu item under the File menu. You can also quit by using the Quit Idle menu item under the Idle menu.

Since the Macintosh keyboard has a command key, ⌘, as well as a control key, **ctrl**, there are two keyboard mappings for IDLE. You can use the Configure IDLE... item under the Options menu to select any of the built-in Key Sets. Selecting the IDLE Classic Mac settings may be more comfortable for Mac OS users.

GNU/Linux. We'll avoid the GNOME and KDE subtleties. Instead, we'll focus on running IDLE from the Terminal tool. Since the file path is rather long, you'll want to edit your `.profile` (or `.bash_profile`) to include the following alias definition.

```
alias idle='env python /usr/lib/python2.5/idlelib/idle.py &'
```

This allows you to run IDLE by entering the command **idle** in a Terminal window.

You can quit IDLE by using the Exit menu item under the File menu.

Basic IDLE Operations

Initially, you'll see the following greeting from IDLE.

```
Python 2.5.1 (r251:54863, Oct  5 2007, 21:08:09)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "copyright", "credits" or "license()" for more information.

    **********************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    **********************************************************

IDLE 1.2.1
>>>
```

The personal firewall notification is a reminder that IDLE uses Internetworking Protocols (IP) as part of its debugger. If you have a software firewall on you development computer, and the firewall software complains, you can allow the connection.

IDLE has a simple and relatively standard text editor, which does Python syntax highlighting. It also has a Python Shell window which manages an interactive Python session. You will see that the Python Shell window has a Shell and a Debug menu.

When you use the New menu item in the File menu, you'll see a file window, which has a slightly different menu bar. A file window has name which is a file name (or *untitled*), and two unique menus, a Run and a Format menu.

Generally, you'll use IDLE in two ways:

- You'll enter Python statements in the Python Shell window.
- You'll create files, and run those module files using the Run Module item in the Run menu. This option is usually **F5**.

The Shell Window

The Python Shell window in IDLE presents a >>> prompt. At this prompt, you can enter Python expressions or statements for evaluation. This window has a complete command history, so you can use the **up arrow** to select a previous statement and make changes.

You can refer back to Command-Line Interaction; those interactions will look and behave the same in IDLE as they do on the command line.

The Shell Window is essentially the command-line interface wrapped in a scrolling window. The IDLE interface, however, provides a consistent working environment, which is independent of each operating system's command-line interface.

The Shell and Debug menus provides functions you'll use when developing larger programs. For our first steps with Python, we won't need either of these menus. We'll talk briefly about the functions, but can't really make use of them until we've learned more of the language.

The Shell Menu. The Shell menu is used to restart the Python interpreter, or scroll back through the shell's log to locate the most recent restart. This is important when you are developing a module that is used as a library. When you change that module, you need to reset the shell so that the previous version is forgotten and the new version can be imported into a fresh, empty interpreter.

Generally, being able to work interactively is the best way to develop working programs. It encourages you to create tidy, simple-looking components which you can exercise directly.

The Debug Menu. The Debug menu provides some handy tools for watching how Python executes a program.

- The Go to File/Line item is used to locate the source file where an exception was raised. You click on the exception message which contains the file name and select the Go to File/Line menu item, and IDLE will open the file and highlight the selected line.
- The Debugger item opens an interactive debugger window that allows you to step through the executing Python program.
- The Stack Viewer item opens a window that displays the current Python stack. This shows the arguments and working variables in the Python interpereter. The stack is organized into local and global *namespaces*, a conceot we need to delve into in Chapter 6, *Variables, Assignment and Input*.
- The Auto-open Stack Viewer option will open the Stack Viewer automatically when a program raises an unhandled exception. How exceptions are raised and handled is a concept we'll delve into in Chapter 17, *Exceptions*.

The File Windows

Each file window in IDLE is a simple text editor with two additional menus. The Format menu has a series of items for fairly common source text manipulations. The formatting operations include indenting, commenting, handling tabs and formatting text paragraphs.

The Run menu makes it easy to execute the file you are editing.

- The Python Shell menu item brings up the Python Shell window.
- The Check Module item checks the syntax for your file. If there are any errors, IDLE will highlight the offending line so you can make changes. Additionally, this option will check for inconsistent use of tabs and spaces for indentation.
- The Run Module, **F5**, runs the entire file. You'll see the output in the Python Shell window.

Script Mode

In interactive mode, Python displays the results of expressions. In script mode, however, Python doesn't automatically display results.

In order to see output from a Python script, we'll introduce the **print** statement. This statement takes a list of values and prints their string representation on the standard output file. The standard output is typically directed to the Terminal window. We'll revisit this in depth in The **print** Statement.

```
print "PI = ", 355.0/113.0
```

We can have the Python interpreter execute our script files. Application program scripts can be of any size or complexity. For the following examples, we'll create a simple, two-line script, called `example1.py`.

Example 3.1. example1.py

```
print 65, "F"
print ( 65 - 32 ) * 5 / 9, "C"
```

There are several ways we can start the Python interpreter and have it evaluate our script file.

- Explicitly from the command line. In this case we'll be running Python and providing the name of the script as an argument.
- Implicitly from the command line. In this case, we'll either use the GNU/Linux shell comment (sharp-bang marker) or we'll depend on the file association in Windows.
- Manually from within IDLE. It's important for newbies to remember that IDLE shouldn't be part of the final delivery of a working application. However, this is a great way to start development of an application program.

Running Python scripts from the command-line applies to all operating systems. It is the core of delivering final applications. We may add an icon for launching the application, but under the hood, an application program is essentially a command-line start of the Python interpreter.

Explicit Command Line

The simplest way to execute a script is to provide the script file name as a parameter to the **python** interpreter. In this style, we explicitly name both the interpreter and the input script. Here's an example.

Example 3.2. Command Line Execution

```
python example1.py
```

This will provide the `example1.py` file to the Python interpreter for execution.

Implicit Command-Line Execution

We can streamline the command that starts our application. For POSIX-standard operating systems (GNU/Linux, UNIX and MacOS), we make the script file itself executable and directing the shell to locate the Python interpreter for us. For Windows users, we associate our script file with the `python.exe` interpreter. There are one or two steps to this.

1. Associate your file with the Python interpreter. Except for Windows, you make sure the first line is the following: `#!/usr/bin/env python`. For Windows, you must assure that `.py` files are associated with `python.exe` and `.pyw` files are associated with `pythonw.exe`.

The whole file will look like this:

```
#!/usr/bin/env python
print 65, "F"
print ( 65 - 32 ) * 5 / 9, "C"
```

2. For POSIX-standard operating systems, do a **chmod +x *example1.py*** to make the file *example1.py* executable. You only do this once, typically the first time you try to run the file. For Windows, you don't need to do this.

Now you can run a script in most GNU/Linux environments by saying:

```
./example1.py
```

Windows Setup. Windows users will need to be sure that `python.exe` is on their `PATH`. This is done with the System control panel. Click on the Advanced tab. Click on the Environment Variables… button. Click on the System variables `Path` line, and click the Edit… button. This will often have a long list of items, sometimes starting with "%SystemRoot%". At the end of this list, add ";" and the direction location of `Python.exe`. On my machine, I put it in `C:\Python23`.

For Windows programmers, the windows command interpreter uses the last letters of the file name to associate a file with an interpreter. You can have Windows run the `python.exe` program whenever you double-click a `.py` file. This is done with the **Folder Options** control panel. The File Types tab allows you to pair a file type with a program that processes the file.

POSIX Setup. We have to be sure that the Python interpreter is in value of the `PATH` that our shell uses. We can't delve into the details of each of the available UNIX Shells. However, the general rule is that the person who administers your POSIX computer should have installed Python and updated the `/etc/profile` to make Python available to all users. If, for some reason that didn't get done, you can update your own `.profile` to add Python to your `PATH` variable.

The #! technique depends on the way all of the POSIX shells handle scripting languages. When you enter a command that is the name of a file, the shell must first check the file for the "x" (execute) mode; this was the mode you added with **chmod +x**. When execute mode is true, the shell must then check the first few bytes to see what kind of file it is. The first few bytes are termed the *magic number*; deep in the bowels of GNU/Linux there is a database that shows what the magic number means, and how to work with the various kinds of files. Some files are binary executables, and the operating system handles these directly. Other files, beginning with "#!", are script files. The rest of this first line names the program that will interpret the script. In this case, we asked the **env** program to find the **python** interpreter. The shell finds the named program and runs it automatically, passing the name of script file as the last argument to the interpreter it found.

The very cool part of this trick is that "#!" is a *comment* to Python. This first line is, in effect, directed at the shell, and ignored by Python. The shell glances at it to see what the language is, and Python studiously ignores it, since it was intended for the shell.

Another Script Example

Throughout the rest of this book, we're going to use this script processing mode as the standard way to run Python programs. Many of the examples will be shown as though a file was sent to the interpreter.

For debugging and testing, it is sometimes useful to *import* the program definitions, and

do some manipulations interactively. We'll touch on this in the section called "Hacking Mode".

Here's a second example. We'll create a new file and write another small Python program. We'll call it `example2.py`.

Example 3.3. example2.py

```
#!/usr/bin/env python
"""Compute the odds of spinning red (or black) six times in a row
on an American roulette wheel. """
print (18.0/38.0)**6
```

This is a one-line Python program with a two line module document string. That's a good ratio to strive for.

After we finish editing, we mark this as executable using `chmod +x example2.py`. Since this is a property of the file, this is remains true no matter how many times we edit, copy or rename the file.

When we run this, we see the following.

```
$ ./example2.py
0.0112962280375
```

Which says that spinning six reds in a row is about a one in eighty-nine probability.

Syntax Formalities

What is a Statement?

Informally, we've seen that simple Python statements must be complete on a single line. As we will see in following chapters, compound statements are built from simple and compound statements.

Fundamentally, Python has a simple equivalence between the lexical line structure and the statements in a Python program. This forces us to write readable programs with one statement per line. There are nine formal rules for the lexical structure of Python.

1. Simple statements must be complete on a single Logical Line. Starting in Chapter 7, *Truth, Comparison and Conditional Processing* we'll look at compound statements, which have indented suites of statements, and which span multiple Logical Lines. The rest of these rules will define how Logical Lines are built from Physical Lines through a few Line Joining rules.

2. Physical Lines are defined by the platform; they'll end in standard `\n` or the Windows ASCII CR LF sequence (`\r\n`).

3. Comments start with the # character outside a quoted string; comments end at the end of the physical line. These are not part of a statement; they may occur on a line by themselves or at the end of a statement.

4. Coding-Scheme Comments. Special comments that are by VIM or EMACS can be included in the first or second line of a Python file. For example, `# -*- coding: latin1 -*-`

5. Explicit Line Joining. A `\` at the end of a physical line joins it to the next physical line to make a logical line. This *escapes* the usual meaning of the line end sequence. The two or three-character sequences (`\\n` or `\\r\n`) are treated as a single space.

6. Implicit Line Joining. Expressions with `()`'s, `[]`'s or `{}`'s can be split into multiple physical lines.

7. Blank Lines. When entering statements interactively, an extra blank line is treated as the end of an indented block in a compound statement. Otherwise, blank lines have no signficance.

8. Indentation. The embedded suite of statements in a compound statement must be indented by a consistent number of spaces or tabs. When entering statements interactively or in an editor that knows Python syntax (like IDLE), the indentation will happen automatically; you will outdent by typing a single **backspace**. When using another text editor, you will be most successful if you configure your editor to use four spaces in place of a tab. This gives your programs a consisent look and makes them portable among a wide variety of editors.

9. Whitespace at the beginning of a line is part of indentation, and is significant. Whitespace elsewhere within a line is not significant. Feel free to space things out so that they read more like English and less like computer-ese.

Exercises

Command-Line Exercises

1. **Simple Commands.** Enter the following one-line commands to Python:

   - copyright
   - license
   - credits
   - help

2. **Simple Expressions.** Enter one-line commands to Python to compute the following:

   - 12345 + 23456
   - 98765 - 12345
   - 128 * 256
   - 22 / 7
   - 355 / 113

   - (18-32)*5/9

   - -10*9/5+32

IDLE Exercises

1. **Create an Exercises Directory.** Create a directory (or folder) for keeping your various exercise scripts. Be sure it is not in the same directory in which you installed Python.

2. **Use IDLE's Shell Window.** Start IDLE. Refer back to the exercises in the section called "Command-Line Interaction". Run these exercises using IDLE.

3. **Use IDLE's File Window.** Start IDLE. Note the version number. Use New Window under the File menu to create a simple file. The file should have the following content.

   ```
   """ My First File """
   print   doc
   ```

   Save this file in your exercises directory; be sure the name ends with `.py`. Run your file with the Run Module menu item in the Run menu, usually **F5**.

Script Exercises

1. **Print Script.** Create and run Python file with commands like the following examples: print 12345 + 23456; print 98765 - 12345; print 128 * 256; print 22 / 7.

2. **Another Simple Print Script.** Create and run a Python file with commands like the following examples: `print "one red", 18.0/38.0`; `print "two reds in a row",(18.0/38.0)**2`.

3. **Interactive Differences.** First, run IDLE (or Python) interactively and enter the following "commands": `copyright`, `license`, `credits`. These are special global objects that print interesting things on the interactive Python console.

   Create a Python file with the three commands, each one on a separate line: `copyright`, `license`, `credits`. When you run this, it doesn't produce any output, nor does it produce an error.

   Now create a Python file with three commands, each on a separate line: **print copyright**, **print license**, **print credits**.

   Interestingly, these three global variables have different behavior when used in a script. This is rare. By default, there are just three more variables with this kind of behavior: `quit`, `exit` and `help`.

4. **Numeric Types.** Compare the results of 22/7 and 22.0/7. Explain the differences in the output.

Other Tools

This section lists some additional tools which are popular ways to create, maintain and execute Python programs. While IDLE is suitable for many purposes, you may prefer an IDE with a different level of sophistication.

Any Platform

The Komodo Edit is an IDE that is considerably more sophisticated than IDLE. It is — in a way — too sophisticated for this book. Our focus is on the language, not high-powered IDE's. As with IDLE, this is a tool that runs everywhere, so you can move seamlessly from GNU/Linux to Wiundows to the Mac OS with a single, powerful tool.

See www.komodo.com for more information on ordering and downloading.

Windows

Windows programmers might want to use a tool like Textpad. See www.textpad.com for information on ordering and downloading. Be sure to also download the `python.syn` file from www.textpad.com/add-ons, which has a number of Python syntax coloring configurations.

To use Textpad, you have two setup steps. First, you'll need to add the Python document class. Second you'll need to tell Textpad about the Python tool.

The Python Document Class. You need to tell Textpad about the Python document class. Use the Configure menu; the New Document Class… menu item lets you add Python documents to Textpad. Name your new document class `Python` and click Next. Give your class members named `*.py` and click Next. Locate your `python.syn` file and click Next. Check the new Python document class, and click Next if everything looks right to create a new Textpad document class.

The Python Tool. You'll want to add the Python interpreter as a Textpad tool. Use the

Configure menu again, this time selecting the Preferences… item. Scroll down the list of preferences on the left and click on Tools. On the right, you'll get a panel with the current set of tools and a prominent Add button on the top right-hand side. Click Add, and select Program… from the menu that appears. You'll get a dialog for locating a file; find the `Python.exe` file. Click Okay to save this program as a Textpad tool.

You can check this by using Configure menu and Preferences… item again. Scroll down the list to find Tools. Click the + sign and open the list of tools. Click the Python tool and check the following:

- The Command is the exact path to your copy of `Python.exe`
- The Parameters contains `$File`
- The Initial Folder contains `$FileDir`
- The "capture output" option should be checked

You might also want to turn off the Sound Alert option; this will beep when a program finishes running. I find this makes things a little too noisy for most programs.

Macintosh

Macintosh programmers might want to use a tool like BBEdit. BBEdit can also run the programs, saving the output for you. See www.barebones.com for more information on BBEdit.

To use BBEdit, you have two considerations when writing Python programs.

You must be sure to decorate each Python file with the following line: `#!/usr/bin/env python`. This tells BBEdit that the file should be interpreted by Python. We'll mention this again, when we get to script-writing exericses.

The second thing is to be sure you set the "chdir to Script's Folder" option when you use the the run... item in the #! ("shebang") menu. Without this, scripts are run in the root directory, not in the directory that contains your script file.

Style Notes: Wise Choice of File Names

There is considerable flexibility in the language; two people can arrive at different presentations of Python source. Throughout this book we will present the guidelines for formatting, taken from the Python Enhancement Proposal (PEP) 8, posted on python.sourceforge.net/pep.

We'll include guidelines that will make your programming consistent with the Python modules that are already part of your Python environment. These guidelines should also also make your programming look like other third-party programs available from vendors and posted on the Internet.

Python programs are meant to be readable. The language borrows a lot from common mathematical notation and from other programming languages. Many languages (C++ and Java) for instance, don't require any particular formatting; line breaks and indendentation become merely conventions; bad-looking, hard-to-read programs are common. On the other hand, Python makes the line breaks and indentations part of the language, forcing you to create programs that are easier on the eyes.

General Notes. We'll touch on many aspects of good Python style as we introduce each piece of Python programming. We haven't seen much Python yet, but we do need some guidance to prevent a few tiny problems that could crop up.

First, Python (like all of Linux) is case sensitive. Some languages that are either all uppercase, or insensitive to case. We have worked with programmers who actually find it helpful to hse the Caps Lock key on their keyboard to expedite working in an all-upper-case world. Please don't do this. Python should look like English, where lower-

case letters predominate.

Second, Python makes use of indentation. Most programmers indent very nicely, and the compiler or interpreter ignores this. Python doesn't ignore it. Indentation is useful for write clear, meaning documents and programs are no different.

Finally, your operating system allows a fairly large number of characters to appear in a file name. Until we start writing modules and packages, we can call our files anything that the operating system will tolerate. Starting in Part IV, "Components, Modules and Packages", however, we'll have to limit ourselves to filename that use only letters, digits and _'s. There can be just one ending for the file: `.py`.

A file name like `exercise_1.py` is better than the name `execise-1.py`. We can run both programs equally well from the command line, but the name with the hyphen limits our ability to write larger and more sophisticated programs.

Chapter 4. Simple Numeric Expressions and Output

print Statement and Numeric Operations

Table of Contents

Basic *expressions* are the most central and useful feature of modern programming languages. To see the results of expressions, we'll use a simple output statement. This chapter starts out with the section called "The **print** Statement", which covers the **print** statement. the section called "Numeric Types and Operators" covers the basic numeric data types and operators that are integral to writing expressions Python. the section called "Numeric Conversion Functions" covers conversions between the various numeric types. the section called "Built-In Functions" covers some of the built-in functions that Python provides.

The print Statement

Before delving into numbers, we'll look briefly at the **print** statement. We'll cover just the essential syntax of the **print** statement in this section. While the **print** statement is very handy, it has some odd syntax quirks that we have to defer until later.

> **Note**
>
> Python 3.0 will replace this irregular statement with a built-in function that is perfectly regular, making it simpler to explain and use. The **print** statement is handy, but it has some confusing details that will be simplified.

print Syntax Overview

The **print** statement takes a list of values and, well, prints them. Speaking strictly, it does two things: it converts the objects to strings and puts the characters of those strings on *standard output*. Generally, standard output is the console window where Python was started, although there are ways to change this that are beyond the scope of this book.

Here's a quick summary of the more important features of **print** statement syntax. In short, the keyword, `print`, is followed by a comma-separated list of expressions.

`print` ⟨*`expression`*`,...`⟩

> **Note**
>
> This syntax summary isn't completely correct because it implies that the list of expressions is *terminated* with a comma. Rather than fuss around with complex syntax diagrams (that's what the Python reference manual is for) we've show an approximation that is close enough.
>
> The `,` in a **print** statement is used to *separate* the various expressions.
>
> A `,` can also be used at the end of the **print** statement to change the formatting; this is an odd-but-true feature that is unique to **print** statement syntax.
>
> It's hard to capture all of this sublety in a single syntax diagram.

One of the simplest kind of expressions is a quoted string. You can use either apostrophes (`'`) or quotes (`"`) to surround strings. This gives you some flexibility in your strings. You can put an apostrophe into a quoted string, and you can put quotes into an apostrophe'd string without the special *escapes* that some other languages require. The full set of quoting rules and alternatives, however, will have to wait for Chapter 12, *Strings*.

For example, the following trivial program prints three strings and two numbers.

```
print "Hi, Mom", "Isn't it lovely?", 'I said, "Hi".', 42, 91056
```

Multi-Line Output. Ordinarily, each **print** statement produces one line of output. You can end the **print** statement with a trailing `,` to combine the results of multiple **print** statements into a single line. Here are two examples.

```
print "335/113=",
print 335.0/113.0

print "Hi, Mom", "Isn't it lovely?",
print 'I said, "Hi".', 42, 91056
```

Since the first **print** statement ends with a **,** it does not produce a complete line of output. The second **print** statement finishes the line of output.

Redirecting Output. The **print** statement's output goes to the operating system's standard output file. How do we send output to the system's standard error file? This involves some more advanced concepts, so we'll introduce it with a two-part recipe that we need to look at in more depth. We'll revisit these topics in Part IV, "Components, Modules and Packages".

First, you'll need access to the standard error object; you get this via the following statement.

```
import sys
```

Second, there is an unusual piece of syntax called a "chevron print" which can be used to redirect output to standard error.

print >>*file*, [*expression*,]

The file can be either `sys.stdout` or `sys.stderr`. Here is an example of a small script which produces messages on both stderr and stdout.

Example 4.1. mixedout.py

```
#!/usr/bin/env python
"""Mixed output in stdout and stderr."""
import sys
print >>sys.stderr, "This is an error message"
print "This is stdout"
print >>sys.stdout, "This is also stdout"
```

When you run this inside IDLE, you'll notice that the stderr is colored red, where the stdout is colored black. You'll also notice that the order of the output in IDLE doesn't match the order in our program. Most POSIX operating systems buffer stdout, but does not buffer stderr. Consequently, stdout messages don't appear until the buffer is full, or the program exits.

print Notes and Hints

A program produces a number of kinds of output. The **print** statement is a handy jumping-off point. Generally, we'll replace this with more advanced techiques.

- Final Reports. Our desktop applications may produce text-based report files. These are often done with **print** statements.
- PDF or other format output files. A desktop application which produces PDF or other format files will need to use additional libraries to produce PDF files. For example, ReportLab offers PDF-production libraries. These applications won't make extensive use of **print** statements.
- Error messages and processing logs. Logs and errors are often directed to the standard error file. You won't often use the **print** statement for this, but use the `logging` library.
- Debugging messages. Debugging messages are often handled by the `logging` library.

The **print** statement is a very basic tool for debugging a complex Python program. Feel free to use **print** statements heavily to create a clear picture of what a program is actually doing. Ultimately, you are likely to replace **print** statements with other, more sophisticated methods.

Python Enhancement Proposal (PEP) 3105 suggests writing a `print` function which replaces the irregular **print** statement syntax with a regular function call. This function can then be altered in simple ways to produce more sophisticated output. We'll provide an example of this approach in the section called "Advanced Parameter Handling For Functions".

Numeric Types and Operators

Python provides four basic types of numbers: plain integers, long integers, floating point numbers and complex numbers.

Numbers all have several things in common. Principally, the standard arithmetic operators of `+`, `-`, `*`, `/`, `%` and `**` are all available for all of these numeric types. Additionally, numbers can be compared, using comparison operators that we'll look at in the section called "Comparisons". Also, numbers can be coerced from one type to another.

More sophisticated math is separated into the `math` module, which we will cover later. However, a few advanced math functions are an integral part of Python, including `abs(`*x*`)` and `pow(`*x*`,` *y*`)`.

Integers

Plain integers are at least 32 bits long. The range is at least -2,147,483,648 to 2,147,483,647 (approximately ±2 billion).

Python represents integers as strings of decimal digits. A number does not include any punctuation, and cannot begin with a leading zero (0). Leading zeros are used for base 8 and base 16 numbers. We'll look at this below.

```
>>> 255+100
355
>>> 397-42
355
>>> 71*5
355
>>> 355/113
3
```

While most features of Python correspond with common expectations from mathematics and other programming languages, the division operator, /, poses certain problems. Specifically, the distinction between the algorithm and the data representation need to be made explicit. Division can mean either exact floating-point results or integer results. Mathematicians have evolved a number of ways of describing precisely what they mean when discussing division. We need similar expressive power in Python.We'll look at more details of division operators in the section called "Division Operators".

Octal and Hexadecimal. For historical reasons, Python supports programming in octal and hexadecimal. I like to think that the early days of computing were dominated by people with 8 or 16 fingers.

A number with a leading 0 (zero) is octal, base 8, and uses the digits 0 to 7. `0123` is octal and equal to 83 decimal.

A number with a leading `0x` or `0X` is hexadecimal, base 16, and uses the digits 0 through 9, plus a, A, b, B, c, C, d, D, e, E, f, and F. `0x2BC8` is hexadecimal and equal to 11208.

Hex, octal and long notations can be combined. `0x234C678D098BAL`, for example is `620976988526778L`.

> **Important**
>
> Watch for leading zeros in numbers. If you transcribe programs from other languages, they may use leading zeros on decimal numbers.

Function Notation. The absolute value operation is done using a slightly different notation than the conventional mathematical operators like + and - that we saw above. It uses functional notation, sometimes called *prefix* notation. A mathematician would write $|n|$. Here's the formal Python definition:

`abs` (*`number`*) → number

> Return the absolute value of the argument.

This tells us that `abs(x)` has one parameter that must be a numeric value and it returns a numeric value. It won't work with strings or sequences or any of the other Python data types we'll cover in Chapter 11, *Sequences: Strings, Tuples and Lists*.

Here's an example using the `abs(x)` function.

```
>>> print abs(10-28/2)
4
```

The expression inside the parenthesis is evaluated first (yielding -4). Then the `abs(x)` function is applied to -4. This evaluates to 4.

Long Integers

One of the useful data types that Python offers are long integers. Unlike ordinary integers with a limited range, long integers have arbitrary length; they can have as many digits as necessary to represent an exact answer. However, these will operate more slowly than plain integers. Long integers end in L or l. Upper case L is preferred, since the lower-case l looks too much like the digit 1. Python is graceful about converting to long integers when it is necessary.

How many different combinations of 32 bits are there? The answer is there are 2^{32} different combinations of 32 on-off bits. When we try `2**32` in Python, the answer is too large for ordinary integers, and we get an anwer in long integers.

```
>>> print 2**32
4294967296L
```

There are about 4 billion ways to arrange 32 bits. How many bits in 1K of memory? 1024×8 bits. How many combinations of bits are possible in 1K of memory?

```
print 2L**(1024*8)
```

I won't attempt to reproduce the output from Python. It has 2,467 digits. There are a lot of different combinations of bits in only 1K of memory. The computer I'm using has 256×1024 K of memory; there are a lot of combinations of bits available in that memory.

Python will silently convert between ultra-fast integers and slow-but-large long integers. You can force a conversion using the `int` or `long` factory functions.

Floating-Point Numbers

Python offers floating-point numbers, often implemented as "double-precision" numbers, typically using 64 bits. Floating-point numbers are based on *scientific notation*, where numbers are written as a *mantissa* and an *exponent*. Generally, powers of 10 are used with the exponent, giving us numbers that look like this: $625\times10^{-4}=0.0625$.

When we write a number that includes a decimal point, Python uses floating point representation. We can also use a form of scientific notation with an explicit mantissa and exponent. Here are some examples:

```
.0625
0.0625
6.25E-2
625E-4
```

The last example isn't properly normalized, since the mantissa isn't between 0 and 10.

Generally, a number, n, is some mantissa, g, and an exponent of c. For human consumption, we use a base of 10.

Equation 4.1. Scientific Notation

$$n=g\times10^{c}$$

Internally, most computers use a base of 2, not 10. This leads to slight errors in converting certain values, which are exact in base 10, to approximations in base 2.

For example, 1/5th doesn't have a precise representation. This isn't generally a problem because we have string formatting operations which can make this tiny representation error invisible to users.

```
>>> 1./5
0.20000000000000001
```

Complex Numbers

Besides plain integers, long integers and floating point numbers, Python also provides for imaginary and complex numbers. These use the European convention of ending with J or j. People who don't use complex numbers should skip this section.

`3.14J` is an imaginary number = $3.14\sqrt{-1}$.

A complex number is created by adding a real and an imaginary number: 2 + 14*j*. Note that Python always prints these in ()'s; for example `(2+14j)`.

The usual rules of complex math work perfectly with these numbers.

```
>>> print (2+3j)*(4+5j)
(-7+22j)
```

Python even includes the complex conjugate operation on a complex number. This operation follows the complex number separated by a dot (`.`). This notation is used because the conjugate is treated like a method function of a complex number object (we'll return to this method and object terminology in Chapter 21, *Classes*). For example:

```
>>> print 3+2j.conjugate()
(3-2j)
```

Numeric Conversion Functions

We can convert a number from one type to another. A conversion may involve a loss of precision because we've reduced the number of bits available. A conversion may also add a false sense of precision by adding bits which don't have any real meaning.

We'll call these *factory functions* because they are a factory for creating new objects from other objects. The idea of factory function is a very general one, and these are just the first of many examples of this pattern.

Also, this section is only an introduction. A more complete list of conversion functions is provided in the section called "String Conversion Functions".

Conversion Function Definitions

There are a number of conversions from one numeric type to another.

`int (x)` → integer

> Generates an integer from the object *x*. If *x* is a floating point number, digits to the right of the decimal point are truncated as part of creating an integer. If the floating point number is more than about 10 digits, a long integer object is created to retain the precision. If *x* is a long integer that is too large to be represented as an integer, there's no conversion. If *x* is a string, the string is parsed to create an integer value. Complex values can't be turned into integers directly.

`float (x)` → float

> Generates a float from object *x*. If *x* is an integer or long integer, a floating point number is created. Note that long integers can have a large number of digits, but floating point numbers only have approximately 16 digits; there can be some loss of precision. Complex values can't turned into floating point numbers directly.

`long (x)` → long

> Generates a long integer from *x*. If *x* is a floating point number, digits to the right of the decimal point are truncated as part of creating a long integer.

`complex (real, [imag])` → complex

> Generates a complex number from *real* and *imag*.

If this "`complex (real, [imag])`" syntax synopsis is confusing, you might want to see Syntax For Newbies.

> **Syntax For Newbies**
>
> We'll show optional parameters to functions by surrounding them with []'s. In the case of `complex`, there will be two different ways to use the function, with one parameter, *real*, or with two parameters, *real*, *imag*. We don't actually enter the []'s, they're just hints as to what the two forms of the function are.
>
> In the case of "`complex (real, [imag])`", we have two choices.

```
>>> complex( 2.0, 3.0 )
(2+3j)
>>> complex( 4.0 )
(4+0j)
```

Conversion Function Examples

`float` (*x*) creates a floating point number equal to the string or number *x*. If a string is given, it must be a valid floating point number: digits, decimal point, and an exponent expression. Examples: `float("6.02E24")`, `float(23)`. This is handy when doing division to prevent getting the simple integer quotient.

For example:

```
>>> print float(22)/7, 22/7
3.14285714286 3
```

`int` (*x*) creates an integer equal to the string or number *x*. If a string is given, it must be a valid decimal integer string. Examples: `int("1243")`, `int(3.14159)`.

`long` (*x*) creates a long integer equal to the string or number *x*. If a string is given, it must be a valid decimal integer. The expression `long(2)` has the same value as the literal `2L`. Examples: `long(6.02E23)`, `long(2)`.

For example:

```
>>> print long(2)**64
18446744073709551616
```

This shows the range of values possible with 64-bit integers, available on larger computers.

Complex is not as simple as the others. A complex number has two parts, real and imaginary. Conversion to complex typically involves two parameters.

`complex` (*r*, [*i*]) creates a complex number with the real part of *r*; if the second parameter, *i*, is given, this is the imaginary part of the complex number, otherwise the imaginary part is zero. The string must be a valid complex number. Examples:

```
>>> print complex(3,2), complex(4), complex("3+4j")
(3+2j) (4+0j) (3+4j)
```

Note that the second parameter, with the imaginary part of the number, is optional. This leads to a number of different ways to call this function. In the example above, we used three variations: two numeric parameters, one numeric parameter and one string parameter.

Built-In Functions

Python has a number of *built-in* functions, which are an integral part of the Python interpreter. We can't look at all of them because many are related to features of the language that we haven't addressed yet.

One of the built-in mathematical functions will have to wait for complete coverage until we've introduced the more complex data types, specifically *tuples*, in Chapter 13, *Tuples*. The `divmod` (*x*, *y*) function returns a tuple object with the quotient and remainder in division.

Built-In Math Functions

The bulk of the math functions are in a separate module, called `math`, which we will cover in the section called "The `math` Module". The formal definitions of mathematical built-in functions are provided below.

`abs` (*number*) → number

Return the absolute value of the argument, |*x*|.

`pow` (*x*, *y*, [*z*]) → number

Raise *x* to the *y* power. If *z* is present, this is done modulo *z*, $x^y \% z$.

`round` (*`number`*, [*`ndigits`*]) → float

Round *number* to *ndigits* beyond the decimal point.

`cmp` (*`x`*, *`y`*) → integer

Compare *x* and *y*, returning a number. If the number is less than 0, then *x*<*y*; if the number is zero, then *x* == *y*; if the number is positive, then *x*>*y*.

The `round` (*`number`*, [*`ndigits`*]) function rounds a number to the nearest whole number. If the n parameter is given, this is the number of decimal places to round to. If n is positive, this is decimal places to the right of the decimal point. If n is negative, this is the number of places to the left of the decimal point. Examples: `round(678.456,2)` yields `678.46`; `round(678.456,-1)` yields 680.

The `cmp`(*`x`*, *`y`*) function is handy for the comparisons used when sorting objects into order. For example, `cmp(2,35)` yields `-1`, telling us that the first value is less than the second value.

String Conversion Functions

The string conversion functions provide alternate representations for numeric values. This list expands on the function definitions in the section called "Numeric Conversion Functions".

`hex` (*`number`*) → string

Create a hexadecimal string representation of *number*. A leading '0x' is placed on the string as a reminder that this is hexadecimal. `hex(684)` yields the string `'0x2ac'`.

`oct` (*`number`*) → string

Create a octal string representation of *number*. A leading '0' is placed on the string as a reminder that this is octal not decimal. `oct(509)` yields the string `'0775'`.

`int` (*`string`*, [*`base`*]) → integer

Generates an integer from the string *x*. If *base* is supplied, *x* must be in the given base. If *base* is omitted, *x* must be decimal.

`str` (*`object`*) → string

Generate a string representation of the given object. This is the a "readable" version of the value.

`repr` (*`object`*) → string

Generate a string representation of the given object. Generally, this is the a Python expression that can reconstruct the value; it may be rather long and complex.

The `int` function has two forms. The `int` (*`x`*) form converts a decimal string, *x*, to an integer. For example `int('25')` is `25`. The `int` (*`x`*, *`b`*) form converts a string, *x*, in base *b* to an integer. For example `int('010101',2)` yields 21. `int('321',4)` is 57, `int('2ac',16)` is `684`.

The `str`(*`x`*) and `repr`(*`x`*) functions convert any Python object to a string. The `str`(*`x`*)

version is typically more readable, where the `repr(x)` version is an internalized representation. For most garden-variety numeric values, there is no difference. For the more complex data types, however, the resultsof `repr` and `str` can be very different. For classes you write (see Chapter 21, *Classes*), your class definition must provide these string representation functions.

Collection Functions

These are two built-in functions which operate on a collection of data elements.

`max` (*sequence*) → value

Return the largest value in *sequence*.

`min` (*sequence*) → value

Return the smallest value in *sequence*.

The `max` and `min` functions accept any number of values and return the largest or smallest of the values. `max(1,2,3)` yields 3, `min(1,2,3)` yields `1`.

Expression Exercises

There are two sets of exercises. The first section, the section called "Basic Output and Functions", covers simpler exercises to reinforce Python basics. The secoind section, the section called "Numeric Types and Expressions", covers more complex numeric expressions.

Basic Output and Functions

1. **Print Expression Results.** In Simple Expressions, we entered some simple expressions into the Python interpreter. Change these simple expressions into print statements.

   Be sure to print a label or identifier with each answer. Here's a sample.

   ```
   print "9-1's * 9-1's = ", 111111111*111111111
   ```

2. **Evaluate and Print Expressions.** Write short scripts to print the results of the following expressions. In most places, changing integers to floating point produces a notably different result. For example `(296/167)**2` and `(296.0/167.0)**2`. Use long as well as complex types to see the differences.

   - `355/113 * ( 1 - 0.0003/3522 )`
   - `22/17 + 37/47 + 88/83`
   - `(553/312)**2`

3. **Numeric Conversion.** Write a print statement to print the mixed fraction 3 5/8 as a floating point number and as an integer.

4. **Numeric Truncation.** Write a print statement to compute `(22.0/7.0)-int(22.0/7.0)`. What is this value? Compare it with `22.0/7.0`. What general principal does this illustrate?

5. **Illegal Conversions.** Try illegal conversions like `int('A')` or `int( 3+4j )`. Why are exceptions raised? Why can't a simple default value like zero or `None` be used instead?

6. **Evaluate and Print Built-in Math Functions.** Write short scripts to print the

results of the following expressions.

- `pow( 2143/22, 0.25 )`
- `pow(553/312,2)`
- `pow( long(3), 64 )`
- `long( pow(float(3), 64) )`

Why do the last two produce different results? What does the difference between the two results tell us about the number of digits of precision in floating-point numbers?

7. **Evaluate and Print Built-in Conversion Functions.** Here are some more expressions for which you can print the results.
   - hex(1234)
   - int(hex(1234), 16)
   - long('0xab')
   - int('0xab')
   - int('0xab', 16)
   - int('ab', 16)
   - cmp(2, 3)

Numeric Types and Expressions

1. **Stock Value.** Compute value from number of shares × purchase price for a stock.

   Once upon a time, stock prices were quoted in fractions of a dollar, instead of dollars and cents. Create a simple print statement for 125 shares purchased at 3 and 3/8. Create a second simple print statement for 150 shares purchased at 2 1/4 plus an additional 75 shares purchased at 1 7/8.

   Don't manually convert 1/4 to 0.25. Use a complete expression of the form 2+1/4.0, just to get more practice writing expressions.

2. **Convert Between °C and °F.** Convert temperatures from one system to another.

   Conversion Constants: 32°F = 0°C, 212°F = 100°C.

   The following two formulae converts between °C (Celsius) and °F (Fahrenheit).

   Equation 4.2. Convert °C (Celsius) to °F (Fahrenheit)

   $$F = 32 + \frac{(212 - 32)}{100} \times C$$

   Equation 4.3. Convert °F (Fahrenheit) to °C (Celsius)

   $$C = (F - 32) \times \frac{100}{212 - 32}$$

   Create a print statement to convert 18° C to °F.

   Create a print statement to convert -4° F to °C.

3. **Periodic Payment on a Loan.** How much does a loan really cost?

   Here are three versions of the standard mortgage payment calculation, with *m*=payment, *p*=principal due, *r*=interest rate, *n*=number of payments.

   Equation 4.4. Mortage Payment, version 1

   $$m = p\left(\frac{r}{1-(1+r)^{-n}}\right)$$

   Equation 4.5. Mortgage, payments due at the end of each period

   $$m = \frac{-r\,p\,(r+1)^n}{(r+1)^n - 1}$$

   Equation 4.6. Mortgage, payments due at the beginning of each period

   $$m = \frac{-r\,p\,(r+1)^n}{[(r+1)^n - 1](r+1)}$$

   Use any of these forms to compute the mortgage payment, *m*, due with a principal, *p*, of $110,000, an interest rate, *r*, of 7.25% annually, and payments, *n*, of 30 years. Note that banks actually process things monthly. So you'll have to divide the interest rate by 12 and multiply the number of payments by 12.

   Note that the results are negative numbers. You are, after all, paying *down* a principle.

4. **Surface Air Consumption Rate.** SACR is used by SCUBA divers to predict air used at a particular depth.

   For each dive, we convert our air consumption at that dive's depth to a normalized air consumption at the surface. Given depth (in feet), *d*, starting tank pressure (psi), *s*, final tank pressure (psi), *f*, and time (in minutes) of *t*, the SACR, *c*, is given by the following formula.

   Equation 4.7. Surface Air Consumption Rate

   $$c = \frac{33(s-f)}{t(d+33)}$$

   Typical values for pressure are a starting pressure of 3000, final pressure of 500.

   A medium dive might have a depth of 60 feet, time of 60 minutes.

   A deeper dive might be to 100 feet for 15 minutes.

   A shallower dive might be 30 feet for 60 minutes, but the ending pressure might be 1500. A typical *c* (consumption) value might be 12 to 18 for most people.

   Write print statements for each of the three dive profiles given above: medium, deep and shallow.

   Given the SACR, *c*, and a tank starting pressure, *s*, and final pressure, *f*, we can plan a dive to depth (in feet), *d*, for time (in minutes), *t*, using the following formula. Usually the 33(*s*-*f*)/*c* is a constant, based on your SACR and tanks.

   Equation 4.8. Time and Depth from SACR

$$33\frac{(s-f)}{c} = t(d+33)$$

For example, tanks you own might have a starting pressure of 2500 and and ending pressure of 500, you might have a *c* (SACR) of 15.2. You can then find possible combinations of time and depth which you can comfortably dive.

Write two print statements that shows how long one can dive at 60 feet and 70 feet.

5. **Force on a Sail.** How much force is on a sail?

   A sail moves a boat by transferring force to its mountings. The sail in the front (the jib) of a typical fore-and-aft rigged sailboat hangs from a stay. The sail in the back (the main) hangs from the mast. The forces on the stay (or mast) and sheets move the boat. The sheets are attached to the clew of the sail.

   The force on a sail, *f*, is based on sail area, *a* (in square feet) and wind speed, *w* (in miles per hour).

   Equation 4.9. Sail Clew Load

   $$f = w^2 \times 0.004a$$

   For a small racing dinghy, the smaller sail in the front might have 61 square feet of surface. The larger, mail sail, might have 114 square feet.

   Write a print statement to figure the force generated by a 61 square foot sail in 15 miles an hour of wind.

6. **Craps Odds.** What are the odds of winning on the first throw of the dice?

   There are 36 possible rolls on 2 dice that add up to values from 2 to 12. There is just 1 way to roll a 2, 6 ways to roll a 7, and 1 way to roll a 12. We'll take this as given until a later exercise where we have enough Python to generate this information.

   Without spending a lot of time on probability theory, there are two basic rules we'll use time and again. If any one of multiple alternate conditions needs to be true, usually expressed as “or”, we add the probabilities. When there are several conditions that must all be true, usually expressed as “and”, we multiply the probabilities.

   Rolling a 3, for instance, is rolling a 1-2 *or* rolling a 2-1. We add the probabilities: 1/36+1/36 = 2/36 = 1/18.

   On a come out roll, we win immediately if 7 or 11 is rolled. There are two ways to roll 11 (2/36) or 6 ways to roll 7 (6/36).

   Write a print statement to print the odds of winning on the come out roll. This means rolling 7 or rolling 11. Express this as a fraction, not as a decimal number; that means adding up the numerator of each number and leaving the denominator as 36.

7. **Roulette Odds.** How close are payouts and the odds?

   An American (double zero) roulette wheel has numbers 1-36, 0 and 00. 18 of the 36 numbers are red, 18 are black and the zeroes are green. The odds of spinning red, then are 18/38. The odds of zero or double zero are 2/36.

   Red pays 2 to 1, the real odds are 38/18.

Write a print statement that shows the difference between the pay out and the real odds.

You can place a bet on 0, 00, 1, 2 and 3. This bet pays 6 to 1. The real odds are 5/36.

Write a print statement that shows the difference between the pay out and the real odds.

The Print Function

If you have Python 2.6, you can start using with the `print` function. The print function is not part of Python 2.6, it's a future expansion. To ease the transition from older to newer language features, there is a `__future__` module available. You can get this future expansion by using the following statement at the beginning of your script or working session in IDLE.

```
from __future__ import print_function
```

This will alert Python that you want to use the `print` function instead of the **print** statement. We'll look at the **import** statement in depth in Part IV, "Components, Modules and Packages".

The print function has the following formal definition.

`print (`*`object`*`,...,[,`*`sep`*`,][,`*`end`*`,][,`*`file`*`])` → number

Convert the objects to strings and write the strings to the given file.

If *`sep`* is not specified, it is a single space. If *`end`* is not specified, it is a single `\n`. If the *`file`* is not specified it is `sys.stdout`.

Examples:

```
from __future__ import print_function
import sys

print("335/113=",335.0/113.0)

print("Hi, Mom", "Isn't it lovely?", end='')
print('I said, "Hi".', 42, 91056)

print("Red Alert!", file-sys.stderr)
```

The **print** statement uses a trailing , to suppress the newline that defines the end of a line of output. To do this with the `print` function, use `end=''`.

To send output to standard error, use `file=sys.stderr`.

Expression Style Notes

Spaces are used sparingly in expressions. Spaces are never used between a function name and the ()'s that surround the arguments. It is considered poor form to write:

```
int (22.0/7)
```

The preferred form is the following:

```
int(22.0/7)
```

A long expression may be broken up with spaces to enhance readability. For example, the following separates the multiplication part of the expression from the addition part

with a few wisely-chosen spaces.

```
b**2 - 4*a*c
```

Chapter 5. Advanced Expressions

The `math` and `random` Modules, Bit-Level Operations, Division

Table of Contents

This chapter covers some more advanced topics. the section called "The `math` Module" covers two important modules, `math` and `random`. the section called "Division Operators" covers the important distinction between the division operators. We also provide some supplemental information that is more specialized. the section called "Bit Manipulation Operators" covers some additional bit-fiddling operators that work on the basic numeric types. the section called "Expression Style Notes" has some notes on style.

Using Modules

A Python module extends the Python execution environment by adding new classes, functions and helpful constants. We tell the Python interpreter to fetch a module with a variation on the **import** statement. There are several variations on **import**, which we'll cover in depth in Part IV, "Components, Modules and Packages".

For now, we'll use the simple **import**:

```
import m
```

This will import module `m`. Only the module's name, `m` is made available. Every name inside the module `m` must be *qualified* by prepending the module name and a `.`. So if module `m` had a function called `spam`, we'd refer to it as `m.spam`.

There are dozens of standard Python modules. We'll get to the most important ones in Part IV, "Components, Modules and Packages". For now, we'll focus on extending the math capabilities of the basic expressions we've looked so far.

The `math` Module

The `math` module is made available to your programs with:

```
import math
```

The `math` module contains the following trigonometric functions

`math.acos` (*x*) → number

arc cosine of *x*.

`math.asin` (*x*) → number

arc sine of *x*.

`math.atan` (*x*) → number

arc tangent of *x*.

`math.atan2 (y, x )` → number

arc tangent of *y*/*x*.

`math.cos (x )` → number

cosine of *x*.

`math.cosh (x )` → number

hyperbolic cosine of *x*.

`math.exp (x )` → number

e**x, inverse of log(*x*).

`math.hypot (x, y )` → number

Euclidean distance, sqrt(*x***x* + *y***y*), length of the hypotenuse of a right triangle with height of *y* and length of *x*.

`math.log (x )` → number

natural logarithm (base e) of *x*, inverse of exp(*x*).

`math.log10 (x )` → number

natural logarithm (base 10) of *x*, inverse of 10**x.

`math.pow (x, y )` → number

*x****y*.

`math.sin (x )` → number

sine of *x*.

`math.sinh (x )` → number

hyperbolic sine of *x*.

`math.sqrt (x )` → number

square root of *x*. This version returns an error if you ask for sqrt(-1), even though Python understands complex and imaginary numbers. A second module, `cmath`, includes a version of `sqrt(x)` which correctly creates imaginary numbers.

`math.tan (x )` → number

tangent of *x*.

`math.tanh (x )` → number

hyperbolic tangent of *x*.

Additionally, the following constants are also provided.

`math.pi`

the value of pi, 3.1415926535897931

`math.e`

the value of e, 2.7182818284590451, used for the `exp(x)` and `log(x)` functions.

The math module contains the following other functions for dealing with floating point numbers.

`math.ceil (x)` → number

next larger whole number. `math.ceil(5.1) == 6`, `math.ceil(-5.1) == -5.0`.

`math.fabs (x)` → number

absolute value of the real *x*.

`math.floor (x)` → number

next smaller whole number. `math.floor(5.9) == 5`, `math.floor(-5.9) == -6.0`.

`math.fmod (x, y)` → number

floating point remainder after division of *x*/*y*. This depends on the platform C library and may return a different result than the Python `x % y`.

`math.modf (x)` → (number, number)

creates a tuple with the fractional and integer parts of *x*. Both results carry the sign of *x* so that *x* can be reconstructed by adding them.

`math.frexp (x)` → (number, number)

this function unwinds the usual base-2 floating point representation. A floating point number is *m**2***e*, where *m* is always a fraction between 1/2 and 1, and *e* is an integer power of 2. This function returns a tuple with *m* and *e*. The inverse is `ldexp(m,e)`.

`math.ldexp (m, e)` → number

*m**2***e*, the inverse of `frexp(x)`.

The `random` Module

The `random` module is made available to your program with:

```
import random
```

The `random` module contains the following functions for working with simple distributions of random numbers. There are numerous other, more sophisticated distributions available, but some later exercises will only use these functions.

`random.choice (sequence)` → value

chooses a random value from the sequence *sequence*. Example: `random.choice( ['red', 'black', 'green'] )`.

`random.random` → number

a random floating point number, *r*, $0 \le r < 1.0$.

`random.randrange ([start,] stop [,step])` → integer

choose a random element from `range` (*start*, *stop*, *step*). Examples: `randrange(6)` returns a number, n, $0 \le n < 6$. `randrange(1,7)` returns a number, n, $1 \le n < 7$. `randrange(10,100,5)` returns a number, n, between 10 and 95 incremented by 5's, $10 \le 5k < 100$.

`random.uniform` (*a*, *b*) → number

a random floating point number, r, $a \le r < b$.

The `randrange` has two optional values, making it particularly flexible. Here's an example of some of the alternatives.

Example 5.1. demorandom.py

```
#!/usr/bin/env python
import random
# Simple Range 0 <= r < 6
print random.randrange(6), random.randrange(6)
# More complex range 1 <= r < 7
print random.randrange(1,7), random.randrange(1,7)
# Really complex range of even numbers between 2 and 36
print random.randrange(2,37,2)
# Odd numbers from 1 to 35
print random.randrange(1,36,2)
```

This demonstrates a number of ways of generating random numbers. It uses the basic `random.randrange` with a variety of different kinds of arguments.

Advanced Expression Exercises

1. **Evaluate These Expressions.** The following expressions are somewhat more complex, and use functions from the math module.

   ```
   math.sqrt( 40.0/3.0 - math.sqrt(12.0) )

   6.0/5.0*( (math.sqrt(5)+1) / 2 )**2

   math.log( 2198 ) / math.sqrt( 6 )
   ```

2. **Run demorandom.py.** Run the `demorandom.py` script several times and save the results. Then add the following statement to the script and run it again several times. What happens when we set an explicit seed?

   ```
   #!/usr/bin/env python
   import random
   random.seed(1)
   ...everything else the same
   ```

   Try the following variation, and see what it does.

   ```
   #!/usr/bin/env python
   import random, time
   random.seed(time.clock())
   ...everything else the same
   ```

3. **Wind Chill.** Wind chill is used by meteorologists to describe the effect of cold and wind combined.

   Given the wind speed in miles per hour, V, and the temperature in °F, T, the Wind Chill, w, is given by the formula below.

   Equation 5.1. Wind Chill, Old Model

$$0.081 \times (3.71 \times \sqrt{V} + 5.81 - 0.25 \times V) \times (T - 91.4) + 91.4$$

Equation 5.2. Wind Chill, New Model

$$35.74 + 0.6215 \times T - 35.75 (V^{0.16}) + 0.4275 \times T \times (V^{0.16})$$

Wind speeds are for 0 to 40 mph, above 40, the difference in wind speed doesn't have much practical impact on how cold you feel.

Write a print statement to compute the wind chill felt when it is -2 °F and the wind is blowing 15 miles per hour.

4. **How Much Does The Atmosphere Weigh?** From *Slicing Pizzas, Racing Turtles, and Further Adventures in Applied Mathematics*, [Banks02].

   Air Pressure (at sea level) $P_0 = 1.01325 \times 10^5$ newtons/m^2. A newton is 1 kg·m/sec^2 or the force acting on a kg of mass.

   Gravity acceleration (at sea level) $g = 9.82$m/sec^2. This converts force from newtons to kg of mass.

   Equation 5.3. Mass of air per square meter (in Kg)

   $$M_{m2} = P_0 \times g$$

   Given the mass of air per square meter, we need to know how many square meters of surface to apply this mass to.

   Radius of Earth $R = 6.37 \times 10^6$m.

   Equation 5.4. Area of a Sphere

   $$A = 4 \pi r^2$$

   Equation 5.5. Mass of atmosphere (in Kg)

   $$M_a = P_0 \times g \times A$$

   Check: somewhere around 10^{18}kg.

5. **How much does the atmosphere weigh? Part 2.** From *Slicing Pizzas, Racing Turtles, and Further Adventures in Applied Mathematics*, [Banks02].

   The exercise How Much Does The Atmosphere Weigh? assumes the earth to be an entirely flat sphere. The averge height of the land is actually 840m. We can use the ideal gas law to compute the pressure at this elevation and refine the number a little further.

   Equation 5.6. Pressure at a given elevation

   $$P = P_0 \times e^{-\left(\frac{mg}{RT}\right)x}$$

   molecular weight of air $m = 28.96 \times 10^{-3}$ kg/mol.

   gas constant $R = 8.314$ joule/°Kmol.

   gravity $g = 9.82$ m/sec^2.

   temperature (°K) $T = 273° + °C$ (assume 15°C for temperature).

elevation z = 840 m.

This pressure can be used for the air over land, and the pressure computed in How Much Does The Atmosphere Weigh? can be used for the air over the oceans. How much land has this reduced pressure? Reference material gives the ocean area (A_o = 3.61×10^{14} m^2) and the land area (A_l = 1.49×10^{14} m^2).

Equation 5.7. Weight of Atmosphere, adjusted for land elevation

$$M_l = P_0 \times g \times A_o + P \times g \times A_l$$

Bit Manipulation Operators

We've already seen the usual math operators: +, -, *, /, %, **; as well as the `abs(x)` and `pow(x, y)` functions. There are several other operators available to us. Principally, these are for manipulating the individual bits of an integer value.

The unary ~ operator flops all the bits in a plain or long integer. 1's become 0's and 0's become 1's. Since most hardware uses a technique called 2's complement, this is mathematically equivalent to adding 1 and switching the number's sign.

```
>>> print ~0x12345678
-305419897
```

There are binary bit manipulation operators, also. These perform simple Boolean operations on all bits of the integer at once.

The binary & operator returns a 1-bit if the two input bits are both 1.

```
>>> print 0&0, 1&0, 1&1, 0&1
0 0 1 0
```

Here's the same kind of example, combining sequences of bits. This takes a bit of conversion to base 2 to understand what's going on.

```
>>> print 3&5
1
```

The number 3, in base 2, is `0011`. The number 5 is `0101`. Let's match up the bits from left to right:

```
  0 0 1 1
& 0 1 0 1
  -------
  0 0 0 1
```

The binary ^ operator returns a 1-bit if one of the two inputs are 1 but not both. This is sometimes called the exclusive or.

```
>>> print 3^5
6
```

Let's look at the individual bits

```
  0 0 1 1
^ 0 1 0 1
  -------
  0 1 1 0
```

Which is the binary representation of the number 6.

The binary | operator returns a 1-bit if either of the two inputs is 1. This is sometimes called the inclusive or. Sometimes this is written *and/or*.

```
>>> print 3|5
7
>>>
```

Let's look at the individual bits.

```
  0 0 1 1
| 0 1 0 1
  -------
  0 1 1 1
```

Which is the binary representation of the number 7.

There are also bit shifting operations. These are mathematically equivalent to multiplying and dividing by powers of two. Often, machine hardware can execute these operations faster than the equivalent multiply or divide.

The << is the left-shift operator. The left argument is the bit pattern to be shifted, the right argument is the number of bits.

```
>>> print 0xA << 2
40
```

`0xA` is hexadecimal; the bits are 1-0-1-0. This is 10 in decimal. When we shift this two bits to the left, it's like multiplying by 4. We get bits of 1-0-1-0-0-0. This is 40 in decimal.

The >> is the right-shift operator. The left argument is the bit pattern to be shifted, the right argument is the number of bits. Python always behaves as though it is running on a 2's complement computer. The left-most bit is always the sign bit, so sign bits are shifted in.

```
>>> print 80 >> 3
10
```

The number 80, with bits of 1-0-1-0-0-0-0, shifted right 3 bits, yields bits of 1-0-1-0, which is 10 in decimal.

There are some other operators available, but, strictly speaking, they're not arithmetic operators, they're logic operations. We'll return to them in Chapter 7, *Truth, Comparison and Conditional Processing*.

Division Operators

In general, the data type of an expresion depends on the types of the arguments. This rule meets our expectations for most operators: when we add two integers, the result should be an integer. However, this doesn't work out well for division because there are two different expectations. Sometimes we expect division to create precise answers, usually the floating-point equivalents of fractions. Other times, we want a rounded-down integer result.

The classical Python definition of `/` depended entirely on the arguments. 685/252 was 2 because both arguments where integers. However, 685./252. was 2.7182539682539684 because the arguments were floating point.

This definition often caused problems for applications where data types were used that the author hadn't expected. For example, a simple program doing Celsius to Fahrenheit conversions will produce different answers depending on the input. If one user provides 18 and another provides 18.0, the answers were different, even though all of the inputs all had the equal numeric values.

```
>>> print 18*9/5+32
64
>>> print 18.0*9/5 + 32
64.4
>>> 18 == 18.0
True
>>>
```

This unexpected inaccuracy was generally due to the casual use of integers where floating-point numbers were more appropriate. (This can also occur using integers where complex numbers were implictly expected.) An explicit conversion function (like `float(x)`) can help prevent this. The idea, however, is for Python be a simple and sparse language, without a dense clutter of conversions to cover the rare case of an unexpected data type.

Starting with Python 2.2, a new division operator was added to clarify what is required: `/` and `//`. The `/` operator should return floating-point results; the `//` operator, will return rounded-down results. In Python 2.5 and 2.6, this distinction is turned off by default. In Python 3.0, this will be turned on.

To help with the transition, two tools were made available. These tools allow the division operator, `/`, to follow the old rules or the new rules. This gives programmers a way to keep older applications running; it also gives them a way to explicitly declare that their program uses the newer operator definition. There are two parts to this: a statememt that can be placed in a program, as well as a command-line option that can be used when starting the Python interpreter.

Program Statements. To ease the transition from older to newer language features, there is a `__future__` module available. This module includes a `division` definition that changes the definition of the `/` operator from classical to future. You can include the following **from** statement to state that your program depends on the future definition of division. We'll look at the **import** statement in depth in Part IV, “Components, Modules and Packages”.

```
from __future__ import division
print 18*9/5+32
print 18*9//5+32
```

This produces the following output. The first line shows the new use of the `/` operator to produce floating point results, even if both arguments are integers. The second line shows the `//` operator, which produces rounded-down results.

```
64.4
64
```

The **from __future__** statement will set the expectation that your script uses the new-style floating-point division operator. This allows you to start writing programs with version 2.5 that will work correctly with all future versions. By version 3.0, this import statement will no longer be necessary, and these will have to be removed from the few modules that used them.

Command Line Options. Another tool to ease the transition is a command-line option used when running the Python interpreter. This can force old-style interpretation of the `/` operator or to warn about old-style use of the `/` operator between integers. It can also force new-style use of the `/` operator and report on all potentially incorrect uses of the `/` operator.

The Python interpreter command-line option of `-Q` will force the `/` operator to be treated classically (“old”), or with the future (“new”) semantics. If you run Python with `-Qold`, the `/` operator's result depends on the arguments. If you run Python with `-Qnew`, the `/` operator's result will be floating point. In either case, the `//` operator returns a rounded-

down integer result.

You can use `-Qold` to force old modules and programs to work with version 2.2 and higher. When Python 3.0 is released, however, this transition will no longer be supported; by that time you should have fixed your programs and modules.

To make fixing easier, the `-Q` command-line option can take two other values: `warn` and `warnall`. If you use `-Qwarn`, then the `/` operator applied to integer arguments will generate a run-time warning. This will allow you to find and fix situations where the `//` operator might be more appropriate. If you use `-Qwarnall`, then all instances of the `/` operator generate a warning; this will give you a close look at your programs.

You can include the command line option when you run the Python interpreter. For Linux and MacOS users, you can also put this on the `#!` line at the beginning of your script file.

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

Chapter 6. Variables, Assignment and Input

The =, augmented = and del Statements

Table of Contents

Variables hold the state of our program. In the section called “Variables” we'll introduce variables, then in the section called “The **Assignment** Statement” we'll cover the basic *assignment* statement for changing the value of a variable. This is followed by an exercise section that refers back to exercises from Chapter 4, *Simple Numeric Expressions and Output*. In the section called “Input Functions” we introduce some primitive interactive input functions that are built-in. This is followed by some simple exercises that build on those from section the section called “The **Assignment** Statement”. We'll cover the multiple assignment statement in the section called “Multiple Assignment Statement”. We'll round on this section with the **del** statement, for removing variables in the section called “The **del** Statement”.

Variables

As a multi-statement program makes progress from launch to completion, it does so by undergoing changes of state. The state of our program as a whole is the state of all of the program's variables. When one variable changes, the overall state has changed.

Variables are the names your program assigns to the results of an expression. Every variable is created with an initial value. Variables will change to identify new objects and the objects identified by a variable can change their internal state. These three kinds of state changes (variable creation, object assignment, object change) happen as inputs are accepted and our program evaluates expressions. Eventually the state of the variables indicates that we are done, and our program can exit.

A Python variable name must be at least one letter, and can have a string of numbers, letters and _'s to any length. Names that start with _ or __ have special significance. Names that begin with _ are typically private to a module or class. We'll return to this notion of privacy in Chapter 21, *Classes* and Chapter 28, *Modules*. Names that begin with __ are part of the way the Python interpreter is built.

Example variable names include `a`, `pi`, `aVeryLongName`, `a_name`, `__str__` and `_hidden`.

Tracing Execution

We can trace the execution of a program by simply following the changes of value of all the variables in the program. For programming newbies, it helps to create a list of variables and write down their changes when studying a program. We'll show some examples in the next section.

Python creates new objects as the result of evaluating an expression. Python creates new variables with an **assignment** statement; it also assigns an object to the variable. Python removes variables with a **del** statement.

Some Consequences. A Python variable is little more than a name which refers to an object. The central issue is to recognize that the underlying object is the essential part of our program; a variable name is just a meaningful label. This has a number of important consequences.

One consequence of a variable being simple a label is that any number of variables can refer to the same object. In other languages (C, C++, Java) there are two kinds of values: primitive and objects, and there are distinct rules for handling the two kinds of values. In Python, every variable is a simple reference to an underlying object. When talking about simple immutable objects, like the number 3, multiple variables referring to a common object is functionally equivalent to having a distinct copy of a primitive value. When talking about mutable objects, like lists, mappings, or complex objects, distinct variable references can change the state of the common object.

Another consequences is that the Python object fully defines it's own *type*. The object's type defines the representation, the range of values and the allowed operations on the object. The type is established when the object is created. For example, floating point addition and long integer objects have different representations, operations of adding these kinds of numbers are different, the objects created by addition are of distinct types. Python uses the type information to choose which addition operation to perform on two values. In the case of an expression with mixed types Python uses the type information to coerce one or both values to a common type.

We've already worked with the four numeric types: plain integers, long integers, floating point numbers and complex numbers. We've touched on the string type, also. There are several other built-in types that we will look at in detail in Part II, “Data Structures”. Plus, we can use class definitions to define new types to Python, something we'll look at in Part III, “Data + Processing = Objects”.

We commonly say that a *static* language associates the type information with the variable. Only values of a certain type can be assigned to a given variable. Python, in

contrast, is a *dynamic* language; a variable is just a label or tag attached to the object. Any variable can be associated with an object of any type.

The final consequence of variables referring to objects is that a variable's scope can be independent of the object itself. This means that variables which are in distinct namespaces can refer to the same object. When a function completes execution and the namespace is deleted, the variables are deleted, and the number of variables referring to an object is reduced. Additional variables may still refer to an object, meaning that the object will continue to exist. When only one variable refers to an object, then removing the last variable removes the last reference to the object, and the object can be removed from memory.

Also note that a expressions generally create new objects; if an object is not saved in a variable, it silently vanishes. We can safely ignore the results of a function.

Scope and Namespaces. A Python variable is a name which refers to an object. To be useful, each variable must have a *scope* of visibility. The scope is defined as the set of statements that can make use of this variable. A variable with *global scope* can be referenced anywhere. On the other hand, variable with *local scope* can only be referenced in a limited suite of statements.

This notion of scope is essential to being able to keep a intellectual grip on a program. Programs of even moderate complexity need to keep pools of variables with separate scopes. This allows you to reuse variable names without risk of confusion from inadvertantly changing the value of a variable used elsewhere in a program.

Python collects variables into pools called *namespaces*. A new namespace is created as part of evaluating the body of a function or module, or creating a new object. Additionally, there is one global namespace. This means that each variable (and the state that it implies) is isolated to the execution of a single function or module. By separating all locally scoped variables into separate namespaces, we don't have an endless clutter of global variables.

In the rare case that you need a global variable, the **global** statement is available to assign a variable to the global namespace.

When we introduce functions in Chapter 9, *Functions*, classes in Chapter 21, *Classes* and modules in Part IV, "Components, Modules and Packages", we'll revisit this namespace technique for managing scope. In particular, see the section called "Functions and Namespaces" for a digression on this.

The Assignment Statement

Assignment is fundamental to Python; it is how the objects created by an expression are preserved. We'll look at the basic assignment statement, plus the augmented assignment statement. Later, in Multiple Assignment Statement, we'll look at multiple assignment.

Basic Assignment

We create and change variables primarily with the *assignment* statement. This statement provides an expression and a variable name which will be used to label the value of the expression.

```
variable = expression
```

Here's a short script that contains some examples of assignment statements.

Example 6.1. example3.py

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

```
# Computer the value of a block of stock
shares= 150
price= 3 + 5.0/8.0
value= shares * price
print value
```

We have an object, the number `150`, which we assign to the variable `shares`. We have an expression `3+5.0/8.0`, which creates a floating-point number, which we save in the variable `price`. We have another expression, `shares * price`, which creates a floating-point number; we save this in `value` so that we can print it. This script created three new variables.

Since this file is new, we'll need to do the **chmod +x example3.py** once, after we create this file. Then, when we run this progam, we see the following.

```
$ ./example3.py
543.75
$
```

Augmented Assignment

Any of the usual arithmetic operations can be combined with assignment to create an *augmented assignment* statement.

For example, look at this augmented assignment statement:

```
a += v
```

This statement is a shorthand that means the same thing as the following:

```
a = a + v
```

Here's a larger example

Example 6.2. portfolio.py

```
#!/usr/bin/env python
# Total value of a portfolio made up of two blocks of stock
portfolio = 0
portfolio += 150 * 2 + 1/4.0
portfolio += 75 * 1 + 7/8.0
print portfolio
```

First, we'll do the **chmod +x portfolio.py** on this file. Then, when we run this progam, we see the following.

```
$ ./portfolio.py
376.125
$
```

The other basic math operations can be used similarly, although the purpose gets obscure for some operations. These include -=, *=, /=, %=, &=, ^=, |=, <<= and >>=.

Here's a lengthy example. This is an extension of Craps Odds in the section called "Numeric Types and Expressions".

In craps, the first roll of the dice is called the "come out roll". This roll can be won immediately if one rolls 7 or 11. It can be lost immediately if one roll 2, 3 or 12. The remaining numbers establish a point and the game continues.

Example 6.3. craps.py

```
#!/usr/bin/env python
# Compute the odds of winning on the first roll
win = 0
win += 6/36.0 # ways to roll a 7
win += 2/36.0 # ways to roll an 11
print "first roll win", win

# Compute the odds of losing on the first roll
lose = 0
lose += 1/36.0 # ways to roll 2
lose += 2/36.0 # ways to roll 3
lose += 1/36.0 # ways to roll 12
print "first roll lose", lose

# Compute the odds of rolling a point number (4, 5, 6, 8, 9 or 10)
point = 1 # odds must total to 1
point -= win # remove odds of winning
point -= lose # remove odds of losting
print "first roll establishes a point", point
```

There's a 22.2% chance of winning, and a 11.1% chance of losing. What's the chance of establishing a point? One way is to figure that it's what's left after winning or loosing. The total of all probabilities always add to 1. Subtract the odds of winning and the odds of losing and what's left is the odds of setting a point.

Here's another way to figure the odds of rolling 4, 5, 6, 8, 9 or 10.

```
point = 0
point += 2*3/36.0 # ways to roll 4 or 10
point += 2*4/36.0 # ways to roll 5 or 9
point += 2*5/36.0 # ways to roll 6 or 8
print point
```

By the way, you can add the statement `print win + lose + point` to confirm that these odds all add to 1. This means that we have defined all possible outcomes for the come out roll in craps.

> **Tracing Execution**
>
> We can trace the execution of a program by simply following the changes of value of all the variables in the program. Here's an example for Example 6.3, "craps.py"
>
> - win: 0, ~~0.16~~, 0.22
> - list: 0, ~~0.027~~, ~~0.083~~, 0.111
> - point: ~~1~~, ~~0.77~~, 0.66

As with many things Python, there is some additional subtlety to this, but we'll cover those topics later. For example, *multiple-assignment* statement is something we'll look into in more deeply in Chapter 13, *Tuples*.

Input Functions

Python provides two simplistic built-in functions to accept input and set the value of variables. These are not really suitable for a complete application, but will do for our initial explorations.

Typically, interactive programs which run on a desktop use a complete graphic user interface (GUI), often written with the `Tkinter` module or the `pyGTK` module. Interactive

programs which run over the Internet use HTML forms. The primitive interactions we're showing with `input` and `raw_input` are only suitable for relatively simple programs.

Note that some IDE's buffer the program's output, making these functions appear to misbehave. For example, if you use Komodo, you'll need to use the "Run in a New Console" option. If you use BBEdit, you'll have to use the "Run in Terminal" option.

You can enhance these functions somewhat by including the statement `import readline`. This module silently and automatically enhances these input functions to give the user the ability to scroll backwards and reuse previous inputs.

You can also `import rlcompleter`. This module allows you to define sophisticated keyword auto-completion for these functions.

The raw_input Function

The first way to get interactive input is the `raw_input(`*`[prompt]`*`)` function. This function accepts a string parameter, which is the user's prompt, written to standard output. The next line available on standard input is returned as the value of the function.

The `raw_input(`*`[prompt]`*`)` function reads from a file often called `stdin`. When running from the command-line, this will be the keyboard, and what you type will be echoed in the command window or Terminal window. If you try, however, to run these examples from Textpad, you'll see that Textpad doesn't have any place for you to type any input. In BBEdit, you'll need to use the Run In Terminal item in the #! menu.

Here's an example script that uses `raw_input(`*`[prompt]`*`)`.

Example 6.4. rawdemo.py

```
#!/usr/bin/env python
# show how raw_input works
a= raw_input( "yes?" )
print a
```

When we run this script from the shell prompt, it looks like the following.

```
MacBook-3:Examples slott$ python rawdemo.py
yes?why not?
why not?
$
```

This program begins by evaluating the `raw_input` function. When `raw_input` is applied to the parameter of `"yes?"`, it writes the prompt on standard output, and waits for a line of input. We entered `why not?`. Once that line was complete, the input string is returned as the value of the function.

The `raw_input` function's value was assigned to the variable `a`. The second statement printed that variable.

If we want numeric input, we must convert the resulting string to a number.

Example 6.5. stock.py

```
#!/usr/bin/env python
# Compute the value of a block of stock
shares = int( raw_input("shares: ") )
price = float( raw_input("dollars: ") )
price += float( raw_input("eights: ") )/8.0
print "value", shares * price
```

We'll **chmod +x stock.py** this program; then we can run it as many times as we like to get results.

```
MacBook-3:Examples slott$ ./stock.py
shares: 150
dollars: 24
eights: 3
value 3656.25
$
```

The `raw_input` mechanism is very limited. If the string returned by `raw_input` is not suitable for use by `int`, an exception is raised and the program stops running. We'll cover exception handling in detail in Chapter 17, *Exceptions*.

Here's what it looks like.

```
MacBook-3:Examples slott$ ./stock.py
shares: a bunch


Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "stock.py", line 3, in ?
    shares = int( raw_input("shares: ") )
ValueError: invalid literal for int(): a bunch


MacBook-3:Examples slott$
```

The input Function

In addition to the `raw_input` function, which returns the exact string of characters, there is the `input(`*`[prompt]`*`)` function. This also accepts an optional parameter, which is the prompt. It writes the prompt, then reads a line of input. It then applies the Python `eval(`*`string`*`)` function to evaluate the input. Typically, we expect to decode a string of digits as an integer or floating point number. Actually, it will evaluate any legal Python expression! Of course, illegal expressions will raise an exception and the program will stop running.

The value of the `input (`*`p`*`)` function is `eval( raw_input(`*`p`*`) )`. Here's an example:

Example 6.6. inputdemo.py

```
#!/usr/bin/env python
shares = input('shares: ')
print shares
```

When we **chmod +x inputdemo.py** and run this program, we see the following.

```
$ ./inputdemo.py
shares: 2+1/2.0
2.5
$
```

In this case, the `input` function evaluated the expression `2+1/2.0`.

The `input` function is an unreliable tool. For yourself, the author of the program, problems with `input` are easily solved. For someone else, the vagaries of what is legal and how Python responds to things that are not legal can be frustrating. In Chapter 17, *Exceptions*, we'll add some sophistication that will solve some of these problems.

Multiple Assignment Statement

The basic assignment statement does more than assign the result of a single expression to a single variable. The assignment satement also copes nicely with assigning multiple variables at one time. The left and right side must have the same number of elements. For example, the following script has several examples of multiple assignment.

Example 6.7. line.py

```
#!/usr/bin/env python
# Compute line between two points.
x1,y1 = 2,3 # point one
x2,y2 = 6,8 # point two
m,b = float(y1-y2)/(x1-x2), y1-float(y1-y2)/(x1-x2)*x1

print "y=",m,"*x+",b
```

When we run this program, we get the following output

```
MacBook-3:Examples slott$ ./line.py
y = 1.25 *x+ 0.5
$
```

We set variables `x1`, `y1`, `x2` and `y2`. Then we computed `m` and `b` from those four variables. Then we printed the `m` and `b`.

The basic rule is that Python evaluates the entire right-hand side of the **=** statement. Then it matches values with destinations on the left-hand side. If the lists are different lengths, an exception is raised and the program stops.

Because of the complete evaluation of the right-hand side, the following construct works nicely to swap to variables. This is often quite a bit more complicated in other languages.

```
a,b = 1,4
b,a = a,b
print a,b
```

We'll return to this in Chapter 13, *Tuples*, where we'll see additional uses for this feature.

The del Statement

An **assignment** statement creates or locates a variable and then assigns a new object to the variable. This change in state is how our program advances from beginning to termination. Python also provides a mechanism for removing variables, the **del** statement.

The **del** statement looks like this:

`del` ⟨*`object`*,...⟩

Each *`object`* is any kind of Python object. Usually these are variables, but they can be functions, modules, classes.

The **del** statement works by *unbinding* the name, removing it from the set of names known to the Python interpreter. If this variable was the last remaining reference to an object, the object will be removed from memory. If, on the other hand, other variables still refer to this object, the object won't be deleted.

C++ Comparison

Programmers familiar with C++ will be pleased to note that memory management is silent and automatic, making programs much more reliable with much less effort. This removal of objects is called *garbage collection*, something that can be rather difficult to manage in larger applications. When garbage collection is done incorrectly, it can lead to *dangling references*: a variable that refers to an object that was deleted prematurely. Poorly designed garbage collection can also lead to *memory leaks*, where unreferenced objects are not properly removed from memory. Because of the automated garbage collection in Python, it suffers from none of these memory management problems.

The **del** statement is typically used only in rare, specialized cases. Ordinary namespace management and garbage collection are generally sufficient for most purposes.

Interactive Mode Revisited

When we first looked at interactive Python in the section called "Command-Line Interaction" we noted that Python executes assignment statements silently, but prints the results of an expression statement. Consider the following example.

```
>>> pi=355/113.0
>>> area=pi*2.2**2
>>> area
15.205309734513278
```

The first two inputs are complete statements, so there is no response. The third input is just an expression, so there is a response.

It isn't obvious, but the value assigned to `pi` isn't correct. Because we didn't see anything displayed, we didn't get any feedback from our computation of `pi`.

Python, however, has a handy way to help us. When we type a simple expression in interactive Python, it secretly assigns the result to a temporary variable named _. This isn't a part of scripting, but is a handy feature of an interactive session.

This comes in handy when exploring something rather complex. Consider this interactive session. We evaluate a couple of expressions, each of which is implicitly assigned to _. We can then save the value of _ in a second variable with an easier-to-remember name, like `pi` or `area`.

```
>>> 335/113.0
2.9646017699115044
>>> 355/113.0
3.1415929203539825
>>> pi=_
>>> pi*2.2**2
15.205309734513278
>>> area=_
```

Note that we created a floating point object (2.964...), and Python secretly assigned this object to _. Then, we computed a new floating point object (3.141...), which Python assigned to _. What happened to the first float, 2.964...? Python garbage-collected this object, removing it from memory.

The second float that we created (3.141) was assigned to _. We then assigned it to `pi`, also, giving us two references to the object. When we computed another floating-point value (15.205...), this was assigned to _. Does this mean our second float, 3.141... was garbage collected? No, it wasn't garbage collected; it was still referenced by the variable `pi`.

Variables, Assignment and Input Function Exercises

Variables and Assignment

1. **Extend Previous Exercises.** Rework the exercises in the section called "Numeric Types and Expressions".

   Each of the previous exercises can be rewritten to use variables instead of expressions using only constants. For example, if you want to tackle the Fahrenheit to Celsius problem, you might write something like this:

   ```
   #!/usr/bib/env python
   # Convert 8 C to F
   C=8
   F=32+C*float(9/5)
   print "celsius",C,"fahrenheit",F
   ```

   You'll want to rewrite these exercises using variables to get ready to add input functions.

2. **State Change.** Is it true that all programs simply establish a state?

   It can argued that a controller for a device (like a toaster or a cruise control) simply *maintains* a steady state. The notion of state change as a program moves toward completion doesn't apply because the software is always on. Is this the case, or does the software controlling a device have internal state changes?

   For example, consider a toaster with a thermostat, a "browness" sensor and a single heating element. What are the inputs? What are the outputs? Are there internal states while the toaster is making toast?

Input Functions

Refer back to the exercises in the section called "Numeric Types and Expressions" for formulas and other details. Each of these can be rewritten to use variables and an input conversion. For example, if you want to tackle the Fahrenheit to Celsius problem, you might write something like this:

```
C = input('Celsius: ')
F = 32+C*float(9/5)
print "celsius",C,"fahrenheit",F
```

1. **Stock Value.** Input the number of shares, dollar price and number of 8th's. From these three inputs, compute the total dollar value of the block of stock.

2. **Convert from °C to °F.** Write a short program that will input °C and output °F. A second program will input °F and output °C.

3. **Periodic Payment.** Input the principal, annual percentage rate and number of payments. Compute the monthly payment. Be sure to divide rate by 12 and multiple payments by 12.

4. **Surface Air Consumption Rate.** Write a short program will input the starting pressure, final pressure, time and maximum depth. Compute and print the SACR.

   A second program will input a SACR, starting pressure, final pressure and depth. It will print the time at that depth, and the time at 10 feet more depth.

5. **Wind Chill.** Input a temperature and a wind speed. Output the wind chill.

6. **Force from a Sail.** Input the height of the sail and the length. The surface area is 1/2×h×l. For a wind speed of 25 MPH, compute the force on the sail. Small boat

sails are 25-35 feet high and 6-10 feet long.

Variables and Assignment Style Notes

Spaces are used sparingly in Python. It is common to put spaces around the assignment operator. The recommended style is

```
c = (f-32)*5/9
```

Do not take great pains to line up assignment operators vertically. The following has too much space, and is hard to read, even though it is fussily aligned.

```
a                    = 12
b                    = a*math.log(a)
aVeryLongVariable    = 26
d                    = 13
```

This is considered poor form because Python takes a lot of its look from natural languages and mathematics. This kind of horizontal whitespace is hard to follow: it can get difficult to be sure which expression lines up with which variable. Python programs are meant to be reasonably compact, more like reading a short narrative paragraph or short mathematical formula than reading a page-sized UML diagram.

Variable names are often given as `mixedCase`; variable names typically begin with lower-case letters. The `lower_case_with_underscores` style is also used, but is less popular.

In addition, the following special forms using leading or trailing underscores are recognized:

- `single_trailing_underscore_`: used to avoid conflicts with Python keywords. For example: `print_ = 42`
- `__double_leading_and_trailing_underscore__`: used for special objects or attributes, e.g. `__init__`, `__dict__` or `__file__`. These names are reserved; do not use names like these in your programs unless you specifically mean a particular built-in feature of Python.

Chapter 7. Truth, Comparison and Conditional Processing

Truth, Comparison and the if Statement. for and while Statements; The break, continue and pass Statements; The assert Statement

Table of Contents

The elements of Python we've seen so far give us some powerful capabilities. We can write programs that implement a wide variety of requirements. State change is not always as simple as the examples we've seen in Chapter 6, *Variables, Assignment and Input*. When we run a script, all of the statements are executed unconditionally. Our programs can't handle alternatives or conditions. The other thing we can't do is write programs which do their processing "for all" pieces of data. For example, when we compute an average, we compute a sum *for all* of the values.

Python provides decision-making mechanisms similar to other programming languages. In the section called "Truth and Logic" we'll look at truth, logic and the logic operators. The exercises that follow examine some subtleties of Python's evaluation rules. In the section called "Comparisons" we'll look at the comparison operators. Then, the section called "Conditional Processing: the **if** Statement" describes the **if** statement. In the section called "The **assert** Statement" we'll introduce a handy diagnostic tool, the **assert** statement.

In the next chapter, Chapter 8, *Looping*, we'll look at looping constructs.

Truth and Logic

Many times the exact change in state that our program needs to make depends on a condition. A condition is a Boolean expression; an expression that is either `True` or `False`. Generally conditions are on comparisons among variables using the comparison operations.

We'll look at the essential definitions of truth, the logic operations and the comparison operations. This will allow us to build conditions.

Truth

Python represents truth and falsity in a variety of ways.

- **False.** Also 0, the special value `None`, zero-length strings `""`, zero-length lists `[]`, zero-length tuples `()`, empty mappings `{}` are all treated as `False`.
- **True.** Anything else that is not equivalent to `False`.

We try to avoid depending on relatively obscure rules for determining `True` vs. `False`. We prefer to use the two explicit keywords, `True` and `False`. Note that a previous version of Python didn't have the boolean literals, and some older open-source programs will define these values.

Python provides a factory function to collapse these various forms of truth into one of the two explicit boolean objects.

`bool` (*`object`*) → boolean

> Returns `True` when the argument *`object`* is one the values equivalent to truth, `False` otherwise.

Logic

Python provides three basic logic operators that work on this Boolean domain. Note that this Boolean domain, with just two values, `True` and `False`, and these three operators form a complete algebraic system, sometimes called Boolean algebra, after the mathemetician George Boole. The operators supported by Python are **not**, **and** and **or**. We can fully define these operators with rule statements or truth tables.

This truth table shows the evaluation of **not** ***x*** for both vales of ***x***.

```
print "x", "not x"
print True, not True
print False, not False
```

x	not x
True	False
False	True

This table shows the evaluation of ***x*** **and** ***y*** for all combination of `True` and `False`.

```
print "x", "y", "x and y"
print True, True, True and True
print True, False, True and False
print False, True, False and True
print False, False, False and False
```

x	y	x and y
True	True	True
True	False	False
False	True	False
False	False	False

An important feature of **and** is that it does not evaluate all of its parameters before it is applied. If the left-hand side is `False` or one of the equivalent values, the right-hand side is not evaluated, and the left-hand value is returned. We'll look at some examples of this later.

For now, you can try things like the following.

```
print False and 23
print 23 and False
```

This will show you that the first false value is what Python returns for **and**.

This table shows the evaluation of ***x*** **or** ***y*** for all combination of `True` and `False`.

x	y	x or y
True	True	True
True	False	True
False	True	True
False	False	False

Parallel with the **and** operator, **or** does not evaluate the right-hand parameter if the left-hand side is `True` or one of the equivalent values.

As a final note, **and** is a high priority operator (analogous to multiplication) and **or** is lower priority (analogous to addition). When evaluating expressions like `a or b and c`, the **and** operation is evaluated first, followed by the **or** operation.

Exercises

1. **Logic Short-Cuts.** We have several versions of false: False, 0, None, '', (), [] and {}. We'll cover all of the more advanced versions of false in Part II, "Data

Structures". For each of the following, work out the value according to the truth tables and the evaluation rules. Since each truth or false value is unique, we can see which part of the expression was evaluated.

- `False and None`
- `0 and None or () and []`
- `True and None or () and []`
- `0 or None and () or []`
- `True or None and () or []`
- `1 or None and 'a' or 'b'`

Comparisons

We'll look at the basic comparison operators. We'll also look at the partial evaluation rules of the logic operators to show how we can build more useful expressions. Finally, we'll look at floating-point equality tests, which are sometimes done incorrectly.

Basic Comparisons

We compare values with the comparison operators. These correspond to the mathematical functions of <, ≤, >, ≥, = and ≠. Conditional expressions are often built using the Python comparison operators: <, <=, >, >=, == and != for less than, less than or equal to, greater than, greater than or equal to, equal to and not equal to.

```
>>> p1=22./7.
>>> p2=355/113.
>>> p1
3.1428571428571428
>>> p2
3.1415929203539825
>>> p1 < p2
False
>>> p1 >= p2
True
```

When applying a comparison operator, we see a number of steps.

1. Evaluate both argument values.
2. Apply the comparison to create a boolean result.
    a. Convert both parameters to the same type. Numbers are converted to progressively longer types: plain integer to long integer to float to complex.
    b. Do the comparison.
    c. Return True or False.

We call out these three steps explicitly because there are some subtleties in comparison among unlike types of data; we'll come to this later when we cover sequences, mappings and classes in Part II, "Data Structures". Generally, it doesn't make sense to compare unlike types of data. After all, you can't ask "Which is larger, the Empire State Building or the color green?"

Comparisons can be combined in Python, unlike most other programming languages. We can ask: `0 <= a < 6` which has the usual mathematical meaning. We're not forced to use the longer form: `0 <= a and a < 6` .

```
>>> 3 < p1 < 3.2
True
>>> 3 < p1 and p1 < 3.2
True
```

This is useful when a is actually some complex expression that we'd rather not repeat.

Also note that the preceding example had a mixture of integers and floating-point numbers. The integers were coerced to floating-point in order to evaluate the expressions.

Partial Evaluation

We can combine the logic operators, comparisons and math. This allows us to use comparisons and logic to prevent common mathematical blunders like attempting to divide by zero, or attempting to take the square root of a negative number.

For example, let's start with this program that will figure the average of 95, 125 and 132.

```
sum = 95 + 125 + 132
count = 3
average = float(sum)/count
print average
```

Initially, we set the variables sum and count. Then we compute the average using sum and count.

Assume that the statement that computes the average (average=...), is part of a long and complex program. Sometimes that long program will try to compute the average of no numbers at all. This has the same effect as the following short example.

```
sum, count = 0, 0
average = float(sum)/count
print average
```

In the rare case that we have no numbers to average we don't want to crash when we foolishly attempt to divide by zero. We'd prefer to have some more graceful behavior.

Recall from the section called "Truth and Logic" that the **and** operator doesn't evaluate the right-hand side unless the left-hand side is True. Stated the other way, the **and** operator *only* evaluates the right side if the left side is True. We can guard the division like this:

```
average = count != 0 and sum/count
print average
```

This is an example that can simplify certain kinds of complex processing. If the count is non-zero, the left side is true and the right side must be checked. If the count is zero, the left side is False, the result of the complete **and** operation is False.

This is a consequence of the meaning of the word *and*. The expression *a and b* means that a is true as well as b is true. If a is false, the value of b doesn't really matter, since the whole expression is clearly false. A similar analysis holds for the word *or*. The expression *a or b* means that one of the two is true; it also means that neither of the two is false. If a is true, then the value of b doesn't change the truth of the whole expression.

The statement "It's cold and rainy" is completely false when it is warm; rain doesn't matter to falsifying the whole statement. Similarly, "I'm stopping for coffee or a newspaper" is true if I've stopped for coffee, irrespective of whether or not I stop for a newspaper.

Floating-Point Comparisons

Exact equality between floating-point numbers is a dangerous concept. After a lengthy computation, round-off errors in floating point numbers may have infinitesimally small differences. The answers are close enough to equal for all practical purposes, but every

single one of the 64 bits may not be identical.

The following technique is the appropriate way to do floating point comparisons.

```
abs(a-b)<0.0001
```

Rather than ask if the two floating point values are the same, we ask if they're close enough to be considered the same. For example, run the following tiny program.

Example 7.1. floatequal.py

```
#!/usr/bin/env python
# Are two floating point values really completely equal?
a,b = 1/3.0, .1/.3
print a,b,a==b
print abs(a-b)<0.00001
```

When we run this program, we get the following output

```
$ python floatequal.py
0.333333333333 0.333333333333 False
True
$
```

The two values appear the same when printed. Yet, on most platforms, the **==** test returns `False`. They are not precisely the same. This is a consequence of representing real numbers with only a finite amount of binary precision. Certain repeating decimals get truncated, and these truncation errors accumulate in our calculations.

There are ways to avoid this problem; one part of this avoidance is to do the algebra necessary to postpone doing division operations. Division introduces the largest number erroneous bits onto the trailing edge of our numbers. The other part of avoiding the problem is never to compare floating point numbers for exact equality.

Conditional Processing: the if Statement

Many times the program's exact change in state depends on a condition. Conditional processing is done by setting statements apart in suites with conditions attached to the suites. The Python syntax for this is an **if** statement.

The if Statement

The basic form of an **if** statement provides a condition and a suite of statements that are executed when the condition is true. It looks like this:

`if` *`expression`* : *`suite`*

The *`suite`* is an indented block of statements. Any statement is allowed in the block, including indented **if** statements. You can use either tabs or spaces for indentation. The usual style is four spaces.

This is our first *compound statement*. See Python Syntax Rules for some additional guidance on syntax for compound statements.

The **if** statement evaluates the condition *`expression`* first. When the result is `True`, the *`suite`* of statements is executed. Otherwise the suite is skipped.

For example, if two dice show a total of 7 or 11, the throw is a winner.

```
d1 and d2 are two dice

if d1+d2 == 7 or d1+d2 == 11:
```

```
    print "winner", d1+d2
```

Here we have a typically complex expression. The **or** operator evaluates the left side first. Python evaluates and applies the high-precendence arithmetic operator before the lower-precendence comparison operator. If the left side is true (`d1+d2` is 7), the **or** expression is true, and the suite is executed. If the left side is false, then the right side is evaluated. If it is true (`d1+d2` is 11), the **or** expression is true, and the suite is executed. Otherwise, the suite is skipped.

Python Syntax Rules

Python syntax is very simple. We've already seen how basic expressions and some simple statements are formatted. Here are some syntax rules and examples. Look at Syntax Formalities for an overview of the lexical rules.

Compount statements, including **if**, **while**, **for**, have an indented suite of statements. You have a number of choices for indentation; you can use tab characters or spaces. While there is a lot of flexibility, the most important thing is to be consistent.

We'll show an example with spaces, shown via ␣, and tabs shown with ⊢.

```
a=0
if␣a=0:
␣␣␣print␣"a␣is␣zero"
else:
␣␣␣print␣"a␣is␣not␣zero"

if␣a%2=0:
⊢print␣"a␣is␣even"
else:
⊢print␣"a␣is␣odd"
```

IDLE uses four spaces for indentation automatically. If you're using another editor, you can set it to use four spaces, also.

The elif Clause

Often there are several conditions that need to be handled. This is done by adding **elif** clauses. This is short for “else-if”. We can add an unlimited number of **elif** clauses. The **elif** clause has almost the same syntax as the **if** clause.

`elif` *`expression`* : *`suite`*

Here is a somewhat more complete rule for the come out roll in a game of craps:

```
result= None
if d1+d2 == 7 or d1+d2 == 11:
    result= "winner"
elif d1+d2 == 2 or d1+d2 == 3 or d1+d2 == 12:
    result= "loser"
print result
```

First, we checked the condition for winning; if the first condition is true, the first suite is executed and the entire **if** statement is complete. If the first condition is false, then the second condition is tested. If that condition is true, the second suite is executed, and the entire **if** statement is complete. If neither condition is true, the **if** statement has no effect.

The else Clause

Python also gives us the capability to put a “catch-all” suite at the end for all other conditions. This is done by adding an **else** clause. The else clause has the following

syntax.

```
else: suite
```

Here's the complete come-out roll rule, assuming two values `d1` and `d2`.

```
point= None
if d1+d2 == 7 or d1+d2 == 11:
    print "winner"
elif d1+d2 == 2 or d1+d2 == 3 or d1+d2 == 12:
    print "loser"
else:
    point= d1+d2
    print  "point is", point
```

Here, we use the **else:** suite to handle all of the other possible rolls. There are six different values (4, 5, 6, 8, 9, or 10), a tedious typing exercise if done using **or**. We summarize this with the **else:** clause.

While handy in one respect, this **else:** clause is also dangerous. By not explicitly stating the condition, it is possible to overlook simple logic errors.

Consider the following complete **if** statement that checks for a winner on a field bet. A field bet wins on 2, 3, 4, 9, 10, 11 or 12. The payout odds are different on 2 and 12.

```
outcome= 0
if d1+d2 == 2 or d1+d2 == 12:
    outcome= 2
    print "field pays 2:1"
elif d1+d2==4 or d1+d2==9 or d1+d2==10 or d1+d2==11:
    outcome= 1
    print "field pays even money"
else:
    outcome= -1
    print "field loses"
```

Here we test for 2 and 12 in the first clause; we test for 4, 9, 10 and 11 in the second. It's not obvious that a roll of 3 is missing from the even money pay out. This fragment incorrectly treats 3, 5, 6, 7 and 8 alike in the **else:**. While the **else:** clause is used commonly as a catch-all, a more proper use for **else:** is to raise an exception because a condition was not matched by any of the **if** or **elif** clauses.

The pass Statement

The **pass** statement does nothing. Sometimes we need a placeholder to fill the syntactic requirements of a compound statement. We use the **pass** statement to fill in the required suite of statements.

The syntax is trivial.

```
pass
```

Here's an example of using the pass statement.

```
if n%2 == 0:
    pass # Ignore even values
else:
    count += 1 # Count the odd values
```

Yes, technically, we can invert the logic in the if-clause. However, sometimes it is more clear to provide the explicit "do nothing" than to determine the inverse of the condition in the **if** statement.

As programs grow and evolve, having a **pass** statement can be a handy reminder of places where a program can be expanded.

The assert Statement

An assertion is a condition that we're claiming should be true at this point in the program. Typically, it summarizes the state of the program's variables. Assertions can help explain the relationships among variables, review what has happened so far in the program, and show that **if** statements and **for** or **while** loops have the desired effect.

When a program is correct, all of the assertions are true no matter what inputs are provided. When a program has an error, at least one assertion winds up false for some combination of inputs.

Python directly supports assertions through an **assert** statement. There are two forms:

`assert` *`condition`*

`assert` ⟨*`condition`*⟩ ⟨ , *`expression`*⟩

If the *`condition`* is `False`, the program is in error; this statement raises an `AssertionError` exception. If the *`condition`* is `True`, the program is correct, this statement does nothing more.

If the second form of the statement is used, and an *`expression`* is given, an exception is raised using the value of the expression. We'll cover exceptions in detail in Chapter 17, *Exceptions*. If the expression is a string, it becomes an the value associated with the `AssertionError` exception.

> **Note**
>
> There is an even more advanced feature of the **assert** statement. If the expression evaluates to a class, that class is used instead of `AssertionError`. This is not widely used, and depends on elements of the language we haven't covered yet.

Here's a typical example:

```
max= 0
if a < b: max= b
if b < a: max= a
assert (max == a or max == b) and max >= a and max >= b
```

If the assertion condition is true, the program continues. If the assertion condition is false, the program raises an `AssertionError` exception and stops, showing the line where the problem was found.

Run this program with `a` equal to `b` and not equal to zero; it will raise the `AssertionError` exception. Clearly, the **if** statements don't set `max` to the largest of `a` and `b` when `a = b`. There is a problem in the **if** statements, and the presence of the problem is revealed by the assertion.

The if-else Operator

There are situations where an expression involves a simple condition and a full-sized **if** statement is distracting syntatic overkill. Python has a handy logic operator that evalutes a condition, then returns either of two values depending on that condition.

Most arithmetic and logic operators have either one or two values. An operation that applies to a single value is called *unary*. For example `-a` and `abs(b)` are examples of

unary operations: unary negation and unary absolute value. An operation that applies to two values is called binary. For example, `a*b` shows the binary multiplication operator.

The if-else operator trinary. It involves a conditional expression and two alternative expressions. Consequently, it doesn't use a single special character, but uses two keywords: `if` and `else`.

The basic form of the operator is *`expression` `if` `condition` `else` `expression`*. Python evaluates the condition first. If the condition is `True`, then the left-hand expression is evaluated, and that's the value of the operation. If the condition is `False`, then the else expression is evaluated, and that's the value of the operation.

Here are a couple of examples.

- ```
  average = sum/count if count != 0 else None
  ```
- ```
  oddSum = oddSum + ( n if n % 2 == 1 else 0 )
  ```

The intent is to have an English-like reading of the statement. "The average is the sum divided by the count if the count is non-zero; otherwise the average is None".

The wordy alterative to the first example is the following.

```
if count != 0:
    average= sum/count
else:
    average= None
```

This seems like three extra lines of code to prevent an error in the rare situation of there being no values to average.

Similarly, the wordy version of the second example is the following:

```
if n % 2 == 0:
    pass
else:
    oddSum = oddSum + n
```

For this second example, the original statement registered our intent very clearly: we were summing the odd values. The long-winded if-statement tends to obscure our goal by making it just one branch of the if-statement.

Condition Exercises

1. **Develop an "or-guard".** In the example above we showed the "and-guard" pattern:

   ```
   average = count != 0 and float(sum)/count
   ```

   Develop a similar technique using **or**.

   Compare this with the **if-else** operator.

2. **Come Out Win.** Assume `d1` and `d2` have the numbers on two dice. Assume this is the come out roll in Craps. Write the expression for winning (7 or 11). Write the expression for losing (2, 3 or 12). Write the expression for a point (4, 5, 6, 8, 9 or 10).

3. **Field Win.** Assume `d1` and `d2` have the numbers on 2 dice. The field pays on 2, 3, 4, 9, 10, 11 or 12. Actually there are two conditions: 2 and 12 pay at one set of odds (2:1) and the other 5 numbers pay at even money. Write two two conditions under which the field pays.

4. **Hardways.** Assume `d1` and `d2` have the numbers on 2 dice. A hardways proposition is 4, 6, 8, or 10 with both dice having the same value. It's the hard way to get the number. A hard 4, for instance is `d1+d2 == 4 and d1 == d2`. An easy 4 is `d1+d2 == 4 and d1 != d2`.

   You win a hardways bet if you get the number the hard way. You lose if you get the number the easy way or you get a seven. Write the winning and losing condition for one of the four hard ways bets.

5. **Sort Three Numbers.** This is an exercise in constructing if-statements. Using only simple variables and if statements, you should be able to get this to work; a loop is not needed.

   Given 3 numbers (X, Y, Z), assign variables `x`, `y`, `z` so that $x \le y \le z$ and `x`, `y`, and `z` are from X, Y, and Z. Use only a series of if-statements and assignment statements.

   Hint. You must define the conditions under which you choose between $x \leftarrow X$, $x \leftarrow Y$ or $x \leftarrow Z$. You will do a similar analysis for assigning values to `y` and `z`. Note that your analysis for setting `y` will depend on the value set for `x`; similarly, your analysis for setting `z` will depend on values set for `x` and `y`.

6. **Come Out Roll.** Accept `d1` and `d2` as input. First, check to see that they are in the proper range for dice. If not, print a message.

   Otherwise, determine the outcome if this is the come out roll. If the sum is 7 or 11, print winner. If the sum is 2, 3 or 12, print loser. Otherwise print the point.

7. **Field Roll.** Accept `d1` and `d2` as input. First, check to see that they are in the proper range for dice. If not, print a message.

   Otherwise, check for any field bet pay out. A roll of 2 or 12 pays 2:1, print "pays 2"; 3, 4, 9, 10 and 11 pays 1:1, print "pays even"; everything else loses, print "loses"

8. **Harways Roll.** Accept `d1` and `d2` as input. First, check to see that they are in the proper range for dice. If not, print a message.

   Otherwise, check for a hard ways bet pay out. Hard 4 and 10 pays 7:1; Hard 6 and 8 pay 9:1, easy 4, 6, 8 or 10, or any 7 loses. Everything else, the bet still stands.

9. **Partial Evaluation.** This partial evaluation of the **and** and **or** operators appears to violate the evaluate-apply principle espoused in The Evaluate-Apply Cycle. Instead of evaluating all parameters, these operators seem to evaluate only the left-hand parameter before they are applied. Is this special case a problem? Can these operators be removed from the language, and replaced with the simple **if**-statement? What are the consequences of removing the short-circuit logic operators?

Condition Style Notes

Now that we have introduced compound statements, you may need to make an adjustment to your editor. Set your editor to use spaces instead of tabs. Most Python is typed using four spaces instead of the ASCII tab character (^I). Most editors can be set so that when you hit the **Tab** key on your keyboard, the editor inserts four spaces. IDLE is set up this way by default. A good editor will follow the indents so that once you indent, the next line is automatically indented.

We'll show the spaces explicitly as ␣'s in the following fragment.

```
if␣a␣>=␣b:
␣␣␣␣m␣=␣a
```

```
if␣b␣>=␣a:
␣␣␣␣m␣=␣b
```

This is has typical spacing for a piece of Python programming.

Note that the colon (:) immediately follows the condition. This is the usual format, and is consistent with the way natural languages (like English) are formatted.

These **if** statements can be collapsed to one-liners, in which case they would look like this:

```
if␣a␣>=␣b:␣m␣=␣a
if␣b␣>=␣a:␣m␣=␣b
```

It helps to limit your lines to 80 positions or less. You may need to break long statements with a \at the end of a line. Also, parenthesized expressions can be continued onto the next line without a \. Some programmers will put in extra ()'s just to make line breaks neat.

While spaces are used sparingly, they are always used to set off comparison operators and boolean operators. Other mathmatical operators may or may not be set off with spaces. This makes the comparisons stand out in an **if** statement or **while** statement.

```
if␣b**2-4*a*c␣<␣0:
␣␣␣␣print␣"no␣root"
```

This shows the space around the comparison, but not the other arithmetic operators.

Chapter 8. Looping

The for and while Statements; The break, continue and pass Statements; The assert Statement

Table of Contents

The elements of Python we've seen so far give us some powerful capabilities. We can write programs that implement a wide variety of requirements. State change is not always as simple as the examples we've seen in Chapter 6, *Variables, Assignment and Input*. When we run a script, all of the statements are executed unconditionally. Our programs can't handle alternatives or conditions. The other thing we can't do is write programs which do their processing “for all” pieces of data. For example, when we compute an average, we compute a sum *for all* of the values.

Python provides iteration (sometimes called looping) similar to other programming languages. In the section called “Iterative Processing: For All and There Exists” we'll describe the **for** and **while** statements. This is followed by some of the most interesting and challenging short exercises in this book. We'll add some iteration control in the section called “More Iteration Control: **break** and **continue**”, describing the **break** and **continue** statements. We'll conclude this chapter with a digression on the correct ways to develop iterative and conditional statements in the section called “A Digression”.

Iterative Processing: For All and There Exists

There are two common qualifiers used for logical conditions. These are sometimes called the universal and existential qualifiers. We can call the "for all" and "there exists". We can also call them the "all" and "any" qualifiers.

A program may involve a state that is best described as a “for all” state, where a number of repetitions of some task are required. For example, if we were to write a program to simulate 100 rolls of two dice, the terminating condition for our program would be that we had done the simulation *for all* 100 rolls.

Similary, we may have a condition that looks for existence of a single example. We might want to know if a file contains a line with "ERROR" in it. In this case, we want to write a program with a terminating condition would be that *there exists* an error line in the log file.

It turns out that All and Any are logical inverses. We can always rework a "for any" condition to be a "for all" condition. A program that determines if there exists an error line is the same as a program that determines that all lines are not error lines.

Any time we have a “for all” or "for any" condition, we have an iteration: we will be iterating through the set of values, evaluating the condition. We have a choice of two Python statements for expressing this iteration. One is the **for** statement and the other is the **while** statement.

Iterative Processing: The for Statement

The simplest **for** statement looks like this:

`for` *`variable`* in *`sequence`* : *`suite`*

The *`suite`* is an indented block of statements. Any statement is allowed in the block, including indented **for** statements.

The *`variable`* is a variable name. The *`suite`* will be executed iteratively with the variable set to each of the values in the given *`sequence`*. Typically, the *`suite`* will use the *`variable`*, expecting it to have a distinct value on each pass.

There are a number of ways of creating the necessary *`sequence`* of values. The most common is to use the `range` function to generate a suitable list. We can also create the list manually, using a sequence display; we'll show some examples here. We'll return to the details of sequences in Chapter 11, *Sequences: Strings, Tuples and Lists*.

The `range` function has 3 forms:

- `range` (*`x`*) generates *`x`* distinct values, from 0 to *`x`*-1, incrementing by 1.
- `range` (*`x`*, *`y`*) generates *`y`*-*`x`* distinct values from *`x`* to *`y`*-1, incrementing by 1.
- `range` (*`x`*, *`y`*, *`i`*) generates values from *`x`* to *`y`*-1, incrementing by *`i`*: [*`x`*, *`x`*+*`i`*, *`x`*+2*`i`*, ... *`x`*+*k`i`* < *`y`*]

A sequence display looks like this: **[** ***`expression`*****, ...]**. It's a list of expressions, usually simply numbers, separated by commas. The square brackets are essential for marking a sequence.

This first example creates a sequence of 6 values from 0 to just before 6. The **for** statement iterates through the sequence, assigning each value to the local variable `i`. The **print** statement has an expression that adds one to `i` and prints the resulting value. Note that the suite of statements in the body of the **for** statement is simply a **print** statement, so we can combine it all on one line.

```
for i in range(6): print i+1
```

The second example creates a sequence of 6 values from 1 to just before 7. The **for** statement iterates through the sequence, assigning each value to the local variable `j`. The **print** statement prints the value.

```
for j in range(1,7):
    print j
```

This example creates a sequence of 36/2=18 values from 1 to just before 36 stepping by 2. This will be a list of odd values from 1 to 35. The **for** statement iterates through the sequence, assigning each value to the local variable `o`. The **print** statement prints all 18 values.

```
for o in range(1,36,2):
    print o
```

This example uses an explicit sequence of values. These are all of the red numbers on a standard roulette wheel. It then iterates through the sequence, assigning each value to the local variable `r`. The **print** statement prints all 18 values followed by the word "red".

```
for r in [1,3,5,7,9,12,14,16,18,19,21,23,25,27,30,32,34,36]:
    print r, "red"
```

Here's a more complex example, showing nested **for** statements. This enumerates all the 36 outcomes of rolling two dice. The outer **for** statement creates a sequence of 6 values, and iterates through the sequence, assigning each value to the local variable `d1`. For each value of `d1`, the inner loop creates a sequence of 6 values, and iterates through that sequence, assigning each value to `d2`. The **print** statement will be executed 36 times.

```
for d1 in range(6):
    for d2 in range(6):
        print d1+1,d2+1,'=',d1+d2+2
```

Here's the example alluded to earlier, which does 100 simulations of rolling two dice. The **for** statement creates the sequence of 100 values, assigns each value to the local variable `i`; note that the suite of statements never actually uses the value of `i`. The value of `i` marks the state changes until the loop is complete, but isn't used for anything else.

```
import random
for i in range(100):
    d1= random.randrange(6)+1
    d2= random.randrange(6)+1
    print d1+d2
```

There are a number of more advanced forms of the **for** statement, which we'll cover in the section on sequences in Chapter 11, *Sequences: Strings, Tuples and Lists*.

Iterative Processing: The while Statement

The **while** statement looks like this:

`while` *`expression`* : *`suite`*

The *`suite`* is an indented block of statements. Any statement is allowed in the block, including indented **while** statements.

As long as the *`expression`* is true, the *`suite`* is executed. This allows us to construct a suite that steps through all of the necessary tasks to reach a terminating condition. It is important to note that the suite of statements must include a change to at least one of the variables in the **while** *`expression`*. When it is possible to execute the suite of statements without changing any of the variables in the **while** *`expression`*, the loop will not terminate.

Let's look at some examples.

```
t, s = 1, 1
while t != 9:
    t, s = t + 2, s + t
```

The loop is initialized with `t` and `s` each set to 1. We specify that the loop continues "while `t` ≠ 9". In the body of the loop, we increment `t` by 2, so that it will be an odd value; we increment `s` by `t`, summing a sequence of odd values.

When this loop is done, `t` is 9, and `s` is the sum of odd numbers less than 9: 1+3+5+7. Also note that the **while** condition depends on `t`, so changing `t` is absolutely critical in the body of the loop.

Here's a more complex example. This sums 100 dice rolls to compute an average.

```
s, r = 0, 0
while r != 100:
    d1,d2=random.randrange(6)+1,random.randrange(6)+1
    s,r = s + d1+d2, r + 1
print s/r
```

We initialize the loop with `s` and `r` both set to zero. The **while** statement specifies that during the loop `r` will not be 100; when the loop is done, `r` will be 100. The body of the loop sets `d1` and `d2` to random numbers; it increments `s` by the sum of those dice, and it increments `r` by 1. When the loop is over, `s` will be the sum of 100 rolls of two dice. When we print, `s/r` we print the average rolled on two dice. The loop condition depends on `r`, so each trip through the loop must update `r`.

More Iteration Control: break and continue

Python offers several statements for more subtle loop control. The point of these statements is to permit two common simplifications of a loop. In each case, these statements can be replaced with **if** statements; however, those **if** statement versions might be considered rather complex for expressing some fairly common situations.

The **break** statement terminates a loop prematurely. This allows a complex condition to terminate a loop. A **break** statement is always found within **if** statements within the body of a **for** or **while** loop. A **break** statement is typically used when the terminating condition is too complex to write as an expression in the **while** clause of a loop. A **break** statement is also used when a **for** loop must be abandoned before the end of the sequence has been reached.

The **continue** statement skips the rest of the loop's suite. Like a **break** statement, a **continue** statements is always found within an **if** statement within a **for** or **while** loop. The **continue** statement is used instead of deeply nested **else** clauses.

Examples break and continue. Here's an example that has a complex **break** condition. We are going to see if we get six odd numbers in a row, or spin the roulette wheel 100 times.

We'll look at this in some depth because it pulls a number of features together in one program. This program shows both **break** and **continue** constructs. Most programs can actually be simplified by eliminating the **break** and **continue** statements. In this case, we didn't simplify, just to show how the statements are used.

Note that we have a two part terminating condition: 100 spins *or* six odd numbers in a row. The hundred spins is relatively easy to define using the `range` function. The six odd numbers in a row requires testing and counting and then, possibly, ending the loop. The overall ending condition for the loop, then, is the number of spins is 100 or the count of odd numbers in a row is six.

Example 8.1. sixodd.py

```
import random
oddCount= 0
for s in range(100):
    lastSpin= s
    n= random.randrange(38)
    # Zero
    if n == 0 or n == 37: # treat 37 as 00
        oddCount = 0
        continue
    # Odd
    if n%2 == 1:
        oddCount += 1
        if oddCount == 6: break
        continue
    # Even
    assert n%2 == 0 and 0 < n <= 36
    oddCount = 0
print oddCount, lastSpin
```

❶ We import the `random` module, so that we can generate a random sequence of spins of a roulette wheel. We initialize `oddCount`, our count of odd numbers seen in a row. It starts at zero, because we haven't seen any add numbers yet.

❷ The **for** statement will assign 100 different values to `s`, such that $0 \le s < 100$. This will control our experiment to do 100 spins of the wheel.

❸ Note that we save the current value of `s` in a variable called `lastSpin`, setting up part of our post condition for this loop. We need to know how many spins were done, since one of the exit conditions is that we did 100 spins and never saw six odd values in a row. This "never saw six in a row" exit condition is handled by the **for** statement itself.

❹ We'll treat 37 as if it were 00, which is like zero. In Roulette, these two numbers are neither even nor odd. The `oddCount` is set to zero, and the loop is continued. This **continue** statement resumes loop with the next value of `s`. It restarts processing at the top of the **for** statement suite.

❺ We check the value of `oddCount` to see if it has reached six. If it has, one of the exit conditions is satisfied, and we can break out of the loop entirely. We use the **break** statement will stop executing statements in the suite of the **for** statement. If `oddCount` is not six, we don't break out of the loop, we use the **continue** statement to restart the **for** statement suite from the top with a new value for `s`.

❻ We threw in an **assert** (see the next section, the section called “The **assert** Statement”, for more information) that the spin, `n`, is even and not 0 or 37. This is kind of a safety net. If either of the preceding **if** statements were incorrect, or a **continue** statement was omitted, this statement would uncover that fact. We could do this with another **if** statement, but we wanted to introduce the **assert** statement.

At the end of the loop, `lastSpin` is the number of spins and `oddCount` is the most recent count of odd numbers in a row. Either `oddCount` is six or `lastSpin` is 99. When `lastSpin` is 99, that means that spins 0 through 99 were examined; there are 100 different numbers between 0 and 99.

Iteration Exercises

1. **Greatest Common Divisor.** The greatest common divisor is the largest number which will evenly divide two other numbers. Examples: GCD(5, 10) = 5, the largest number that evenly divides 5 and 10. GCD(21, 28) = 7, the largest number that divides 21 and 28.

   GCD's are used to reduce fractions. Once you have the GCD of the numerator and denominator, they can both be divided by the GCD to reduce the fraction to simplest form. 21/28 reduces to 3/4.

Procedure 8.1. Greatest Common Divisor of two integers, *p* and *q*

1. **Loop.** Loop until $p = q$.
    a. **Swap.** If $p < q$ then swap p and q.
    b. **Subtract.** If $p > q$ then subtract q from p.
2. **Result.** Print p

2. **Extracting the Square Root.** This is a procedure for approximating the square root. It works by dividing the interval which contains the square root in half. Initially, we know the square root of the number is somewhere between 0 and the number. We locate a value in the middle of this interval and determine of the square root is more or less than this midpoint. We continually divide the intervals in half until we arrive at an interval which is small enough and contains the square root. If the interval is only 0.001 in width, then we have the square root accurate to 0.001

    Procedure 8.2. Square Root of a number, *n*

    1. **Two Initial Guesses**

        $g1 \leftarrow 0$

        $g2 \leftarrow n$

        At this point, $g1 \times g1 - n \leq 0 \leq g2 \times g2 - n$

    2. **Loop.** Loop until $\text{abs}(g1 \times g1 - n) \div n < 0.001$
        a. **Midpoint.** $mid \leftarrow (g1+g2) \div 2$
        b. **Midpoint Squared vs. Number.** $sqrt\_mid \leftarrow mid \times mid - number$
        c. **Which Interval?**

            if $sqrt\_mid \leq 0$ then $g1 \leftarrow mid$

            if $sqrt\_mid \geq 0$ then $g2 \leftarrow mid$

            if *sqrt_mid* is zero, *mid* is the exact answer!
    3. **Result.** Print $g1$

3. **Sort Four Numbers.** This is a challenging exercise in if-statement construction. For some additional insight, see [Dijkstra76], page 61.

    Given 4 numbers (W, X, Y, Z)

    Assign variables `w`, `x`, `y`, `z` so that `w` ≤ `x` ≤ `y` ≤ `z` and `w`, `x`, `y`, `z` are from W, X, Y, and Z. Do not use an array. One way to guarantee the second part of the above is to initialize `w`, `x`, `y`, `z` to W, X, Y, Z, and then use swapping to rearrange the variables.

    Hint: There are only a limited combination of out-of-order conditions among four variables. You can design a sequence of if statements, each of which fixes one of the out-of-order conditions. This sequence of if statements can be put into a loop. Once all of the out-of-order conditions are fixed, the numbers are in order, the loop can end.

4. **Highest Power of 2.** This can be used to determine how many bits are required to

represent a number. We want the highest power of 2 which is less than or equal to our target number. For example $64 \le 100 < 128$. The highest power of $2 \le 100$ is 2^6.

Given a number n, find a number p such that $2^p \le n < 2^p+1$.

This can be done with only addition and multiplication by 2. Multiplication by 2, but the way, can be done with the << shift operator. Do not use the `pow` function, or even the ** operator, as these are too slow for our purposes.

Consider using a variable c, which you keep equal to 2^p. An initialization might be $p \leftarrow 1$, $c \leftarrow 2$. When you increment p by 1, you also double c.

Develop your own loop. This is actually quite challenging, even though the resulting program is tiny. For additional insight, see [Gries81], page 147.

5. **How Much Effort to Produce Software?** The following equations are the basic COCOMO estimating model, described in [Boehm81]. The input, K, is the number of 1000's of lines of source, total source lines÷1000.

   Equation 8.1. Development Effort

   $$E = 2.4K^{1.05}$$

   Equation 8.2. Development Cost

   $$C = E \times R \times 152$$

   Equation 8.3. Project Duration

   $$D = 2.5E^{0.38}$$

   Equation 8.4. Staffing

   $$S = \frac{E}{D}$$

   Where K is the number of 1000's of lines of source, E is effort in staff-months, D is duration in calendar months, S is the average staff size, R is the billing rate, C is the cost in dollars (assuming R $ per hour, and 152 working hours per staff-month).

   Evaluate these functions for projects which range in size from 8,000 lines (K=8) to 64,000 lines (K=64) in steps of 8. Produce a table with lines of source, Effort, Duration, Cost and Staff size.

6. **Wind Chill Table.** Wind chill is used by meteorologists to describe the effect of cold and wind combined. Given the wind speed in miles per hour, V, and the temperature in °F, T, the Wind Chill, w, is given by the formula below. See Wind Chill in the section called "Numeric Types and Expressions" for more information.

   $$35.74 + 0.6215 \times T - 35.75(V^{0.16}) + 0.4275 \times T \times (V^{0.16})$$

   Wind speeds are for 0 to 40 mph, above 40, the difference in wind speed doesn't have much practical impact on how cold you feel.

   Evaluate this for all values of V (wind speed) from 0 to 40 mph in steps of 5, and all values of T (temperature) from -10 to 40 in steps of 5.

7. **Celsius to Fahrenheit Conversion Tables.** We'll make two slightly different

conversion tables.

For values of Celsius from -20 to +30 in steps of 5, produce the equivalent Fahrenheit temperature. The following formula converts C (Celsius) to F (Fahrenheit).

$$F = 32 + \frac{(212 - 32)}{100} \times C$$

For values of Fahrenheit from -10 to 100 in steps of 5, produce the equivalent Celsius temperatures. The following formula converts F (Fahrenheit) to C (Celsius).

$$C = (F - 32) \times \frac{100}{212 - 32}$$

8. **Dive Planning Table.** Given a surface air consumption rate, *c*, and the starting, *s*, and final, *f*, pressure in the air tank, a diver can determine maximum depths and times for a dive. For more information, see Surface Air Consumption Rate in the section called "Numeric Types and Expressions".

   Accept *c*, *s* and *f* from input, then evaluate the following for *d* from 30 to 120 in steps of 10. Print a table of *t* and *d*.

   For each diver, *c* is pretty constant, and can be anywhere from 10 to 20, use 15 for this example. Also, *s* and *f* depend on the tank used, typical values are *s*=2500 and *f*=500.

   $$t = \frac{33(s - f)}{c(d + 33)}$$

9. **Computing π.** Each of the following series compute increasingly accurate values of π (3.1415926...)

   - $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots$
   - $\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \ldots$
   - $\pi = \sum_{0 \le k < \infty} \left(\frac{1}{16}\right)^k \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$
   - $\pi = 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \ldots$

10. **Computing e.** A logarithm is a power of some base. When we use logarithms, we can effectively multiply numbers using addition, and raise to powers using multiplication. Two Python built-in functions are related to this: `math.log` and `math.exp`. Both of these compute what are called natural logarithms, that is, logarithms where the base is *e*. This constant, *e*, is available in the math module, and it has the following formal definition:

    Equation 8.5. Definition of e

    $$e = \sum_{0 \le k < \infty} \frac{1}{k!}$$

    For more information on the Σ operator, see the section called "Digression on The Sigma Operator".

Equation 8.6. Definition of Factorial, n!

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

4!=4×3×2×1. By definition, 0! = 1.

If we add up the values 1/0! + 1/1! + 1/2! + 1/3! + 1/4! + ... we get the value of e. Clearly, when we get to about 1/10!, the fraction is so small it doesn't contribute much to the total.

We can do this with two loops, an outer loop to sum up the 1/*k*!'s, and an inner loop to compute the *k*!'s.

However, if we have a temporary value of *k*!, each time through the loop we can multiply the temporary by *k*, and then add 1/*temp* to the sum.

You can check your work against `math.e` ~ 2.71828... or `math.exp (1)`.

11. **Hailstone Numbers.** For additional information, see [Banks02].

    Start with a small number, n, $1 \le n \le 30$.

    There are two transformation rules that we will use:

    - If n is odd, multiple by 3 and add 1 to create a new value for n.
    - If n is even, divide by 2 to create a new value for n.

    Perform a loop with these two transformation rules until you get to $n = 1$. You'll note that when $n = 1$, you get a repeating sequence of 1, 4, 2, 1, 4, 2, ...

    You can test for oddness using the % (remainder) operation. If `n % 2 == 1`, the number is odd, otherwise it is even.

    The two interesting facts are the "path length", the number of steps until you get to 1, and the maximum value found during the process.

    Tabulate the path lengths and maximum values for numbers 1..30. You'll need an outer loop that ranges from 1 to 30. You'll need an inner loop to perform the two steps for computing a new n until $n == 1$; this inner loop will also count the number of steps and accumulate the maximum value seen during the process.

    Check: for 27, the path length is 111, and the maximum value is 9232.

Condition and Loops Style Notes

As additional syntax, the **for** and **while** statements permits an **else** clause. This is a suite of statements that are executed when the loop terminates normally. This suite is skipped if the loop is terminated by a **break** statement. The **else** clause on a loop might be used for some post-loop cleanup. This is so unlike other programming languages, that it is hard to justify using it.

An **else** clause always raises a small problem when it is used. It's never perfectly clear what conditions lead to execution of an **else** clause. The condition that applies has to be worked out from context. For instance, in **if** statements, one explicitly states the exact condition for all of the **if** and **elif** clauses. The logical inverse of this condition is assumed as the **else** condition. It is, unfortunately, left to the person reading the program to work out what this condition actually is.

Similarly, the **else** clause of a **while** statement is the basic loop termination condition, with all of the conditions on any **break** statements removed. The following kind of analysis can be used to work out the condition under which the else clause is executed.

```
while not BB:
    if C1: break
    if C2: break
else:
    assert BB and not C1 and not C2
assert BB or C1 or C2
```

Because this analysis can be difficult, it is best to avoid the use of **else** clauses in loop constructs.

A Digression

For those new to programming, here's a short digression, adapted from chapter 8 of Edsger Dijkstra's book, *A Discipline of Programming* [Dijkstra76].

Let's say we need to set a variable, `m`, to the larger of two input values, `a` and `b`. We start with a state we could call "`m` undefined". Then we want to execute a statement after which we are in a state of "(`m==a` or `m==b`) and `m≥a` and `m≥b`".

Clearly, we need to choose correctly between two different **assignment** statements. We need to do either **m=a** or **m=b**. How do we make this choice? With a little logic, we can derive the condition by taking each of these statement's effects out of the desired end-state.

For the statement **m=a** to be the right statement to use, we show the effect of the statement by replacing `m` with the value `a`, and examining the end state: (`a==a` or `a==b`) and `a≥a` and `a≥b`. Removing the parts that are obviously true, we're left with `a≥b`. Therefore, the assignment **m=a** is only useful when `a≥b`.

For the statement **m=b** to be the right statement to establish the necessary condition, we do a similar replacement of `b` for `m` and examine the end state: (`b==a` or `b==b`) and `b≥a` and `b≥b`. Again, we remove the parts that are obviously true and we're left with `b≥a`. Therefore, the assignment **m=b** is only useful when `b≥a`.

Each assignment statement can be "guarded" by an appropriate condition.

```
if a>=b: m=a
elif b>=a: m=b
```

Is the correct statement to set `m` to the larger of `a` or `b`.

Note that the hard part is establishing the post condition. Once we have that stated correctly, it's relatively easy to figure the basic kind of statement that might make some or all of the post condition true. Then we do a little algebra to fill in any guards or loop conditions to make sure that only the correct statement is executed.

Successful Loop Design. There are several considerations when using the **while** statement. This list is taken from David Gries', *The Science of Programming* [Gries81].

1. The body condition must be initialized properly.
2. At the end of the suite, the body condition is just as true as it was after initialization. This is called the *invariant*, because it is always true during the loop.
3. When this body condition is true and the while condition is false, the loop will have completed properly.
4. When the while condition is true, there are more iterations left to do. If we wanted to, we could define a mathematical function based on the current state that computes how many iterations are left to do; this function must have a value greater than zero when the while condition is true.
5. Each time through the loop we change the state of our variables so that we are getting closer to making the while condition false; we reduce the number of iterations left to do.

While these conditions seem overly complex for something so simple as a loop, many programming problems arise from missing one of them.

Gries recommends putting comments around a loop showing the conditions before and after the loop. Since Python provides the **assert** statement; this formalizes these comments into actual tests to be sure the program is correct.

Designing a Loop. Let's put a particular loop under the microscope. This is a small example, but shows all of the steps to loop construction. We want to find the least power of 2 greater than or equal to some number greater than 1, call it `x`. This power of 2 will tell us how many bits are required to represent `x`, for example.

We can state this mathematically as looking for some number, `n`, such that $2^{n-1} < x \leq 2^n$. This says that if `x` is a power of 2, for example 64, we'd find 2^6. If `x` is another number, for example 66, we'd find $2^6 < 66 \leq 2^7$, or $64 < 66 \leq 128$.

We can start to sketch our loop already.

```
assert x > 1
... initialize ...
... some loop ...
assert 2**(n-1) < x <= 2**n
```

We work out the initialization to make sure that the invariant condition of the loop is initially true. Since `x` must be greater than or equal to 1, we can set `n` to 1. $2^{1-1}=2^0=1 <$ `x`. This will set things up to satisfy rule 1 and 2.

```
assert x > 1
n= 1
... some loop ...
assert 2**(n-1) < x <= 2**n
```

In loops, there must be a condition on the body that is invariant, and a terminating condition that changes. The terminating condition is written in the **while** clause. In this case, it is invariant (always true) that `2**(n-1) < x`. That means that the other part of our final condition is the part that changes.

```
assert x > 1
n= 1
while not ( x <= 2**n ):
    n= n + 1
    assert 2**(n-1) < x
assert 2**(n-1) < x <= 2**n
```

The next to last step is to show that when the **while** condition is true, there are more than zero trips through the loop possible. We know that `x` is finite and some power of 2 will satisfy this condition. There's some `n` such that `n-1 < log2 (x) < = n` that limits the trips through the loop.

The final step is to show that each cycle through the loop reduces the trip count. We can argue that increasing `n` gets us closer to the upper bound of `log2 (x)`.

We should add this information on successful termination as comments in our loop.

Chapter 9. Functions

Table of Contents

The heart of programming is the *evaluate-apply* cycle, where function arguments are evaluated and then the function is applied to those argument values. We'll review this in the section called "Semantics".

In the section called "Function Definition: The **def** and **return** Statements" we introduce the syntax for defining and using a function. We introduce some of the alternate argument forms available for handling optional parameters in the section called "More Features". Then, in the section called "Providing Argument Values by Keyword" we show how Python can use keyword parameters as well as positional parameters.

In the section called "Object Method Functions" we will describe how to use *method functions* as a prelude to Part II, "Data Structures"; real details on method functions are deferred until Chapter 21, *Classes*.

Further sophistication in how Python handles parameters has to be deferred to the section called "Advanced Parameter Handling For Functions", as it depends on a knowledge of dictionaries, introduced in Chapter 15, *Mappings and Dictionaries*.

We'll also defer examination of the **yield** statement until Chapter 18, *Generators and the **yield** Statement*. The **yield** statement creates a special kind of function, one that is most useful when processing complex data structures, something we'll look at in Part II, "Data Structures".

Semantics

A function, in a mathematical sense, is often described as a mapping from domain values to range values. Given a domain value, the function returns the matching range value. If we think of the square root function, it maps a positive number, n, to another number, s, such that $s^2 = n$. If we think of multplication as a function, it maps a pair of values, a and b, to a new value, c, such that c is the product of a and b.

In Python, this narrow definition is somewhat relaxed. Python lets us create functions which do not need a domain value, but create new objects. It also allows us to have functions that don't return values, but instead have some other effect, like reading user input, or creating a directory, or removing a file.

What We Provide. In Python, we create a new function by providing three pieces of information: the name of the function, a list of zero or more variables, called *parameters*, with the domain of input values, and a suite of statements that creates the output values. This definition is saved for later use. We'll show this first in the section called "Function Definition: The **def** and **return** Statements".

Typically, we create function definitions in script files because we don't want to type them more than once. Almost universally, we **import** a file with our function definitions so we can use them.

We use a function in an expression by following the function's name with `()`'s. The

Python interpreter evaluates the argument values in the ()'s, then applies the function. We'll show this second in the section called "Function Use".

Applying a function means that the interpreter first evaluates all of the argument values, then assigns the argument values to the function parameter variables, and finally evaluates the suite of statements that are the function's body. In this body, any **return** statements define the resulting range value for the function. For more information on this evaluate-apply cycle, see The Evaluate-Apply Cycle.

Namespaces and Privacy. Note that the parameter variables used in the function definition, as well as any variables in a function are private to that function's suite of statements. This is a consequence of the way Python puts all variables in a namespace. When a function is being evaluated, Python creates a temporary namespace. This namespace is deleted when the function's processing is complete. The namespace associated with application of a function is different from the global namespace, and different from all other function-body namespaces.

While you can change the standard namespace policy (see the section called "The **global** Statement",) it generally will do you more harm than good. A function's interface is easiest to understand if it is only the parameters and return values and nothing more. If all other variables are local, they can be safely ignored.

Terminology: argument and parameter. We have to make a firm distinction between an argument *value*, an object that is created or updated during execution, and the defined parameter *variable* of a function. The argument is the object used in particular application of a function; it may be referenced by other variables or objects. The parameter is a variable name that is part of the function, and is a local variable within the function body.

The Evaluate-Apply Cycle

The evaluate-apply cycle shows how any programming language computes the value of an expression. Consider the following expression:

```
math.sqrt( abs( b*b-4*a*c ) )
```

What does Python do?

For the purposes of analysis, we can restructure this from the various mathematical notation styles to a single, uniform notation. We call this *prefix* notation, because all of the operations prefix their operands. While useful for analysis, this is cumbersome to write for real programs.

```
math.sqrt( abs( sub( mul(b,b), mul(mul(4,a),c) ) ) )
```

We've replaced `x*y` with `mul(x,y)`, and replaced `x-y` with `sub(x,y)`. This allows us to more clearly see how evaluate-apply works. Each part of the expression is now written as a function with one or two arguments. First the arguments are evaluated, then the function is applied to those arguments.

In order for Python to evaluate this `math.sqrt(...)` expression, it evaluates the argument, `abs(...)`, and then applies `math.sqrt` to it. This leads Python to a nested evaluate-apply process for the `abs(...)` expression. We'll show the whole process, with indentation to make it clearer.

1. value of math.sqrt(...) = get value of abs(...)
    a. value of abs(...) = get value of sub(...)
        i. value of sub(...) = get values of mul(b,b); get value of

mul(mul(...),c)

A. value of mul(b,b) = get value of *b*; then apply mul()

B. value of mul(mul(4,a),c) = get value of mul(4,a); get value of *c*

I. value of mul(4,a) = get value of *a*; then apply mul()

C. apply mul() to mul(4,a) and value of *c*

ii. apply sub() to mul(b,b) and mul(mul(...),c)

b. apply abs() to the value of of sub(...)

2. apply sqrt() to the value of abs(...)

The apply part of the evalute-apply cycle is sometimes termed a function *call*. The idea is that the main procedure "calls" the body of a function; the function does its work and returns to the main procedure. This is also called a function *invocation*.

Function Definition: The def and return Statements

We create a function with a **def** statement. This provides the name, parameters and the suite of statements that creates the function's result.

`def` *`name`* (⟨*`parameter`*...⟩): *`suite`*

The *`name`* is the name by which the function is known. The *`parameters`* is a list of variable names; these names are the local variables to which actual argument values will be assigned when the function is applied. The *`suite`* (which must be indented) is a block of statements that computes the value for the function.

The first line of a function is expected to be a document string (generally a triple-quoted string) that provides basic documentation for the function. This is traditionally divided in two sections, a summary section of exactly one line and the detail section. We'll return to this style guide in the section called "Functions Style Notes".

The **return** statement specifies the result value of the function. This value will become the result of applying the function to the argument values. This value is sometimes called the effect of the function.

`return` ⟨*`expression`*⟩

Let's look at a complete, although silly, example.

```
def odd( spin ):
    """Return true if this spin is odd."""
    if spin % 2 == 1:
        return True
    return False
```

We name this function `odd`, and define it to accept a single parameter, named *`spin`*. We provide a docstring with a short description of the function. In the body of the function, we test to see if the remainder of *`spin`*/2 is 1; if so, we return `True`. Otherwise, we return `False`.

Function Use

When Python evaluates `odd(n)`, first it evaluates `n`; second, it assigns this argument value to the local parameter of `odd` (named `spin`); third, it applies `odd`, the suite of statements is executed, ending with return 1 or return 0; fourth, this result returned to the calling statement so that it can finish it's execution.

We would use this function like this.

```
s = random.randrange(37)
# 0 <= s <= 36, single-0 roulette
if s == 0:
    print "zero"
elif odd(s):
    print s, "odd"
else:
    print s, "even"
```

We evaluate a function named random.randrange to create a random number, `s`. The **if** clause handles the case where `s` is zero. The first **elif** clause evaluates our `odd` function. To do this evaluation, Python must set `spin` to the value of `s` and execute the suite of statements that are the body of `odd`. The suite of statements will return either `True` or `False`. Since the **if** and **elif** clauses handle zero and odd cases, all that is left is for `s` to be even.

Function Varieties

"Ordinary" Functions. Functions which follow the classic mathematical definitions map input argument values to a resulting value. These are, perhaps, the most common kind of functions, and don't have a special name. We'll look at prodedure and factory functions which are atypical in that they may lack a result or lack input values.

Procedure Functions. One common kind of function is one that doesn't return a result, but instead carries out some procedure. This function would omit any **return** statement. Or, if **return** statements are used to exit from the function, they would have no value to return. Carrying out an action is sometimes termed a side-effect of the function. The primary effect is always the value returned.

Here's an example of a function that doesn't return a value, but carries out a procedure.

```
def report( spin ):
    """Report the current spin."""
    if spin == 0:
        print "zero"
        return
    if odd(spin):
        print spin, "odd"
        return
    print spin, "even"
```

This function, `report`, has a parameter named *`spin`*, but doesn't return a value. Here, the **return** statements exit the function but don't return values.

This kind of function would be used as if it was a new Python language statement, for example:

```
for i in range(10):
    report( random.randrange(37) )
```

Here we execute the `report` function as if it was a new kind of statement. We don't evaluate it as part of an expression.

There's actually no real subtlety to this distinction. Any expression can be used as a Python statement. A function call is an expression, and an expression is a statement.

This greatly simplifies Python syntax. The docstring for a function will explain what kind of value the function returns, or if the function doesn't return anything useful.

> **Subroutines and Functions**
>
> Programmers who started their careers in languages like FORTRAN or Pascal may recall a firm distinction between functions and subroutines. Python (as well as Java and C++) have eliminated this distinction.

The simple **return** statement, by the way, returns the special value `None`. This default value means that you can define your function like `report`, above, use it in an expression, and everything works nicely because the function does return a value.

```
for i in range(10):
    t= report( random.randrange(37) )
print t
```

You'll see that `t` is `None`.

Factory Functions. Another common form is a function that doesn't take a parameter. This function is a factory which generates a value, often by accessing some obscured object. Sometimes this object is encapsulated in a module or class. In the following example, we'll access the random number generator encapsulated in the `random` module.

```
def spinWheel():
    """Return a string result from a roulette wheel spin."""
    t= random.randrange(38)
    if t == 37:
        return "00"
    return str(t)
```

This function's evaluate-apply cycle is simplified to just the apply phase. To make 0 (zero) distinct from 00 (double zero), it returns a string instead of a number.

Generators. A generator function contains the **yield** statement. These functions look like conventional functions, but they have a different purpose in Python. We will examine this in detail in Chapter 18, *Generators and the **yield** Statement*.

These functions have a persistent internal processing state; ordinary functions can't keep data around from any previous calls without resorting to global variables. Further, these functions interact with the **for** statement. Finally, these functions don't make a lot of sense until we've worked with sequences in Chapter 11, *Sequences: Strings, Tuples and Lists*.

Some Examples

Here's a big example of using the `odd`, `spinWheel` and `report` functions.

Example 9.1. functions.py

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

def odd( spin ):
    """odd(number) -> boolean."""
    return spin%2 == 1

def report( spin ):
    """Reports the current spin on standard output.  Spin is a String"""
    if int(spin) == 0:
        print "zero"
```

```
        return
    if odd(int(spin)):
        print spin, "odd"
        return
    print spin, "even"

def spinWheel():
    """Returns a string result from a roulette wheel spin."""
    t= random.randrange(38)
    if t == 37:
        return "00"
    return str(t)

for i in range(12):
    n= spinWheel()
    report( n )
```

❶ We've defined a function named `odd`. This function evaluates a simple expression; it returns `True` if the value of it's parameter, *`spin`*, is odd.

❷ The function called `report` uses the `odd` function to print a line that describes the value of the parameter, *`spin`*. Note that the parameter is private to the function, so this use of the variable name *`spin`* is technically distinct from the use in the `odd` function. However, since the `report` function provides the value of *`spin`* to the `odd` function, their two variables often happen to have the same value.

❸ The `spinWheel` function creates a random number and returns the value as a string.

❹ The "main" part of this program is the for loop at the bottom, which calls `spinWheel`, and then `report`. The `spinWheel` function uses `random.randrange`; the `report` function uses the `odd` function. This generates and reports on a dozen spins of the wheel.

For most of our exercises, this free-floating main script is acceptable. When we cover modules, in Part IV, "Components, Modules and Packages", we'll need to change our approach slightly to something like the following.

```
def main():
    for i in range(12):
        n= spinWheel()
        report( n )

main()
```

This makes the main operation of the script clear by packaging it as a function. Then the only free-floating statement in the script is the call to `main`.

Hacking Mode

On one hand we have interactive use of the Python interpreter: we type something and the interpreter responds immediately. We can do simple things, but when our statements get too long, this interaction can become a niusance. We introduced this first, in the section called "Command-Line Interaction".

On the other hand, we have scripted use of the interpreter: we present a file as a finished program to execute. While handy for getting useful results, this isn't the easiest way to get a program to work in the first place. We described this in the section called "Script Mode".

In between the interactive mode and scripted mode, we have a third operating mode, that we might call *hacking mode*. The idea is to write most of our script and then exercise portions of our script interactively. In this mode, we'll develop script files, but we'll exercise them in an interactive environment. This is handy for developing and debugging function definitions.

The basic procedure is as follows.

1. In our favorite editor, write a script with our function definitions. We often leave this editor window open. IDLE, for example, leaves this window open for us to look at.

2. Open a Python shell. IDLE, for example, always does this for us.

3. In the Python Shell, **import** the script file. In IDLE, this is effectively what happens when we run the module with **F5**.

4. In the Python Shell, test the function interactively. If it works, we're done.

5. If the functions in our module didn't work, we return to our editor window, make any changes and save the file.

6. In the Python Shell, clear out the old definition by restarting the shell. In IDLE, we can force this with **F6**. This happens automatically when we run the module using **F5**.

7. Go back to step 3, to import and test our definitions.

The interactive test results are often saved and put into the docstring for the file with our function definitions. We usually copy the contents of the Python Shell window and paste it into our module's or function's docstring. This record of the testing can be validated using the `doctest` module.

Example. Here's the sample function we're developing. If you look carefully, you might see a serious problem. If you don't see the problem, don't worry, we'll find it by doing some debugging.

In IDLE, we created the following file.

Example 9.2. function1.py Initial Version

```
# Some Function Or Other
def odd( number ):
    """odd(number) -> boolean

    Returns True if the given number is odd.
    """
    return number % 2 == "1"
```

We have two windows open: function1.py and Python Shell.

Here's our interactive testing session. In our function1.py window, we hit F5 to run the module. Note the line that shows that the Python interpreter was restarted; forgetting any previous definitions. Then we exercised our function with two examples.

```
Python 2.5.1 (r251:54863, Oct  5 2007, 21:08:09)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.1.4
```

```
>>> ================================ RESTART ================================
>>>
>>> odd(2)
False
>>> odd(3)
False
```

Clearly, it doesn't work, since 3 is odd. When we look at the original function, we can see the problem.

The expression `number % 2 == "1"` should be `number % 2 == 1`. We need to fix `function1.py`. Once the file is fixed, we need to remove the old stuff from Python, re-import our function and rerun our test. IDLE does this for us when we hit **F5** to rerun the module. It shows this with the prominent restart message.

If you are not using IDLE, you will need to restart Python to clear out the old definitions. Python optimizes import operations; if it's seen the module once, it doesn't import it a second time. To remove this memory of which modules have been imported, you will need to restart Python.

More Features

Python provides a mechanism for optional parameters. This allows us to create a single function which has several alternative forms. In other languages, like C++ or Java, these are called overloaded functions; they are actually separate function definitions with the same name but different parameter forms. In Python, we can write a single function that accepts several parameter forms.

Python has three mechanisms for dealing with optional parameters and a variable number of parameters. We'll cover the basics of optional parameters in this section. The other mechanisms for dealing with variable numbers of parameters will be deferred until the section called “Advanced Parameter Handling For Functions” because these mechanisms use some more advanced data structures.

Python functions can return multiple values. We'll look at this, also.

Default Values for Parameters

One way the we can handle optional parameters is by providing a default value for a parameter. If no argument is supplied for the parameter, the default value is used.

```
def report( spin, count=1 ):
    print spin, count, "times in  a row"
```

This silly function can be used in two ways:

```
report( n )
report( n, 2 )
```

The first form provides a default argument of 1 for the *`count`* parameter. The second form has an explicit argument value of 2 for the *`count`* parameter.

If a parameter has no default value, it is not optional. If a parameter has a default value, it is optional. In order to disambiguate the assignment of arguments to parameters, Python uses a simple rule: all required parameters must be first, all optional parameters must come after the required parameters.

The `int` function does this. We can say `int("23")` to do decimal conversion and `int("23",16)` to do hexadecimal conversion. Clearly, the second argument to `int` has a default value of 10.

When we look at the Python `range` function, we see a more sophisticated version of

this.

`range (x)` is the same as `range (0, x, 1)`

`range (x, y)` is the same as `range (x, y, 1)`

It appears from these examples that the *first* parameter is optional. The authors of Python use a pretty slick trick for this that you can use also. The `range` function behaves as though the following function is defined.

```
def range(x, y=None, z=None):
    if y==None:
        start, stop, step = 0, x, 1
    elif z==None:
        start, stop, step = x, y, 1
    else:
        start, stop, step = x, y, z
    Real work is done with start, stop and step
```

By providing a default of `None`, the function can determine whether a value was supplied or not supplied. This allows for complex default handling within the body of the function.

Conclusion. Python must find a value for all parameters. The basic rule is that the values of parameters are set in the order in which they are declared. Any missing parameters will have their default values assigned. These are called positional parameters, since the position is the rule used for assigning argument values when the function is applied.

If a mandatory parameter (a parameter without a default value) is missing, this is a basic `TypeError`. For example.

Example 9.3. badcall.py

```
#!/usr/bin/env python
def hack(a,b):
    print a+b

hack(3)
```

When we run this example, we see the following.

```
$ python badcall.py

Traceback (most recent call last):
  File "badcall.py", line 5, in ?
    hack(3)
TypeError: hack() takes exactly 2 arguments (1 given)

$
```

Providing Argument Values by Keyword

In addition to supplying argument values by position, Python also permits argument values to be specified by name. Using explicit keywords can make programs much easier to read.

First, we'll define a function with a simple parameter list:

```
import random

def averageDice( samples=100 ):
    """Return the average of a number of throws of 2 dice."""
```

```
    s = 0
    for i in range(samples):
        d1,d2 = random.randrange(6)+1,random.randrange(6)+1
        s += d1+d2
    return float(s)/float(samples)
```

Next, we'll show three different kinds of arguments: keyword, positional, and default.

```
test1 = averageDice( samples=200 )
test2 = averageDice( 300 )
test3 = averageDice()
```

When the `averageDice` function is evaluated to set `test1`, the keyword form is used. The second call of the `averageDice` function uses the positional form. The final example relies on a default for the parameter.

Conclusion. This gives us a number of variations including positional parameters and keyword parameters, both with and without defaults. Positional parameters work well when there are few parameters and their meaning is obvious. Keyword parameters work best when there are a lot of parameters, especially when there are optional parameters.

Good use of keyword parameters mandates good selection of keywords. Single-letter parameter names or obscure abbreviations do not make keyword parameters helpfully informative.

Here are the rules we've seen so far:

1. Supply values for all parameters given by name, irrespective of position.
2. Supply values for all remaining parameters by position; in the event of duplicates, raise a `TypeError`.
3. Supply defaults for any parameters that have defaults defined; if any parameters still lack values, raise a `TypeError`.

There are still more options available for handling variable numbers of parameters. It's possible for additional positional parameters to be collected into a sequence object. Further, additional keyword parameters can be collected into a dictionary object. We'll get to them when we cover dictionaries in the section called “Advanced Parameter Handling For Functions”.

Returning Multiple Values

One common desire among programmers is a feature that allows a function to return multiple values. Python has some built-in functions that have this property. For example, `divmod` returns the divisor and remainder in division. We could imagine a function, `rollDice` that would return two values showing the faces of two dice.

In Python, it is done by returning a `tuple`. We'll wait for Chapter 13, *Tuples* for complete information on `tuples`. The following is a quick example of how multiple assignment works with functions that return multiple values.

Example 9.4. rolldice.py

```
import random

def rollDice():
    return ( 1 + random.randrange(6), 1 + random.randrange(6) )

d1,d2=rollDice()
print d1,d2
```

This shows a function that creates a two-valued `tuple`. You'll recall from the section called "Multiple Assignment Statement" that Python is perfectly happy with multiple expressions on the right side of **=**, and multiple destination variables on the left side. This is one reason why multiple assignment is so handy.

Function Exercises

1. **Fast Exponentiation.** This is the fastest way to raise a number to an integer power. It requires the fewest multiplies, and does not use logarithms.

   Procedure 9.1. Fast Exponentiation of integers, raises *n* to the *p* power

   1. **Base Case.** If $p = 0$, return 1.0.
   2. **Odd.** If p is odd, return $n \times$ `fastExp (n, p-1 )`.
   3. **Even.** If p is even, compute $t \leftarrow$ `fastExp (n, p÷2 )`; return $t \times t$.

2. **Greatest Common Divisor.** The greatest common divisor is the largest number which will evenly divide two other numbers. You use this when you reduce fractions. See Greatest Common Divisor in for an alternate example of this exercise's algorithm. This version can be slightly faster than the loop we looked at earlier.

   Procedure 9.2. Greatest Common Divisor of two integers, *p* and *q*

   1. **Base Case.** If $p = q$, return p.
   2. **Swap.** If $p < q$, return `GCD (q, p)`.
   3. **Subtract.** If $p > q$, return `GCD (p, p-q)`.

3. **Factorial Function.** Factorial of a number n is the number of possible arrangements of 0 through n things. It is computed as the product of the numbers 1 through n. That is, $1 \times 2 \times 3 \times ... \times n$. We touched on this in Computing e in the section called "Iteration Exercises". This function defintion can simplify the program we wrote for that exercise.

   Procedure 9.3. Factorial of an integer, *n*

   1. **Base Case.** If $n = 0$, return 1.
   2. **Multiply.** If $n > 0$, return $n \times$ `factorial (n-1)`.

4. **Fibonacci Series.** Fibonacci numbers have a number of interesting mathematical properties. The ratio of adjacent Fibonacci numbers approximates the golden ratio ($(1+\sqrt{5})/2$, about 1.618), used widely in art and architecture.

   Procedure 9.4. The *n*th Fibonacci Number, F_n

   1. **F_0 Case.** If $n = 0$, return 0.
   2. **F_1 Case.** If $n = 1$, return 1.
   3. **$F_n = F_{n-1}$ and F_{n-2}.** If $n > 2$, return `fibonacci (n-1)` + `fibonacci (n-2)`.

5. **Ackermann's Function.** An especially complex algorithm that computes some really big results. This is a function which is specifically designed to be complex. It cannot easily be rewritten as a simple loop. Further, it produces extremely large results because it describes extremely large exponents.

Procedure 9.5. Ackermann's Function of two numbers, *m* and *n*

1. **Base Case.** If $m = 0$, return $n+1$.
2. **N Zero Case.** If $m \neq 0$ and $n = 0$, return `ackerman (m-1, 1)`.
3. **N Non-Zero Case.** If $m \neq 0$ and $n \neq 0$, return `ackerman (m-1, ackerman (m, n-1))`.

6. **Compute the Maximum Value of Some Function.** Given some integer-valued function `f(x)`, we want to know what value of `x` has the largest value for `f(x)` in some interval of values. For additional insight, see [Dijkstra76].

   Imagine we have an integer function of an integer, call it `f(x)`. Here are some examples of this kind of function.

   - `def f1(x): return x`
   - `def f2(x): return -5/3*x-3`
   - `def f3(x): return -5*x*x+2*x-3`

   The question we want to answer is what value of `x` in some fixed interval returns the largest value for the given function? In the case of the first example, `def f1(x): return x`, the largest value of `f1(x)` in the interval $0 \leq x < 10$ occurs when x is 9. What about `f3(x)` in the range ($-10 \leq x \leq 10$)

   Procedure 9.6. Max of a Function in the interval *low* to *high*

   1. **Initialize**

      $x \leftarrow low$

      $max \leftarrow x$

      $maxFx \leftarrow$ `f(max)`

   2. **Loop.** While $low \leq x < high$.

      a. **New Max?** If `f(x)` $> maxFx$, then

         $max \leftarrow x$

         $maxFx \leftarrow$ `f(max)`

      b. **Next X.** Increment x by 1.

   3. **Return.** Return x as the value at which `f(x)` had the largest value.

7. **Integration Approximation.** This is a simple rectangular rule for finding the area under a curve which is continuous on some closed interval.

   We will define some function which we will integrate, call it `f (x)`. Here are some examples.

   - `def f1(x): return x*x`
   - `def f2(x): return 0.5 * x * x`
   - `def f3(x): return exp( x )`
   - `def f4(x): return 5 * sin( x )`

   When we specify $y=f(x)$, we are specifying two dimensions. The y is given by the

function's values. The x dimension is given by some interval. If you draw the function's curve, you put two limits on the x axis, this is one set of boundaries. The space between the curve and the y axis is the other boundary.

The x axis limits are a and b. We subdivide this interval into s rectangles, the width of each is $h=(b-a)\div s$. We take the function's value at the corner as the average height of the curve over that interval. If the interval is small enough, this is reasonably accurate.

Procedure 9.7. Integration Function in the interval *a* to *b* in *s* steps

1. **Initialize**

   $x \leftarrow a$

   $h \leftarrow (b-a)\div s$

   $sum \leftarrow 0.0$

2. **Loop.** While $a \le x < b$.

   a. **Update Sum.** Increment *sum* by $f(x) \times h$.

   b. **Next X.** Increment x by h.

3. **Return.** Return *sum* as the area under the curve $f(x)$ for $a \le x < b$.

8. **Field Bet Results.** In the dice game of Craps, the Field bet in craps is a winner when any of the numbers 2, 3, 4, 9, 10, 11 or 12 are rolled. On 2 and 12 it pays 2:1, on the other number, it pays 1:1.

   Define a function `win (dice, num, pays )`. If the value of `dice` equals `num`, then the value of `pays` is returned, otherwise 0 is returned. Make the default for `pays` a 1, so we don't have to repeat this value over and over again.

   Define a function `field (dice)`. This will call `win` 7 times: once with each of the values for which the field pays. If the value of dice is a 7, it returns -1 because the bet is a loss. Otherwise it returns 0 because the bet is unresolved. It would start with

   ```
   def field( dice ):
       win( dice, 2, pays=2 )
       win( dice, 3, pays=1 )
       ...
   ```

   Create a function `roll` that creates two dice values from 1 to 6 and returns their sum. The sum of two dice will be a value from 2 to 12.

   Create a main program that calls `roll` to get a dice value, then calls `field` with the value that is rolled to get the payout amount. Compute the average of several hundred experiments.

9. **`range` Function Keywords.** Does the range function permit keywords for supplying argument values? What are the keywords?

10. **Optional First Argument.** Optional parameters must come last, yet range fakes this out by appearing to have an optional parameter that comes first. The most common situation is `range(5)`, and having to type `range(0,5)` seems rather silly. In this case, convenience trumps strict adherence to the rules. Is this a good thing? Is strict adherence to the rules more or less important than convenience?

Object Method Functions

We've seen how we can create functions and use those functions in programs and other functions. Python has a related technique called *methods* or *method functions*. The functions we've used so far are globally available. A method function, on the other hand, belongs to an object. The object's class defines what methods and what properties the object has.

We'll cover method functions in detail, starting in Chapter 21, *Classes*. For now, however, some of the Python data types we're going to introduce in Part II, "Data Structures" will use method functions. Rather than cover too many details, we'll focus on general principles of how you use method functions in this section.

The syntax for calling a method function looks like this:

`someObject.aMethod (`*`argument list`*`)`

A single `.` separates the owning object (`someObject`) from the method name (`aMethod`). Some methods have no arguments, others have complex arguments.

We glanced at a simple example with complex numbers. The complex conjugate function is actually a method function of the complex number object. The example is in the section called "Complex Numbers".

In the next chapter, we'll look at various kinds of sequences. Python defines some generic method functions that apply to any of the various classes of sequences. The `string` and `list` classes, both of which are special kinds of sequences, have several methods functions that are unique to strings or lists.

For example:

```
>>> "Hi Mom".lower()
'hi mom'
```

Here, we call the `lower` method function, which belongs to the string object `"Hi Mom"`.

When we describe modules in Part IV, "Components, Modules and Packages", we'll cover module functions. These are functions that are imported with the module. The `array` module, for example, has an `array` function that creates `array` objects. An `array` object has several method functions. Additionally, an `array` object is a kind of sequence, so it has all of the methods common to sequences, also.

`file` objects have an interesting life cycle, also. A `file` object is created with a built-in function, `file`. A file object has numerous method functions, many of which have side-effects of reading from and writing to external files or devices. We'll cover files in Chapter 19, *Files*, listing most of the methods unique to file objects.

Functions Style Notes

The suite within a compound statement is typically indented four spaces. It is often best to set your text editor with tab stops every four spaces. This will usually yield the right kind of layout.

We'll show the spaces explicitly as ␣'s in the following fragment.

```
def␣max(a,␣b):
␣␣␣␣if␣a␣>=␣b:
␣␣␣␣␣␣␣␣m␣=␣a
␣␣␣␣if␣b␣>=␣a:
␣␣␣␣␣␣␣␣m␣=␣b
␣␣␣␣return m
```

This is has typical spacing for a piece of Python programming.

Also, limit your lines to 80 positions or less. You may need to break long statements with a `\` at the end of a line. Also, parenthesized expressions can be continued onto the next line without a `\`. Some programmers will put in extra ()'s just to make line breaks neat.

Names. Function names are typically `mixedCase`. However, a few important functions were done in `CapWords` style with a leading upper case letter. This can cause confusion with class names, and the recommended style is a leading lowercase letter for function names.

In some languages, many related functions will all be given a common prefix. Functions may be called `inet_addr`, `inet_network`, `inet_makeaddr`, `inet_lnaof`, `inet_netof`, `inet_ntoa`, etc. Because Python has classes (covered in Part III, "Data + Processing = Objects") and modules (covered in Part IV, "Components, Modules and Packages"), this kind of function-name prefix is not used in Python programs. The class or module name is the prefix. Look at the example of `math` and `random` for guidance on this.

Parameter names are also typically *`mixedCase`*. In the event that a parameter or variable name conflicts with a Python keyword, the name is extended with an _. In the following example, we want our parameter to be named *`range`*, but this conflicts with the builtin function `range`. We use a trailing _ to sort this out.

```
def integrate( aFunction, range_ ):
    """Integrate a function over a range."""
    ...
```

Blank lines are used sparingly in a Python file, generally to separate unrelated material. Typicaly, function definitions are separated by single blank lines. A long or complex function might have blank lines within the body. When this is the case, it might be worth considering breaking the function into separate pieces.

Docstrings. The first line of the body of a function is called a *docstring*. The recommended forms for docstrings are described in Python Extension Proposal (PEP) 257.

Typically, the first line of the docstring is a pithy summary of the function. This may be followed by a blank line and more detailed information. The one-line summary should be a complete sentence.

```
def fact( n ):
    """fact( number ) -> number

    Returns the number of permutations of n things."""
    if n == 0: return 1L
    return n*fact(n-1L)

def bico( n, r ):
    """bico( number, number ) -> number

    Returns the number of combinations of n things
    taken in subsets of size r.
    Arguments:
    n -- size of domain
    r -- size of subset
    """
    return fact(n)/(fact(r)*fact(n-r))
```

The docsting can be retrieved with the `help` function.

`help (`*`object`*`)`

> Type `help` for interactive help, or `help (`*`object`*`)` for help about *`object`*.

Here's an example, based on our `fact` shown above.

```
>>> help(fact)

Help on function fact in module __main__:

fact(n)
    fact( number ) -> number

    Returns the number of permutations of n things.
```

Chapter 10. Additional Notes On Functions

The global Statement

Table of Contents

We cover some more advanced techniques available in Python in the section called "Returning Multiple Values". In the section called "Functions and Namespaces" we'll describe some of the internal mechanisms Python uses for storing variables. We'll introduce the **global** statement in the section called "The **global** Statement". We'll include a digression on the two common argument binding mechanisms: call by value and call by reference in the section called "Call By Value and Call By Reference". Finally, we'll cover some aspects of functions as first-class objects in the section called "Function Objects".

Functions and Namespaces

This is an overview of how Python determines the meaning of a name. We'll omit some details to hit the more important points. For more information, see section 4.1 of the *Python Language Reference*.

The important issue is that we want variables created in the body of a function to be private to that function. If all variables are global, then each function runs a risk of accidentally disturbing the value of a global variable. In the COBOL programming language (without using separate compilation or any of the moden extensions) all variables are globally declared in the data division, and great care is required to prevent accidental or unintended use of a variable.

To achieve privacy and separation, Python maintains several dictionaries of variables. These dictionaries define the context in which a variable name is understood. Because these dictionaries are used for resolution of variables, which name objects, they are called *namespaces*. A global namespace is available to all modules that are part of the currently executing Python script. Each module, class, function, lambda, or anonymous block of code given to the **exec** command has its own private namespace.

Names are resolved using the nested collection of namespaces that define an execution environment. The Python always checks the most-local dictionary first, ending with the global dictionary.

Consider the following script.

```
def deep( hi, mom ):
    do some work

def shallow( hows, things ):
    deep( hows, 1 )
```

```
    deep( things, coffee )

hows="a"
coffee="b"
shallow( hows, coffee )
```

❸ This is the main function, it executes in the global namespace, where two variables are defined, along with two functions, `deep` and `shallow`.

❷ The `shallow` function has a local namespace, where two variables are defined: `hows` and `things`. When `shallow` is called from the main script, the local `hows` hides the global variable with the same name. The reference to `coffee` is not resolved in the local namespace, but is resolved in the global namespace. This is called a *free* variable, and is sometimes a symptom of poor software design.

❶ The `deep` function has a local namespace, where two variables are defined: `hi` and `mom`. When `deep` is called from `shallow`, there are three nested scopes that define the environment: the local namespace for `deep`, the local namespace for `shallow`, and the global namespace for the main script.

Built-in Functions. If you evaluate the function `globals`, you'll see the mapping that contains all of the global variables Python knows about. For these early programs, all of our variables are global.

If you simply evaluate `locals`, you'll see the same thing. However, if you call `locals` from within the body of a function, you'll be able to see the difference between local and global variables.

The following example shows the creation of a gobal variable `a`, and a global function, `q`. It shows the local namespace in effect while the function is executing. In this local namespace we also have a variable named `a`.

```
>>> a=22.0
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__d
oc__': None, 'a': 22.0}

>>> def q( x, y ):
...     a = x / y
...     print locals()
...
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'q':
 <function q at 0x007E5EF0>, '__doc__': None, 'a': 22.0}

>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'q':
 <function q at 0x007E5EF0>, '__doc__': None, 'a': 22.0}

>>> q(22.0,7.0)
{'a': 3.1428571428571428, 'y': 7.0, 'x': 22.0}

>>>
```

The function `vars` (*`context`*) accepts a parameter which is the name of a specific local context: a module, class, or object. It returns the local variables for that specific context. The local variables are kept in a local variable named `__dict__`. The `vars` function retrieves this.

The `dir` (*`object`*) function examines the `__dict__` of a specific object to locate all local variables as well as other features of the object.

Assignment statements, as well as **def** and **class** statements, create names in the local dictionary. The **del** statement removes a name from the local dictionary.

Some Consequences. Since each imported module exists in it's own namespace, all

functions and classes within that module must have their names qualified by the module name. We saw this when we imported `math` and `random`. To use the `sqrt` function, we must say `math.sqrt`, providing the module name that is used to resolve the name `sqrt`.

This module namespace assures that everything in a module is kept separate from other modules. It makes our programs clear by qualifying the name with the module that defined the name.

The module namespace also allow a module to have relatively global variables. A module, for example, can have variables that are created when the module is imported. In a sense these are global to all the functions and classes in the module. However, because they are only known within the module's namespace, they won't conflict with variables in our program or other modules.

Having to qualify names within a module can become annoying when we are making heavy use of a module. Python has ways to put elements of a module into the global namespace. We'll look at these in Part IV, "Components, Modules and Packages".

The global Statement

The suite of statements in a function definition executes with a local namespace that is different from the global namespace. This means that all variables created within a function are local to that function. When the suite finishes, these working variables are discarded.

The overall Python session works in the global namespace. Every other context (e.g. within a function's suite), there is a distinct local namespace. Python offers us the **global** statement to change the namespace search rule.

`global` *`name`*...

The **global** statement tells Python that the following names are part of the global namespace. The following example shows two functions that share a global variable.

```
ratePerHour= 45.50
def cost( hours ):
    global ratePerHour
    return hours * ratePerHour
def laborMaterials( hours, materials ):
    return cost(hours) + materials
```

Warning

The **global** statement has a consequence of tightly coupling pieces of software. This can lead to difficulty in maintenance and enhancement of the program. Classes and modules provide better ways to assemble complex programs.

As a general policy, we discourage use of the **global** statement.

Call By Value and Call By Reference

Beginning programmers can skip this section. This is a digression for experienced C and C++ programmers.

Most programming languages have a formal mechanism for determining if a parameter receives a copy of the argument (*call by value*) or a reference to the argument object (*call by name* or *call by reference*). The distinction is important because programming languages like C++ allow us to provide a reference to a global variable as input to a function and have that global variable updated by the function without an obvious

assignment statement.

The following scenario is entirely hypothetical for Python programmers, but a very real problem for C and C++ programmers. Imagine we have a function `to2`, with this kind of definition in C.

```
int to2( int *a ) {
    /* set parameter a's value to 2 */
    *a= 2;
    return 0;
}
```

This function changes the value of the variable `a` to 2. This would be termed a *side-effect* because it is in addition to any value the function might return normally.

When we do the following in C

```
int x= 27;
int z= to2( &x );
printf( "x=%i, z=%i", x, z );
```

We get the unpleasant side-effect that our function `to2` has changed the argument variable, `x`, and the variable wasn't in an **assignment** statement! We merely called a function, using `x` as an argument. In C, the **&** operator is a hint that a variable might be changed. Further, the function definition *should* contain the keyword **const** when the reference is properly read-only. However, these are burdens placed on the programmer to assure that the program compiles correctly.

Python does not permit this kind of call-by-reference problem in the first place. The arguments to a function are always references to the original underlying object, not the variable. In the Python version of the above example, the variable `x` is a reference to an integer object with a value of 27. The parameter variable in the `to2` function is a copy of the reference, and it is local to the function's scope. The original variable, `x`, cannot be changed by the function, and the original argument object, the integer 27, is immutable, and can't be changed either.

If an argument value is a mutable object, the parameter is a reference to that object, and the function has access to methods of that object. The methods of the object can be called, but the original object cannot be replaced with a new object.

Programmers intimate with the implementation details behind C know that manipulating a copy of a value directly can be far more efficient than working through references. This is true only when the language has some high-performance data types that would benefit from call by value processing. Unlike C or Java, Python has no data types that benefit from this kind of special case handling. All data elements are passed by reference in a simple, uniform manner.

It also means that, in general, all variable updates must be done explicitly via an **assignment** statement. This makes variable changes clear.

Function Objects

One interesting consequence of the Python world-view is that a function is an object of the class `function`, a subclass of `callable`. The common feature that all `callables` share is that they have a very simple interface: they can be called. Other `callables` include the built-in functions, generator functions (which have the **yield** statement instead of the **return** statement) and things called lambdas.

Sometimes we don't want to call and evaluate a function. Sometimes we want to do other things to or with a function. For example, the various factory functions (`int`, `long`, `float`, `complex`) can be used with the `isinstance` function instead of being

called to create a new object.

For example, `isinstance(2,int)` has a value of `True`. It uses the `int` function, but doesn't apply the `int` function.

A function object is created with the **def** statement. Primarily, we want to evaluate the function objects we create. However, because a function is an object, it has attributes, and it can be manipulated to a limited extent.

From a syntax point of view, a name followed by `()`'s is a function call. You can think of the `()`'s as the "call" operator: they require evaluation of the arguments, then they apply the function.

name (***arguments***)

When we use a function name without `()`'s, we are talking about the `function` object. There are a number of manipulations that you might want to do with a function object.

Call The Function. By far, the most common use for a function object is to call it. When we follow a function name with `()`'s, we are calling the function: evaluating the arguments, and applying the function. Calling the function is the most common manipulation.

Alias The Function. This is dangerous, because it can make a program obscure. However, it can also simplify the evoluation and enhancement of software. Imagine that the first version of our program had two functions named `rollDie` and `rollDice`. The definitions might look like the following.

```
def rollDie():
    return random.randrange(1,7)
def rollDice():
    return random.randrange(1,7) + random.randrange(1,7)
```

When we wanted to expand our program to handle five-dice games, we realized we could generalize this function.

```
def rollNDice( n=2 ):
    t= 0
    for d in range(n):
        t += random.randrange( 1, 7 )
    return t
```

It is important to remove the duplicated algorithm in all three versions of our dice rolling function. Since `rollDie` and `rollDice` are just special cases of `rollNDice`, we should replace them with something like the following.

```
def rollDie():
    return rollNDice( 1 )
def rollDice():
    return rollNDice()
```

This revised definition of `rollDice` is really just an another name for the `ollNDice`. Because a function is an object assigned to a variable, we can have multiple variables assigned to the function. Here's how we create an alias to a function.

```
rollDice = rollNDice
```

Warning

Function alias definitions helps maintaining compatibility between old and new releases of software. It is not something that should be done as a general practice; we need to be careful providing multiple

names for a given function. This can be a simplification. It can also be a big performance improvement for certain types of functions that are heavily used deep within nested loops.

Function Attributes. A function object has a number of attributes. We can interrogate those attributes, and to a limited extend, we can change some of these attributes. For more information, see section 3.2 of the *Python Language Reference* and section 2.3.9.3 of the *Python Library Reference*.

`func_doc`, `__doc__`

Docstring from the first line of the function's body.

`func_name`, `__name__`

Function name from the **def** statement.

`__module__`

Name of the module in which the function name was defined.

`func_defaults`

Tuple with default values to be assigned to each argument that has a default value. This is a subset of the parameters, starting with the first parameter that has a default value.

`func_code`

The actual code object that is the suite of statements in the body of this function.

`func_globals`

The dictionary that defines the global namespace for the module that defines this function. This is `m.__dict__` of the module which defined this function.

`func_dict`, `__dict__`

The dictionary that defines the local namespace for the attributes of this function.

You can set and get your own function attributes, also.

```
def rollDie():
    return random.randrange(1,7)
rollDie.version= "1.0"
rollDie.authoor= "sfl"
```

Data Structures

The Data View

Computer programs are built with two essential features: data and processing. We started with processing elements of Python. We're about to start looking at data structures.

In Part I, "Language Basics", we introduced almost all of the procedural elements of the Python language. We started with expressions, looking at the various operators and data types available. We described fourteen of the 21 statements that make up the Python language.

- Expression statements - for example, a function evaluation where there is no return value.

- The **import** statement - used to include a module into another module or program.
- The **print** statement - used to provide visible output.
- The **assignment** statements, from simple to augmented - used to set the type and value of a variable.
- The **del** statement - used to remove a variable, function, module or other object.
- The **if** statement - for conditionally performing suites of statements.
- The **pass** statement - which does nothing, but is a necessary placeholder for an if or while suite that is empty.
- The **assert** statement - used to confirm the program is in the expected state.
- The **for** and **while** statements - for performing suites of statements using a sequence of values or while a condition is held true.
- The **break** and **continue** statements - for short-cutting loop execution.
- The **def** statement - used to create a function.
- The **return** statement - used to exit a function, possibly providing the return value.
- The **global** statement - used to adjust the scoping rules, allowing local access to global names. We discourage its use in the section called "The **global** Statement".

The Other Side of the Coin. The next chapters focus on adding various data types to the basic Python language. The subject of data representation and data structures is possibly the most profound part of computer programming. Most of the *killer applications* - email, the world wide web, relational databases, are basically programs to create, read and transmit complex data structures.

We will make extensive use of the object classes that are built-in to Python. This experience will help us design our own object classes in Part III, "Data + Processing = Objects".

We'll work our way through the following data structures.

- **Sequences.** In Chapter 11, *Sequences: Strings, Tuples and Lists* we'll extend our knowledge of data types to include an overview various kinds of sequences: strings, tuples and lists. Sequences are collections of objects accessed by their numeric position within the collection.

    - In Chapter 12, *Strings* we describe the `string` subclass of sequence. The exercises include some challenging string manipulations.

    - We describe fixed-length sequences, called `tuples` in Chapter 13, *Tuples*.

    - In Chapter 14, *Lists* we describe the variable-length sequence, called a `list`. This `list` sequence is one of the powerful features that sets Python apart from other programming languages. The exercises at the end of the `list` section include both simple and relatively sophisticated problems.

- **Mappings.** In Chapter 15, *Mappings and Dictionaries* we describe mappings and dictionary objects, called `dicts`. We'll show how dictionaries are part of some advanced techniques for handling arguments to functions. Mappings are collections of value objects that are accessed by key objects.

- **Sets.** We'll cover `set` objects in Chapter 16, *Sets*. Sets are simple collections of unique objects with no additional kind of access.

- **Exceptions.** We'll cover `exception` objects in Chapter 17, *Exceptions*. We'll also show the exception handling statements, including **try**, **except**, **finally** and **raise** statements. Exceptions are both simple data objects and events that control the execution of our programs.

- **Iterables.** The **yield** statement is a variation on **return** that simplifies certain kinds of generator algorithms that process or create create *iterable* data structures. We can iterate through almost any kind of data collection. We can also define our

own unique or specialized iterations. We'll cover this in Chapter 18, *Generators and the **yield** Statement*.

- **Files.** The subject of files is so vast, that we'll introduce file objects in Chapter 19, *Files*. Files are so centrally important that we'll return files in Part IV, "Components, Modules and Packages". We'll look at several of the file-related modules in Chapter 33, *File Handling Modules* as well as Chapter 34, *File Formats: CSV, Tab, XML, Logs and Others*.

In Chapter 20, *Advanced Sequences* we describe more advanced sequence techniques, including multi-dimensional processing, additional sequence-processing functions, and sorting.

Deferred Topics. There are a few topics that need to be deferred until later. The **class** statement will be covered in detail in chapters on object oriented programming, starting with Part III, "Data + Processing = Objects". We'll look at the **with** statement there, also. We'll revisit the **import** statement in detail in Part IV, "Components, Modules and Packages". Additionally, we'll cover the **exec** statement in Chapter 28, *Modules*.

Table of Contents

Chapter 11. Sequences: Strings, Tuples and Lists

The Common Features of Sequences

Table of Contents

Before digging into the details, we'll introduce the common features of three of the structured data types that manipulate sequences of values. In the chapters that follow we'll look at Chapter 12, *Strings*, Chapter 13, *Tuples* and Chapter 14, *Lists* in detail. In Chapter 15, *Mappings and Dictionaries*, we'll introduce another structured data type for manipulating mappings between keys and values.

In the section called “Semantics” we will provide an overview of the semantics of sequences. We describes the common features of the sequences in the section called “Overview of Sequences”.

The sequence is central to programming and central to Python. A number of statements and functions we have covered have sequence-related features that we have glossed over, danced around, and generally avoided.

We'll revisit a number of functions and statements we covered in previous sections, and add the power of sequences to them. In particular, the **for** statement is something we glossed over in the section called “Iterative Processing: For All and There Exists”.

Semantics

A sequence is a container of objects which are kept in a specific order. We can identify objects in a sequence by their position or index. Positions are numbered from zero in Python; the element at index zero is the first element.

We call these containers because they are a single object which contains (or collects) any number of other objects. The “any number” clause means that they can collect zero other objects, meaning that an empty container is just as valid as a container with one or thousands of objects.

In some programming languages, they use words like "vector" or "array" to refer to sequential containers. Further in other languages there are very specific implementations of sequential containers. For example, in C or Java, the primitive array has a statically

allocated number of positions. In Java, a reference outside that specific number of positions raises an exception. In C, however a reference outside the defined positions of an array is an error that may never be detected. Really.

There are four commonly-used subspecies of sequence containers. The `string`, the `Unicode string`, the `tuple` and the `list`. A `string` is a container of single-byte ASCII characters. A `Unicode string` is a container of multi-byte Unicode (or Universal Character Set) characters. A `tuple` and a `list` are more general containers.

When we create a `tuple` or `string`, we've created an *immutable*, or static object. We can examine the object, looking at specific characters or objects. We can't change the object. This means that we can't put additional data on the end of a `string`. What we can do, however, is create a new `string` that is the concatenation of the two original `strings`.

When we create a `list`, on the other hand, we've created a *mutable* object. A `list` can have additional objects appended to it or inserted in it. Objects can be removed from a `list`, also. A `list` can grow and shrink; the order of the objects in the `list` can be changed without creating a new `list` object.

One other note on `string`. While `string` are sequences of characters, there is no separate character data type. A character is simply a `string` of length one. This relieves programmers from the C or Java burden of remembering which quotes to use for single characters as distinct from multi-character `strings`. It also eliminates any problems when dealing with Unicode multi-byte characters.

Overview of Sequences

All the varieties of sequences (`strings`, `tuples` and `lists`) have some common characteristics. We'll identify the common features first, and then move on to cover these in detail for each individual type of sequence. This section is a road-map for the following three sections that cover `strings`, `tuples` and `lists` in detail.

Literal Values. Each sequence type has a literal representation. The details will be covered in separate sections, but the basics are these: `strings` are quoted: `"string"`; `tuples` are in `()`'s: `(1,'b',3.1)`; `lists` are in `[]`'s: `[1,'b',3.1]`. A `tuple` or a `list` is a sequences of various types of items. A `string` is a sequence of characters only.

Operations. Sequences have three common operations: `+` will concatenate sequences to make longer sequences. `*` is used with a number and a sequence to repeat the sequence several times. Finally, the `[]` operator is used to select elements from a sequence.

The [ ] operator can extract a single item, or a subset of items by slicing. There are two forms of `[]`. The single item format is ***sequence*** **[** ***index*** **]**. Items are numbered from 0. The slice format is ***sequence*** **[** ***start*** **:** ***end*** **]**. Items from *start* to *end*-1 are chosen to create a new sequence as a slice of the original sequence; there will be *end*−*start* items in the resulting sequence.

Positions can be numbered from the end of the `string` as well as the beginning. Position -1 is the last item of the sequence, -2 is the next-to-last item. Here's how it works: each item has a positive number position that identifies the item in the sequence. We'll also show the negative position numbers for each item in the sequence. For this example, we're looking at a four-element sequence like the `tuple (3.14159,"two words",2048,(1+2j))`.

forward position	0	1	2	3
reverse position	-4	-3	-2	-1
item	3.14159	"two	2048	(1+2j)

		words"		

Why do we have two different ways of identifying each position in the sequence? If you want, you can think of it as a handy short-hand. The last item in any sequence, `S` can be identified by the formula `S[ len(S)-1 ]`. For example, if we have a sequence with 4 elements, the last item is in position 3. Rather than write `S[ len(S)-1 ]`, Python lets us simplify this to `S[-1]`.

You can see how this works with the following example.

```
>>> a=(3.14159,"two words",2048,(1+2j))
>>> a[0]
3.1415899999999999
>>> a[-3]
'two words'
>>> a[2]
2048
>>> a[-1]
(1+2j)
```

Built-in Functions. `len`, `max` and `min` apply to all varieties of sequences. `len` returns the length of the sequence. `len("Wednesday")` is 9. `max` returns the largest value in the sequence. `max( (1,2,3) )` is 3. `min`, analogously, returns the smallest value in the sequence. `min( [19,9,99] )` is 9.

Comparisons. The standard comparisons (<, <=, >, <=, ==, !=) apply to sequences. These all work by doing item-by-item comparison within the two sequences. The item-by-item rule results in `strings` being sorted alphabetically, and `tuples` and `lists` sorted in a way that is similar to `strings`.

There are two additional comparisons: **in** and **not in**. These check to see if a single value occurs in the sequence. The **in** operator returns a `True` if the item is found, `False` if the item is not found. The **not in** operator returns `True` if the item is not found in the sequence.

Methods. The `string` and `list` classes have method functions that operate on the object's value. For instance `"abc".upper()` executes the `upper` method belonging to the `string` literal `"abc"`. The result is `'ABC'`. The exact dictionary of methods is unique to each class of sequences.

Statements. `tuples` and `lists` are central to certain Python statements, like the **assignment** statement and the **for** statement. These were details that we skipped over in the section called "The **Assignment** Statement" and the section called "Iterative Processing: For All and There Exists". The additional `tuple`-specific details of these statements will be covered in Chapter 13, *Tuples*.

Modules. There is a `string` module with several `string`-specific functions. Most of these functions are now member functions of the `string` type, except for a special-purpose function used to create translation tables. Additionally, this module has a number of constants to define various subsets of the ASCII character set, including digits, printable characters, whitespace characters and others.

Factory Functions. There are also built-in factory (or conversion) functions for the sequence objects.

`repr` (*`object`*) → string

> Return the canonical `string` representation of the object. For most object types, `eval(repr(object)) == object`.

`str` (*`object`*) → string

Return a nice `string` representation of the object. If the argument is a `string`, the return value is the same object.

`list` (*`sequence`*) → list

Return a new `list` whose items are the same as those of the argument sequence.

`tuple` (*`sequence`*) → tuple

Return a new `tuple` whose items are the same as those of the argument sequence. If the argument is a `tuple`, the return value is the same object.

Exercises

1. **Tuples and Lists.** What is the value in having both immutable sequences (`tuples`) and mutable sequences (`lists`)? What are the circumstances under which you would want to change a `string`? What are the problems associated with `strings` that grow in length? How can storage for variable length `strings` be managed?

2. **Unicode Strings.** What is the value in making a distinction between Unicode strings and ASCII strings? Does it improve performance to restrict a `string` to single-byte characters? Should all strings simply be Unicode strings to make programs simpler? How should file reading and writing be handled?

3. **Statements and Data Structures.** In order to introduce the **for** statement in the section called "Iterative Processing: For All and There Exists", we had to dance around the sequence issue. Would it make more sense to introduce the various sequence data structures first, and then describe statements that process the data structure later?

   Something has to be covered first, and is therefore more fundamental. Is the processing statement more fundamental to programming, or is the data structure?

Style Notes

Try to avoid extraneous spaces in `lists` and `tuples`. Python programs should be relatively compact. Prose writing typically keeps ()'s close to their contents, and puts spaces after commas, never before them. This should hold true for Python, also. The preferred formatting for `lists` and `tuples`, then, is `(1,2,3)` or `(1, 2, 3)`. Spaces are not put after the enclosing `[ ]`'s or `( )`'s. Spaces are not put before `,`'s.

Chapter 12. Strings

Table of Contents

We'll look at `strings` from a number of viewpoints: semantics, literal values, operations, comparison operators, built-in functions, methods and modules. Additionally, we have a digression on the immutability of `strings`.

String Semantics

A `string` is an immutable sequence of characters. Since it is a sequence, all of the common operations to sequences apply. We can append `strings` together, select characters from a `string`. When we select a slice from a `string`, we've extracted a substring.

String Literal Values

A `string` is a sequence of characters. The literal value for a `string` is written by surrounding the value with quotes or apostrophes. There are several variations to provide some additional features.

Basic String

> `"xyz"` or `'xyz'`. A basic `string` must be completed on a single line, or continued with a `\` as the very last character of a line.

Multi-Line String, Triple-Quoted String

> `"""xyz"""` or `'''xyz'''`. A multi-line `string` continues on until the concluding triple-quote or triple-apostrophe.

Unicode String

> `u"Unicode"`, `u'Unicode'`, `u"""Unicode"""`, etc. Unicode is the Universal Character Set; each character requires from 1 to 4 bytes of storage. ASCII is a single-byte character set; each of the 256 ASCII characters requires a single byte of storage. Unicode permits any character in any of the languages in common use around the world.

Raw String

> `r"raw\nstring"`, r'raw\nstring', etc. The backslash characters (\) are *not* interpreted by Python, but are left as is. This is handy for Windows files names that contain \'s. It is also handy for regular expressions that make extensive use of backslashes. Example: `'\n'` is a one-character `string` with a non-printing newline; `r'\n'` is a two-character `string`.

Outside of raw strings, non-printing characters and Unicode characters that aren't found on your keyboard are created using *escapes*. A table of escapes is provided below. These are Python representations for unprintable ASCII characters. They're called escapes because the `\` is an escape from the usual meaning of the following character.

Escape	Meaning
`\` at end of a line	The end-of-line is ignored, the `string` continues on the next line
`\\`	Backslash (\)
`\'`	Apostrophe (')
`\"`	Quote (")
`\a`	ASCII Bell (BEL), an audible signal. Some OS's translate this to a screen flash or ignore it completely.
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\`*ooo*	ASCII character with octal value *ooo*. Exactly three octal digits are

	required.
\x*hh*	ASCII character with hex value *hh*

Note that adjacent strings are automatically put together to make a longer string.

"ab" "cd" "ef" is the same as "abcdef".

For Unicode, a special \u*xxxx* escape is provided. This requires the four digit Unicode character identification. 日本 is written in Python as u'\u65e5\u672c' using two Unicode characters provided via escapes. There are a variety of Unicode encoding schemes, for example, UTF-8, UTF-16 and LATIN-1. The codecs module provides mechanisms for encoding and decoding Unicode strings.

String Operations

There are a number of operations on strings, operations which create strings and operations which create other objects from strings.

There are three operations (+, *, [ ]) that work with all sequences (including strings) and a unique operation, %, that can be performed only with strings.

The + operator creates a new string as the concatenation of the arguments.

```
>>> "hi " + 'mom'
'hi mom'
```

The * operator between strings and numbers (number * string or string * number) creates a new string that is a number of repetitions of the input string.

```
>>> print 3*"cool!"
'cool!cool!cool!'
```

The [ ] operator can extract a single character or a slice from the string. There are two forms. The single item format is ***string*** **[** ***index*** **]**. Items are numbered from 0 to len(*string*). Items are also numbered in reverse from −len(*string*) to -1. The slice format is ***string*** **[** ***start*** **:** ***end*** **]**. Characters from *start* to *end*-1 are chosen to create a new string as a slice of the original string; there will be *end*−*start* characters in the resulting string. If *start* is omitted it is the beginning of the string (position 0), if *end* is omitted it is the end of the string (position -1).

```
>>> s="adenosine"
>>> s[2]
'e'
>>> s[:5]
'adeno'
>>> s[-5:]
'osine'
>>> s[5:]
'sine'
```

The String Formatting Operation, %. The % operator is sometimes call string interpolation, since it interpolates literal text and converted values. We prefer to call it string formatting, since that is a more apt description. This formatting is taken straight from the C library's printf function.

This operator has two forms. You can use it with a string and value as well as a string and a tuple. We'll cover tuples in detail later, but for now, it is a comma-separated collection of values in ()'s.

The `string` on the left-hand side of `%` contains a mixture of *literal text* plus *conversion specifications*. A conversion specification begins with `%`. For example, integers are converted with `%i`. Each conversion specification will use a corresponding value from the `tuple`. The first conversion uses the first value of the `tuple`, the second conversion uses the second value from the `tuple`. For example:

```
import random
d1, d2 = random.randrange(1,6), random.randrange(1,6)
r= "die 1 shows %i, and die 2 shows %i" % ( d1, d2 )
```

The first `%i` will convert the value for `d1` to a `string` and insert the value, the second `%i` will convert the value for `d2` to a `string`. The `%` operator returns the new `string` based on the format, with each conversion specification replaced with the appropriate values.

Conversion Specifications. Each conversion specification has from one to four elements, following this pattern:

%[flags][width[.precision]]code

The `%` and the final code in each conversion specification are required. The other elements are optional.

The optional *`flags`* element can have any combination of the following values:

`-`

Left adjust the converted value in a field that has a length given by the *`width`* element. The default is right adjustment.

`+`

Show positive signs (sign will be + or -). The default is to show negative signs only.

␣ (a *space*)

Show positive signs with a space (sign will be ␣ or -). The default is negative signs only.

`#`

Use the Python literal rules (0 for octal, 0x for hexadecimal, etc.) The default is decoration-free notation.

`0`

Zero-fill the the field that has a length given by the *`width`* element. The default is to space-fill the field. This doesn't make a lot of sense with the - (left-adjust) flag.

The optional *`width`* element is a number that specifies the total number of characters for the field, including signs and decimal points. If omitted, the width is just big enough to hold the output number. If a `*` is used instead of a number, an item from the `tuple` of values is used as the width of the field. For example, `"%*i" % ( 3, d1 )` uses the value 3 from the `tuple` as the field width and `d1` as the value to convert to a `string`.

The optional *`precision`* element (which must be preceded by a dot, `.` if it is present) has a few different purposes. For numeric conversions, this is the number of digits to the right of the decimal point. For `string` conversions, this is the maximum number of characters to be printed, longer `strings` will be truncated. If a `*` is used instead of a number, an item from the `tuple` of values is used as the precision of the conversion. For example, `"%*.*f" % ( 6, 2, avg )` uses the value 6 from the `tuple` as the field width, the value 2 from the `tuple` as the precision and `avg` as the value.

The standard conversion rules also permit a long or short indicator: `l` or `h`. These are tolerated by Python so that these formats will be compatible with C, but they have no effect. They reflect internal representation considerations for C programming, not external formatting of the data.

The required one-letter *`code`* element specifies the conversion to perform. The codes are listed below.

`%`

Not a conversion, this creates a % in the resulting `string`. Use `%%` to put a `%` in the output `string`.

`c`

Convert a single-character `string`. This will also convert an integer value to the corresponding ASCII character. For example, `"%c" % ( 65, )` results in `"A"`.

`s`

Convert a `string`. This will convert non-`string` objects by implicitly calling the `str` function.

`r`

Call the `repr` function, and insert that value.

`i, d`

Convert a numeric value, showing ordinary decimal output. The code i stands for integer, d stands for decimal. They mean the same thing; but it's hard to reach a consensus on which is "correct".

`u`

Convert an *unsigned* number. While relevant to C programming, this is the same as the `i` or `d` format conversion.

`o`

Convert a numeric value, showing the octal representation. `%#0` gets the Python-style value with a leading zero.

`x, X`

Convert a numeric value, showing the hexadecimal representation. `%#X` gets the Python-style value with a leading `0X`; `%#x` gets the Python-style value with a leading `0x`.

`e, E`

Convert a numeric value, showing scientific notation. `%e` produces *`±d.dde±xx`* , `%E` produces *`±d.dddE±xx`* . For example `6.02E23`.

`f, F`

Convert a numeric value, using ordinary decimal notation. In case the number is gigantic, this will switch to `%g` or `%G` notation.

`g, G`

"Generic" floating-point conversion. For values with an exponent larger than -4, and smaller than the *`precision`* element, the `%f` format will be used. For values

with an exponent smaller than -4, or values larger than the *precision* element, the %e or %E format will be used.

Here are some examples.

```
"%i: %i win, %i loss, %6.3f" % (count,win,loss,float(win)/loss)
```

This example does four conversions: three simple integer and one floating point that provides a width of 6 and 3 digits of precision. ±0.000 is the expected format. The rest of the string is literally included in the output.

```
"Spin %3i: %2i, %s" % (spin,number,color)
```

This example does three conversions: one number is converted into a field with a width of 3, another converted with a width of 2, and a string is converted, using as much space as the string requires.

String Comparison Operations

The standard comparisons (<, <=, >, >=, ==, !=) apply to strings. These comparisons use the standard character-by-character comparison rules for ASCII or Unicode.

There are two additional comparisons: **in** and **not in**. These check to see if a single character string occurs in a longer string. The **in** operator returns a True when the character is found, False if the character is not found. The **not in** operator returns True if the character is not found.

```
>>> "a" in 'xyxxyabcxyzzy'
True
```

String Built-in Functions

The following built-in functions are relevant to string manipulation

chr(*i*) → character

Return a string of one character with ordinal i; 0 ≤ *i* < 256.

len(*object*) → integer

Return the number of items of a sequence or mapping.

ord(*c*) → integer

Return the integer ordinal of a one character string

repr(*object*) → string

Return the canonical string representation of the object. For most object types, eval(repr(object)) == object.

str(*object*) → string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

unichr(*i*) → Unicode string

Return a Unicode string of one character with ordinal i; 0 ≤ *i* < 65536.

unicode(*string*, [*encoding*,] [*errors*]) → Unicode string

> Creates a new Unicode object from the given encoded `string`. *`encoding`* defaults to the current default `string` encoding and *`errors`*, defining the error handling, to 'strict'.

For character code manipulation, there are three related functions: `chr`, `ord` and `unichr`. `chr` returns the ASCII character that belongs to an ASCII code number. `unichr` returns the Unicode character the belongs to a Unicode number. `ord` transforms an ASCII character to its ASCII code number, or transforms a Unicode character to its Unicode number.

The `len` function returns the length of the `string`.

```
>>> len("abcdefg")
7
>>> len(r"\n")
2
>>> len("\n")
1
```

The `str` function converts any object to a `string`.

```
>>> a= str(355.0/113.0)
>>> a
'3.14159292035'
>>> len(a)
13
```

The `repr` function also converts an object to a `string`. However, `repr` usually creates a `string` suitable for use as Python source code. For simple numeric types, it's not terribly interesting. For more complex, types, however, it reveals details of their structure. It can also be invoked using the *reverse quotes* (`), also called accent grave, (underneath the tilde, ~, on most keyboards).

```
>>> a="""a very
... long string
... on multiple lines"""
>>> print repr(a)
'a very\012long string\012on multiple lines'
>>> print `a`
'a very\012long string\012on multiple lines'
```

This representation shows the newline characters (`\012`) embedded within the triple-quoted `string`. If we simply print `a` or `str(`*`a`*`)`, we would see the string interpreted instead of represented.

```
>>> a="""a very
... long string
... on multiple lines"""
>>> print a
a very
long string
on multiple lines
```

The `unicode(`*`string`*`, [`*`encoding`*`, ] [`*`errors`*`])` function converts the *`string`* to a specific Unicode external representation. The default *`encoding`* is 'UTF-8' with 'strict' error handling. Choices for *`errors`* are 'strict', 'replace' and 'ignore'. Strict raises an exception for unrecognized characters, replace substitutes the Unicode replacement character (`\uFFFD`) and ignore skips over invalid characters. The `codecs` and `unicodedata` modules provide more functions for working with Unicode.

String Methods

A `string` object has a number of method functions. These can be grouped arbitrarily into transformations, which create new `strings` from old, and information, which returns a fact about a `string`. In all of the following method functions, we'll assume a `string` object named *`s`*.

The following `string` transformation functions create a new `string` from an existing `string`.

`s``.capitalize` → string

Create a copy of the `string` *`s`* with only its first character capitalized.

`s``.center(`*`width`*`)` → string

Create a copy of the `string` *`s`* centered in a `string` of length *`width`*. Padding is done using spaces.

`s``.encode(`*`encoding`*`, [`*`errors`*`])` → string

Return an encoded `string` version of *`s`*. Default *`encoding`* is the current default `string` encoding. *`errors`* may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a `ValueError`. Other possible values are 'ignore' and 'replace'.

`s``.expandtabs ([`*`tabsize`*`])` → string

Return a copy of *`s`* where all tab characters are expanded using spaces. If *`tabsize`* is not given, a tab size of 8 characters is assumed.

`s``.join(`*`sequence`*`)` → string

Return a `string` which is the concatenation of the `strings` in the *`sequence`*. The separator between elements is *`s`*.

`s``.ljust(`*`width`*`)` → string

Return *`s`* left justified in a `string` of length *`width`*. Padding is done using spaces.

`s``.lower` → string

Return a copy of *`s`* converted to lowercase.

`s``.lstrip` → string

Return a copy of *`s`* with leading whitespace removed.

`s``.replace(`*`old`*`, `*`new`*`, [`*`maxsplit`*`])` → string

Return a copy of `string` *`s`* with all occurrences of substring *`old`* replaced by *`new`*. If the optional argument *`maxsplit`* is given, only the first *`maxsplit`* occurrences are replaced.

`s``.rjust(`*`width`*`)` → string

Return *`s`* right justified in a `string` of length *`width`*. Padding is done using spaces.

`s``.rstrip` → string

Return a copy of *`s`* with trailing whitespace removed.

s.strip → string

Return a copy of *s* with leading and trailing whitespace removed.

s.swapcase → string

Return a copy of *s* with uppercase characters converted to lowercase and vice versa.

s.title → string

Return a titlecased version of *s*, i.e. words start with uppercase characters, all remaining cased characters have lowercase.

s.translate(*table*, [*deletechars*]) → string

Return a copy of the string *s*, where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation *table*, which must be a string of length 256. The translation tables are built using the string.maketrans function in the string module.

s.upper → string

Return a copy of *s* converted to uppercase.

The following accessor methods provide information about a string.

s.count(*sub*, [*start*,] [*end*]) → int

Return the number of occurrences of substring *sub* in string *s*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

s., (endswith*suffix*, [*start*,] [*end*]) → boolean

Return True if *s* ends with the specified *suffix*, otherwise return False. With optional *start*, or *end*, test *s*[*start*:*end*]. The suffix can be a single string or a sequence of individual strings.

s., (find*sub*, [*start*,] [*end*]) → int

Return the lowest index in *s* where substring *sub* is found, such that *sub* is contained within *s*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 on failure.

s., (index*sub*, [*start*,] [*end*]) → int

Like *s*. find but raise ValueError when the substring is not found.

s., (isalnum) → boolean

Return True if all characters in *s* are alphanumeric and there is at least one character in *s*, False otherwise.

s., (isalpha) → boolean

Return True if all characters in *s* are alphabetic and there is at least one character in *s*, False otherwise.

s., (isdigit) → boolean

Return True if all characters in *s* are digits and there is at least one character in *s*,

`False` otherwise.

s.`islower()` → boolean

Return `True` if all characters in *s* are lowercase and there is at least one cased character in *s*, `False` otherwise.

s.`isspace()` → boolean

Return `True` if all characters in *s* are whitespace and there is at least one character in *s*, `False` otherwise.

s.`istitle()` → boolean

Return `True` if *s* is a titlecased `string`, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones, `False` otherwise.

s.`isupper()` → boolean

Return `True` if all characters in *s* are uppercase and there is at least one cased character in *s*, `False` otherwise.

s.`rfind`*sub*`[`, *start*`[`, *end*`]])` → int

Return the highest index in *s* where substring *sub* is found, such that *sub* is contained within *s*`[`*start*`:`*end*`]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 on failure.

s.`rindex`*sub*`[`, *start*`[`, *end*`]])` → int

Like *s.*`rfind` but raise `ValueError` when the substring is not found.

s.`startswith`*prefix*`[`, *start*`[`, *end*`]])` → boolean

Return `True` if *s* starts with the specified *prefix*, otherwise return `False`. With optional *start*, or *end*, test *s*`[`*start*`:`*end*`]`. The prefix can be a single string or a sequence of individual strings.

The following generators create another kind of object, usually a sequence, from a `string`.

s.`partition(`*sep*`)` → 3-tuple

Return a `three-tuple` of the text prior to the first occurance of *sep* in the `string` *s*, the *sep* as the delimiter, and the text after the first occurance of the separator. If the separator doesn't occur, all of the input string is in the first element of the 3-tuple; the other two elements are empty strings.

s.`split(`*sep*`[`, *maxsplit*`])` → sequence

Return a `list` of the words in the `string` *s*, using *sep* as the delimiter `string`. If *maxsplit* is given, at most *maxsplit* splits are done. If *sep* is not specified, any whitespace `string` is a separator.

s.`splitlines (`*[keepends]*`)` → sequence

Return a `list` of the lines in *s*, breaking at line boundaries. Line breaks are not included in the resulting `list` unless *keepends* is given and true.

Here's another example of using some of the `string` methods and slicing operations.

```
for arg in sys.argv[1:]:
```

```
    argCap= arg.upper()
    if argCap[:2] == "-D":
        if argCap[2:] == "MYSQL":
            print "MySQL Connection"
        elif argCap[2:] == "POSTGRES":
            print "Postgres Connection"
        elsif argCap[2:] == "SQLITE"
            print "SQLite Connection"
        else:
            print "'%s' is an unknown -D option" % argCap[2:]
    else:
        parse other option types
```

We use the `upper` function to translate the provided parameter to upper case. This simplifies the comparison between the parameters. We take slices of each parameter `string` to compare the initial portion to see if it is `-D`, and we compare the final portion to a number of literal `strings`. Additionally, we use the formatting operation, **%**, to format an error report.

String Modules

There is an older module named `string`. Almost all of the functions in this module are directly available as methods of the `string` type. The one remaining function of value is the `maketrans` function, which creates a translation table to be used by the `translate` method of a `string`. Beyond that, there are a number of public module variables which define various subsets of the ASCII characters.

`maketrans(`*`from`*`,` *`to`* `)` → string

> Return a translation table (a `string` 256 characters long) suitable for use in `string.translate`. The `strings` *`from`* and *`to`* must be of the same length.

The following example shows how to make and then apply a translation table.

```
>>> from string import maketrans
>>> t= maketable("aeiou","xxxxx")
>>> phrase= "now is the time for all good men to come to the aid of their party"
>>> phrase.translate(t)
'nxw xs thx txmx fxr xll gxxd mxn tx cxmx tx thx xxd xf thxxr pxrty'
```

More importantly, this module contains a number of definitions of the characters in the ASCII character set. These definitions serve as a central, formal repository for facts about the character set. Note that there are general definitions, applicable to Unicode character setts, different from the ASCII definitions.

`ascii_letters`

> The set of all letters, essentially a union of `ascii_lowercase` and `ascii_uppercase`.

`ascii_lowercase`

> The lowercase letters in the ASCII character set: `'abcdefghijklmnopqrstuvwxyz'`

`ascii_uppercase`

> The uppercase letters in the ASCII character set: `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`

`digits`

The digits used to make decimal numbers: `'0123456789'`

`hexdigits`

The digits used to make hexadecimal numbers: `'0123456789abcdefABCDEF'`

`letters`

This is the set of all letters, a union of `lowercase` and `uppercase`, which depends on the setting of the locale on your system.

`lowercase`

This is the set of lowercase letters, and depends on the setting of the locale on your system.

`octdigits`

The digits used to make octal numbers: `'01234567'`

`printable`

All printable characters in the character set. This is a union of digits, letters, punctuation and whitespace.

`punctuation`

All punctuation in the ASCII character set, this is `'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'`

`uppercase`

This is the set of uppercase letters, and depends on the setting of the locale on your system.

`whitespace`

A collection of characters that cause spacing to happen. For ASCII this is `'\t\n\x0b\x0c\r '`

String Exercises

1. **Check Amount Writing.** Translate a number into the English phrase.

   This example algorithm fragment is only to get you started. This shows how to pick off the digits from the right end of a number and assemble a resulting `string` from the left end of the `string`.

   Note that the right-most two digits have special names, requiring some additional cases above and beyond the simplistic loop shown below. For example, 291 is "two hundred ninety one", where 29 is "twenty nine". The word for "2" changes, depending on the context.

   As a practical matter, you should analyze the number by taking off three digits at a time, the expression `(number % 1000)` does this. You would then format the three digit number with words like "million", "thousand", etc.

   Procedure 12.1. English Words For An Amount, *n*

   1. **Initialization**

      Set `result` ← ""

      Set `tensCounter` ← 0

2. **Loop.** While $n > 0$

    a. **Get Right Digit.** Set `digit` ← n % 10, the remainder when divided by 10.

    b. **Make Phrase.** Translate `digit` to a `string` from "zero" to "nine". Translate `tensCounter` to a `string` from "" to "thousand".

    c. **Assemble Result.** Prepend `digit string` and `tensCounter string` to the left end of the `result string`.

    d. **Next Digit.** $n \leftarrow n \div 10$. `tensCounter` ← `tensCounter` + 1.

3. **Result.** Return `result` as the English translation of n.

2. **Roman Numerals.** This is similar to translating numbers to English. Instead we will translate them to Roman Numerals.

    The Algorithm is similar to Check Amount Writing (above). You will pick off successive digits, using %10 and /10 to gather the digits from right to left.

    The rules for Roman Numerals involve using four pairs of symbols for ones and five, tens and fifties, hundreds and five hundreds. An additional symbol for thousands covers all the relevant bases.

    When a number is followed by the same or smaller number, it means addition. "II" is two 1's = 2. "VI" is 5 + 1 = 6.

    When one number is followed by a larger number, it means subtraction. "IX" is 1 before 10 = 9. "IIX" isn't allowed, this would be "VIII".

    For numbers from 1 to 9, the symbols are "I" and "V", and the coding works like this.

    a. "I"
    b. "II"
    c. "III"
    d. "IV"
    e. "V"
    f. "VI"
    g. "VII"
    h. "VIII"
    i. "IX"

    The same rules work for numbers from 10 to 90, using "X" and "L". For numbers from 100 to 900, using the sumbols "C" and "D". For numbers between 1000 and 4000, using "M".

    Here are some examples. 1994 = MCMXCIV, 1956 = MCMLVI, 3888= MMMDCCCLXXXVIII

Digression on Immutability of Strings

In Chapter 12, *Strings* and Chapter 13, *Tuples* we noted that `strings` and `tuples` are both immutable. They cannot be changed once they are created. Programmers experienced in other languages sometimes find this to be an odd restriction.

Two common questions that arise are how to expand a `string` and how to remove characters from a `string`.

Generally, we don't expand or contract a `string`, we create a new `string` that is the concatenation of the original `strings`. For example:

```
>>> a="abc"
>>> a=a+"def"
>>> a
'abcdef'
```

In effect, Python gives us `strings` of arbitrary size. It does this by dynamically creating new `strings` instead of modifying existing `strings`.

Some programmers who have extensive experience in other languages will ask if creating a new `string` from the original `strings` is the most efficient way to accomplish this. Or they suggest that it would be "simpler" to allow mutable `strings` for this kind of concatenation. The short answer is that Python's storage management makes this use if immutable `strings` the simplest and most efficient. We'll discuss this in some depth in the section called "Digression on Immutability of Strings".

Responses to the immutability of `tuples` and `lists` vary, including some of the following frequently asked questions.

Q: Since lists do everything tuples do and are mutable, why bother with tuples?
Q: Wouldn't it be more efficient to allow mutable strings?

Q: Since `lists` do everything `tuples` do and are mutable, why bother with `tuples`?

A: Immutable `tuples` are more efficient than variable-length `lists`. There are fewer operations to support. Once the `tuple` is created, it can only be examined. When it is no longer referenced, the normal Python garbage collection will release the storage for the `tuple`.

Many applications rely on fixed-length `tuples`. A program that works with coordinate geometry in two dimensions may use 2-tuples to represent (x,y) coordinate pairs. Another example might be a program that works with colors as 3-tuples, (r,g,b), of red, green and blue levels. A variable-length `list` is not appropriate for these kinds of fixed-length tuple.

Q: Wouldn't it be "more efficient" to allow mutable `strings`?

A: There are a number of axes for efficiency: the two most common are time and memory use.

A mutable `string` could use less memory. However, this is only true in the benign special case where we are only replacing or shrinking the `string` within a fixed-size buffer. If the `string` expands beyond the size of the buffer the program must either crash with an exception, or it must switch to dynamic memory allocation. Python simply uses dynamic memory allocation from the start. C programs often have serious security problems created by attempting to access memory outside of a `string` buffer. Python avoids this problem by using dynamic allocation of immutable `string` objects.

Processing a mutable `string` could use less time. In the cases of changing a `string` in place or removing characters from a `string`, a fixed-length buffer would require somewhat less memory management overhead. Rather than indict Python for offering immutable `strings`, this leads to some productive thinking about `string` processing in general.

In text-intensive applications we may want to avoid creating separate `string` objects. Instead, we may want to create a single `string` object -- the input buffer -- and work with slices of that buffer. Rather than create `strings`, we can create `slice` objects that describe starting and ending offsets within the one-and-only input buffer.

If we then need to manipulate these slices of the input buffer, we can create new `strings` only as needed. In this case, our application program is designed for efficiency. We use the Python `string` objects when we want flexibility and simplicity.

Chapter 13. Tuples

Table of Contents

We'll look at `tuples` from a number of viewpoints: semantics, literal values, operations, comparison operators, statements, built-in functions and methods. Additionally, we have a digression on the immutability of `strings`. Additionally, we have a digression on the Σ operator.

Tuple Semantics

A `tuple` is a container for a fixed sequence of data objects. The name comes from the Latin suffix for multiples: dou*ble*, tri*ple*, quadr*uple*, quin*tuple*. Mathematicians commonly consider *ordered pairs*; for instance, most analytical geometry is done with Cartesian coordinates (*x*,*y*), an ordered pair, double, or 2-tuple.

An essential ingredient here is that a `tuple` has a fixed and known number of elements. For example a 2-dimensional geometric point might have a `tuple` with *x* and *y*. A 3-dimensional point might be a `tuple` with *x*, *y*, and *z*. The size of the `tuple` can't change without fundamentally redefining the problem we're solving.

A `tuple` is an immutable sequence of Python objects. Since it is a sequence, all of the common operations to sequences apply. Since it is immutable, it cannot be changed. Two common questions that arise are how to expand a `tuple` and how to remove objects from a `tuple`.

When someone asks about changing an element inside a `tuple`, either adding, removing or updating, we have to remind them that the `list`, covered in Chapter 14, *Lists*, is for dynamic sequences of elements. A `tuple` is generally applied when the number of elements is fixed by the nature of the problem. For example, 2-dimensional geometry, or a 4-part internet address, or a Red-Green-Blue color code. We don't change `tuples`, we create new ones.

This `tuple` processing even pervades the way functions are defined. We can have positional parameters collected into a `tuple`, something we'll cover in the section called "Advanced Parameter Handling For Functions".

Tuple Literal Values

A `tuple` is created by surrounding objects with `()`'s and separating the items with commas (`,`). An empty `tuple` is empty `()`'s. An interesting question is how Python tells an expression from a 1-`tuple`. A 1-element `tuple` has a single comma: `(1,)`. An expression lacks the comma: `(1)`. A pleasant consequence of this is that an extra comma at the end of every `tuple` is legal; for example, `(9, 10, 56, )`.

Examples:

```
xy= (2, 3)
personal= ('Hannah',14,5*12+6)
singleton= ("hello",)
```

The elements of a `tuple` do not have to be the same type. A `tuple` can be a mixture of any Python data types, including other `tuples`.

Tuple Operations

There are three standard sequence operations (`+`, `*`, `[]`) that can be performed with `tuples` as well as `lists` and `strings`.

The `+` operator creates a new `tuple` as the concatenation of the arguments. Here's an example.

```
>>> ("chapter",8) + ("strings","tuples","lists")
('chapter', 8, 'strings', 'tuples', 'lists')
```

The `*` operator between `tuples` and numbers (number `*` `tuple` or `tuple` `*` number) creates a new `tuple` that is a number of repetitions of the input `tuple`.

```
>>> 2*(3,"blind","mice")
(3, 'blind', 'mice', 3, 'blind', 'mice')
```

The `[]` operator selects an item or a slice from the `tuple`. There are two forms. The first format is ***`tuple`*** **[** ***`index`*** **]**. Items are numbered from 0 at beginning through the length. They are also number from -1 at the end backwards to `-len(`*`tuple`*`)`. The slice format is ***`tuple`*** **[** ***`start`*** **:** ***`end`*** **]**. Elements from *`start`* to *`end`*-1 are chosen; there will be *end*–*start* elements in the resulting `tuple`. If *`start`* is omitted, it is the beginning of the `tuple` (position 0), if *`end`* is omitted, it is the end of the tuple (position -1).

```
>>> t=( (2,3), (2,"hi"), (3,"mom"), 2+3j, 6.02E23 )
>>> t[2]
(3, 'mom')
>>> print t[:3], 'and', t[3:]
((2, 3), (2, 'hi'), (3, 'mom')) and ((2+3j), 6.02e+23)
>>> print t[-1], 'then', t[-3:]
6.02e+23 then ((3, 'mom'), (2+3j), 6.02e+23)
```

Tuple Comparison Operations

The standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`, **in**, **not in**) work exactly the same among `tuples` as they do among `strings`. The `tuples` are compared element by element. If the corresponding elements are the same type, ordinary comparison rules are used. If the corresponding elements are different types, the type names are compared, since there is almost no other rational basis for comparison.

```
>>> a=(1,2,3,4,5)
>>> b=(9,8,7,6,5)
>>> if a < b: print "a smaller"
>>> else: print "b smaller"
a smaller
>>> print 3 in a
True
>>> print 3 in b
False
```

Here's a longer example.

Example 13.1. redblack.py

```
#!/usr/bin/env python
import random
n= random.randrange(38)
if n == 0:
    print '0', 'green'
elif n == 37:
    print  '00', 'green'
elif n in ( 1,3,5,7,9, 12,14,16,18, 19,21,23,25,27, 30,32,34,36 ):
    print n, 'red'
else:
    print n, 'black'
```

This script will create a random number, setting aside the zero and double zero. If the number is in the `tuple` of red spaces on the roulette layout, this is printed. If none of the other rules are true, the number is in one of the black spaces.

Clearly the heart of this script is the extended if-statement which contains the tuple of red positions on the roulette wheel. This should be rewritten as a function.

Tuple Statements

The Assignment Sratement. There is a variation on the **assignment** statement called a **multiple-assignment** statement that works nicely with `tuples`. We looked at this in the section called "Multiple Assignment Statement". Multiple variables are set by decomposing the items in the `tuple`.

```
>>>
x,y=(1,2)
>>>
x
1
>>>
y
2
```

An essential ingredient here is that a `tuple` has a fixed and known number of elements. For example a 2-dimensional geometric point might have a `tuple` with x and y. A 3-dimensional point might be a `tuple` with x, y, and z.

This works well because the right side of the assignment statement is fully evaluated before the assignments are performed. This allows things like swapping two variables with `x,y=y,x`.

The for Statement. The **for** statement also works directly with sequences like `tuples`. The `range` function that we have used creates a `list` (a kind of sequence covered in the next section). A `tuple` is also a sequence and can be used in a **for** statement.

```
s= 0
for i in ( 1,3,5,7,9, 12,14,16,18, 19,21,23,25,27, 30,32,34,36 ):
    s += i
print "total",s
```

Tuple Built-in Functions

A number of built-in functions create or process `tuples`.

`len(`*`object`*`)` → integer

Return the number of items of a sequence or mapping.

`max(`*`tuple`*`)` → value

Return its largest item in the sequence.

`min(`*`tuple`*`)` → value

Return its smallest item in the sequence.

`any(`*`tuple`*`)` → boolean

Return `True` if there exists an item in the sequence which is `True`.

`all(`*`tuple`*`)` → boolean

Return `True` if all items in the sequence are `True`.

`divmod(`*`x`*`, `*`y`*`)` → (*div*, *mod*)

Return the `tuple ((x-x%y)/y, x%y)`. Invariant: `div*y + mod == x`. This is the quotient and the remainder in division.

The `divmod` functions is often combined with multiple assignment. For example:

```
>>> q,r= divmod(355,113)
>>> print q,"*113+",r,"=355"
3 *113+ 16 =22
```

Tuple Exercises

These exercises implement some basic statistical algorithms. For some background on the Sigma operator, Σ, see the section called “Digression on The Sigma Operator”.

1. **Blocks of Stock.** A block of stock as a number of attributes, including a purchase date, a purchase price, a number of shares, and a ticker symbol. We can record these pieces of information in a `tuple` for each block of stock and do a number of simple operations on the blocks.

   Let's dream that we have the following portfolio.

Purchase Date	Purchase Price	Shares	Symbol	Current Price
25 Jan 2001	43.50	25	CAT	92.45
25 Jan 2001	42.80	50	DD	51.19
25 Jan 2001	42.10	75	EK	34.87
25 Jan 2001	37.58	100	GM	37.58

   We can represent each block of stock as a 5-tuple with purchase date, purchase price, shares, ticker symbol and current price.

```
portfolio= [ ( "25-Jan-2001", 43.50, 25, 'CAT', 92.45 ),
( "25-Jan-2001", 42.80, 50, 'DD', 51.19 ),
( "25-Jan-2001", 42.10, 75, 'EK', 34.87 ),
( "25-Jan-2001", 37.58, 100, 'GM', 37.58 )
]
```

   Develop a function that examines each block, multiplies shares by purchase price and determines the total purchase price of the portfolio.

   Develop a second function that examines each block, multiplies shares by purchase price and shares by current price to determine the total amount gained or lost.

2. **Mean.** Computing the mean of a `list` of values is relatively simple. The mean is the sum of the values divided by the number of values in the `list`. Since the statistical formula is so closely related to the actual loop, we'll provide the formula, followed by an overview of the code.

   Equation 13.1. Mean

   $$\mu_x = \frac{\sum_{0 \le i < n} x_i}{n}$$

   The definition of the Σ mathematical operator leads us to the following method for computing the mean:

   Procedure 13.1. Computing Mean

   1. intialize sum, *s*, to zero
   2. for each value, *i*, in the range 0 to the number of values in the `list`, *n*:
      a. add element x_i to *s*
   3. return *s* divided by the number of elements, *n*

3. **Standard Deviation.** The standard deviation can be done a few ways, but we'll use the formula shown below. This computes a deviation measurement as the square of the difference between each sample and the mean. The sum of these measurements is then divided by the number of values times the number of degrees of freedom to get a standardized deviation measurement. Again, the formula summarizes the loop, so we'll show the formula followed by an overview of the code.

   Equation 13.2. Standard Deviation

   $$\sigma_x = \sqrt{\frac{\sum_{0 \le i < n} (x_i - \mu_x)^2}{n - 1}}$$

   The definition of the Σ mathematical operator leads us to the following method for computing the standard deviation:

   Procedure 13.2. Computing Standard Deviation

   1. compute the mean, *m*
   2. intialize sum, *s*, to zero
   3. for each value, x_i in the `list`:
      a. compute the difference from the mean, *d* as x_i - *m*
      b. add d^2 to *s*. This is typically done as *d***d* in Java, since there is no "squared" operator. In Python, this can be *d***2.
   4. compute the variance as *s* divided by (*n* - 1). This *n* - 1 value reflects the statistical notion of "degrees of freedom", which is beyond the scope of this book.
   5. return the square root of the variance.

   The `math` module contains the `math.sqrt` funtion. For some additional

information, see the section called "The `math` Module".

Digression on The Sigma Operator

For those programmers new to statistics, this section provides background on the Sigma operator, Σ.

Equation 13.3. Basic Summation

$$\sum_{i=0}^{n} f(i)$$

The Σ operator has three parts to it. Below it is a bound variable, *i* and the starting value for the range, written as *i*=0. Above it is the ending value for the range, usually something like *n*. To the right is some function to execute for each value of the bound variable. In this case, a generic function, *f*(*i*). This is read as "sum *f*(*i*) for *i* in the range 0 to *n* ".

One common definition of Σ uses a closed range, including the end values of 0 and *n*. However, since this is not a helpful definition for software, we will define Σ to use a half-open interval. It has exactly *n* elements, including 0 and *n*-1; mathematically, $0 \le i < n$.

Consequently, we prefer the following notation, but it is not often used. Most statistical and mathematical texts seem to use 1-based indexing, consequently some care is required when translating formulae to programming languages that use 0-based indexing.

Equation 13.4. Summation with Half-Open Interval

$$\sum_{0 \le i < n} f(i)$$

Our two statistical algorithms have a form more like the following. In this we are applying some function, *f*(), to each value, x_i of an array. When computing the mean, there is no function. When computing standard deviation, the function involves subtracting and multiplying.

Equation 13.5. Summing Elements of an Array, *x*

$$\sum_{0 \le i < n} f(x_i)$$

We can transform this definition directly into a for loop that sets the bound variable to all of the values in the range, and does some processing on each value of the `list` of integers. This is the Python implemention of Sigma. This computes two values, the sum, `sum`, and the number of elements, `n`.

Example 13.2. Python Sigma Iteration

```
sum= 0
for i in range(len(theList)): ❶
    xi= theList[i] ❷
    # fxi = processing of xi  ❸
    sum += fxi
n= len(theList)
```

❶ Get the length of `theList`. Execute the body of the loop for all values of `i` in the range 0 to the number of elements-1.

❷ Fetch item `i` from `theList` and assign it to `xi`.

❸ For simple mean calculation, this statement does nothing. For standard deviation, however, this statement computes the measure of deviation from the average.

More advanced Python programmers may be aware that there are several ways to shorten this loop down to a single expression using the `reduce` function as well as list comprehensions.

An Optimization. In the usual mathematical notation, an integer index, *i* is used. In Python it isn't necessary to use the formal integer index. Instead, an iterator can be used to visit each element of the `list`, without actually using an explicit numeric counter. The processing simplifies to the following.

Example 13.3. Python Sample Values by Iterator

```
for xi in theList: ❶
    # fxi = processing of xi ❷
    sum += fxi
n= len(theList)
```

❶ Execute the loop assigning each item of of `theList` to `xi`.

❷ For simple mean calculation, this statement does nothing. For standard deviation, however, this statement computes the measure of deviation from the average.

Chapter 14. Lists

Table of Contents

We'll look at `lists` from a number of viewpoints: semantics, literal values, operations, comparison operators, statements, built-in functions and methods. Additionally, we have a digression on the immutability of `strings`.

List Semantics

A `list` is a container for variable length sequence of Python objects. A `list` is mutable, which means that items within the `list` can be changed. Also, items can be added to the `list` or removed from the `list`.

A `list` is an mutable sequence of Python objects. Since it is a sequence, all of the common operations to sequences apply. Since it is mutable, it can be changed: objects can be added to, removed from and replaced in a `list`.

Sometimes we'll see a `list` with a fixed number of elements, like a two-dimensional point with two elements, *x* and *y*. When someone asks about a fixed-length `list`, we have to remind them that the `tuple`, covered in Chapter 14, *Lists*, is for static sequences of elements.

A great deal of Python's internals a `list`-based. The **for** statement, in particular, expects a sequence, and we often create a `list` by using the `range` function. When we split a `string` using the `split` method, we get a `list` of substrings.

List Literal Values

A `list` is created by surrounding them with `[]`'s and separating the values with commas (`,`). A `list` can be created, expanded and reduced. An empty `list` is simply `[]`. As

with `tuples`, an extra comma at the end of the `list` is graciously ignored.

Examples:

```
myList = [ 2, 3, 4, 9, 10, 11, 12 ]
history = [ ]
```

The elements of a `list` do not have to be the same type. A `list` can be a mixture of any Python data types, including `lists`, `tuples`, `strings` and numeric types.

A `list` permits a sophisticated kind of display called a *comprehension*. We'll revisit this in some depth in the section called "List Comprehensions". As a teaser, consider the following:

```
>>> print [ 2*i+1 for i in range(6) ]
[1, 3, 5, 7, 9, 11]
```

This statement creates a `list` using a list comprehension. A comprehension starts with a candidate `list` (`range(`*6*`)`, in this example) and derives the `list` values from the candidate `list` (using `2*i+1` in this example). A great deal of power is available in comprehensions, but we'll save the details for a later section.

List Operations

The three standard sequence operations (`+`, `*`, `[]`) can be performed with `lists`, as well as `tuples` and `strings`.

The `+` operator creates a new `list` as the concatenation of the arguments. The

```
>>> ["field"] + [2, 3, 4] + [9, 10, 11, 12]
['field', 2, 3, 4, 9, 10, 11, 12]
```

The `*` operator between `lists` and numbers (number `*` `list` or `list` `*` number) creates a new `list` that is a number of repetitions of the input `list`.

```
>>> 2*["pass","don't","pass"]
['pass', "don't", 'pass', 'pass', "don't", 'pass']
```

The `[]` operator selects an character or a slice from the `list`. There are two forms. The first format is ***list*** **[** ***index*** **]**. Elements are numbered from 0 at beginning through the length. They are also number from -1 at the end backwards to `-len(`*list*`)`. The slice format is ***list*** **[** ***start*** **:** ***end*** **]**. Items from *start* to *end*-1 are chosen; there will be *end*−*start* elements in the resulting `list`. If *start* is omitted it is the beginning of the `list` (position 0), if *end* is omitted it is the end of the `list`.

In the following example, we've constructed a `list` where each element is a `tuple`. Each `tuple` could be a pair of dice.

```
>>> l=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> l[2]
(2, 2)
>>> print l[:3], 'split', l[3:]
[(6, 2), (5, 4), (2, 2)] split [(1, 3), (6, 5), (1, 4)]
>>> l[-1]
(1, 4)
>>> l[-3:]
[(1, 3), (6, 5), (1, 4)]
```

List Comparison Operations

The standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`, **in**, **not in**) work exactly the same among

lists, tuples and strings. The lists are compared element by element. If the corresponding elements are the same type, ordinary comparison rules are used. If the corresponding elements are different types, the type names are compared, since there is no other rational basis for comparison.

```
d1= random.randrange(6)+1
d2= random.randrange(6)+1
if d1+d2 in [2, 12] + [3, 4, 9, 10, 11]:
    print "field bet wins on ", d1+d2
else:
    print "field bet loses on ", d1+d2
```

This will create two random numbers, simulating a roll of dice. If the number is in the list of field bets, this is printed. Note that we assemble the final list of field bets from two other lists. In a larger application program, we might separate the different lists of winners based on different payout odds.

List Statements

The Assignment Statement. The variation on the **assignment** statement called **multiple-assignment** statement also works with lists. We looked at this in the section called "Multiple Assignment Statement". Multiple variables are set by decomposing the items in the list.

```
>>> x,y=[1,"hi"]
>>> x
1
>>> y
"hi"
```

This will only work of the list has a fixed and known number of elements. This is more typical when working with tuples, which are immutable, rather than lists, which can vary in length.

The for Statement. The **for** statement also works directly with sequences like list. The range function that we have used creates a list. We can also create lists other ways. We'll touch on various list construction techniques at several points in the text.

```
s= 0
for i in [2,3,5,7,11,13,17,19]:
    s += i
print "total",s
```

The del Statement. The **del** statement removes items from a list. For example

```
>>> i = range(10)
>>> del i[0],i[2],i[4],i[6]
>>> i
[1, 2, 4, 5, 7, 8]
```

This example reveals how the **del** statement works.

The i variable starts as the list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

Remove i[0] and the variable is [1, 2, 3, 4, 5, 6, 7, 8, 9].

Remove i[2] (the value 3) from this new list, and get [1, 2, 4, 5, 6, 7, 8, 9].

Remove i[4] (the value 6) from this new list and get [1, 2, 4, 5, 7, 8, 9].

Finally, remove i[6] and get [1, 2, 4, 5, 7, 8].

List Built-in Functions

A number of built-in functions create or deal with `lists`. The following functions apply to all sequences, including `tuples` and `strings`.

`len(`*`object`*`)` → integer

Return the number of items of a sequence or mapping.

`max(`*`list`*`)` → value

Return the largest item in the sequence.

`min(`*`list`*`)` → value

Return the smallest item in the sequence.

`any(`*`tuple`*`)` → boolean

Return `True` if there exists an item in the sequence which is `True`.

`all(`*`tuple`*`)` → boolean

Return `True` if all items in the sequence are `True`.

`range(`*`[start]`*`, `*`stop`*`, `*`[step]`*`)` → list of int

The arguments must be plain integers. If the *`step`* argument is omitted, it defaults to 1. If the *`start`* argument is omitted, it defaults to 0. The full form returns a `list` of plain integers [*start*, *start* + *step*, *start* + 2 * *step*, ...]. If *`step`* is positive, the last element is the largest *start* + *i* * *step* less than *`stop`*; if *`step`* is negative, the last element is the largest *start* + *i* * *step* greater than *`stop`*. *`step`* must not be zero (or else `ValueError` is raised).

List Methods

A `list` object has a number of member methods. These can be grouped arbitrarily into transformations, which change the `list`, and information, which returns a fact about a `list`. In all of the following method functions, we'll assume a `list` object named *`l`*.

The following `list` transformation functions update a `list` object. In the case of the `pop` method, it both returns information as well as updates the `list`.

`l``.append(`*`object`*`)`

Update `list` *`l`* by appending *`object`* to end of the `list`.

`l``.extend(`*`list`*`)`

Extend `list` *`l`* by appending *`list`* elements. Note the difference from `append(`*`object`*`)`, which treats the argument as a single `list` object.

`l``.insert (`*`index`*`, `*`object`*`)`

Update `list` *`l`* by inserting *`object`* before position *`index`*. If index is greater than `len(`*`list`*`)`, the object is simply appended. If index is less than zero, the object is prepended.

`l``.pop (`*`[index]`*`)` → item

Remove and return item at *`index`* (default last, -1) in `list` *`l`*. An exception is

raised if the `list` is already empty.

l.remove (*value*) → item

Remove first occurrence of *value* from `list` *l*. An exception is raised if the value is not in the `list`.

l.reverse

Reverse the items of the `list` *l*. This is done "in place", it does not create a new `list`.

l.sort(*[cmpfunc]*)

Sort the items of the `list` *l*. This is done "in place", it does not create a new `list`. If a comparison function, *compare* is given, it must behave like the built-in `cmp`: `cmpfunc`(*x*, *y*) → -1, 0, 1.

The following accessor methods provide information about a `list`.

l.count(*value*) → integer

Return number of occurrences of *value* in `list` *l*.

l.index(*value*) → integer

Return index of first occurrence of *value* in `list` *l*.

Stacks and Queues. The `append` and `pop` functions can be used to create a standard push-down *stack*, or last-in-first-out (LIFO) `list`. The `append` function places an item at the end of the `list` (or top of the stack), where the `pop` function can remove it and return it.

```
>>> stack= []
>>> stack.append(1)
>>> stack.append("word")
>>> stack.append( ("a","2-tuple") )
>>> stack.pop()
('a', '2-tuple')
>>> stack.pop()
'word'
>>> stack.pop()
1
```

The `append` and `pop`(*0*) functions can be used to create a standard *queue*, or first-in-first-out (FIFO) `list`. The `append` function places an item at the end of the queue. A call to `pop`(*0*) removes the first item from the queue it and returns it.

```
>>> queue=[]
>>> queue.append(1)
>>> queue.append("word")
>>> queue.append( ("a","2-tuple") )
>>> queue.pop(0)
1
>>> queue.pop(0)
'word'
>>> queue.pop(0)
('a', '2-tuple')
```

List Exercises

1. **Accumulating Unique Values.** This uses the Bounded Linear Search algorithm to

locate duplicate values in a sequence. This is a powerful technique to eliminate sorting from a wide variety of summary-type reports. Failure to use this algorithm leads to excessive processing in many types of applications.

Procedure 14.1. Unique Values of a Sequence, *seq*

1. **Initalize**

   set `a` ← an empty sequence.

2. **Loop.** For each value, `v`, in *seq*.

   We'll use the Bounded Linear Search for `v` in `a`.

   a. **Initialize**

      `i` ← 0.

      Append `v` to the `list` `a`.

   b. **Search.** while `a[i]` ≠ `v`, increment `i`.

      At this point `a[i]` = `v`. The question is whether `i` = `len(`*a*`)` or not.

   c. **New Value?** if `i` = `len(`*a*`)`, then `v` is unique.

   d. **Existing Value?** if `i` ≠ `len(`*a*`)`, then `v` is a duplicate of `a[i]`; `a[-1]` can be removed.

3. **Result.** Return array `a`, which has unique values from *seq*.

2. **Binary Search.** This is not as universally useful as the Bounded Linear Search (above) because it requires the data be sorted.

   Procedure 14.2. Binary Search a sorted Sequence, *seq*, for a target value, *tgt*

   1. **Initialize**

      `l`←0.

      `h`←`len(`*seq*`)`.

      `m`←`(l+h)÷2`.

   2. **Loop.** While `l+1<h` and *seq*`[m]` ≠ *tgt*

      a. If *tgt* < *seq*`[m]`, then `h`←`m`

      b. If *tgt* > *seq*`[m]`, then `l`←`m`

      c. `m`←`(l+h)÷2`.

   3. If *tgt* = *seq*`[m]`, then return `m`

      If *tgt* ≠ *seq*`[m]`, then return `-1` as a code for "not found".

3. **Quicksort.** The super-fast sort routine

   As a series of loops it is rather complex. As a recursion it is quite short. This is the same basic algorithm in the C libraries.

   Quicksort proceeds by partitioning the `list` into two regions: one has all of the

high values, the other has all the low values. Each of these regions is then individually sorted into order using the quicksort algorithm. This means the each region will be subdivided and sorted.

For now, we'll sort an array of simple numbers. Later, we can generalize this to sort generic objects.

Procedure 14.3. Quicksort a List, *a* between elements *lo* and *hi*

1. **Partition**

   Initialize. `middle` ← (`hi`+`lo`)÷2.

   `ls` ← `lo`

   `hs` ← `hi`

   Swap To Partition. while `ls` < `hs`

   a. If *a*[ls].key ≤ *a*[middle].key, increment ls

   b. If *a*[ls].key > *a*[middle].key, swap ls with middle

   c. If *a*[hs].key ≥ *a*[middle].key, decrement hs

   d. If *a*[hs].key < *a*[middle].key, swap hs with middle

2. **Quicksort Each Partition**

   QuickSort(*a*, lo, middle)

   QuickSort(*a*, middle+1, hi)

4. **Recursive Search.** This is also a binary search: it works using a design called "divide and conquer". Rather than search the whole `list`, we divide it in half and search just half the `list`. This version, however is defined with a recusive function instead of a loop. This can often be faster than the looping version shown in exercise 2.

   Procedure 14.4. Recursive Search a List, *seq* for a target, *tgt*, in the region between elements *lo* and *hi*

   1. **Empty Region?** If *lo*+1 = *hi*, then return -1 as a code for "not found".
   2. **Middle Element.** `m` ← (*lo*+*hi*)÷2
   3. **Found?** If *seq*[m] = *tgt*, then return `m`
   4. **Lower Half?** If *seq*[m] < *tgt*, then return `recursiveSearch` (*seq*, *tgt*, *lo*, *m*)
   5. **Upper Half?** If *seq*[m] > *tgt*, then return `recursiveSearch` (*seq*, *tgt*, *m*, *hi*)

5. **Sieve of Eratosthenes.** This is an algorithm which locates prime numbers. A prime number can only be divided evenly by 1 and itself. We locate primes by making a table of all numbers, and then crossing out the numbers which are multiples of other numbers. What is left must be prime.

   Procedure 14.5. Sieve of Eratosthenes - List Version

   1. **Initialize**

Create a `list`, `prime` of 5000 booleans, all `True`, initially.

`p` ← 2

2. **Generate Primes.** while 2 ≤ `p` < 5000

   a. **Find Next Prime.** while not `prime[p]` and 2 ≤ `p` < 5000: increment `p`.

      At this point, `p` is prime.

   b. **Remove Multiples**

      `k` ← `p` + `p`

      while `k` < 5000

      i. `prime[k]` ← false

      ii. `k` ← `k` + `p`

   c. **Next `p`.** increment `p`

3. **Report**

   At this point, for all `p` if `prime[ p ]` is true, `p` is prime.

   while 2 ≤ `p` < 5000

   a. if`prime[ p ]`, print `p`

6. **Polynomial Arithmetic.** We can represent numbers as polynomials. We can represent polynomials as arrays of their coefficients. This is covered in detail in [Knuth73], section 2.2.4 algorithms A and M.

   Example: $4x^3 + 3x + 1$ has the following coefficients: (4, 0, 3, 1).

   $2x^2 - 3x - 4$ is represented as (2, -3, -4).

   The sum is $4x^3 + 2x^2 - 3$, (4, 2, 0, -3).

   The product is $8x^5 - 12x^4 - 10x^3 - 7x^2 - 15x - 4$, (8, -12, -10, -7, -15, -4).

   You can apply this to large decimal numbers. In this case, `x` is assumed to be a power of 10, and the coefficients must all be between 0 and `x`-1. For example, 1987 is $1x^3 + 9x^2 + 8x + 7$, when x = 10.

   Procedure 14.6. Add Polynomials, *`p`*, *`q`*

   1. **Result Size.** `rSize` ← the larger of `len(`*`p`*`)` and `len(`*`q`*`)`.

   2. **Pad P?** If `len(`*`p`*`)` < `rSize`, create `p1` from zeroes on the left + *`p`*.

      Else, `p1` ← *`p`*.

   3. **Pad Q?** If `len(`*`q`*`)` < `rSize`, create `q1` from zeroes on the left + *`q`*.

      Else, `q1` ← *`q`*.

   4. **Add.** Add matching elements from `p1` and `q1` to create result, `r`

   5. **Result.** Return `r` as the sum of *`p`* and *`q`*.

Procedure 14.7. Multiply Polynomials, *x*, *y*

1. **Result Size.** `rSize` ← the sum of `len(`*x*`)` + `len(`*y*`)`.

   Initialize the result array, `r`, to all zeroes, with a size of `rSize`.

2. **For all elements of x.** while 0 ≤ `i` < `len(`*x*`)` loop

   a. **For all elements of y.** do while 0 ≤ `j` < `len(`*y*`)` loop

      i. add *x*`[i]` * *y*`[j]` to `r[i+j]`

3. **Result.** Return `r` as the product of *x* and *y*.

7. **Random Number Evaluation.** Before using a new random number generator, it is wise to evaluate the degree of randomness in the numbers produced. A variety of clever algorithms look for certain types of expected distributions of numbers, pairs, triples, etc. This is one of many random number tests.

   Use `random.random` to generate an array of random samples. These numbers will be uniform over the interval 0..1

   Procedure 14.8. Distribution test of a sequence of random samples, *u*

   1. **Initialize**

      Initialize `count` to a `list` of 10 zeroes.

   2. **Examine Samples.** for each sample value, `v` in *u*

      a. **Coerce Into Range.** Multiply `v` by 10 and truncate to get a value `x` in 0..9 range

      b. **Count.** increment `count[x]`

   3. **Report.** Display counts, % of total, deviation from expected count of 1/10th of available samples

8. **Dutch National Flag.** A challenging problem, one of the hardest in this set. This is from Edsger Dijkstra's book, *A Discipline of Programming* [Dijkstra76].

   Imagine a board with a row of holes filled with red, white, and blue pegs. Develop an algorithm which will swap pegs to make three bands of red, white, and blue (like the Dutch flag). You must also satisfy this additional constraint: each peg must be examined exactly once.

   Without the additional constraint, this is a relatively simple sorting problem. The additional constraint requires that instead of a simple sort which passes over the data several times, we need a more clever sort.

   Hint: You will need four partitions in the array. Initially, every peg is in the "Unknown" partition. The other three partitions ("Red", "White" and "Blue") are empty. As the algorithm proceeds, pegs are swapped into the Red, White or Blue partition from the Unknown partition. When you are done, the unknown partition is reduced to zero elements, and the other three partitions have known numbers of elements.

Chapter 15. Mappings and Dictionaries

Table of Contents

Many algorithms need to map a key to a data value. This kind of mapping is supported by the Python dictionary, `dict`. We'll look at dictionaries from a number of viewpoints: semantics, literal values, operations, comparison operators, statements, built-in functions and methods.

We are then in a position to look at two applications of the dictionary. We'll look at how Python uses dictionaries along with sequences to handle arbitrary `lists` of parameters to functions in the section called "Advanced Parameter Handling For Functions". This is a very sophisticated set of tools that let us define functions that are very flexible and easy to use.

Dictionary Semantics

A dictionary, called a `dict`, maps a *key* to a *value*. The key can be any type of Python object that computes a consistent hash value. The value referenced by the key can be any type of Python object.

There is a subtle terminology issue here. Python has provisions for creating a variety of different types of mappings. Only one type of mapping comes built-in; that type is the `dict`. The terms are almost interchangeable. However, you may develop or download other types of mappings, so we'll be careful to focus on the `dict` class.

Working with a `dict` is similar to working with a sequence. Items are inserted into the `dict`, found in the `dict` and removed from the `dict`. A `dict` object has member methods that return a sequence of keys, or values, or `(`*`key`*`,`*`value`*`)` `tuples` suitable for use in a **for** statement.

Unlike a sequence, a `dict` does not preserve order. Instead of order, a `dict` uses a *hashing* algorithm to identify a place in the `dict` with a rapid calculation of the key's hash value. The built-in function, `hash` is used to do this calculation. Items in the `dict` are inserted in an position related to their key's apparently random hash values.

Later in this chapter we'll see how Python uses `tuples` and dictionaries to handle variable numbers of arguments to functions.

Some Alternate Terminology. A `dict` can be thought of as a container of **(*`key`*:*`value`*)** pairs. This can be a helpful way to imagine the information in a mapping. Each pair in the list is the mapping from a key to an associated value.

A `dict` can be called an associative array. Ordinary arrays, typified by sequences, use a numeric index, but a `dict`'s index is made up of the key objects. Each key is associated with (or "mapped to") the appropriate value.

Some Consequences. Each key object has a hash value, which is used to place tje key and the value in the `dict`. Consequently, the keys must have consistent hash values; they must be immutable objects. You can't use `list` or `dict` objects as keys. You can

use tuple, string and frozenset objects, since they are immutable. Additionally, when we get to class definitions (in Chapter 21, *Classes*), we can make arrangements for our objects to return an immutable hash value.

A common programming need is a heterogeneous container of data. Database records are an example. A record in a database might have a boat's name (as a string), the length overall (as a number) and an inventory of sails (a list of strings). Often a record like this will have each element (known as a *field*) identified by name. A C or C++ struct accomplishes this. This kind of named collection of data elements may be better handled with a class (someting we'll cover in Chapter 21, *Classes*). However, a mapping is also useful for managing this kind of heterogeneous data with named fields.

Note that many languages make record definitions a statically-defined container of named fields. A Python dict is dynamic, allowing us to add field names at run-time.

A common alternative to hashing is using some kind of ordered structure to maintain the keys. This might be a tree or list, which would lead to other kinds of mappings. For example, there is an ordered dictionary, odict, that can be downloaded from http://www.voidspace.org.uk/python/odict.html.

Dictionary Literal Values

A dict literal is created by surrounding a key-value list with {}'s; a keys is separated from its value with :'s, and the *key*:*value* pairs are separated with commas (,). An empty dict is simply {}. As with lists and tuples, an extra , inside the {}'s is tolerated.

Examples:

```
diceRoll = { (1,1): "snake eyes", (6,6): "box cars" }
myBoat = { "NAME":"KaDiMa", "LOA":18,
 "SAILS":["main","jib","spinnaker"] }
theBets = { }
```

The diceRoll variable is a dict with two elements. One element has a key of a tuple (1,1) and a value of a string, "snake eyes". The other element has a key of a tuple (6,6) and a value of a string "box cars".

The myBoat variable is a dict with three elements. One element has a key of the string "NAME" and a value of the string "KaDiMa". Another element has a key of the string "LOA" and a value of the integer 18. The third element has a key of the string "SAILS" and the value of a list ["main", "jib", "spinnaker"].

The theBets is an empty dict.

The values and keys in a dict do not have to be the same type. Keys must be a type that can produce a hash value. Since lists and dict objects are mutable, they are not permitted as keys. All other non-mutable types (especially strings, frozensets and tuples) are legal keys.

Dictionary Operations

A dict only permits a single operation: []. This is used to add, change or retrieve items from the dict. The slicing operations that apply to sequences don't apply to a dict.

Examples of dict operations.

```
>>> d= { }
>>> d[2]= [ (1,1) ]
>>> d[3]= [ (1,2), (2,1) ]
>>> d
{2: [(1, 1)], 3: [(1, 2), (2, 1)]}
```

```
>>> d[2]
[(1, 1)]
>>> d["2 or 3"]= d[2] + d[3]
>>> d["2 or 3"]
[(1, 1), (1, 2), (2, 1)]
```

This example starts by creating an empty `dict`, `d`. Into `d[2]` it inserts a `list` with a single `tuple`. Into `d[3]` it inserts a `list` with two `tuples`. When the entire `dict` is printed it shows the two key:value pairs, one with a key of 2 and another with a key of 3.

Then it creates an entry with a key of the `string "2 or 3"`. The value for this entry is computed from the values of `d[2] + d[3]`. Since these two entries are `lists`, the `lists` can be combined with the + operator. The resulting expression is stored into the `dict`.

When we print the final `dict`, we see that there are three key:value pairs: one with a key of 3, one with a key of 2 and one with a key of `"2 or 3"`.

```
{3: [(1, 2), (2, 1)], 2: [(1, 1)],
'2 or 3': [(1, 1), (1, 2), (2, 1)]}
```

This ability to use any object as a key is a powerful feature, and can eliminate some needlessly complex programming that might be done in other languages.

Here are some other examples of picking elements out of a `dict`.

```
>>> myBoat = { "NAME":"KaDiMa",
...     "LOA":18, "SAILS":["main","jib","spinnaker"] }
...

>>> myBoat["NAME"]
KaDiMa
>>> for s in myBoat["SAILS"]:
...     print s
...

main
jib
spinnaker
```

String Formatting with Dictionaries. The `string` formatting operator, **%**, can be applied to a `dict` as well as a sequence. When this operator was introduced in Chapter 12, *Strings*, the format specifications were applied to a `tuple` or other sequence. When used with a `dict`, each format specification is given an additional option that specifies which `dict` element to use. The general format for each conversion specification is:

%(element)[flags][width[.precision]]code

The *`flags`*, *`width`*, *`precision`* and *`code`* elements are defined in Chapter 12, *Strings*. The *`element`* field must be enclosed in ()'s; this is the key to be selected from the `dict`.

For example:

```
print "%(NAME)s, %(LOA)d" % myBoat
```

This will find `myBoat[NAME]` and use `%s` formatting; it will find `myBoat[LOA]` and use `%d` number formatting. We'll return to another application of this when we talk about classes in Chapter 21, *Classes*.

Dictionary Comparison Operations

Some of the standard comparisons (<, <=, >, >=, ==, !=) don't have a lot of meaning between two dictionaries. Since there may be no common keys, nor even a common data type for keys, dictionaries are simply compared by length. The `dict` with fewer elements is considered less than a `dict` with more elements.

The membership comparisons (**in**, **not in**) don't apply to dictionaries at all. It would be unclear whether the key or the value is referenced. Dictionaries have more sophisticated membership tests; we'll cover these under member methods, below.

Dictionary Statements

The for Statement. We can't meaningfully iterate through the elements in a `dict`, since there's no implicit order. Worse, it isn't obvious that we want to iterate through the keys or through the values in the `dict`.

Consequently, we have three different ways to visit the elements in a `dict`, all based on three `dict` method functions. Here are the choices:

- The key:value pairs. We can use the `items` method to iterate through the sequence of 2-tuples that contain each key and the associated value.
- The keys. We can use the `keys` method to iterate through the sequence of keys.
- The values. We can use the `values` method to iterate throught he sequence of values in each key:value pair.

Here's an example of using the key:value pairs. This relies on the `tuple`-based **for** statement that we looked at in the section called “Tuple Statements”. We'll iterate through the `dict`, update it, and iterate through it a second time. In this case, coincidentally, the new key-value pair wound up being shown at the end of the `dict`.

```
>>> myBoat = { "NAME":"KaDiMa", "LOA":18,
 "SAILS":["main","jib","spinnaker"] }
>>> for key,value in myBoat.items():
...     print key, " = ", value


LOA  =  18
NAME  =  KaDiMa
SAILS  =  ['main', 'jib', 'spinnaker']
>>> myBoat['YEAR']=1972
>>> for key,value in myBoat.items():
...     print key, " = ", value


LOA  =  18
NAME  =  KaDiMa
SAILS  =  ['main', 'jib', 'spinnaker']
YEAR  =  1972
```

The del Statement. The **del** statement removes items from a `dict`. For example

```
>>> i = { "two":2, "three":3, "quatro":4 }
>>> del i["quatro"]
>>> print i
{'three': 3, 'two': 2}
```

In this example, we use the key to remove the item from the `dict`.

The member function, `pop`, does this also.

Dictionary Built-in Functions

Here are the built-in functions that deal with dictionaries.

`len` (*`object`*) → integer

> Return the number of items of a sequence or mapping.

`dict` (*`valList`*) → dictionary

> Each element of the `list` must be a 2-element `tuple`; create a `dict` with the first element as the key and the second element as the value. Note that the `zip` function produces a `list` of 2-`tuples` from two parallel `lists`.

```
>>> dict( [('first',0), ('second',1), ('third',2)] )
{'second': 1, 'third': 2, 'first': 0}
>>> dict( zip(['fourth','fifth','sixth'],[3,4,5]) )
{'sixth': 5, 'fifth': 4, 'fourth': 3}
```

Dictionary Methods

A `dict` object has a number of member methods. Many of these maintain the values in a `dict`. Others retrieve parts of the `dict` as a sequence, for use in a **for** statement. In the following definitions, *`d`* is a `dict` object.

The following transformation functions update a `dict` object.

`d``.clear`

> Remove all items from the `dict`.

`d``.copy` → dictionary

> Copy the `dict` to make a new `dict`. This is a *shallow copy*. All objects in the new `dict` are references to the same objects as the original `dict`.

`d``.setdefault(`*`key`*`, [`*`default`*`])` → object

> Similar to *`d`*`.get(`*`key`*`)` and *`d`*`[`*`key`*`]` - get the item with the given key. However, this sets a default value to the supplied object.

`d``.update(`*`new`*`, [`*`default`*`])` → object

> Merge values from the new `dict` into the original `dict`, adding or replacing as needed. It is equivalent to the following Python statement. `for k in new.keys(): d[k]= new[k]`

`d``.pop(`*`key`*`, [`*`value`*`])` → object

> Remove the given key from the `dict`, returning the associated value. If the key does not exist, return the optional value provided in the `pop` call.

The following accessor methods provide information about a `dict`.

`d``.get(`*`key`*`, [`*`default`*`])` → object

> Get the item with the given *`key`*, similar to *`d`*`[`*`key`*`]`. If the key is not present, supply *`default`* instead.

`d``.has_key(`*`key`*`)` → boolean

> If there is an entry in the `dict` with the given *`key`*, return `True`, otherwise return `False`.

d.items → sequence

Return all of the items in the dict as a sequence of (key,value) tuples. Note that these are returned in no particular order.

d.keys → sequence

Return all of the keys in the dict as a sequence of keys. Note that these are returned in no particular order.

d.values → sequence

Return all the values from the dict as a sequence. Note that these are returned in no particular order.

Dictionary Exercises

1. **Word Frequencies.** Update the exercise in Accumulating Unique Values to count each occurance of the values in aSequence. Change the result from a simple sequence to a dict. The dict key is the value from aSequence. The dict value is the count of the number of occurances.

   If this is done correctly, the input sequence can be words, numbers or any other immutable Python object, suitable for a dict key.

   For example, the program could accept a line of input, discarding punctuation and breaking them into words in space boundaries. The basic string operations should make it possible to create a simple sequence of words.

   Iterate through this sequence, placing the words into a dict. The first time a word is seen, the frequency is 1. Each time the word is seen again, increment the frequency. Produce a frequency table.

   To alphabetize the frequency table, extract just the keys. A sequence can be sorted (see section 6.2). This sorted sequence of keys can be used to extract the counts from the dict.

2. **Stock Reports.** A block of publicly traded stock has a variety of attributes, we'll look at a few of them. A stock has a ticker symbol and a company name. Create a simple dict with ticker symbols and company names.

   For example:

   ```
   stockDict = { 'GM': 'General Motors',
    'CAT':'Caterpillar', 'EK':"Eastman Kodak" }
   ```

   Create a simple list of blocks of stock. These could be tuples with ticker symbols, prices, dates and number of shares. For example:

   ```
   purchases = [ ( 'GE', 100, '10-sep-2001', 48 ),
    ( 'CAT', 100, '1-apr-1999', 24 ),
    ( 'GE', 200, '1-jul-1998', 56 ) ]
   ```

   Create a purchase history report that computes the full purchase price (shares times dollars) for each block of stock and uses the stockDict to look up the full company name. This is the basic relational database join algorithm between two tables.

   Create a second purchase summary that which accumulates total investment by ticker symbol. In the above sample data, there are two blocks of GE. These can easily be combined by creating a dict where the key is the ticker and the value is

the `list` of blocks purchased. The program makes one pass through the data to create the `dict`. A pass through the `dict` can then create a report showing each ticker symbol and all blocks of stock.

3. **Date Decoder.** A date of the form 8-MAR-85 includes the name of the month, which must be translated to a number. Create a `dict` suitable for decoding month names to numbers. Create a function which uses `string` operations to split the date into 3 items using the "-" character. Translate the month, correct the year to include all of the digits.

   The function will accept a date in the "dd-MMM-yy" format and respond with a `tuple` of (*y*, *m*, *d*).

4. **Dice Odds.** There are 36 possible combinations of two dice. A sim→ple pair of loops over range(6)+1 will enumerate all combinations. The sum of the two dice is more interesting than the actual combination. Create a `dict` of all combinations, using the sum of the two dice as the key.

   Each value in the `dict` should be a `list` of `tuples`; each `tuple` has the value of two dice. The general outline is something like the following:

```
d= {}
Loop with d1 from 1 to 6
    Loop with d2 from 1 to 6
        newTuple ← ( d1, d2 ) # create the tuple
        oldList ← dictionary entry for sum d1+d2
        newList ← oldList + newTuple
        replace entry in dictionary with newList

Loop over all values in the dictionary
    print the key and the length of the list
```

Advanced Parameter Handling For Functions

In the section called “More Features” we hinted that Python functions can handle a variable number of argument values in addition to supporting optional argument values. Earlier, when we defined a function that had optional parameters, it had a definite number of parameters, but some (or all) could be omitted because we provided default values for them.

If we provide too many positional parameters to a function, Python raises an exception. Consider the following example. We defined a function of three positional parameters, and then evaluated it with more than three argument values.

```
>>> def avg(a,b,c):
        return (a+b+c)/3.0

>>> avg(10,11,12)
11.0
>>> avg(10,11,12,13,14,15,16)

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    avg(10,11,12,13,14,15,16)
TypeError: avg() takes exactly 3 arguments (7 given)
```

Unlimited Number of Positional Argument Values

Python lets us define a function that handles an unknown and unlimited number of argument values. Examples of built-in functions with a unlimited number of argument values are `max` and `min`.

Rather than have Python raise an exception, we can request the additional positional

argument values be collected into a `tuple`. To do this, you provide a final parameter definition of the form ***extras** . The * indicates that this parameter variable is the place to capture the extra argument values. The variable, here called *extras*, will receive a sequence with all of the extra positional argument values.

You can only provide one such variable (if you provided two, how could Python decide which of these two got the extra argument values?) You must provide this variable after the ordinary positional parameters in the function definition.

Examples of Unlimited Positional Arguments. The following function accepts an unlimited number of positional arguments; it collects these in a single `tuple` parameter, `args`.

```
def myMax( *args ):
    max= arg[0]
    for a in args[1:]:
        if a > max: max= a
    return max
```

We take the first element of the `args` `tuple` as our current guess at the maximum value, `max`. We use a **for** loop that will set the variable `a` to each of the other arguments. If `a` is larger than our current guess, we update the current guess, `max`. At the end of the loop, the post condition is that we have visited every element in the `list` `args`; the current guess must be the largest value.

Here's another example. In this case we have a fixed parameter in the first position and all the extra parameters collected into a `tuple` called `vals`.

```
def printf( format, *vals ):
    print format % vals
```

This should look familiar to C programmers. Now we can write the following, which may help ease the transition from C to Python.

```
printf( "%s = %d", "some string", 2 )
printf( "%s, %s, %d %d", "thing1", "thing2", 3, 22 )
```

Unlimited Number of Keyword Argument Values

In addition to collecting extra positional argument values into a single parameter, Python can also collect extra keyword argument values into a `dict`. If you want a container of keyword arguments, you provide a parameter of the form ****extras** . Your variable, here called *extras*, will receive a `dict` with all of the keyword parameters.

The following function accepts any number opf keyword arguments; they are collected into a single parameter.

```
def rtd( **args ):
    if args.has_key( "rate" ) and args.has_key( "time" ):
        args["distance"]= args["rate"]*args["time"]
    elif args.has_key( "rate" ) and args.has_key( "distance" ):
        args["time"]= args["distance"]/args["rate"]
    elif args.has_key( "time" ) and args.has_key( "distance" ):
        args["rate"]= args["distance"]/args["time"]
    else:
        raise Exception("%r does not compute" % ( args, ) )
    return args
```

Here's two examples of using this `rtd` function.

```
>>>
print rtd( rate=60, time=.75 )
```

```
{'distance': 45.0, 'rate': 60.0, 'time': 0.75}
>>>
print rtd( distance=173, time=2+50/60.0 )
{'distance': 173, 'rate': 61.058823529411761,
 'time': 2.8333333333333335}
```

The keyword arguments are collected into a `dict`, named `args`. We check for some combination of "rate", "time" and "distance" by using the `dict` method `has_key`. If the `dict` of keyword arguments, `args`, has the given keyword, `has_key` returns true. For each combination, we can solve for the remaining value and update the `dict` by insert the additional key and value into the `dict`.

Using a Container Instead of Individual Arguments

We can also force a sequence to be broken down into individual parameters. We can use a special version of the * operator when evaluating a function. Here's an example of forcing a 3-`tuple` to be assigned to three positional parameters.

```
>>> def avg3(a,b,c):
...     return (a+b+c)/3.0

>>> avg3( *(4,5,7) )
5.333333333333333
```

In this example, we told Python to break down our 3-`tuple`, and assign each value of the `tuple` to a separate parameter variable.

As with the * operator, we can use ** to make a `dict` become a series of keyword parameters to a function.

```
>>> d={ 'a':5, 'b':6, 'c':9 }
>>> avg3( **d )
6.666666666666667
```

In this example, we told Python to assign each element of the `dict`, `d`, to a parameter of our function.

We can mix and match this with ordinary parameter assignment, also. Here's an example.

```
>>> avg3( 2, b=3, **{'c':4} )
3.0
```

Here we've called our function with three argument values. The parameter `a` will get its value from a simple positional parameter. The parameter `b` will get its value from a keyword argument. The parameter `c` will get its value from having the `dict` `{'c':4}` turned into keyword parameter assignment.

We'll make more use of this in the section called “Inheritance”.

Creating a print function

The **print** statement has irregular, complex syntax. Also, because it's a statement, making blanket changes requires tedious, manual search and replace on the program source. Writing a print function can make the **print** statement slightly easier to cope with.

We want to mirror the capabilities of the existing print statement, so we need to accept an unlimited number of positional parameters, as well as some optional keyword parameters. One keyword parameter can be used in place of the "chevron" feature to define the output file. The other keyword parameters can provide formatting information:

what character goes between fields and what character (if any) goes at the end of the line.

We want to have a syntax summary like this.

`_print(`*`args`*`,` *`sep=string`*`,` *`end=string`*`,` *`file=file`*`)`

> Converts the argument values to strings, and writes them to the given file. If no file is provided, writes to standard output. The *`sep`* parameter is the column separator, which is a space by default. The *`end`* parameter is the end-of-line character, which is a `\n` by default.

A simple version of our print replacement function would look something like the following. This example looks forward to using the `map` function, which we won't look at in detail until the section called "Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`".

Example 15.1. printFunction.py

```
import sys
def _print( *args, **kw ):
    out= kw.get('file',sys.stdout)
    linesep= kw.get('end','\n')
    colsep= kw.get('sep',' ')
    out.write( colsep.join( map(str,args) ) )
    out.write( linesep )
```

Our intent is to have a function that we can use as follows. We want to provide a long sequence of positional parameters first, followed by some optional keyword parameters.

```
_print( "Python Version", end="" )
_print( sys.version )
_print( "This is", "Stderr", out=sys.stderr, )
_print( "This is", "Stdout", out=sys.stdout, )
```

Here are the original **print** statements that we replaced.

```
print "Python Version",
print sys.version
print >>sys.stderr, "This is", "Stderr"
prinr >>sys.stdout, "This is", "Stdout"
```

Note that we can't simply say `def _print ( *args, sep=' ', end='\n', file=None )` as the definition of this function. We can't have the "all extra positional parameters" listed first in the function definition, even though this is our obvious intent. To achieve the syntax we want, we have to write a function which collects all positional parameters into one sequence of values, and all keyword parameters into a separate dictionary.

Chapter 16. Sets

Table of Contents

Many algorithms need to work with simple containers of data values, irrespective of order or any key. This is a simple set of objects, which is supported by the Python set container. We'll look at Sets from a number of viewpoints: semantics, literal values, operations, comparison operators, statements, built-in functions and methods.

Set Semantics

A `set` is, perhaps the simplest possible container, since it contains objects in no particular order with no particular identification. Objects stand for themselves. With a sequence, objects are identified by position. With a mapping, objects are identified by some key. With a `set`, objects stand for themselves.

Since each object stands for itself, elements of a `set` cannot be duplicated. A `list` or `tuple`, for example, can have any number of duplicate objects. For example, the `tuple ( 1, 1, 2, 3 )` has four elements, which includes two copies of the integer `1`; if we create a `set` from this `tuple`, the `set` will only have three elements.

A `set` has large number of operations for unions, intersections, and differences. A common need is to examine a `set` to see if a particular object is a member of that `set`, or if one `set` is contained within another `set`.

A `set` is mutable, which means that it cannot be used as a key for a `dict` (see Chapter 15, *Mappings and Dictionaries* for more information.) In order to use a `set` as a `dict` key, we can create a `frozenset`, which is an immutable copy of the original `set`. This allows us to accumulate a `set` of values to create a `dict` key.

Set Literal Values

There are no literal values for `sets`. A `set` value is created by using the `set` or `frozenset` factory functions. These can be applied to any *iterable* container, which includes any sequence, and also includes files.

We'll return to "iterable" when we look at the **yield** statement in Chapter 18, *Generators and the **yield** Statement*.

`set(` *`iterable`*`)` → set

Transforms the given iterable (sequence, file or `set`) into a `set`.

```
>>> set( ("hello", "world", "of", "words", "of", "world") )
set(['world', 'hello', 'words', 'of'])
```

Note that we provided a six-`tuple` to the `set` function, and we got a `set` with the four unique objects. The `set` is shown with a `list` literal, to remind us that a `set` is mutable.

`frozenset (`*`iterable`*`)` → set

Transforms the given iterable (sequence, file or `set`) into an immutable `frozenset`.

Set Operations

There are a large number of `set` operations, including union (`|`), intersection (`&`), difference (`-`), symmetric difference (`^`). These are unusual operations, so we'll look at them in some detail. In addition to this operator notation, there are method functions which do the same things. We'll look at the method function versions below.

We'll use the following two `sets` to show these operators.

```
>>> fib=set( (1,1,2,3,5,8,13) )
```

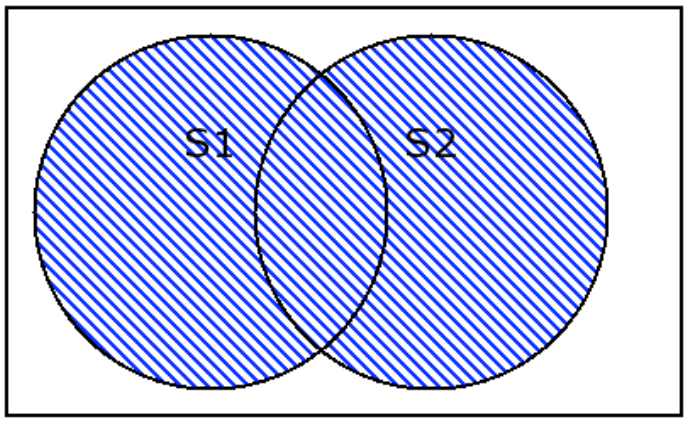
```
>>> prime=set( (2,3,5,7,11,13) )
```

Union, |

The resulting `set` has elements from both source `sets`. An element is in the result if it is one `set` *or* the other.

```
>>> fib | prime
set([1, 2, 3, 5, 7, 8, 11, 13])
```

Figure 16.1. Set Union, S1|S2

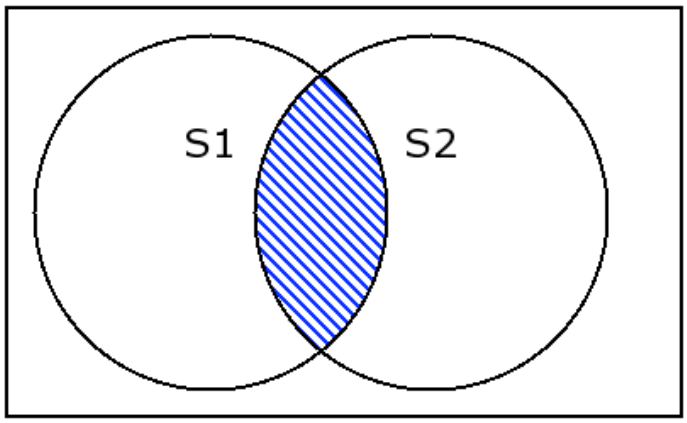


Intersection, &

The resulting `set` has elements that are common to both source `sets`. An element is in the result if it is in one `set` *and* the other.

```
>>> fib & prime
set([2, 3, 5, 13])
```

Figure 16.2. Set Intersection, S1&S2

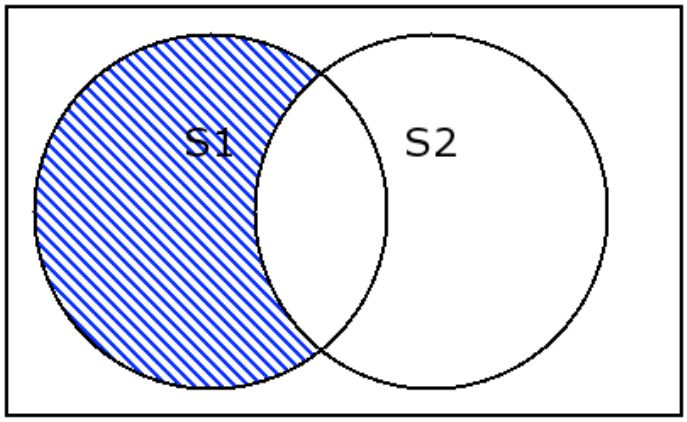


Difference, -

The resulting `set` has elements of the left-hand `set` with all elements from the right-hand `set` removed. An element will be in the result if it is in the left-hand `set` and not in the right-hand `set`.

```
>>> fib - prime
set([8, 1])
>>> prime - fib
set([11, 7])
```

Figure 16.3. Set Difference, S1-S2

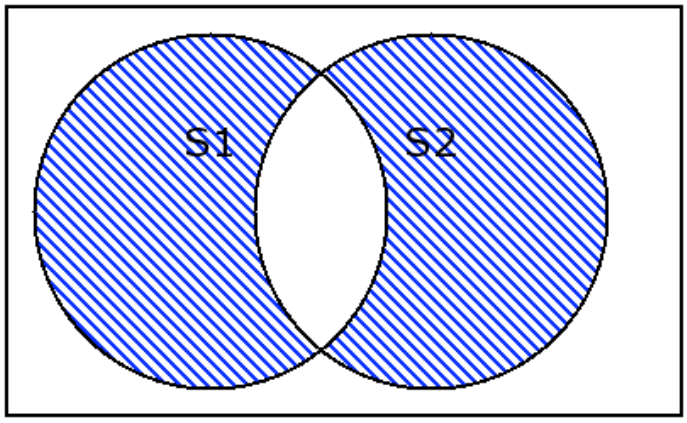


Symmetric Difference, ^

The resulting `set` has elements which are unique to each `set`. An element will be in the result `set` if either it is in the left-hand `set` and not in the right-hand `set` or it is in the right-hand `set` and not in the left-hand `set`. Whew!

```
>>> fib ^ prime
set([8, 1, 11, 7])
```

Figure 16.4. Set Symmetric Difference, S2^S2



Set Comparison Operators

Therera re a number of `set` comparisons. All of the standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`, **in**, **not in**) work with `sets`, but the interpretation of the operators is based on `set` theory. For example, the comparisons determine if we have subset or superset (`<=`, `>=`) relationships between two `sets`.

The basic in and not in operators are the basic membership tests. For example, the `set` `craps` is all of the ways we can roll craps on a come out roll. We've modeled a throw of the dice as a 2-`tuple`. We can now test a specific throw to see if it is craps.

```
>>> craps= set( [ (1,1), (2,1), (1,2), (6,6) ] )
>>> (1,2) in craps
True
>>> (3,4) in craps
False
```

The ordering operators (`<`, `<=`, `>`, `>=`) comoare two `sets` to determine their superset or

subset relationship. These operators reflect the two definitions of subset (and superset). S1 is a subset of S2 if every element of S1 is in S2. The basic subset test is the <= operator; it says nothing about "extra" elements in S2. S1 is a proper subset of S2 if every element of S1 is in S2 and S2 has at least one additional element, not in S1. The proper subset test is the < operator.

Here we've defined `three` to hold both of the dice rolls that total 3. When we compare `three` with `craps`, we see the expected relationships: `three` is a subset `craps` as well as a proper subset of `craps`.

```
>>> three= set( [ (2,1), (1,2) ] )
>>> three < craps
True
>>> three <= craps
True
>>> craps <= craps
True
>>> craps < craps
False
```

We can also see that any given `set` is a subset of itself, but is never a proper subset of itself.

Set Statements

The **for** statement works directly with `sets`, because they are iterable. A `set` is not a sequence, but it is like a sequence because we can iterate through the elements using a **for** statement.

Here we create three `sets`: `even`, `odd` and `zero` to reflect some standard outcomes in Roulette. The union of all three `sets` is the complete `set` of possible spins. We can iterate through this resulting `set`.

```
>>> even= set( range(2,38,2) )
>>> odd= set( range(1,37,2) )
>>> zero= set( (0,'00') )
>>> for n in even|odd|zero:
        print n
```

Set Built-in Functions

A number of built-in functions create or process `sets`.

`len(`*`object`*`)` → integer

Return the number of items of a `set`.

`max(`*`set`*`)` → value

With a `set`, return its largest item

`min(`*`set`*`)` → value

With a `set`, return its smallest item.

Set Methods

A `set` object has a number of member methods. In the following definitions, *`s`* is a `set` object.

The following transformation functions update a `set`.

s.clear

Remove all items from the set.

s.copy → set

Copy the set to make a new set. This is a *shallow copy*. All objects in the new set are references to the same objects as the original set.

s.pop → object

Remove an arbitrary object from the set, returning the object. If the set was already empty, this will raise a KeyError exception.

s.add(*new*)

Adds element *new* to the set. If the object is already in the set, nothing happens.

s.remove(*old*)

Removes element *old* from the set. If the object *old* is not in the set, this will raise a KeyError exception.

s.discard(*old*)

Removes element *old* from the set. If the object *old* is not in the set, nothing happens.

s.update(*new*) → object

Merge values from the *new* set into the original set, adding elements as needed. It is equivalent to the following Python statement. s |= new.

s.intersection_update(*new*) → object

Update *s* to have the intersection of *s* and *new*. In effect, this discards elements from *s*, keeping only elements which are common to *new* and *s*. It is equivalent to the following Python statement. s &= new.

s.difference_update(*new*) → object

Update *s* to have the difference between *s* and *new*. In effect, this discards elements from *s* which are also in *new*. It is equivalent to the following Python statement. s -= new.

s.symmetric_difference_update(*new*) → object

Update *s* to have the symmetric difference between *s* and *new*. In effect, this both discards elements from *s* which are common with *new* and also inserts elements into *s* which are unique to *new*. It is equivalent to the following Python statement. s ^= new.

The following accessor methods provide information about a set.

s.issubset(*set*) → boolean

If *s* is a subset of *set*, return True, otherwise return False. Essentially, this is s <= set.

s.issuperset(*set*) → boolean

If *s* is a superset of *set*, return True, otherwise return False. Essentially, this is s

`>= set`.

s`.union(`*new*`)` → set

If *new* is a proper `set`, return `s | new`. If *new* is a sequence or other iterable, make a new `set` from the value of *new*, then return the union, `s | new`. This does not update *s*.

```
>>> prime.union( (1, 2, 3, 4, 5) )
set([1, 2, 3, 4, 5, 7, 11, 13])
```

s`.intersection(`*new*`)` → set

If *new* is a proper `set`, return `s & new`. If *new* is a sequence or other iterable, make a new `set` from the value of *new*, then return the intersection, `s & new`. This does not update *s*.

s`.difference(`*new*`)` → set

If *new* is a proper `set`, return `s - new`. If *new* is a sequence or other iterable, make a new `set` from the value of *new*, then return the difference, `s - new`. This does not update *s*.

s`.symmetric_difference(`*new*`)` → set

If *new* is a proper `set`, return `s ^ new`. If *new* is a sequence or other iterable, make a new `set` from the value of *new*, then return the symmetric difference, `s ^ new`. This does not update *s*.

Set Exercises

1. **Dice Rolls.** In Craps, each roll of the dice belongs to one of several `sets` of rolls that are used to resolve bets. There are only 36 possible dice rolls, but it's annoying to define the various `sets` manually. Here's a multi-step procedure that produces the various `sets` of dice rolls around which you can define the game of craps.

   First, create a sequence with 13 empty `sets`, call it `dice`. Something like `[ set() ]*13` doesn't work because it makes 13 copies of a single `set` object. You'll need to use a for statement to evaluate the `set` function 13 different times. What is the first index of this sequence? What is the last entry in this sequence?

   Second, write two, nested, for-loops to iterate through all 36 combinations of dice, creating 2-`tuples`. The 36 2-`tuples` will begin with `(1,1)` and end with `(6,6)`. The sum of the two elements is an index into `dice`. We want to add each 2-`tuple` to the appropriate `set` in the `dice` sequence.

   When you're done, you should see results like the following:

   ```
   >>> dice[7]
   set([(5, 2), (6, 1), (1, 6), (4, 3), (2, 5), (3, 4)])
   ```

   Now you can define the various rules as `sets` built from other `sets`.

   `lose`

   On the first roll, you lose if you roll 2, 3 or 12. This is the `set dice[2] | dice[3] | dice[12]`. The game is over.

   `win`

   On the first roll, you win if you roll 7 or 11. The game is over.

`point`

On the first roll, any other result (4, 5, 6, 8, 9, or 10) establishes a point. The game runs until you roll the point or a seven.

`craps`

Once a point is established, you win if you roll the point's number. You lose if you roll a 7.

Once you have these three `sets` defined, you can simulate the first roll of a craps game with a relatively elegant-looking program. You can generate two random numbers to create a 2-`tuple`. You can then check to see if the 2-`tuple` is in the `lose` or `win` `sets`.

If the come-out roll is in the `point` `set`, then the sum will let you pick a `set` from the dice sequence. For example, if the come-out roll is `(2,2)`, the sum is 4, and you'd assign `dice[4]` to the variable `point`; this is the `set` of winners for the rest of the game. The `set` of losers for the rest of the game is always the `craps` `set`.

The rest of the game is a simple loop, like the come-out roll loop, which uses two random numbers to create a 2-`tuple`. If the number is in the `point` `set`, the game is a winner. If the number is in the `craps` `set`, the game is a loser, otherwise it continues.

2. **Roulette Results.** In Roulette, each spin of the wheel has a number of attributes like even-ness, low-ness, red-ness, etc. You can bet on any of these attributes. If the attribte on which you placed bet is in the `set` of attributes for the number, you win.

   We'll look at a few simple attributes: red-black, even-odd, and high-low. The even-odd and high-low attributes are easy to compute. The red-black attribute is based on a fixed set of values.

   ```
   redNumbers= set( [1,3,5,7,9,12,14,16,18,19,21,23,25,27,30,32,34,36] )
   ```

   We have to distinguish between 0 and 00, which makes some of this decision-making rather complex. We can, for example, use ordinary integers for the numbers 0 to 36, and append the `string` "00" to this `set` of numbers. For example, `set( range(37) ) | set( ['00'] )`. This `set` is the entire Roulette wheel, we can call it `wheel`.

   We can define a number of `sets` that stand for bets: `red`, `black`, `even`, `odd`, `high` and `low`. We can iterate though the values of wheel, and decide which `sets` that value belongs to.

   - If the spin is non-zero and `spin % 2 == 0`, add the spin to the `even` `set`.
   - If the spin is non-zero and `spin % 2 != 0`, add the spin to the `odd` `set`.
   - If the spin is non-zero and it's in the `redNumbers` `set`, add the spin to the `red` `set`.
   - If the spin is non-zero and it's not in the `redNumbers` `set`, add the value to the `black` `set`.
   - If the spin is non-zero and `spin <= 18`, add the value to the `low` `set`.
   - If the spin is non-zero and `spin > 18`, add the value to the `high` `set`.

   Once you have these six `sets` defined, you can use them to simulate Roulette. Each round involves picking a random spin with something like `random.choice(`

`list(wheel) )`. You can then see which `set` the spin belongs to. If the spin belongs to a `set` on which you've bet, the spin is a winner, otherwise it's a loser.

These six `sets` all pay 2:1. There are a some `sets` which pay 3:1, including the 1-12, 13-24, 25 to 36 ranges, as well as the three columns, spin % 3 == 0, spin % 3 == 1 and spin % 3 == 2. There are still more bets on the Roulette table, but the `sets` of spins for those bets are rather complex to define.

3. **Sieve of Eratosthenes.** Look at Sieve of Eratosthenes. We created a `list` of candidate prime numbers, using a sequence with 5000 boolean flags. We can, without too much work, simplify this to use a `set` instead of a `list`.

   Procedure 16.1. Sieve of Eratosthenes - Set Version

   1. **Initialize**

      Create a `set`, `prime` which has integers between 2 and 5000.

      `p` ← 2

   2. **Generate Primes.** while 2 ≤ `p` < 5000

      a. **Find Next Prime.** while is is not in `prime` and 2 ≤ `p` < 5000: increment `p`.

         At this point, `p` is prime.

      b. **Remove Multiples**

         `k` ← `p` + `p`

         while `k` < 5000

            i. Remove `k` from the `set prime`

            ii. `k` ← `k` + `p`

      c. **Next `p`.** increment `p`

   3. **Report**

      At this point, the `set prime` has the prime numbers. We can return the `set`.

   In this case, step 2b, Remove Multiples, can be revised to create the `set` of multiples, and use `difference_update` to remove the multiples from `prime`. You can, also, use the `range` function to create multiples of `p`, and create a `set` from this sequence of multiples.

Chapter 17. Exceptions

Table of Contents

A Digression

A well-written program should produce valuable results even when exceptional conditions occur. A program depends on numerous resources: memory, files, other packages, input-output devices, to name a few. Sometimes it is best to treat a problem with any of these resources as an exception, which interrupts the normal sequential flow of the program.

In the section called "Exception Semantics" we introduce the semantics of exceptions. We'll show the basic exception-handling features of Python in the section called "Basic Exception Handling" and the way exceptions are raised by a program in the section called "Raising Exceptions". We descibe most of the built-in exceptions in the section called "Built-in Exceptions". In addition to exercises in the section called "Exception Exercises", we also include style notes in the section called "Style Notes" and a digression on problems that can be caused by poor use of exceptions in the section called "A Digression".

Exception Semantics

An *exception* is an event that interrupts the ordinary sequential processing of a program. When an exception is *raised*, Python will *handle* it immediately. Python does this by examining **except** clauses associated with **try** statements to locate a suite of statements that can process the exception. If there is no **except** clause to handle the exception, the program stops running, and a message is displayed on the standard error file.

An exception has two sides: the dynamic change to the sequence of execution and an object that contains information about the exceptional situation. The dynamic change is initiated by the **raise** statement, and can finish with the handlers that process the raised exception. If no handler matches the exception, the program's execution effectively stops at the point of the **raise**. In addition to the dynamic side of an exception, an object is created by the **raise** statement; this is used to carry any information associated with the exception.

Consequences. The use of exceptions has a two important consequences. First, we need to clarify where exceptions can be raised. Since various places in a program will raise exceptions, and these can be hidden deep within a function or class, their presence should be announced by specifying the possible exceptions in the docstring. Second, multiple parts of a program will have handlers to cope with various exceptions. These handlers should handle just the meaningful exceptions. Some exceptions (like `RuntimeError` or `MemoryError`) generally can't be handled within a program; when these exceptions are raised, the program is so badly broken that there is no real recovery.

Exceptions are a powerful tool for dealing with rare, atypical conditions. Exceptions should be considered as different from the expected or ordinary conditions that a program handles. For example, if a program accepts input from a person, exception processing is not appropriate for validating their inputs. There's nothing rare or uncommon about a person making mistakes while attempting to enter numbers or dates. On the other hand, an unexpected disconnection from a network service is a good candidate for an exception; this is a rare and atypical situation. Examples of good exceptions are those which are raised in response to problems with physical resources like files and networks.

Python has a large number of built-in exceptions, and a programmer can create new exceptions. Generally, it is better to create new exceptions rather than attempt to stretch or bend the meaning of existing exceptions.

Basic Exception Handling

Exception handling is done with the **try** statement. The **try** statement encapsulates several pieces of information. Primarily, it contains a suite of statements and a group of

exception-handling clauses. Each exception-handling clause names a class of exceptions and provides a suite of statements to execute in response to that exception.

The basic form of a **try** statement looks like this:

`try:` *`suite`*

`except` *`exception`* ⟨`,`*`target`*⟩ `:` *`suite`*

`except:` *`suite`*

Each *`suite`* is an indented block of statements. Any statement is allowed in the suite. While this means that you can have nested **try** statements, that is rarely necessary, since you can have an unlimited number of **except** clauses.

If any of the statements in the **try** suite raise an exception, each of the **except** clauses are examined to locate a clause that matches the exception raised. If no statement in the **try** suite raises an exception, the **except** clauses are silently ignored.

The first form of the **except** clause provides a specific exception class which is used for matching any exception which might be raised. If a `target` variable name is provided, this variable will have the exception object assigned to it.

The second form of the **except** clause is the "catch-all" version. This will match all exceptions. If used, this must be provided last, since it will always match the raised exception.

We'll look at the additional **finally** clause in a later sections.

The structure of the complete **try** statement summarizes the philosophy of exceptions. First, try the suite of statements, expecting them work. In the unlikely event that an exception is raised, find an exception clause and execute that exception clause suite to recover from or work around the exceptional situation.

Except clauses include some combination of error reporting, recovery or work-around. For example, a recovery-oriented except clause could delete useless files. A work-around exception clause could returning a complex result for square root of a negative number.

First Example. Here's the first of several related examples. This will handle two kinds of exceptions, `ZeroDivisionError` and `ValueError`.

Example 17.1. exception1.py

```
def avg( someList ):
    """Raises TypeError or ZeroDivisionError exceptions."""
    sum= 0
    for v in someList:
        sum = sum + v
    return float(sum)/len(someList)

def avgReport( someList ):
    try:
        m= avg(someList)
        print "Average+15%=", m*1.15
    except TypeError, ex:
        print "TypeError: ", ex
    except ZeroDivisionError, ex:
        print "ZeroDivisionError: ", ex
```

This example shows the `avgReport` function; it contains a **try** clause that evaluates the `avg` function. We expect that there will be a `ZeroDivisionError` exception if an empty

list is provided to `avg`. Also, a `TypeError` exception will be raised if the list has any non-numeric value. Otherwise, it prints the average of the values in the list.

In the **try** suite, we print the average. For certain kinds of inappropriate input, we will print the exceptions which were raised.

This design is generally how exception processing is handled. We have a relatively simple, clear function which attempts to do the job in a simple and clear way. We have a application-specific process which handles exceptions in a way that's appropriate to the overall application.

Nested try Statements. In more complex programs, you may have many function definitions. If more than one function haa a **try** statement, the nested function evaluations will effectively nest the **try** statements inside each other.

This example shows a function `solve`, which calls another function, `quad`. Both of these functions have a **try** statement. An exception raised by `quad` could wind up in an exception handler in `solve`.

Example 17.2. exception2.py

```
def sum( someList ):
    """Raises TypeError"""
    sum= 0
    for v in someList:
        sum = sum + v
    return sum

def avg( someList ):
    """Raises TypeError or ZeroDivisionError exceptions."""
    try:
        s= sum(someList)
        return float(s)/len(someList)
    except TypeError, ex:
        return "Non-Numeric Data"

def avgReport( someList ):
    try:
        m= avg(someList)
        print "Average+15%=", m*1.15
    except TypeError, ex:
        print "TypeError: ", ex
    except ZeroDivisionError, ex:
        print "ZeroDivisionError: ", ex
```

In this example, we have the same `avgReport` function, which uses `avg` to compute an average of a list. We've rewritten the `avg` function to depend on a `sum` function. Both `avgReport` and `avg` contain **try** statements. This creates a nested context for evaluation of exceptions.

Specifically, when the function sum is being evaluated, an exception will be examined by `avg` first, then examined by `avgReport`. For example, if `sum` raises a `TypeError` exception, it will be handled by `avg`; the `avgReport` function will not see the `TypeError` exception.

Function Design - Exceptions or Status Codes. Note that this example has a subtle bug that illustrates an important point regarding function design. We introduced the bug when we defined `avg` to return either an answer or an error status code in the form of a `string`. Generally, things are more complex when we try to mix valid results and error codes.

Status codes are the only way to report errors in languages that lack exceptions. C, for example, makes heavy use of status codes. The POSIX standard API definitions for

operating system services are oriented toward C. A program making OS requests must examing the results to see if it is a proper values or an indication that an error occurred. Python, however, doesn't have this limitation. Consequently many of the OS functions available in Python modules will raise exceptions rather than mix proper return values with status code values.

In our case, our design for `avg` attepts to return either a valid numeric result or a string result. To be correct we would ave to do two kinds of error checking in `avgReport`. We would have to handle any exceptions and we would also have to examine the results of `avg` to see if they are an error value or a proper answer.

Rather than return status codes, a better design is to simply use exceptions for all kinds of errors. IStatus codes have no real purposes in well-designed programs. In the next section, we'll look at how to define and raise our own exceptions.

Raising Exceptions

The **raise** statement does two things: it creates an `exception` object, and immediately leaves the expected program execution sequence to search the enclosing **try** statements for a matching **except** clause. The effect of a **raise** statement is to either divert execution in a matching **except** suite, or to stop the program because no matching **except** suite was found to handle the exception.

The `exception` object created by **raise** can contain a message string that provides a meaningful error message. In addition to the string, it is relatively simple to attach additional attributes to the exception.

Here are the two forms for the **raise** satement.

`raise` *`exceptionClass`* ⟨, *`value`*⟩

`raise` ⟨*`exception`*⟩

The first form of the **raise** statement uses an exception class name. The optional parameter is the additional value that will be contained in the exception. Generally, this is a `string`.

Here's an example of the **raise** statement.

```
raise ValueError, "oh dear me"
```

This statement raises the built-in exception `ValueError` with an amplifying `string` of `"oh dear me"`. The amplifying `string` in this example, one might argue, is of no use to anybody. This is an important consideration in exception design. When using a built-in exception, be sure that the parameter `string` pinpoints the error condition.

The second form of the raise statement uses an object constructor to create the `Exception` object.

```
raise ValueError( "oh dear me" )
```

Here's a variation on the second form in which additional attributes are provided for the exception.

```
ex= MyNewError( "oh dear me" )
ex.myCode= 42
ex.myType= "O+"
raise ex
```

In this case a handler can make use of the message, as well as the two additional attributes, `myCode` and `myType`.

Defining Your Own Exception. You will rarely have a need to raise a built-in exception. Most often, you will need to define an exception which is unique to your application.

We'll cover this in more detail as part of the object oriented programming features of Python, in Chapter 21, *Classes*. Here's the short version of how to create your own unique exception class.

```
class MyError( Exception ): pass
```

This single statement defines a subclass of `Exception` named `MyError`. You can then raise `MyError` in a **raise** statement and check for `MyError` in **except** clauses.

Here's an example of defining a unique exception and raising this exception with an amplifying `string`.

Example 17.3. quadratic.py

```
import math
class QuadError( Exception ): pass

def quad(a,b,c):
    if a == 0:
        ex= QuadError( "Not Quadratic" )
        ex.coef= ( a, b, c )
        raise ex
    if b*b-4*a*c < 0:
        ex= QuadError( "No Real Roots" )
        ex.coef= ( a, b, c )
        raise ex
    x1= (-b+math.sqrt(b*b-4*a*c))/(2*a)
    x2= (-b-math.sqrt(b*b-4*a*c))/(2*a)
    return (x1,x2)
```

An application that used this function might do something like the following.

```
def quadReport( a, b, c ):
    try:
        x1, x2 = quad( a, b, c )
        print "Roots are", x1, x2
    except QuadError, ex:
        print ex, ex.coef
```

Additional raise Statements. Exceptions can be raised anywhere, including in an **except** clause of a **try** statement. We'll look at two examples of re-raising an exception.

We can use the simple **raise** statement in an **except** clause. This re-raises the original exception. We can use this to do standardized error handling. For example, we might write an error message to a log file, or we might have a standardized exception clean-up process.

```
try:
    attempt something risky
except Exception, ex:
    log_the_error( ex )
    raise
```

This shows how we might write the exception to a standard log in the function `log_the_error` and then re-raise the original exception again. This allows the overall application to choose whether to stop running gracefully or handle the exception.

The other common technique is to transform Python errors into our application's unique errors. Here's an example that logs an error and transforms the built-in `FloatingPointError` into our application-specific error, `MyError`.

```
class MyError( Exception ): pass

try:
    attempt something risky
except FloatingPointError, e:
    do something locally, perhaps to clean up
    raise MyError("something risky failed: %s" % ( e, ) )
```

This allows us to have more consistent error messages, or to hide implementation details.

An Exceptional Example

The following example uses a uniquely named exception to indicate that the user wishes to quit rather than supply input. We'll define our own exception, and define function which rewrites a built-in exception to be our own exception.

We'll define a function, `ckyorn`, which does a "Check for Y or N". This function has two parameters, `prompt` and `help`, that are used to prompt the user and print help if the user requests it. In this case, the return value is always a "Y" or "N". A request for help ("?") is handled automatically. A request to quit is treated as an exception, and leaves the normal execution flow. This function will accept "Q" or end-of-file (usually Control-**D**, but also Control-**Z** on Windows) as the quit signal.

Example 17.4. interaction.py

```
class UserQuit( Exception ): pass

def ckyorn( prompt, help="" ):
    ok= 0
    while not ok:
        try:
            a=raw_input( prompt + " [y,n,q,?]: " )
        except EOFError:
            raise UserQuit
        if a.upper() in [ 'Y', 'N', 'YES', 'NO' ]: ok= 1
        if a.upper() in [ 'Q', 'QUIT' ]:
            raise UserQuit
        if a.upper() in [ '?' ]:
            print help
    return a.upper()[0]
```

We can use this function as shown in the following example.

```
import interaction
answer= interaction.ckyorn(
    help= "Enter Y if finished entering data",
    prompt= "All done?")
```

This function transforms an `EOFError` into a `UserQuit` exception, and also transforms a user entry of "Q" or "q" into this same exception. In a longer program, this exception permits a short-circuit of all further processing, omitting some potentially complex **if** statements.

Details of the `ckyorn` Function. Our function uses a loop that will terminate when we have successfully interpreted an answer from the user. We may get a request for help or perhaps some uninterpretable input from the user. We will continue our loop until we get something meaningful. The post condition will be that the variable `ok` is set to True and the answer, `a` is one of `("Y", "y", "N", "n")`.

Within the loop, we surround our `raw_input` function with a **try** suite,. This allows us to process any kind of input, including user inputs that raise exceptions. The most

common example is the user entering the end-of-file character on their keyboard. For Linux it is control-**D**; for Windows it is control-**Z**.

We handle `EOFError` by raising our `UserQuit` exception. This separates end-of-file on ordinary disk files elsewhere in the program from this end-of-file generated from the user's keyboard. When we get end-of-file from the user, we need to tidy up and exit the program promptly. When we get end-of-file from an ordinary disk file, this will require different processing.

If no exception was raised, we examine the input character to see if we can interpret it. Note that if the user enters 'Q' or 'QUIT', we treat this exactly like as an end-of-file; we raise the `UserQuit` exception so that the program can tidy up and exit quickly.

We return a single-character result only for ordinary, valid user inputs. A user request to quit is considered extraordinary, and we raise an exception for that.

Complete Exception Handling and The finally Clause

A common use case is to have some final processing that must occur irrespective of any exceptions that may arise. The situation usually arises when an external resource has been acquired and must be released. For example, a file must be closed, irrespective of any errors that occur while attempting to read it.

With some care, we can be sure that all exception clauses do the correct final processing. However, this may lead to a some redundant programming. The **finally** clause saves us the effort of trying to carefully repeat the same statement(s) in a number of **except** clauses. This final step will be performed before the try block is finished, either normally or by any exception.

The complete form of a **try** statement looks like this:

`try:` *`suite`*

`except` *`exception`* ⟨`,`*`target`*⟩ `:` *`suite`*

`except:` *`suite`*

`finally:` *`suite`*

Each *`suite`* is an indented block of statements. Any statement is allowed in the suite. While this means that you can have nested **try** statements, that is rarely necessary, since you can have an unlimited number of **except** clauses.

The **finally** clause is always executed. This includes all three possible cases: if the try block finishes with no exceptions; if an exception is raised and handled; and if an exception is raised but not handled. This last case means that every nested try statement with a **finally** clause will have that **finally** clause executed.

Use a **finally** clause to close files, release locks, close database connections, write final log messages, and other kinds of final operations. In the following example, we use the **finally** clause to write a final log message.

```
def avgReport( someList ):
    try:
        print "Start avgReport"
        m= avg(someList)
        print "Average+15%=", m*1.15
    except TypeError, ex:
        print "TypeError: ", ex
    except ZeroDivisionError, ex:
        print "ZeroDivisionError: ", ex
    finally:
```

```
        print "Finish avgReport"
```

Exception Functions

The `sys` module provides one function that provides the details of the exception that was raised. Programs with exception handling will occasionally use this function.

The `sys.exc_info` function returns a 3-`tuple` with the exception, the exception's parameter, and a traceback object that pinpoints the line of Python that raised the exception. This can be used something like the following not-very-good example.

Example 17.5. exception2.py

```
import sys
import math
a= 2
b= 2
c= 1
try:
    x1= (-b+math.sqrt(b*b-4*a*c))/(2*a)
    x2= (-b-math.sqrt(b*b-4*a*c))/(2*a)
    print x1, x2
except:
    e,p,t= sys.exc_info()
    print e,p
```

This uses **multiple assignment** to capture the three elements of the `sys.exc_info` `tuple`, the exception itself in `e`, the parameter in `p` and a Python `traceback` object in `t`.

This "catch-all" exception handler in this example is a bad policy. It may catch exceptions which are better left uncaught. We'll look at these kinds of exceptions in the section called “Built-in Exceptions”. For example, a `RuntimeError` is something you should not bother catching.

Exception Attributes

Exceptions have one interesting attribute. In the following example, we'll assume we have an exception object named *`e`*. This would happen inside an **except** clause that looked like `except` *`SomeException`*`, e:`.

`e``.args`

> A tuple of the argument values provided when the exception was created. Generally this is the error message associated with the exception. However, an exception can be created with a collection of values instead of a message string; these are collected into the `args` attribute.

`e``.message`

> The message string provided when the exception was created.

Built-in Exceptions

The following exceptions are part of the Python environment. There are three broad categories of exceptions.

- Non-error Exceptions. These are exceptions that define events and change the sequence of execution.
- Run-time Errors. These exceptions can occur in the normal course of events, and indicate typical program problems.
- Internal or Unrecoverable Errors. These exceptions occur when compiling the

Python program or are part of the internals of the Python interpreter; there isn't much recovery possible, since it isn't clear that our program can even continue to operate. Problems with the Python source are rarely seen by application programs, since the program isn't actually running.

Here are the non-error exceptions. Generally, you will never have a handler for these, nor will you ever raise them with a **raise** statement.

`StopIteration`

This is raised by an iterator when there is no next value. The **for** statement handles this to end an iteration loop cleanly.

`GeneratorExit`

This is raised when a generator is closed by having the `close` method evaluated.

`KeyboardInterrupt`

This is raised when a user hits control-**C** to send an interrupt signal to the Python interpreter. Generally, this is not caught in application programs because it's the only way to stop a program that is misbehaving.

`SystemExit`

This exception is raised by the `sys.exit` function. Generally, this is not caught in application programs; this is used to force a program to exit.

Here are the errors which can be meaningfully handled when a program runs.

`AssertionError`

Assertion failed. See the **assert** statement for more information in the section called "The **assert** Statement"

`AttributeError`

Attribute not found in an object.

`EOFError`

Read beyond end of file.

`FloatingPointError`

Floating point operation failed.

`IOError`

I/O operation failed.

`IndexError`

Sequence index out of range.

`KeyError`

Mapping key not found.

`OSError`

OS system call failed.

`OverflowError`

Result too large to be represented.

`TypeError`

Inappropriate argument type.

`UnicodeError`

Unicode related error.

`ValueError`

Inappropriate argument value (of correct type).

`ZeroDivisionError`

Second argument to a division or modulo operation was zero.

The following errors indicate serious problems with the Python interepreter. Generally, you can't do anything if these errors should be raised.

`MemoryError`

Out of memory.

`RuntimeError`

Unspecified run-time error.

`SystemError`

Internal error in the Python interpreter.

The following exceptions are more typically returned at compile time, or indicate an extremely serious error in the basic construction of the program. While these exceptional conditions are a necessary part of the Python implementation, there's little reason for a program to handle these errors.

`ImportError`

Import can't find module, or can't find name in module.

`IndentationError`

Improper indentation.

`NameError`

Name not found globally.

`NotImplementedError`

Method or function hasn't been implemented yet.

`SyntaxError`

Invalid syntax.

`TabError`

Improper mixture of spaces and tabs.

`UnboundLocalError`

Local name referenced but not bound to a value.

The following exceptions are part of the implementation of exception objects. Normally, these never occur directly. These are generic categories of exceptions. When you use one of these names in a **catch** clause, a number of more more specialized exceptions will match these.

`Exception`

Common base class for all user-defined exceptions.

`StandardError`

Base class for all standard Python errors. Non-error exceptions (`StopIteration`, `GeneratorExit`, `KeyboardInterrupt` and `SystemExit`) are not subclasses of `StandardError`.

`ArithmeticError`

Base class for arithmetic errors. This is the generic exception class that includes `OverflowError`, `ZeroDivisionError`, and `FloatingPointError`.

`EnvironmentError`

Base class for errors that are input-output or operating system related. This is the generic exception class that includes `IOError` and `OSError`.

`LookupError`

Base class for lookup errors in sequences or mappings, it includes `IndexError` and `KeyError`.

Exception Exercises

1. **Input Helpers.** There are a number of common character-mode input operations that can benefit from using exceptions to simplify error handling. All of these input operations are based around a loop that examines the results of raw_input and converts this to expected Python data.

   All of these functions should accept a prompt, a default value and a help text. Some of these have additional parameters to qualify the `list` of valid responses.

   All of these functions construct a prompt of the form:

   ```
   your prompt [valid input hints,?,q]:
   ```

   If the user enters a ?, the help text is displayed. If the user enters a q, an exception is raised that indicates that the user quit. Similarly, if the `KeyboardInterrupt` or any end-of-file exception is received, a user quit exception is raised from the exception handler.

   Most of these functions have a similar algorithm.

   Procedure 17.1. User Input Function

   1. **Construct Prompt.** Construct the prompt with the hints for valid values, plus `?` and `q`.

   2. **While Not Valid Input.** Loop until the user enters valid input.

      Try the following suite of operations.

      a. **Prompt and Read.** Use raw_input to prompt and read a reply from

the user.

b. **Help?** If the user entered "?", provide the help message.

c. **Quit?** If the user entered "q" or "Q", raise a UserQuit exception.

d. Try the following suite of operations

i. **Convert.** Attempt any conversion.

ii. **Range Check.** If necessary, do any range checks. For some prompts, there will be a fixed list of valid answers. For other prompts, there is no checking required.

If the input is valid, break out of the loop.

In the event of an exception, the user input was invalid.

e. **Nothing?** If the user entered nothing, and there is a default value, return the default value.

In the event of an exception, this function should generally raise a `UserQuit` exception.

3. **Result.** Return the validated user input.

ckdate
: Prompts for and validates a date. The basic version would require dates have a specific format, for example mm/dd/yy. A more advanced version would accept a `string` to specify the format for the input. Much of this date validation is available in the `time` module, which will be covered in Chapter 32, *Dates and Times: the `time` and `datetime` Modules*. This function not reaturn bad dates or other invalid input.

ckint
: Display a prompt; verify and return an integer value

ckitem
: Build a menu; prompt for and return a menu item. A menu is a numbered list of alternative values, the user selects a value by entering the number. The function should accept a sequence of valid values, generate the numbers and return the actual menu item `string`. An additional help prompt of `"??"` should be accepted, this writes the help message and redisplays the menu.

ckkeywd
: Prompts for and validates a keyword from a list of keywords. This is similar to the menu, but the prompt is simply the list of keywords without numbers being added.

ckpath
: Display a prompt; verify and return a pathname. This can use the os.path module for information on construction of valid paths. This should use fstat to check the user input to confirm that it actually exists.

ckrange
: Prompts for and validates an integer in a given range. The range is given as separate values for the lowest allowed and highest allowed value. If either is not given, then that limit doesn't apply. For instance, if only a lowest value is given, the valid input is greater than or equal to the lowest value. If only a highest value is given, the input must be less than or equal to the highest value.

ckstr
: Display a prompt; verify and return a `string` answer. This is similar to the basic raw_input, except that it provides a simple help feature and raises exceptions when the user wants to quit.

cktime

Display a prompt; verify and return a time of day. This is similar to ckdate; a more advanced version would use the time module to validate inputs. The basic version can simply accept a *hh*:*mm*:*ss* time `string` and validate it as a legal time.

ckyorn
: Prompts for and validates yes/no. This is similar to ckkeywd, except that it tolerates a number of variations on yes (YES, y, Y) and a number of variations on no (NO, n, N). It returns the canonical forms: Y or N irrespective of the input actually given.

Style Notes

Built-in exceptions are all named with a leading upper-case letter. This makes them consistent with class names, which also begin with a leading upper-case letter.

Most modules or classes will have a single built-in exception, often called `Error`. This exception will be imported from a module, and can then be qualified by the module name. Modules and module qualification is covered in Part IV, "Components, Modules and Packages". It is not typical to have a complex hierarchy of exceptional conditions defined by a module.

A Digression

Readers with experience in other programming languages may equate an exception with a kind of **goto** statement. It changes the normal course of execution to a (possibly hard to find) exception-handling suite. This is a correct description of the construct, which leads to some difficult decision-making.

Some exception-causing conditions are actually predictable states of the program. The notable exclusions are I/O Error, Memory Error and OS Error. These three depend on resources outside the direct control of the running program and Python interpreter. Exceptions like Zero Division Error or Value Error can be checked with simple, clear **if** statements. Exceptions like Attribute Error or Not Implemented Error should never occur in a program that is reasonably well written and tested.

Relying on exceptions for garden-variety errors — those that are easily spotted with careful design or testing — is often a sign of shoddy programming. The usual story is that the programmer received the exception during testing and simply added the exception processing **try** statement to work around the problem; the programmer made no effort to determine the actual cause or remediation for the exception.

In their defense, exceptions can simplify complex nested **if** statements. They can provide a clear "escape" from complex logic when an exceptional condition makes all of the complexity moot. Exceptions should be used sparingly, and only when they clarify or simplify exposition of the algorithm. A programmer should not expect the reader to search all over the program source for the relevant exception-handling clause.

Future examples, which use I/O and OS calls, will benefit from simple exception handling. However, exception laden programs are a problem to interpret. Exception clauses are relatively expensive, measured by the time spent to understand their intent.

Chapter 18. Generators and the yield Statement

Table of Contents

We've made extensive use of the relationship between the **for** statement and various kinds of *iterable* containers without looking closely at how this works.

In this chapter, we'll look at the semantics of generators, their close relationshp an iterable container, and the **for** statement. We'll look at some additional functions that we can use to create and access data structures that support elegant iteration.

Generator Semantics

The easiest way to define an iterator (and the closely-related concept of generator) is to look at the **for** statement.

Let's look at the following snippet of code.

```
for i in ( 1, 2, 3, 4, 5 ):
    print i
```

Under the hood, the **for** statement engages in the following sequence of interactions with an iterable object like the sequence in the code snippet above.

1. The **for** statement requests an iterator from the object; in this case the object is a `tuple`. The **for** statement does this by evaluating the `iter` function on the given expression. The working definition of iterable is that the object responds to the `iter` function.

2. The **for** statement evaluates the the iterator's `next` method and assigns the value to the target variable; in this case, `i`.

3. The **for** statement evaluates the suite of statements; in this case, the suite is just the **print** statement.

4. The **for** statement continues steps 2 and 3 until an exception is raised. If the exception is a `StopIteration`, this is handled to indicate that the loop has finished normally.

The other side of this relationship is the iterator, which must define a `next` method; this method either returns the next item from a sequence (or other container) or it raises the `StopIteration` exception. Also, an iterator must maintain some kind of internal state to know which item in the sequence will be delivered next.

When we describe a container as iterable, we mean that it responds to the `iter` function by returning an iterator object that can be used by the **for** statement. All of the sequence containers return iterators; `sets` and files also return iterators. In the case of a `dict`, the iterator returns the `dict`'s keys in no particular order.

Defining An Iterator. Generally, we don't directly create iterators, this can be complex. Most often, we define a *generator*. A generator is a function that can be used by the **for** statement as if it were an iterator. A generator looks like a conventional function, with one important difference: a generator includes the **yield** statement.

The essential relationship between a generator and the **for** statement is the same as between an iterator and the **for** statement.

1. The **for** statement calls the generator. The generator begins execution and executes statements up to the first **yield** statement.

2. The **for** statement assigns the value that was returned by the **yield** to the target

variable.

3. The **for** statement evaluates the suite of statements.

4. The **for** statement continues steps 2 and 3 until the generator executes a **return** statement. In a generator, the **return** statement secretly raises the `StopIteration` exception. When a `StopIteration` is raised, it is handled by the **for** statement.

What we Provide. Generator definition is similar to function definition (see Chapter 9, *Functions*); we provide three pieces of information: the name of the generator, a list of zero or more parameters, and a suite of statements that yields the output values.

We use a generator in a **for** statement by following the function's name with `()`'s. The Python interpreter evaluates the argument values in the `()`'s, then applies the generator. This will execute the generator's suite up to the first **yield** statement, which yields the first value from the generator. When the for statement requests the next value, Python will resume execution at the statement after the **yield** statement; the generator will work until it yields another value to the **for** statement.

This back-and-forth between the **for** statement and the generator means that the generator's local variables are all preserved by the **yield** statement. A generator has a peer relationship with the **for** statement; it's local variables are kept when it yields, and disposed of when it returns. This is distinct from ordinary functions, which have a context that is nested within the context that evaluated the function; an ordinary function's local variables are disposed of when it returns.

Example: Using a Generator to Consolidate Information. Lexical scanning and parsing are both tasks that compilers do to discover the higher-level constructs that are present in streams of lower-level elements. A lexical scanner discovers punctuation, literal values, variables, keywords, comments, and the like in a file of characters. A parser discovers expressions and statements in a sequence of lexical elements.

Lexical scanning and parsing algorithms *consolidate* a number of characters into tokens or a number of tokens into a statement. A characteristic of these algorithms is that some state change is required to consolidate the inputs prior to creating each output. A generator provides these characteristics by preserving the generator's state each time an output is yielded.

In both lexical scanning and parsing, the generator function will be looping through a sequence of input values, discovering a high-level element, and then yielding that element. The **yield** statement returns the sequence of results from a generator function, and also saves all the local variables, and even the location of the **yield** statement so that the generator's next request will resume processing right after the **yield**.

Defining a Generator

The presence of the **yield** statement in a function means that the function is actually a generator object, and will have the an iterator-like interface built automatically. In effect it becomes a stateful object with a `next` method defined — so it will work with the **for** statement — and it will raise the `StopIteration` exception when it returns.

The syntax for a function definition is in the section called "Function Definition: The **def** and **return** Statements"; a generator is similar.

`def` *`name`* (⟨*`parameter`*...⟩): *`suite`*

The suite of statements must include at least one **yield** statement.

The **yield** statement specifies the values emitted by the generator. Note that the expression is required.

`yield` *`expression`*

If the **return** statement is used, it ends the generator, and raises the `StopIteration` exception to alert the **for** statement. For obvious reasons, the **return** statement cannot return a value.

Here's a complete, but silly example of a generator.

```
def primes():
    yield 2
    yield 3
    yield 5
    yield 7
    yield 11
    return
```

In this case, we simply yield a fixed sequence of values. After yielding five values, it exits. Here's how this generator is used by a **for** statement.

```
>>> for p in primes():
...     print p


2
3
5
7
11
```

Generator Functions

The `iter` function can be used to acquire an iterator object associated with a container like a sequence, `set`, file or `dict`. We can then manipulate this iterator explicitly to handle some common situations.

`iter(`*`iterable`*`)` → iterator

> Returns the iterator from an object. This iterator interacts with built-in types in obvious ways. For sequences this will return each element in order. For `sets`, it will return each element in no particular order. For dictionaries, it will return the keys in no particular order. For files, it will return each line in order.

Gettimg an explicit iterator — outside a **for** statement — is very handy for dealing with data structures (like files) which have a head-body structure. In this case, there are one or more elements (the head) which are processed one way and the remaining elements (the body) which are processed another way.

We'll return to this in detail in Chapter 19, *Files*. For now, here's a small example.

```
>>> seq=range(10)
>>> seqIter= iter(seq)
>>> seqIter.next()
0
>>> seqIter.next()
1
>>> for v in seqIter:
...     print v


2
3
4
5
6
7
8
```

```
9
```

❶ We create a sequence, `seq`. Any iterable object would work here; any sequence, `set`, `dict` or file.

❷ We create the iterator for this sequence, and assign it to `seqIter`. This object has a `next` method which is used by the **for** statement. We can call this explicitly to get past the heading items in the sequence.

❸ Here, we call `next` explicitly to yield the first two elements of the iterator.

❹ Here, we provide the iterator to the **for** statement. The **for** statement repeatedly calls the `next` method and executes its suite of statements.

Generator Statements

What the for statement really does

In the section called "Iterative Processing: The **for** Statement", we defined a for statement using the following summary:

`for` *`variable`* in *`sequence`* : *`suite`*

This isn't completely accurate, it turns out. We aren't limited to a sequence. The **for** statement actually requires an iterator or generator. Given an object which is not an iterator or generator, it will call the `iter` function to get an iterator over the container.

A more correct syntax summary is the the folliowing:

`for` *`variable`* in *`expression`* : *`suite`*

The Secret of for. Once we've looked at generator functions and iterators, we can see what the **for** statement really does. The purpose of the **for** statement is to visit each value yielded by a generator, assigning each value to the *`variable`*. The **for** statement examines the *`expression`* to see if it is a generator function, or an object. If it is an object, it must respond to the `iter` function by providing a generator function. All of the built-in collections provide the necessary generator function.

Looking forward, we'll see many additional applications of this feature of the **for** statement. As we look at designing our own objects in Part III, "Data + Processing = Objects", we'll want to assure that our objects work well with the **for** statement, also.

Generator Methods

Generators behave as though they have three methods. In the following summaries, *`g`* is the generator (or iterator) object.

g(.next)

The `next` method resumes execution after the **yield** statement. This is called automatically by the **for** statement. The `next` method does one of two things:

- It raises `StopIteration`. An iterator created from a built-in container (sequence, `set`, `dict`, file) does this at the end of the sequence. A generator does this when it returns; either because it finished the suite of statements, or it executed an explicit **return** statement. This exception is handled automatically by the **for** statement.

- It changes its internal state and yields the next value. An iterator updates its positions in the given container. A generator, hopefully, is designed properly to update its internal state and make progress toward completion. It is possible to mis-design a generator so that it never terminates.

g(.close)

The `close` method will force the generator to stop prematurely. This raises a `GeneratorExit` within the generator. If the generator has acquired resources (like a file, socket, lock or database connection), it should use a **try** statement. When the `close` method raises an exception, the event will force the execution of any **finally** clause in the **try** statement, allowing the generator to release the resources it acquired.

g.throw(*type*, ⟨*value*⟩ , ⟨*traceback*⟩)

The `throw` method will force the generator to handle an exception when it reaches the next **yield** statement. This allows a client function to send exception events into the generator.

g.send(*arg*) → next value

This is a variation on the `next` method; it also resumes execution after the **yield** statement. The argument values will become the return value from the **yield** statement. The generator can then use these values to change it's internal state and yield another value or raise `StopIteration`.

Generator Example

Let's look at an example which summarizes some details, yielding the summaries. Assume we have the following `list` of `tuples`. We want to know how many red spins separate a black spin, on average. We need a function which will yield the count of gaps as it examines the spins. We can then call this function repeatedly to get the gap information.

Example 18.1. generator.py

```
spins = [('red', '18'), ('black', '13'), ('red', '7'),
    ('red', '5'), ('black', '13'), ('red', '25'),
    ('red', '9'), ('black', '26'), ('black', '15'),
    ('black', '20'), ('black', '31'), ('red', '3')]

def countReds( aList ):
    count= 0
    for color,number in aList:
        if color == 'black':
            yield count
            count= 0
        else:
            count += 1
    yield count

gaps= [ gap for gap in countReds(spins) ]
print gaps
```

❶ The `spin` variable defines our sample data. This might be an actual record of spins.

❷ We define our `gapCount(`*aList*` )` function. This function initializes `count` to show the number of non-black's before a black. It then steps through the individual spins, in the order presented. For non-black's, the count is incremented.

❸ For black spins, however, we yield the length of the gap between the last black. When we yield a result, the generator produces a result value, and also saves all the other processing information about this function so that it can be continued from this point.

When the function is continued, it resumes right after the **yield** statement: the count will be reset, and the **for** loop will advance to examine the next number in the sequence.

❹ When the sequence is exhausted, we also yield the final count. The first and last gap counts may have to be discarded for certain kinds of statistical analysis.

❺ This shows how we use the generator created by a function with a **yield** statement. In this case, we create a `list` comprehension; the **for** clause will step through the values yielded by the generator until the it exits normally. This sequence of values is collected into a `list` that we can the use for statistical analysis.

Generator Exercises

1. **The Sieve of Eratosthones (Again).** Look at Sieve of Eratosthenes and Sieve of Eratosthenes. We created a `list` or a set of candidate prime numbers. This exercise has three parts: initialization, generating the `list` (or `set`) or prime numbers, then reporting. In the `list` version, we had to filter the sequence of boolean values to determine the primes. In the `set` version, the `set` contained the primes.

   Within the *Generate* step, there is a point where we know that the value of `p` is prime. At this point, we can yield p. If yield each value as we discover it, we eliminate the entire "report" step from the function.

2. **The Generator Version of `range`.** The range function creates a sequence. For very large sequences, this consumes a lot of memory. You can write a version of range which does not create the entire sequence, but instead yields the individual values. Using a generator will have the same effect as iterating through a sequence, but won't consume as much memory.

   Define a generator, `genrange`, which generates the same sequence of values as `range`, without creating a `list` object.

   Check the documentation for the built-in function `xrange`.

3. **Prime Factors.** There are two kinds of positive numbers: prime numbers and composite numbers. A composite number is the product of a sequence of prime numbers. You can write a simple function to factor numbers and yield each prime factor of the number.

   Your `factor` function can accept a number, `n`, for factoring. The function will test values, `f`, between 2 and the square root of `n` to see if the expression `n % f == 0` is true. If this is true. then the factor, `f`, divides `n` with no remainder; `f` is a factor.

   Don't use a simple-looking **for**-loop; the prime factor of 128 is 2, repeated 7 times.

Subroutines and Coroutines

In the section called "Generator Semantics" we noted that a **for** statement and the associated generator are peers. Technically, two functions with this peer relationship are called coroutines. This is in contrast to ordinary functions, where one is subordinate to the other; when a function is evaludatated, it is a subroutine.

The function evaluation we've been using since Chapter 9, *Functions* shows the standard subroutine relationship. The function's evaluation is entirely subordinate to the statement that evaluated the function. Consider the following snippet of code

```
for i in range(10):
    print math.sqrt( float( i ) )
```

We have a **print** statement which evaluates the `math.sqrt` function. The evaluation of the `math.sqrt` function is entirely subordinate to the statement. Each evaluation is `math.sqrt` is distinct, with no memory of any previous evaluation of the function. We

sometimes describe functions like this as *idempotent*: we get the same result every time because there's no memory or history.

A generator, however, can define a different kind of relationship called coroutine instead of subroutine. Evaluation of a couroutine is not subordinate to the statement, but a peer with the statement. We can use this to create functions with "memory." Couroutines are functions which have an internal state.

To define a stateful coroutine, we will use the more general form of the **yield** statement and we'll use `send` instead of `next`. The **yield** statement we looked at in the section called “Defining a Generator” was triggered by a **for** statement calling the generator's `next` method. Each time the **for** statement evaluated the `next` method, the generator resumed execution at the **yield** statement. The value provided by the **yield** statement was the value returned by the `next` function. The relationship is assymetric becuase the generator provides values with no interaction or control.

It turns out that the **yield** statement is really a special kind of operator. The **yield** operator yields a value that will be made available through the `next` or `send` function. Additionally, the `send` function can provide a value which is returned by the **yield** operator. There are two simultaneous data transfers: the **yield** value is the result returned by the `send` function; the values provided by the `send` function and the results returned by the **yield** operator.

Example 18.2. coroutine.py

```
def search(aList,key):
    probeG= probe(len(aList))
    try:
        index= probeG.next()
        while aList[index] != key:
            print "search for", key, ":", index
            index= probeG.send( cmp(aList[index],key) )
        probeG.close()
        return index
    except StopIteration:
        return -1

def probe( size ):
    lo, hi = 0, size
    oldMid, mid = 0, (lo+hi)//2
    while oldMid != mid:
        compare= (yield mid)
        if compare < 0: lo= mid
        else: hi= mid
        oldMid, mid = mid, (lo+hi)//2
```

❶ The search function examines a sorted list to locate the entry with a given key.

❷ We initiate the generator, `probe`, providing the overall size of the list. We save the return value in `probeG`. The return value from an initial call to a generator function is an internal generator object which has a number of methods (next, send, close and throw.) When we use simple generators in a **for** statement, this happens silently and automatically.

❸ We evaluate `probeG`'s `next` method, which yields the first value from the generator. This is a position in the list that we'll probe to see if the requested key is at that position.

❹ If we didn't find the requested key, we send the comparison result to the generator via the generator's `send` method. This will cause the generator to yield a new probe value. The probe function's **yield** operator will receive either -1 or +1 from this `send` method.

❺ We handle the `StopIteration` exception; this may be raised if our `probe` function has run out of values to yield to search.

❻ The `probe` function contains a **yield** operation, so it is a generator. It is initialized

with the size of a list that will be searched. The variables `lo` and `hi` define a subset of the list which is being probed. Initially, it's the whole list. Values sent in from the coroutine, `search`, will define which half of the list will be considered as the new subset list.

❼ We yield the previously computed middle of the list. The result of the **yield** operator will be the comparison result. A negative number means that the probed value was too low, and we need to examine the upper half of the subset. A positive number means we should examine the lower half of the subset. A new middle position is computed; this will be yielded next time the coroutine makes a request.

Linear Search and Binary Search. Note that we've partitioned the algorithm into two elements: the key comparison portion (in searc), and the sequence of probe values. We've provided a binary search version of probe. We can also provide a linear search version of probe. With a few name changes we can create a search function which works with a sorted list or an unsorted list.

The probe function can be renamed to either "sorted", because that's the kind of data it requires, or "binary" because that's the algorithm it embodies.

The following function can be called "unsorted" or "linear", depending on your preference for describing the data it works with or the algorithm.

```
def unsorted( size ):
    for pos in range(size):
        compare= (yield pos)
        if compare == 1: break # went too far
```

We prefer "sorted" and "unsorted" as the coroutine names. We can define a version of search that we woukld use as follows:

```
search( sorted, someList, akey )
search( unsorted, someOtherList, aKey )
```

This relies on providing one of the two probe functions to the search function.

Chapter 19. Files

Table of Contents

Programs often deal with external data; data outside of volatile primary memory. This external data could be persistent data on a file system or transient data on an input-output device. Most operating systems provide a simple, uniform interface to external data via `files`. In the section called “File Semantics”, we provide an overview of the semantics of files. We cover the most important of Python's built-in functions for working with files in the section called “Built-in Functions”. In the section called “File Methods”, we describe some method functions of file objects.

Files are a deep, deep subject. We'll touch on several modules that are related to managing files in Part IV, "Components, Modules and Packages". These include Chapter 33, *File Handling Modules* and Chapter 34, *File Formats: CSV, Tab, XML, Logs and Others*.

File Semantics

In one sense a file is a container for a sequence of bytes. A more useful view, however, is that a `file` is a container of data objects, encoded as a sequence of bytes. Files can be kept on persistent but slow devices like disks. Files can also be presented as a stream of bytes flowing through a network interface. Even the user's keyboard can be processed as if it was a file; in this case the file forces our software to wait until the person types something.

Our operating systems use the abstraction of *file* as a way to unify access to a large number of devices and operating system services. In the Linux world, all external devices, plus a large number of in-memory data structures are accessible through the file interface. The wide variety of things with file-like interfaces is a consequence of how Unix was originally designed. Since the number and types of devices that will be connected to a computer is essentially infinite, device drivers were designed as a simple, flexible plug-in to the operating system. For more information on the ubiquity of files, see the section called "Additional Background".

Files include more than disk drives and network interfaces. Kernel memory, random data generators, semaphores, shared memory blocks, and other things have file interfaces, even though they aren't — strictly speaking — devices. Our OS applies the file abstraction to many things. Python, similarly, extends the file interface to include certain kinds of in-memory buffers.

All GNU/Linux operating systems make all devices available through a standard file-oriented interface. Windows makes most devices available through a reasonably consistent file interface. Python's `file` class provides access to the OS file API's, giving our applications the same uniform access to a variety of devices.

> **Important**
>
> The terminology is sometimes confusing. We have physical files on our disk, the file abstraction in our operating system, and `file` objects in our Python program. Our Python `file` object makes use of the operating system file API's which, in turn, manipulate the files on a disk.
>
> We'll try to be clear, but with only one overloaded word for three different things, this chapter may sometimes be confusing.
>
> We rarely have a reason to talk about a physical file on a disk. Generally we'll talk about the OS abstraction of file and the Python class of `file`.

Standard Files. Consistent with POSIX standards, all Python programs have three files available: `sys.stdin`, `sys.stdout`, `sys.stderr`. These files are used by certain built-in statements and functions. The **print** statement, for example, writes to `sys.stdout`. The `input` and `raw_input` functions both write their prompt to `sys.stdout` and read their input from `sys.stdin`.

These standard files are always available, and Python assures that they are handled consistently by all operating systems. The `sys` module makes these files available for explict use. Newbies may want to check File Redirection for Newbies for some additional notes on these standard files.

File Redirection for Newbies

The presence of a standard input file and two standard output files is a powerful tool for simplifying programs and making them more reusable. Programs which read from standard input and write to standard output can be combined into processing sequences, called pipelines. The POSIX-standard syntax for creating a pipeline is shown below.

```
$ ./step1.py <source.dat | ./step2.py >result.dat
```

This pipeline has two steps: `step1.py` reads input from stdin and writes to stdout. We've told the shell to redirect stdin so that it reads the file `source.dat`. We've also told the shell to connect `step1.py` standard output to `step2.py` standard input. We've also redirected `step2.py` standard output to a file, `result.dat`.

We can simplify our programs when we make the shell responsible for file name and pipeline connections.

File Organization and Structure. Some operating systems provide support for a large variety of file organizations. Different file organizations include different record termination rules, possibly with keys, and possibly fixed length records. The POSIX standard, however, considers a file to be nothing more than a sequence of bytes. It becomes entirely the job of the application program, or libraries outside the operating system to impose any organization on those bytes.

The basic `file` objects in Python consider a file to be a sequence of characters. (These can be ASCII or Unicode characters.) The characters can be processed as a sequence of variable length lines; each line terminated with a newline character. Files moved from a Windows environment may contain lines with an extraneous ASCII carriage return character (`\r`), which is easily removed with the `string strip` method.

Ordinary text files can be managed directly with the built-in `file` objects and their methods for reading and writing lines of data. We will cover this basic text file processing in the rest of this chapter.

Additional Background

The GNU/Linux view of files can be surprising for programmers with a background that focuses on mainframe Z/OS or Windows. This is additional background information for programmers who are new to the POSIX use of the file abstraction. This POSIX view informs how Python works.

In the Z/OS world, files are called *data sets*, and can be managed by the OS catalog or left uncataloged. While this is also true in the GNU/Linux world, the catalog (called a directory) is seamless, silent and automatic, making files far easier to manage than they are in the Z/OS world. In the GNU/Linux world, uncataloged, temporary files are atypical, rarely used, and require special API's.

In the Z/OS world, files are generally limited to disk files and nothing else. This is different from the GNU/Linux use of file to mean almost any kind of external device or service.

Block Mode Files. File devices can be organized into two different kinds of structures: *block mode* and *character mode*. Block mode devices are exemplified by magnetic disks: the data is structured into blocks of bytes that can be accessed in any order. Both the media (disk) and read-write head can move; the device can be repositioned to any block as often as necessary. A disk provides direct (sometimes also called random) access to each block of data.

Character mode devices are exemplified by network connections: the bytes come pouring into the processor buffers. The stream cannot be repositioned. If the buffer fills up and bytes are missed, the lost data are gone forever.

Operating system support for block mode devices includes file directories and file management utilities for deleting, renaming and copying files. Modern operating systems include file navigators (Finders or Explorers), iconic representations of files, and standard GUI dialogs for opening files from within application programs. The operating system also handles moving data blocks from memory buffers to disk and from disk to memory buffers. All of the device-specific vagaries are handled by having a variety of *device drivers* so that a range of physical devices can be supported in a uniform manner by a single operating system software interface.

Files on block mode devices are sometimes called *seekable*. They support the operating system `seek` function that can begin reading from any byte of the file. If the file is structured in fixed-size blocks or records, this seek function can be very simple and effective. Typically, database applications are designed to work with fixed-size blocks so that seeking is always done to a block, from which database rows are manipulated.

Character Mode Devices and Keyboards. Operating systems also provide rich support for character mode devices like networks and keyboards. Typically, a network connection requires a *protocol stack* that interprets the bytes into packets, and handles the error correction, sequencing and retransmission of the packets. One of the most famous protocol stacks is the TCP/IP stack. TCP/IP can make a streaming device appear like a sequential file of bytes. Most operating systems come with numerous client programs that make heavy use of the netrowk, examples include sendmail, ftp, and a web browser.

A special kind of character mode file is the *console*; it usually provides input from the keyboard. The POSIX standard allows a program to be run so that input comes from files, pipes or the actual user. If the input file is a *TTY* (teletype), this is the actual human user's keyboard. If the file is a pipe, this is a connection to another process running concurrently. The keyboard console or TTY is different from ordinary character mode devices, pipes or files for two reasons. First, the keyboard often needs to explicitly echo characters back so that a person can see what they are typing. Second, pre-processing must often be done to make backspaces work as expected by people.

The echo feature is enabled for entering ordinary data or disabled for entering passwords. The echo feature is accomplished by having keyboard events be queued up for the program to read as if from a file. These same keyboard events are automatically sent to update the GUI if echo is turned on.

The pre-processing feature is used to allow some standard edits of the input before the application program receives the buffer of input. A common example is handling the backspace character. Most experienced computer users expect that the backspace key will remove the last character typed. This is handled by the OS: it buffers ordinary characters, removes characters from the buffer when backspace is received, and provides the final buffer of characters to the application when the user hits the Return key. This handling of backspaces can also be disabled; the application would then see the keyboard events as *raw* characters. The usual mode is for the OS to provide *cooked* characters, with backspace characters handled before the application sees any data.

Typically, this is all handled in a GUI in modern applications. However, Python provides some functions to interact with Unix TTY console software to enable and disable echo and process raw keyboard input.

File Formats and Access Methods. In Z/OS (and Open VMS, and a few other operating systems) files have very specific formats, and data access is mediated by the operating system. In Z/OS, they call these access methods, and they have names like BDAM or VSAM. This view is handy in some respects, but it tends to limit you to the

access methods supplied by the OS vendor.

The GNU/Linux view is that files should be managed minimally by the operating system. At the OS level, files are just bytes. If you would like to impose some organization on the bytes of the file, your application should provide the access method. You can, for example, use a database management system (DBMS) to structure your bytes into tables, rows and columns.

The C-language standard I/O library (stdio) can access files as a sequence of individual lines; each line is terminated by the newline character, **\n**. Since Python is built in the C libraries, Python can also read files as a sequence of lines.

Built-in Functions

There are two built-in functions that creates a new file or open an existing file.

`file` (*`filename`*, [*`mode`*,] [*`buffering`*]) → file object

> Create a Python file object associated with an operating system file. *`filename`* is the name of the file. *`mode`* can be 'r', 'w' or 'a' for reading (default), writing or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. Add a 'b' to the mode for binary files. Add a '+' to the mode to allow simultaneous reading and writing. If the *`buffering`* argument is given, 0 means unbuffered, 1 means line buffered, and larger numbers specify the buffer size.

`open` (*`filename`*, [*`mode`*,] [*`buffering`*]) → file object

> Does the same thing as the `file` function. The `file` function is a standard factory function, like `int`, `float`, `str`, `list`, `tuple`, etc. The name of the function matches the class of the object being created. The `open` function, however, is more descriptive of what is really going on in the program.

Creating File Name Strings. A filename `string` can be given as a standard name, or it can use OS-specific punctuation. The standard is to use `/` to separate elements of a file path; Python can do OS-specific translation. Windows, for example, uses `\` for most levels of the path, but has a leading device character separated by a `:`. Rather than force your program to implement the various operating system punctuation rules, Python provides modules to help you construct and process file names. The `os.path` module should be used to construct file names. Best practice is to use the `os.path.join` function to make file names from sequences of `strings`. We'll look at this in Chapter 33, *File Handling Modules*.

The *`filename`* `string` can be a simple file name, also called a *relative path* `string`, where the OS rules of applying a current working directory are used to create a full, absolute path. Or the filename `string` can be a full *absolute path* to the file.

File Mode Strings. The *`mode`* `string` specifies how the file will be accessed by the program. There are three separate issues addressed by the mode `string`: opening, text handling and operations.

Opening. For the opening part of the mode `string`, there are three alternatives:

`r`

> Open for reading. Start at the beginning of the file. If the file does not exist, raise an `IOError` exception. This is implied if nothing else is specified.

`w`

> Open for writing. Start at he beginning of the file. If the file does not exist, create

the file.

`a`

Open for appending. Start at the end of the file. If the file does not exist, create the file.

Text Handling. For the text handling part of the mode `string`, there are two alternatives:

`b`

Do not interpret newlines as end of line. The file is simply a sequence of bytes.

(nothing)

The default, if nothing is specified is to interpret newlines as end of line. The file is a sequence of text lines.

`U`

The capital `U` mode enables "universal newline" processing. This allows your program to cope with the non-standard line-ending characters present in some Windows files. The standard end-of-line is a single newline character, `\n`. In Windows, an additional `\r` character may also be present.

Operations. For the additional operations part of the mode `string`, there are two alternatives:

`+`

Allow both read and write operations.

(nothing)

If nothing is specified, allow only reads for files opened with "r"; allow only writes for files opened with "w" or "a".

Typical combinations include `"rb"` to read binary data and `"w+"` to create a file for reading and writing.

Examples. The following examples create file objects for further processing:

```
myLogin = file( ".login", "r" )
newSource = open( "somefile.c", "w" )
theErrors = file( "error.log", "a" )
```

Each of these opens a named file in the current working directory. The first example opens a file for reading. The second example creates a new file or truncates an existing file prior to writing. The third example creates a new file or opens an existing file for appending.

Buffering files is typically left as a default, specifying nothing. However, for some situations buffering can improve performance. Error logs, for instance, are often unbuffered, so the data is available immediately. Large input files may have large buffer numbers specified to encourage the operating system to optimize input operations by reading a few large chunks of data instead of a large number of smaller chunks.

File Methods

The built-in `file` function creates a `file` object. The resulting object has a number of operations that change the state of the file, read or write data, or return information

about the file. In the following descriptions, *f* is a `file` object, created by the `file`.

Read Methods. The following methods read from a file. As data is read, the file position is advanced from the beginning to the end of the file. The file must be opened with a mode that includes or implies `'r'` for these methods to work.

f.`read`(*[size]*) → string

> Read as many as *size* characters from file *f*. If *size* is negative or omitted, the rest of the file is read.

f.`readline`(*[size]*) → string

> Read the next line or as many as *size* characters from file *f*; an incomplete line can be read. If *size* is negative or omitted, the next complete line is read. If a complete line is read, it includes the trailing newline character, `\n`. If the file is at the end, *f*. `readline` returns a zero length `string`. If the file has a blank line, this will be a `string` of length 1 (the newline character).

f.`readlines`(*[hint]*) → list of strings

> Read the next lines or as many lines from the next *hint* characters from file *f*. The value of *hint* may be rounded up to match an internal buffer size. If *hint* is negative or omitted, the rest of the file is read. All lines will include the trailing newline character, `\n`. If the file is at the end, *f*. `readline2` returns a zero length `list`.

Write Methods. The following methods writing to a file. As data is written, the file position is advanced, possibly growing the file. If the file is opened for write, the position begins at the beginning of the file. If the file is opened for append, the position begins at the end of the file. If the file does not already exist, both writing and appending are equivalent. The file must be opened with a mode that include `'a'` or `'w'` for these methods to work.

f.`flush`

> Flush all accumulated data from the internal buffers of file *f* to the OS file. Depending on your OS, this may also force all the data to be written to the device.

f.`write`(*string*)

> Write the given `string` to file *f*. Buffering may mean that the `string` does not appear on any console until a `close` or `flush` operation is used.

f.`writelines`(*list*)

> Write the `list` of `strings` to file *f*. Buffering may mean that the `strings` do not appear on any console until a `close` or `flush` operation is used.

f.`truncate` 〈*[size]*〉

> Truncate file *f*. If *size* is not given, the file is truncated at the current position. If *size* is given, the file will be truncated at *size*. If the file isn't as large as the given *size*, the results vary by operating system. This function is not available on all platforms.

Position Control Methods. The current position of a file can be examined and changed. Ordinary reads and writes will alter the position. These methods will report the position, and allow you to change the position that will be used for the next operation.

f.`seek`(*offset*, [*whence*])

Change the position from which file *f* will be processed. There are three values for *whence* which determine the direction of the move. If *whence* is zero (the default), move to the absolute position given by *offset*. `f.seek(0)` will rewind file `f`. If *whence* is one, move relative to the current position by *offset* bytes. If offset is negative, move backwards; otherwise move forward. If *whence* is two, move relative to the end of file. `f.seek(0,2)` will advance file *f* to the end, making it possible to append to the file.

f`.tell` → integer

Return the position from which file *f* will be processed. This is a partner to the `seek` method; any position returned by the `tell` method can be used as an argument to the `seek` method to restore the file to that position.

Other Methods. These are additional useful methods of a file object.

f`.close`

Close file *f*. The closed flag is set. Any further operations (except a redundant close) raise an `IOError` exception.

f`.fileno` → integer

Return the internal file descriptor (FD) used by the OS library when working with file *f*. A number of Python modules provide functions that use the OS libraries; the OS libraries need the FD.

f`.isatty` → boolean

Return `True` if file *f* is connected to the console or keyboard.

f`.closed`

This attribute is `True` if file *f* is closed.

f`.mode`

This attribute of file *f* is the mode argument to the `file` function that was used to create the file object.

f`.name`

This attribute of file *f* is the filename argument to the `file` function that was used to create the file object.

Several Examples

We'll look at four examples of file processing. In all cases, we'll read simple text files. We'll show some traditional kinds of file processing programs and how those can be implemented using Python.

Reading a Text File

The following program will examine a standard unix password file. We'll use the explicit `readline` method to show the processing in detail. We'll use the `split` method of the input `string` as an example of parsing a line of input.

Example 19.1. readpswd.py

```
pswd = file( "/etc/passwd", "r" )
for aLine in pswd
    fields= aLine.split( ":" )
```

```
    print fields[0], fields[1]
pswd.close()
```

❶ This program creates a `file` object, `pswd`, that represents the `/etc/passwd` file, opened for reading.
❷ A `file` is a sequence of lines. We can use a `file` in the **for** statement, and the `file` object will return each individual line in response to the `next` method.
❸ The input `string` is split into individual fields using `":"` boundaries. Two particular fields are printed. Field 0 is the username and field 1 is the password.
❹ Closing the file releases any resources used by the file processing.

For non-unix users, a password file looks like the following:

```
root:q.mJzTnu8icF.:0:10:God:/:/bin/csh
fred:6k/7KCFRPNVXg:508:10:% Fredericks:/usr2/fred:/bin/csh
```

Reading a File as a Sequence of Strings

This program shows us that a file is a sequence of individual lines. Because it is an iterable object, the **for** statement will provide the individual lines.

This file will have a CSV (Comma-Separated Values) file format that we will parse. The `csv` module does a far better job than this little program. We'll look at that module in the section called "Comma-Separated Values: The `csv` Module".

A popular stock quoting service on the Internet will provide CSV files with current stock quotes. The files have comma-separated values in the following format:

```
stock, lastPrice, date, time, change, openPrice, daysHi, daysLo, volume
```

The stock, date and time are typically quoted `strings`. The other fields are numbers, typically in dollars or percents with two digits of precision. We can use the Python `eval` function on each column to gracefully evaluate each value, which will eliminate the quotes, and transform a string of digits into a floating-point price value. We'll look at dates in Chapter 32, *Dates and Times: the `time` and `datetime` Modules*.

This is an example of the file:

```
"^DJI",10623.64,"6/15/2001","4:09PM",-66.49,10680.81,10716.30,10566.55,N/A
"AAPL",20.44,"6/15/2001","4:01PM",+0.56,20.10,20.75,19.35,8122800
"CAPBX",10.81,"6/15/2001","5:57PM",+0.01,N/A,N/A,N/A,N/A
```

The first line shows a quote for an index: the Dow-Jones Industrial average. The trading volume doesn't apply to an index, so it is "N/A". The second line shows a regular stock (Apple Computer) that traded 8,122,800 shares on June 15, 2001. The third line shows a mutual fund. The detailed opening price, day's high, day's low and volume are not reported for mutual funds.

After looking at the results on line, we clicked on the link to save the results as a CSV file. We called it `quotes.csv`. The following program will open and read the `quotes.csv` file after we download it from this service.

Example 19.2. readquotes.py

```
qFile= file( "quotes.csv", "r" )
for q in qFile:
    try:
        stock, price, date, time, change, opPrc, dHi, dLo, vol\
        = q.strip().split( "," )
        print eval(stock), float(price), date, time, change, vol
    except ValueError:
        pass
```

```
qFile.close()
```

❶ We open our quotes file, `quotes.csv`, for reading, creating an object named `qFile`.

❷ We use a **for** statement to iterate through the sequence of lines in the file.

❸ The quotes file typically has an empty line at the end, which splits into zero fields, so we surround this with a **try** statement. The empty line will raise a `ValueError` exception, which is caught in the **except** clause and ignored.

❹ Each stock quote, `q`, is a `string`. By using the `strip` operation of the `string`, we create a new string with excess whitespace characters removed. The `string` which is created then performs the `split ( ',' )` operation to separate the fields into a `list`. We use multiple assignment to assign each field to a relevant variable. Note that we strip this file into nine fields, leading to a long statement. We put a `\` to break the statement into two lines.

❺ The name of the stock is a string which includes quotes. In order to gracefully remove the quotes, we use the `eval` function. The price is a string. We use the `float` function to convert this string to a proper numeric value for further processing.

Read, Sort and Write

For COBOL expatriates, here's an example that shows a short way to read a file into an in-memory sequence, sort that sequence and print the results. This is a very common COBOL design pattern, and it tends to be rather long and complex in COBOL.

This example looks forward to some slightly more advanced techniques like `list` sorting. We'll delve into sorting in Chapter 20, *Advanced Sequences*.

Example 19.3. sortquotes.py

```
data= []
qFile= file( "quotes.csv", "r" )
for q in qFile:
    fields= tuple( q.strip().split( "," ) )
    if len(fields) == 9: data.append( fields )
qFile.close()
def priceVolume(a,b):
    return cmp(a[1],b[1]) or cmp(a[8],b[8])
data.sort( priceVolume )
for stock, price, date, time, change, opPrc, dHi,  dLo, vol in data:
    print stock, price, date, time, change, volume
```

❶ We create an empty sequence, `data`, to which we will append `tuples` created from splitting each line into fields.

❷ We create file object that will read all the lines of our CSV-format file.

❸ This **for** loop will set `q` to each line in the file.

❹ The variable `field` is created by stripping whitespace from the line, `q`, breaking it up on the `","` boundaries into separate fields, and making the resulting sequence of field values into a `tuple`.

If the line has the expected nine fields, the `tuple` of fields is appended to the `data` sequence. Lines with the wrong number of fields are typically the blank lines at the beginning or end of the file.

❺ To prepare for the sort, we define a comparison function. This will compare fields 1 and 8, price and volume. This relies on the behavior of the **or** operator: if the comparison of field 1 is equal, the value of `cmp` will be 0, which is equivalent to `False`; so field 8 must be compared.

❻ We can then sort the `data` sequence. The sort function will use our `priceVolume` function to compare records. This kind of sort is covered in depth in the section called "Advanced List Sorting".

❼ Once the sequence of data elements is sorted, we can then print a report showing

our stocks ranked by price, and for stocks of the same price, ranked by volume. We could expand on this by using the `%` operator to provide a nicer-looking report format.

Reading "Records"

In languages like C or COBOL a "record" or "struct" that describe the contents of a file. The advantage of a record is that the fields have names instead of numeric positions. In Python, we can acheive the same level of clarity using a `dict` for each line in the file.

For this, we'll download files from a web-based portfolio manager. This portfolio manager gives us stock information in a file called `display.csv`. Here is an example.

```
+/-,Ticker,Price,Price Change,Current Value,Links,# Shares,P/E,Purchase Price,
-0.0400,CAT,54.15,-0.04,2707.50,CAT,50,19,43.50,
-0.4700,DD,45.76,-0.47,2288.00,DD,50,23,42.80,
0.3000,EK,46.74,0.30,2337.00,EK,50,11,42.10,
-0.8600,GM,59.35,-0.86,2967.50,GM,50,16,53.90,
```

This file contains a header line that names the data columns, making processing considerably more reliable. We can use the column titles to create a `dict` for each line of data. By using each data line along with the column titles, we can make our program quite a bit more flexible. This shows a way of handling this kind of well-structured information.

Example 19.4. readportfolio.py

```
quotes=open( "display.csv", "rU" )
titles= quotes.next().strip().split( ',' )
invest= 0
current= 0
for q in quotes:
    values= q.strip().split( ',' )
    data= dict( zip(titles,values) )
    print data
    invest += float(data["Purchase Price"])*float(data["# Shares"])
    current += float(data["Price"])*float(data["# Shares"])
print invest, current, (current-invest)/invest
```

❶ We open our portfolio file, `display.csv`, for reading, creating a file object named `quotes`.

❷ The first line of input, `quotes.next`, is the set of column titles. We strip any extraneous whitespace characters from this line, creating a new `string`. We perform a `split( ',' )` to create a `list` of individual column title `strings`. This `list` is saved in the variable `titles`.

❸ We also initialize two counters, `invest` and `current` to zero. These will accumulate our initial investment and the current value of this portfolio.

❹ We use a **for** statement to iterate through the remaining lines in `quotes` file. Each line is assigned to `q`.

❺ Each stock quote, `q`, is a `string`. We use the `strip` operation to remove excess whitespace characters; the `string` which is created then performs the `split ( ',' )` operation to separate the fields into a `list`. We assign this `list` to the variable `values`.

❻ We create a `dict`, `data`; the column titles in the `titles list` are the keys. The data fields from the current record, in `values` are used to fill this `dict`. The built-in `zip` function is designed for precisely this situation. This function interleaves values from each `list` to create a new `list` of `tuples`. In this case, we will get a sequence of `tuples`, each `tuple` will be a value from `titles` and the corresponding value from `values`. This `list` of 2-`tuples` creates the `dict`.

Now, we have access to each piece of data using it's proper column tile. The

number of shares is in the column titled `"# Shares"`. We can find this information in `data["# Shares"]`.

❼ We perform some simple calculations on each `dict`. In this case, we convert the purchase price to a number, convert the number of shares to a number and multiply to determine how much we spent on this stock. We accumulate the sum of these products into `invest`.

We also convert the current price to a number and multiply this by the number of shares to get the current value of this stock. We accumulate the sum of these products into `current`.

❽ When the loop has terminated, we can write out the two numbers, and compute the percent change.

File Exercises

1. **File Structures.** What is required to process variable length lines of data in an arbitrary (random) order? How is the application program to know where each line begins?

2. **Device Structures.** Some disk devices are organized into cylinders and tracks instead of blocks. A disk may have a number of parallel platters; a cylinder is the stack of tracks across the platters available without moving the read-write head. A track is the data on one circular section of a single disk platter. What advantages does this have? What (if any) complexity could this lead to? How does an application program specify the tracks and sectors to be used?

   Some disk devices are described as a simple sequence of blocks, in no particular order. Each block has a unique numeric identifier. What advantages could this have?

   Some disk devices can be partitioned into a number of "logical" devices. Each partition appears to be a separate device. What (if any) relevance does this have to file processing?

3. **Portfolio Position.** We can create a simple CSV file that contains a description of a block of stock. We'll call this the portfolio file. If we have access to a spreadsheet, we can create a simple file with four columns: stock, shares, purchase date and purchase price. We can save this as a CSV file.

   If we don't have access to a spreadsheet, we can create this file in IDLE. Here's an example line.

   ```
   stock,shares,"Purchase Date","Purchase Price"
   "AAPL", 100, "10/1/95", 14.50
   "GE", 100, "3/5/02", 38.56
   ```

   We can read this file, multiply shares by purchase price, and write a simple report showing our initial position in each stock.

   Note that each line will be a simple string. When we split this string on the ,'s (using the string `split` method) we get a list of strings. We'll still need to convert the number of shares and the purchase price from strings to numbers in order to do the multiplication.

4. **Aggregated Portfolio Position.** In Portfolio Position we read a file and did a simple computation on each row to get the purchase price. If we have multiple blocks of a given stock, these will be reported as separate lines of detail. We'd like to combine (or aggregate) any blocks of stock into an overall position.

   Programmers familiar with COBOL (or RPG) or similar languages often use a Control-Break reporting design which sorts the data into order by the keys, then

reads the lines of data looking for break in the keys. This design uses very little memory, but is rather slow and complex.

It's far simpler to use a Python dictionary than it is to use the Control-Break algorithm. Unless the number of distinct key values is vast (on the order of hundreds of thousands of values) most small computers will fit the entire summary in a simple dictionary.

A program which produces summaries, then, would have the following design pattern.

a. Create an empty dictionary.

b. Read the portfolio file. For each line in the file, do the following.

   i. Create a tuple from the key fields. If there's only one key field, then this value can be the dictionary's key.

   ii. If this key does not exist in the dictionary, insert the necessary element, and provide a suitable initial value. If you're computing one sum, a simple zero will do. If you're computing multiple sums, a `tuple` of zeroes is appropriate.

   iii. Locate the selected value from the dictionary, accumulate new values into it. For the simplest case (one key, one value being accumulated) this looks like `sum[key] =+ value`.

c. Write the dictionary keys and values as the final report.

Some people like to see the aggregates sorted into order. This is a matter of getting the dictionary keys into a list, sorting the list, then iterating through this sorted list to write the final report.

5. **Portfolio Value.** In the section called "Reading a File as a Sequence of Strings", we looked at a simple CSV-format file with stock symbols and prices. This file has the stock symbol and last price, which serves as a daily quote for this stock's price. We'll call this the stock-price file.

   We can now compute the aggregate value for our portfolio by extracting prices from the stock price file and number of shares from the portfolio file.

   If you're familiar with SQL, this is called a join operation; and most databases provide a number of algorithms to match rows between two tables. If you're familiar with COBOL, this is often done by creating a lookup table, which is an in-memory array of values.

   We'll create a dictionary from the stock-price file. We can then read our portfolio, locate the price in our dictionary, and write our final report of current value of the portfolio. This leads to a program with the following design pattern.

   a. Load the price mapping from the stock-price file.

      i. Create an empty stock price dictionary.

      ii. Read the stock price file. For each line in the file, populate the dictionary, using the stock name as the key, and the most recent sale price is the value.

   b. Process the position information from the portfolio file. See Aggregated Portfolio Position and Portfolio Position for the skeleton of this process.

In the case of a stock with no price, the program should produce a "no price quote" line in the output report. It should not produce a `KeyError` exception.

Chapter 20. Advanced Sequences

Table of Contents

This chapter presents some advanced sequence concepts. In the section called "Lists of Tuples" we describe the relatively common Python data structure built from `lists` of `tuples`. We'll cover a powerful `list` construction method called a *list comprehension* in the section called "List Comprehensions". In the section called "Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`" we'll cover the extremely powerful `map`, `filter` and `reduce` functions. In the section called "Advanced List Sorting" we cover some advanced sequence sorting. In the section called "Multi-Dimensional Arrays or Matrices" we cover simple multidimensional sequences.

Even more complex data structures are available. Numerous modules handle the sophisticated representation schemes described in the Internet standards (called Requests for Comments, RFC's). We'll touch on these in Chapter 30, *The Python Library*.

Lists of Tuples

The `list` of `tuple` structure is remarkably useful. In other languages, like Java, we are forced to either use built-in arrays or create an entire class definition to simply keep a few values togther. One common situation is processing list of simple coordinate pairs for 2-dimensional or 3-dimensional geometries. Additional examples might includes list of tuples the contain the three levels for red, green and blue that define a color. Or, for printing, the four-color `tuple` of the values for cyan, magenta, yellow and black.

As an example of using red, green, blue `tuples`, we may have a list of individual colors that looks like the following.

```
colorScheme = [ (0,0,0), (0x20,0x30,0x20), (0x10,0xff,0xff) ]
```

We've already seen how dictionaries (Chapter 15, *Mappings and Dictionaries*) can be processed as a `list` of `tuples`. The `items` method returns the mapping as a `list` of `tuples`. Additionally, the `zip` built-in function interleaves two or more `lists` to create a `list` of `tuples`.

for statement. A interesting form of the **for** statement is one that exploits multiple assignment to work with a `list` of `tuples`. Consider the following example:

```
for c,f in [ ("red",18), ("black",18), ("green",2) ]:
    print "%s occurs %f" % (c, f/38.0)
```

In this program, we have created a `list` of `tuples`. Each `tuple` is a pair with a color and the number of spaces of that color on a roulette wheel. The **for** statement uses a form of multiple assignment to split up each `tuple` into two variables, `c` and `f`. The **print** statement can then work with these variables separately. This is equivalent to the following:

```
for p in [ ("red",18), ("black",18), ("green",2) ]:
```

```
    c,f = p
    print "%s occurs %f" % (c, f/38.0)
```

In this version the **for** statement sets the variable `p` to each `tuple` in the `list`. We then use multiple assignment in a separate statement to split up the `tuple`, `p`, into `c` and `f`.

The `items` method of a `dict` transforms a `dict` to a sequence of `tuples`. We looked at dictionaries in Chapter 15, *Mappings and Dictionaries*.

```
d = { 'red':18, 'black':18, 'green':2 }
for c,f in d.items():
    print "%s occurs %f" % (c, f/38.0)
```

Sorting Lists of Tuples. We often need to sort a `list` of `tuples`. The basic operation of a `list`'s `sort` method is to use the `cmp` function to compare each element of this list. This will compare two `tuples` element-by-element, starting with the first element of the `tuple`. Often, we want to compare elements of the `tuple` in some way other than beginning from the first element in order.

For example, we may have the folloiwng list, which has names and weights. We want the sorted by the second element, weight.

```
[ ('steve',180), ('xander',190), ('hannah',110), ('cindy',140) ]
```

We'll look at this in depth in the section called "Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`".

List Comprehensions

Python provides a sophisticated mechanism to build a `list` called a *list comphrehension* or *list maker*. A list comprehension is a single expression that combines an expression, **for** statement and an optional **if** statement. This allows a simple, clear expression of the processing that will build up the values of a `list`.

The basic syntax follows the syntax for a `list` literal. It encloses the processing in `[]`'s. Here's the simplest form, omitting the optional **if** clause.

[***expr for-clause*** **]**

The *`for-clause`* mirrors the **for** statement:

for ***v*** **in** ***sequence***

Here are some examples.

```
even = [ 2*x for x in range(18) ]
hardways = [ (x,x) for x in (2,3,4,5) ]
samples = [ random.random() for x in range(10) ]
```

A list comprehension behaves like the following loop:

```
r= []
for v in sequence:
    r.append( expr )
```

The basic process, then, is to iterate through the sequence in the *`for-clause`*, evaluating the expression, *`expr`*. The values that result are assembled into the `list`. If the expression depends on the *`for-clause`*, each value in the `list` can be different. If the expression doesn't depend on the *`for-clause`*, each value will be the same.

Here's an example where the expression depends on the for-clause.

```
>>> a= [ v*2+1 for v in range(10) ]
>>> a
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

This creates the first 10 odd numbers. It starts with the sequence created by `range(10)`. The *for-clause* at assigns each value in this sequence to the local variable `v`. The expression, v*2+1, is evaluated for each distinct value of `v`. The expression values are assembled into the resulting `list`.

Typically, the expression depends on the variable set in the *for-clause*. Here's an example, however, where the expression doesn't depend on the *for-clause*.

```
b= [ 0 for i in range(10) ]
```

This creates a `list` of 10 zeroes. Because the expression doesn't depend on the *for-clause*, this could also be done as

```
b= 10*[0]
```

A comprehension can also have an *if-clause*.

The basic syntax is as follows:

[*expr for-clause if-clause*]

The *for-clause* mirrors the **for** statement:

for *v* in *sequence*

The *if-clause* mirrors the **if** statement:

if *filter*

Here is an example.

```
hardways = [ (x,x) for x in range(1,7) if x+x not in (2, 12) ]
```

This more complex list comprehension behaves like the following loop:

```
r= []
for v in sequence:
    if filter:
        r.append( expr )
```

The basic process, then, is to iterate through the sequence in the *for-clause*, evaluating *the if-clause*. When the *the if-clause* is true, evaluate the expression, *expr*. The values that result are assembled into the `list`.

```
>>> v = [ (x,2*x+1) for x in range(10) if x%3==0 ]
>>> v
[(0, 1), (3, 7), (6, 13), (9, 19)]
```

This works as follows:

1. The `range(10)` creates a sequence of 10 values.
2. The *for-clause* iterates through the sequence, assigning each value to the local variable *x*.
3. The *if-clause* evaluates the filter function, `x%3==0`. If it is false, the value is skipped. If it is true, the expression, at `(x,2*x+1)`, is evaluated.
4. This expression creates a 2-`tuple` of the value of *x* and the value of 2**x*+1.
5. The expression results (a sequence of `tuples`) are assembled into a `list`, and

assigned to `v`.

A list comprehension can have any number of *`for-clauses`* and *`if-clauses`*, in any order. A *`for-clause`* must be first. The clauses are evaluated from left to right.

Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`

The `map`, `filter`, and `reduce` built-in functions are handy functions for processing sequences. These owe much to the world of functional programming languages. The idea is to take a small function you write and apply it to all the elements of a sequence. This saves you writing an explicit loop. The implicit loop within each of these functions may be faster than an explicit **for** or **while** loop.

Additionally, each of these is a pure function, returning a result value. This allows the results of the functions to be combined into complex expressions relatively easily.

It is common to have multiple-step processes on `lists` of values. For instance, filtering a large set of data to locate useful samples, transforming those samples and then computing a sum or average. Rather than write three explicit loops, the result can be computed in a single expression.

Let's say we have two sequences and we have a multi-step process like this:

1. for i in seq1:.... apply function f1 and create sequence r1
2. for i in seq2:.... apply function f2 and create sequence r2
3. for i in len(r1):.... apply function f3 to r1[i] and r2[i]

Instead of these lengthy explicit loops, we can take a functional approach that look like this:

```
f= reduce( f3, zip( map(f1,seq1), map(f2,seq2) ) )
```

Where `f1`, `f2`, and `f3` are functions that define the body of the each of the above loops.

Definitions. These functions transform lists. The `map` and `filter` each apply some function to a sequence to create a new sequence. The `reduce` function applies a function which will reduce the sequence to a single value. The `zip` function interleaves values from lists to create a list of tuples.

The `map`, `zip` and `filter` functions have no internal state, they simply apply the function to each individual value of the sequence. The `reduce` function, in contrast, maintains an internal state which is seeded from an initial value, passed to the function along with each value of the sequence and returned as the final result.

Here are the formal definitions.

`map` (*`function`*, *`sequence`*, [*`sequence...`*]) → list

> Create a new `list` from the results of applying the given *`function`* to the items of the the given *`sequence`*. If more than one sequence is given, the function is called with multiple arguments, consisting of the corresponding item of each sequence. If any sequence is too short, `None` is used for missing value. If the *`function`* is `None`, `map` will create tuples from corresponding items in each list, much like the `zip` function.

`filter` (*`function`*, *`sequence`*) → list

> Return a `list` containing those items of *`sequence`* for which `function` (*`item`*) is true. If *`function`* is `None`, return a `list` of items that are equivalent to `True`.

`reduce` (*`function`*, *`sequence`*, [*`initial`*]) → value

Apply a *function* of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If *initial* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

`zip` (*seq1*, [*seq2*, ...]) → [(*seq1*[0], *seq2*[0],...), ...]

Return a `list` of `tuples`, where each `tuple` contains the matching element from each of the argument sequences. The returned `list` is truncated in length to the length of the shortest argument sequence.

Costs and Benefits. What are the advantages? First, the functional version can be clearer. It's a single line of code that summarizes the processing. Second, and more important, Python can execute the sequence processing functions far faster than the equivalent explicit loop.

You can see that `map` and `filter` are equivalent to simple list comprehensions. This gives you two ways to specify these operations, both of which have approximately equivalent performance. This means that `map` and `filter` aren't *essential* to Python, but hey are widely used.

The `reduce` function is a bit of a problem. It can have remarkably bad performance if it is misused. Consequently, there is some debate about the value of having this function. We'll present the function along with the caveat that it can lead to remarkable slowness.

The `map` Function. The `map` function transforms a sequence into another sequence by applying a function to each item in the sequence. The idea is to apply a mapping tranformation to a sequence. This is is a common design pattern within numerous kinds of programs. Generally, the transformation is the interesting part of the programming, and the loop is just another boring old loop.

The function call `map`(*aFunction*, *aSequence*) behaves as if it had the following definition.

```
def map( aFunction, aSequence ):
    return [ aFunction(v) for v in aSequence ]
```

For example:

```
>>> map( int, [ "10", "12", "14", 3.1415926, 5L ] )
[10, 12, 14, 3, 5]
```

This applies the `int` function to each element of the input sequence (a `list` that contains some `strings`, a floating point value and a long integer value) to create the output sequence (a `list` of integers).

The function used in `map` can be a built-in function, or a user-defined function created with the **def** statement (see Chapter 9, *Functions*).

```
>>> def oddn(x):
...     return x*2+1
...
>>> map( oddn, range(6) )
[1, 3, 5, 7, 9, 11]
```

This example defines a function `oddn`, which creates an odd number from the input. The `map` function applies our `oddn` function to each value of the sequence created by `range`(*6*). We get the first 6 odd numbers.

The `filter` Function. The `filter` function chooses elements from the input sequence

where a supplied function is `True`. Elements for which the supplied function is `False` are discarded.

The function call `filter` (*`aFunction`*, *`aSequence`*) behaves as if it had the following definition.

```
def filter( aFunction, aSequence ):
    return [ v for v in aSequence if aFunction(v) ]
```

For example:

```
>>> def gt2( a ):
...     return a > 2
...
>>> filter( gt2, range(8) )
[3, 4, 5, 6, 7]
```

This example uses a function, `gt2`, which returns `True` for inputs greater than 2. We create a range from 0 to 7, but only keep the values for which the filter function returns `True`.

Here's another example that keeps all numbers that are evenly divisible by 3.

```
>>> def div3( a ):
...     return a % 3 == 0
...
>>> filter( div3, range(10) )
[0, 3, 6, 9]
```

Our function, `div3`, returns `True` when the remainder of dividing a number by 3 is exactly 0. This will keep numbers that are evenly divisible by 3. We create a range, apply the function to each value in the range, and keep only the values where the filter is `True`.

The `reduce` Function. The `reduce` function can be used to implement the common spread-sheet functions that compute sums and products. This function works by seeding a result with an initial value. It then calls the user-supplied function with this result and each value of the sequence. This is remarkably common, but it can't be done as a list comprehension.

The function call `reduce` (*`aFunction`*, *`aSequence`* , [, *`init`*]) behaves as if it had the following definition.

```
def reduce( aFunction, aSequence, init= 0 ):
    r= init
    for s in aSequence:
        r= aFunction( r, s )
    return r
```

The important thing to note is that the function is applied to the internal value, `r`, and each element of the list to compute a new internal value.

For example:

```
>>> def add(a,b):
...     return a+b
...
>>> reduce( add, range(10) )
45
```

This expression computes the sum of the 10 numbers from zero through nine. The function we defined, `add`, adds the previous result and the next sequence value together.

Here's an interesting example that combines `reduce` and `map`. This uses two functions defined in earlier examples, `add` and `oddn`.

```
for i in range(10):
    sq=reduce( add, map(oddn, range(i)), 0 )
    print i, sq
```

Let's look at the evaluation from innermost to outermost. The `range(`*i*`)` generates a `list` of numbers from 0 to `i`-1. The `map` applies the `oddn` function form to create a sequence of `i` odd numbers from 1 to 2`i`+1. The `reduce` then adds this sequence of odd numbers. Interestingly, these sums add to the square of `i`.

The `zip` Function. The `zip` function interleaves values from two sequences to create a new sequence. The new sequence is a sequence of `tuples`. Each item of a `tuple` is the corresponding values from from each sequence.

```
>>> zip( range(5), range(1,20,2) )
[(0, 1), (1, 3), (2, 5), (3, 7), (4, 9)]
```

In this example, we zipped two sequences together. The first sequence was `range(5)`, which has five values. The second sequence was range(1,20,2) which has 10 odd numbers from 1 to 19. Since `zip` truncates to the shorter list, we get five tuples, each of which has the matching values from both lists.

The `map` function behaves a little like `zip` when there is no function provided, just sequences. However, `map` does not truncate, it fills the shorter list with `None` values.

```
>>> map( None, range(5), range(1,20,2) )
[(0, 1), (1, 3), (2, 5), (3, 7), (4, 9), (None, 11), (None, 13), (None, 15),
(None, 17), (None, 19)]
```

Advanced List Sorting

Consider a `list` of `tuples`. We could get such a `list` when processing information that was extracted from a spreadsheet program. For example, if we had a spreadsheet with raw census data, we can easily transform it into a sequence of `tuples` that look like the following.

```
jobData= [
(001,'Albany','NY',162692),
(003,'Allegany','NY',11986),
...
(121,'Wyoming','NY',8722),
(123,'Yates','NY',5094)
]
```

Each `tuple` can be built from a row of the spreadsheet. In this case, we wrote a simple forumla in our spreadsheet to make each row into a `tuple`. We could have used the `csv` module to read a version of spreadsheet saved as a `.csv` file, but cutting and pasting is pretty quick, also.

Once we have each row as a `tuple`, we can put some `[]`'s around the `tuples` to make a `list`. We can then slap an assignment statement around this `list` of rows and turn our spreadsheet into a Python statement.

Sorting this `list` can be done trivially with the `list` `sort` method.

```
jobData.sort()
```

Note that this updates the `list` in place. The `sort` method specifically does not return a result. A common mistake is to say something like: `a= b.sort()`. This always sets the variable `a` to `None`.

This kind of sort will simply compare each `tuple` with each other `tuple`. While easy to use in this form, it doesn't permit more sophisticated sorting. Let's say we wanted to sort by state name, the third element in the `tuple`. We have two strategies for sorting when we don't want the simplistic comparison of elements in order.

1. We can provide a compare function to the `sort` method of a list. This compare function must have the same signature as the built-in `cmp` function, but does the comparison we want.

2. We can provide a "key extraction" function to the `sort` method. This will locate the key value (or a `tuple` of key values) within the given objects.

3. We can decorate each element in the list, making it into a new kind of 2-tuple with the fields on which we want to sort as the first element of this `tuple` and the original data as the second element of the `tuple`.

Sorting With a Compare Function. The `sort` method of a list can accept a comparison function. While this is a very general solution, it is also relatively low performance because of the overheads involved.

We must define a function that behaves like the built-in `cmp` function. In our example, we'll define a comparison which works with the third element of our `jobData tuple`.

```
def sort3( a, b ):
    return cmp( a[2], b[2] )
jobData.sort( sort3 )
```

Note that we pass the function object to the `sort` method. A common mistake is to say `jobData.sort( sort3() )`. If we include the ()'s, we call the function `sort3` once, invoking the eval-apply process. We don't want to call the function once: we want to provide the function to `sort`, so that `sort` can call the function as many times as needed to sort the `list`.

Another common process is to sort information by several key fields. Continuing this example, lets sort the `list` by state name and then number of jobs. This is sometimes called a multiple-key sort. We want our data in order by state. When the states are equal, we want to use the area code to sort the data.

This can be done in two ways. The most common technique is to create a tuple of the various key fields. The other way is to make use of the **or** operator.

Since tuples are compared element-by-element, we can create a `tuple` of the various key fields and turn the built-in `cmp` function loose on this `tuple`. This makes good use of Python's built-in features.

The formal definition for **or** is given in the section called “Truth and Logic”. If the first argument is not true, the second argument must be evaluated. The `cmp` function returns 0 when elements are equal; we can use this to do a series of comparisons. Then the first fields are equal, we can then compare the next field.

```
def cmpStJob1( a, b ):
    aKey= ( a[2], a[3] )
    bKey= ( b[2], b[3] )
    return cmp( aKey, bKey )

def cmpStJob2( a, b ):
    return cmp( a[2], b[2] ) or cmp( a[3], b[3] )
```

Sorting With Key Extraction. The `sort` method of a list can accept a keyword parameter, *`key`*, that provides a key extraction function. This function returns a value which can be used for comparison purposes. To sort our jobData by the third field, we

can use a function like the following.

```
def byState( a ):
    return a[2]

jobData.sort( key=byState )
```

This byState function returns the selected key value, which is then used by sort to order the tuples in the original list. If we want to sort by a multi-part key, we cna do something like the following.

```
def byStateJobs( a ):
    return ( a[2], a[3] )
```

This function will create a two-value `tuple` and use these two values for ordering the items in the `list`.

Sorting With List Decoration. Superficially, this method appears more complex. However it is remarkably flexible, allowing you to combine `sort`, `map` and `filter` operations into a single statement. The idea is to transform the initial `list` of values into a new `list` of 2-tuples, with the first item being the key and the second item being the original `tuple`. The first item, used for sorting, is a decoration placed in front of the original value.

In this example, we decorate our values with a 2-tuple of state names and number of jobs. We can sort this temporary list of 2-tuples. Then we can strip off the decoration and recover the original values.

```
deco= [ ((a[2],a[3]),a) for a in jobData ]
deco.sort()
sorted= [ v for k,v in deco ]
```

When constructing the keys we can do `map`- or `filter`-like operations. SImilarly, we can also to additional calculations when we strip off the decorations.

Multi-Dimensional Arrays or Matrices

There are situations that demand multi-dimensional arrays or matrices. In many languages (Java, COBOL, BASIC) this notion of multi-dimensionality is handled by pre-declaring the dimensions (and limiting the sizes of each dimension). In Python, these are handled somewhat more simply.

If you have a need for more sophisticated processing than we show in this section, you'll need to get the Python `Numeric` module, also known as `NumPy`. This is a Source Forge project, and can be found at `http://numpy.sourceforge.net/`.

Let's look at a simple two-dimensional tabular summary. When rolling two dice, there are 36 possible outcomes. We can tabulate these in a two-dimensional table with one die in the rows and one die in the columns:

	1	**2**	**3**	**4**	**5**	**6**
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

In Python, a multi-dimensional table like this can be implemented as a sequence of sequences. A table is a sequence of rows. Each row is a sequence of individual cells.

This allows us to use mathematical-like notation. Where the mathematician might say $A_{i,j}$, in Python we can say `A[i][j]`. In Python, we want the row `i` from table A, and column `j` from that row.

This looks remarkably like the `list` of `tuples` we discussed in the section called "Lists of Tuples".

List of Lists Example. We can build a table using a nested list comprehension. The following example creates a table as a sequence of sequences and then fills in each cell of the table.

```
table= [ [ 0 for i in range(6) ] for j in range(6) ]
print table
for d1 in range(6):
    for d2 in range(6):
        table[d1][d2]= d1+d2+2
print table
```

This program produced the following output.

```
[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
[[2, 3, 4, 5, 6, 7], [3, 4, 5, 6, 7, 8], [4, 5, 6, 7, 8, 9],
[5, 6, 7, 8, 9, 10], [6, 7, 8, 9, 10, 11], [7, 8, 9, 10, 11, 12]]
```

This program did two things. It created a six by six table of zeroes. It then filled this with each possible combination of two dice. This is not the most efficient way to do this, but we want to illustrate several techniques with a simple example. We'll look at each half in detail.

The first part of this program creates and prints a 6-item `list`, named `table`; each item in the table is a 6-item `list` of zeroes. It uses a list comprehension to create an object for each value of `j` in the range of 0 to 6. Each of the objects is a `list` of zeroes, one for each value of `i` in the range of 0 to 6. After this initialization, the two-dimensional table of zeroes is printed.

The comprehension can be read from inner to outer, like an ordinary expression. The inner `list`, `[ 0 for i in range(6) ]`, creates a simple `list` of six zeroes. The outer `list`, `[ [...] for j in range(6) ]` creates six copies of these inner `lists`.

The second part of this program then iterates over all combinations of two dice, filling in each cell of the table. This is done as two nested loops, one loop for each of the two dice. The outer enumerates all values of one die, `d1`. The loop enumerates all values of a second die, `d2`.

Updating each cell involves selecting the row with `table[d1]`; this is a `list` of 6 values. The specific cell in this `list` is selected by `...[d2]`. We set this cell to the number rolled on the dice, `d1+d2+2`.

Additional Examples. The printed `list` of `lists` is a little hard to read. The following loop would display the table in a more readable form.

```
>>> for row in table:
...     print row
...
[2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9, 10]
[6, 7, 8, 9, 10, 11]
[7, 8, 9, 10, 11, 12]
```

As an exercise, we'll leave it to the reader to add some features to this to print column and row headings along with the contents. As a hint, the `"%2d" % value` `string` operation might be useful to get fixed-size numeric conversions.

Explicit Index Values. We'll summarize our table of die rolls, and accumulate a frequency table. We'll use a simple list with 13 buckets (numbered from 0 to 12) for the frequency of each die roll. We can see that the die roll of 2 occurs just once in our matrix, so we'll expact that `fq[2]` will have the value 1. Let's visit each cell in the matrix and accumulate a frequency table.

There is an alternative to this approach. Rather than strip out each row sequence, we could use explicit indexes and look up each individual value with an integer index into the sequence.

```
fq= 13*[0]
for i in range(6):
    for j in range(6):
        c= table[i][j]
        fq[ c ] += 1
```

We initialize the frequency table, `fq`, to be a `list` of 13 zeroes.

The outer loop sets the variable `i` to the values from 0 to 5. The inner loop sets the variable `j` to the values from 0 to 5.

We use the index value of `i` to select a row from the table, and the index value of `j` to select a column from that row. This is the value, `c`. We then accumulate the frequency occurances in the frequency table, `fq`.

This looks very mathematical and formal. However, Python gives us an alternative, which can be somewhat simpler.

Using List Iterators Instead of Index Values. Since our table is a list of lists, we can make use of the power of the **for** statement to step through the elements without using an index.

```
fq= 13*[0]
print fq
for row in table:
    for c in row:
        fq[c] += 1
print fq[2:]
```

We initialize the frequency table, `fq`, to be a `list` of 13 zeroes.

The outer loop sets the variable `row` to each element of the original `table` variable. This decomposes the table into individual rows, each of which is a 6-element `list`.

The inner loop sets the variable `c` to each column's value within the row. This decomposes the row into the individual values.

We count the actual occurances of each value, `c` by using the value as an index into the frequency table, `fq`. The increment the frequency value by 1.

Mathematical Matrices. We use the explicit index technique for managing the mathematically-defined matrix operations. Matrix operations are done more clearly with this style of explicit index operations. We'll show matrix addition as an example, here, and leave matrix multplication as an exercise in a later section.

```
m1 = [ [1, 2, 3, 0], [4, 5, 6, 0], [7, 8, 9, 0] ]
m2 = [ [2, 4, 6, 0], [1, 3, 5, 0], [0, -1, -2, 0] ]
m3= [ 4*[0] for i in range(3) ]
```

```
for i in range(3):
    for j in range(4):
        m3[i][j]= m1[i][j]+m2[i][j]
```

In this example we created two input matrices, `m1` and `m2`, each three by four. We initialized a third matrix, `m3`, to three rows of four zeroes, using a comprehension. Then we iterated through all rows (using the `i` variable), and all columns (using the `j` variable) and computed the sum of `m1` and `m2`.

Python provides a number of modules for handling this kind of processing. In Part IV, "Components, Modules and Packages" we'll look at modules for more sophisticated matrix handling.

The Lambda

The functions used by `map`, `filter`, `reduce` and a `list`'s `sort` method can also be a special function called a *lambda form*. This permits us to use special one-use-only throw-away functions without the overhead of a **def** statement. A lambda form is like a defined function: it has parameters and computes a value. The body of a lambda, however, can only be a single expression, limiting it to relatively simple operations. If it gets complex, you'll have to define a real function.

Generally, it's clearer to formally define a function rather than try to define a lambda form.

We can play with lambda forms by applying them directly as functions to arguments.

```
>>> from math import pi
>>> print (lambda x: pi*x*x)(5)
78.53981633970
```

This statement creates a **lambda** that accepts a single argument, named *x*, and computes `pi*x*x`. This lambda is applied to an argument of 5. It computes the area of a circle with a radius of 5.

Here's a lambda form used in the map function.

```
>>> print map( lambda x: pi*x*x, range(8) )

[0.0, 3.1415926535897931, 12.566370614359172, 28.274333882308138,
50.26548245743669, 78.539816339744831, 113.09733552923255,
153.93804002589985]
```

This `map` function applies our radius-computing lambda form to the values from 0 to 7 as created by the `range`. The input sequence is mapped to the output sequence by having the lambda function applied to each value.

Parameterizing a Lambda. Sometimes we have a lambda which -- in effect -- has two kinds of parameters: parameters that are elements of a sequence being processed by `map`, `filter` or `reduce` function, and parameters that are more global than the items of a sequence.

Consider this more complex example.

```
spins = [ (23,"red"), (21,"red"), (0,"green"), (24,"black") ]
betOn= "black"
print filter( lambda x, y=betOn: y in x, spins )
betOn= "red"
print filter( lambda x, y=betOn: y in x, spins )
```

First, we create four sample spins of a roulette wheel, and save this `list` in a variable called `spins`. Then we chose a particular bet, saving this in a variable called `betOn`. If the given `betOn` keyword occurs in any of the `tuples` that describe all of the spins, the `tuple` is kept.

The call to `filter` has a lambda form that uses a common Python hack. The `filter` function only passes a single argument value to the function or lambda form. If there are additional parameters declared, they must have default values; in this case, we set the default to the value of our variable, `betOn`.

Let's work through this in a little more detail.

As the `filter` function executes, it enumerates each element from the sequence, as if it had a **for `s` in `spins`:** clause. Each individual item is given to the **lambda**. The **lambda** then does the evaluation of `y in x`; `x` is a `tuple` from the `list` (e.g., (23, "red")) and `y` is a default parameter value, set to the value of `betOn` (e.g., "black"). Every time the `y` value actually appears in the `tuple` `x`, (`("black" in (24,"black")`), the `tuple` is selected to create the resulting `list` from the filter. When the `y` value is not in the `tuple` `x`, the `tuple` ignored

This default parameter hack is required because of the way that Python maintains only two execution contexts: local and global. The lambda's execution takes place in a fresh local context with only its two local parameter variables, `x` and `y`; it doesn't have access to global variables. When the lambda is created, the creation happens in the context where the `betOn` variable is known. So we provide the extra, global parameters as defaults when the lambda is created.

As an alternative to creating `lists` with the `filter` function, similar results can be created with a list comprehension. This is covered just after the following material on `reduce`.

Exercises

1. **All Dice Combinations.** Write a list comprehension that uses nested for-clauses to create a single `list` with all 36 different dice combinations from (1,1) to (6,6).

2. **Temperature Table.** Writea list comprehension that creates a `list` of `tuples`. Each `tuple` has two values, a temperature in Farenheit and a temperature in Celsius.

   Create one `list` for Farenheit values from 0 to 100 in steps of 5 and the matching Celsius values.

   Create another `list` for Celsius values from -10 to 50 in steps of 2 and the matching Farenheit values.

3. **Define max() and min().** Use `reduce` to create versions of the built-ins `max` and `min`.

   You may find this difficult to do this with a simple lambda form. However, consider the following. We can pick a value from a `tuple` like this: `(a,b)[0] == a`, and `(a,b)[1] == b`. What are the values of `(a,b)[a<b]` and `(a,b)[a>b]`?

4. **Compute the Average or Mean.** A number of standard descriptive statistics can be built with `reduce`. These include mean and standard deviation. The basic formulae are given in Chapter 13, *Tuples*.

   Mean is a simple “add-reduction” of the values in a sequence divided by the length.

5. **Compute the Variance and Standard Deviation.** A number of standard

descriptive statistics can be built with `reduce`. These include mean and standard deviation. The basic formulae are given in Chapter 13, *Tuples*.

The standard deviation has a number of alternative definitions. One approach is to sum the values and square this number, as well as sum the squares of each number. Summing squares can be done as a `map` to compute squares and then use a sum function based on `reduce`. Or summing squares can be done with a special `reduce` that both squares and sums.

Also the standard deviation can be defined as the square root of the variance, which is computed as:

Procedure 20.1. Variance of a sequence *a*

1. **Mean.** m ← mean(a)
2. **Total Variance.** s ← sum of $(a[i] - m)^2$ for all i
3. **Average Variance.** divide s by n–1

6. **Compute the Mode.** The mode function finds the most common value in a data set. This can be done by computing the frequency with which each unique value occurs and sorting that `list` to find the most common value. The frequency distribution is easiest done using a mapping, something we'll cover in the next chapter. This can be simplified also using the advanced `list` sorting in the next section of this chapter.

   Procedure 20.2. Mode of a sequence, *a*

   1. **Initialization**

      `fqList` ← empty `list`

   2. **For each value, `v` in *a***

      a. If `v` is element 0 of one of the `tuples` of `fqList`, then

         Get the frequency, `f`, element 1 of the `tuple`.

         Remove the `tuple` (`v`,`f`) from `fqList`.

         Create a new `tuple` (`v`,`f`+1).

         Add the new `tuple` to the `fqList`.

      b. If `v` is not element 0 of one of the `tuples` of `fqList`, then

         Create a new `tuple` (`v`,1).

         Add the new `tuple` to the `fqList`.

   3. Save `tuple` 0 of the `fqList` as the largest `tuple`, `maxFq`.

   4. **For each frequency, `t` in `fqList`**

      a. If `t`'s frequency is larger than the frequency of `maxFq`, then

         `maxFq` ← `t`.

   5. Return `maxFq` as the modal value and the frequency with which it occurs.

7. **Compute the Median.** The median function arranges the values in sorted order. It

locates either the mid-most value (if there are an odd number) or it averages two adjacent values (if there are an even number).

If `len(data) % 2 == 1`, there is an odd number of values, and `(len(data)+1)/2` is the midmost value. Otherwise there is an even number of values, and the `len(data)/2` and `len(data)/2-1` are the two mid-most values which must be averaged.

8. **Unique Values In A Sequence.** In Accumulating Unique Values, we looked at accumulating the unique values in a sequence. Sorting the sequence leads to a purely superficial simplification. Sorting is a relatively expensive operation, but for short sequences, the cost is not much higher than the version already presented.

   Given an input sequence, `seq`, we can easily sort this sequence. This will put all equal-valued elements together. The comparison for unique values is now done between adjacent values, instead of a lookup in the resulting sequence.

   Procedure 20.3. Unique Values of a Sequence, *`seq`*, using `sort`

   1. **Initalize**

      set `result` ← an empty sequence.

      Sort the input sequence, *`seq`*.

   2. **Loop.** For each value, `v`, in *`seq`*.

      a. **Already in `result`?** Is v the last element in `result`? If so, ignore it. If not, append `v` to the sequence `result`.

   3. **Result.** Return array `result`, which has unique values from *`seq`*.

9. **Portfolio Reporting.** In Blocks of Stock, we presented a stock portfolio as a sequence of `tuples`. Plus, we wrote two simple functions to evaluate purchase price and total gain or loss for this portfolio.

   Develop a function (or a lambda form) to sort this porfolio into ascending order by current value (current price * number of shares). This function (or lambda) will require comparing the products of two fields instead of simply comparing two fields.

10. **Matrix Formatting.** Given a 6×6 matrix of dice rolls, produce a nicely formatted result. Each cell should be printed with a format like `"| %2s"` so that vertical lines separate the columns. Each row should end with an '|'. The top and bottom should have rows of "----"'s printed to make a complete table.

11. **Three Dimensions.** If the rolls of two dice can be expressed in a two-dimensional table, then the rolls of three dice can be expressed in a three-dimensional table. Develop a three dimensional table, 6 x 6 x 6, that has all 216 different rolls of three dice.

    Write a loop that extracts the different values and summarizes them in a frequency table. The range of values will be from 3 to 18.

Data + Processing = Objects

Combining Data and Processing into Class Definitions

In Part I, "Language Basics", we examined the core statements in the Python language. In Part II, "Data Structures", we examined the built-in data structures available to us as

programmers. Using these data structures gave us some hands-on experience with a number of classes. After using this variety of built-in objects, we are better prepared to design our own objects.

Chapter 21, *Classes* introduces basics of class definitions and Chapter 22, *Advanced Class Definition* introduces simple inheritance. We extend this discussion further to include several common design patterns that use polymorphism. In Chapter 23, *Some Design Patterns* we cover a few common design patterns. Chapter 24, *Creating or Extending Data Types* describes the mechanism for adding types to Python that behave like the built-in types.

We will spend some additional time on Python's flexible notion of "attribute" in Chapter 25, *Properties and Descriptors*. It turns out that an attribute can be a simple instance variable or it can be a method function that manages an instance variable.

We'll look at Python's decorators in Chapter 26, *Decorators*; this is handy syntax for assuring that specific aspects of a family of classes are implemented consistently. We'll look at the various ways to define properties in objects.properties. Additionally, we'll look how we can manage more sophisticated object protocols in Chapter 27, *Managing Contexts: the **with** Statement*.

Data Types. We've looked at most of these data types in Part II, "Data Structures". This is a kind of road-map of some of the most important built-in features.

- **None.** A unique constant, handy as a placeholder when no other value is appropriate. A number of built-in functions return values of `None` to indicate that no useful work can be done.
- **NotImplemented.** A unique constant, returned by special methods to indicate that the method is not implemented. See the section called "Special Method Names" for more information.
- Numeric types have relatively simple values.
    - **Boolean.** A variety of values are treated as logically false: `False`, 0, `None`, `""`, `()`, `[]`, `{}`. All other values are logically `True`.
    - **Integer.** Typically 32-bit numbers with a range of -2,147,483,648 through 2,147,483,647. On some platforms, these may be 64-bit numbers.
    - **Long.** These are specially coded integers of arbitrary length. They grow as needed to accurately represent numeric results.
    - **Float.** These are floating point, scientific notation numbers. They are represented using the platform's floating point notation, so ranges and precisions vary. Typically these are called "double precision" in other languages, and are often 64-bits long.
    - **Complex.** These are a pair of floating point numbers of the form (*a*+*b*j), where *a* is the real part and *b* is the "imaginary" part.
- **Sequence.** Collections of objects identified by their order or position.
    - Immutable sequences are created as needed and can be used but never changed.
        - **String.** A string is a sequence of individual ASCII characters.
        - **Unicode.** A Unicode string is a sequence of individual Unicode characters.

        - **Tuple.** A tuple is a simple comma-separated sequence of Python items in `()`'s.

    - Mutable sequences can be created, appended-to, changed, and have elements deleted

        - **List.** A list is a comma-separated sequence of Python items in `[]`'s. Operations like `append` and `pop` can be used to change lists.

- **Set and Frozenset.** Collections of objects. The collection is neither ordered nor keyed. Each item stands for itself. A `set` is mutable; we can append-to, change and delete elements from a set. A `frozenset` is immutable.

- **Mapping.** Collections of objects identified by keys instead of order.

    - **Dictionary.** A dictionary is a collection of objects which are indexed by other objects. It is like a sequence of `key:value` pairs, where keys can be found efficiently. Any Python object can be used as the value. Keys have a small restriction: mutable lists and other mappings cannot be used as keys. It is created with a comma-separated list of `key:value` pairs in `{}`'s.

- **Callable.** When we create a function with the **def** statement, we create a `callable` object. We can also define our own classes with a special method of `__call__`, to make a callable object that behaves like a function.

- **File.** Python supports several operations on files, most notably reading, writing and closing. Python also provides numerous modules for interacting with the operating system's management of files.

There are numerous additional data structures that are part of Python's implementation internals; they are beyond the scope of this book.

One of the most powerful and useful features of Python, is its ability to define new *classes*. The next chapters will introduce the class and the basics of object-oriented programming.

Table of Contents

Chapter 21. Classes

Table of Contents

Object-oriented programming permits us to organize our programs around the interactions of *objects*. A *class* provides the definition of the structure and behavior of the objects; each object is an instance of a class. Consequently, a typical program is a number of class definitions and a final main function. The main function creates the objects that will perform the job of the program.

This chapter presents the basic techniques of defining classes. In the section called "Semantics" we define the semantics of objects and the classes which define their attributes (instance variables) and behaviors. In the section called "Class Definition: the **class** Statement" we show the syntax for creating class definitions; we cover the use of objects in the section called "Creating and Using Objects". Python has some system-defined names that classes can exploit to make them behave like built-in Python classes, a few of these are introduced in the section called "Special Method Names". We provide some examples in the section called "Some Examples". Perhaps the most important part of working with objects is how they collaborate to do useful work; we introduce this in the section called "Object Collaboration".

Semantics

Object-oriented programming focuses software design and implementation around the definitions of and interactions between individual objects. An object is said to *encapsulate* a state of being and a set of behaviors; it is both data and processing. Each instance of a class has individual copies of attributes which are tightly coupled with the class-wide operations. We can understand objects by looking at four features, adapted from [Rumbaugh91].

- **Identity.** Each object is unique and is distinguishable from all other objects. In the real world, two identical coffee cups occupy different locations on our desk. In the world of a computer's memory, objects could be identified by their address, which would make them unique.

- **Classification.** This is sometimes called *Encapsulation*. Objects with the same attributes and behavior belong to a common *class*. Each individual object has unique attribute values. We saw this when we looked at the various collection

classes. Two different `list` objects have the same general structure, and the same behavior. Both lists respond to `append`, `pop`, and all of the other methods of a `list`. However, each `list` object has a unique sequence of values.

- **Inheritance.** A class can inherit methods from a parent class, reusing common features. A superclass is more general, a subclass overrides superclass features, and is more specific. With the built-in Python classes, we've looked at the ways in which all immutable sequences are alike.

- **Polymorphism.** A general operation can have variant implementation methods, depending on the class of the object. We saw this when we noted that almost every class on Python has a **+** operation. Between two floating-point numbers the **+** operation adds the numbers, between two lists, however, the **+** operation concatenates the lists.

Python's Implementation. A *class* is the Python-language definition of the features of individual *objects*: the names of the *attributes* and definitions of the *operations*. The generic notion of attribute is implemented in the instance variables for an object. The general notion of operation is implemented in the methods or method functions of an object. All Python objects are instances of some class.

Additionally, a class also constructs new object instances for us. Once we've defined the class, we can then use it as a kind of factory to create new objects.

Python class definitions require us to provide a number of things.

- We provide a distinct name to the class. We define the behavior of each object by defining its method functions. The attributes of each object are called instance variables and are created by an initialization method function when the object is created.

- We list the superclasses from which a subclass inherits features. This is called multiple inheritance and differs from the single-inheritance approach used by languages like Java.

- We provide method functions which define the operations for the class.

- We can define attributes as part of the class definition. Generally, however, our attributes should be instance variables, which are created dynamically as the methods of the class are evaluated. Most commonly, we provide an initialization method function (named `__init__`) which creates our instance variables.

- Python provides a simple version of unique identify.

Note that our instance variables are not a formal part of the class definition. This is because Python is a dynamic language; Python doesn't rely on static declarations of instance variables the way C++ or Java do.

Another consequence of Python's dynamic nature is that polymorphism is based on simple matching of method names. This is distinct from languages like Java or C++ where polymorphism depends on inheritance. Python's approach is sometimes called “duck typing”: if it quacks like a duck and walks like a duck it is a duck. If several objects have the common method names, they are effectively polymorphic with respect to those methods.

Interface and Attributes. The best programming practice is to treat each object as if the internal implementation details where completely opaque. All other objects within an application should use only the interface methods and attributes for interacting with an object. Some languages (like C++ or Java) have a formal distinction between interface and implementation. Python has a limited mechanism for making a distinction between the defined interface and the private implementation of a class.

Python uses a few simple techniques for separating interface from implementation. We can use a leading _ on an instance variable or method function name to make it more-or-less private to the class. Further, you can use properties or descriptors to create more sophisticated protocols for accessing instance variables. We'll wait until Chapter 25, *Properties and Descriptors* to cover these more advanced techniques.

Technically, a class definition creates a `class` object. This Python object defines the class by containing the definitions of the various functions. Additionally, a class object can also own class-level variables; these are, in effect, attributes which are shared by each individual object of that class.

An Object's Lifecycle. Each instance of every class has a lifecycle. The following is typical of most objects.

1. **Definition.** The class definition is read by the Python interpreter or it is built-in to the language. Our class definitions are created by the **class** statement. Examples of built-in classes include files, strings, sequences and mappings.

2. **Construction.** An instance of the class is constructed: Python allocates the namespace for the object's instance variables and associating the object with the class definition. The `__init__` method is executed to initialize any attributes of the newly created instance.

3. **Access and Manipulation.** The instance's methods are called (similar to function calls we covered in Chapter 9, *Functions*), by client objects, functions or the main script. There is a considerable amount of collaboration among objects in most programs. Methods that report on the state of the object are sometimes calles accessors; methods that change the state of the object are sometimes called manipulators.

4. **Garbage Collection.** Eventually, there are no more references to this instance. Typically, the variable with an object reference was part of the body of a function which finished, and the variables no longer exist. Python detects this, and removes the object from memory, freeing up the storage for subsequent reuse. This freeing of memory is termed *garbage collection*, and happens automatically. See Garbage Collection for more information.

Garbage Collection

It is important to note that Python counts references to objects. When object is no longer refrenced, the reference count is zero, the object can be removed from memory. This is true for all objects, especially objects of built-in classes like String. This frees us from the details of memory management as practiced by C++ programmers. When we do something like the following:

```
s= "123"
s= s+"456"
```

The following happens.

1. Python creates the string "123" and puts a reference to this string into the variable `s`.
2. Python creates the string "456".
3. Python performs the string concatenation method between the string referenced by `s` and the string "456", creating a new string "123456".
4. Python assigns the reference to this new "123456" string into the variable `s`.
5. At this point, strings "123" and "456" are no longer referenced by any variables. These objects will be destroyed as part of garbage collection.

Class Definition: the class Statement

We create a class definition with a **class** statement. We provide the class name, the parent classes, and the method function definitions.

`class` *`name(parent)`*`:` *`suite`*

The *`name`* is the name of the class, and this name is used to create new objects that are instances of the class. Traditionally, class names are capitalized and class elements (variables and methods) are not capitalized.

The *`parent`* is the name of the parent class, from which this class can inherit attributes and operations. For simple classes, we define the parent as `object`. Failing to list `object` as a parent class is not — strictly speaking — a problem; using `object` as the superclass does make a few of the built-in functions a little easier to use.

The *`suite`* is a series of function definitions, which define the class. All of these function definitions must have a first positional argument, `self`, which Python uses to identify each object's unique attribute values.

The *`suite`* can also contain assignment statements which create instance variables and provide default values.

The suite typically begins with a comment string (often a triple-quoted string) that provides basic documentation on the class. This string becomes a special attribute, called `__doc__`. It is available via the `help` function.

For example:

```
>>> import random
>>> class Die(object):
        """Simulate a 6-sided die."""
        def roll( self ):
            self.value= random.randrange(1,7)
            return self.value
         def getValue( self ):
              return self.value
```

We imported the `random` module to provide the random number generator.

We defined the simple class named `Die`, and claimed it as a subclass of `object`. The indented suite contains three elements.

- The docstring, which provides a simple definition of the real-world thing that this class represents. As with functions, the docstring is retrieved with the `help` function.
- We defined a method function named `roll`. This method function has the mandatory positional parameter, `self`, which is used to qualifty the instance variables. The `self` variable is a namespace for all of the attributes and methods of this object.
- we defined a method function named `getValue`. This function will return the last value rolled.

When the `roll` method of a `Die` object is executed, it sets that object's instance variable, `self.value`, to a random value. Since the variable name, `value`, is qualified by the instance variable, `self`, the variable is local to the specific instance of the object. If we omitted the qualifier, Python would look in the global namespace for a variable named `value`.

Creating and Using Objects

Once we have a class definition, we can make objects which are instances of that class. We do this by evaluating the class as if it were a function: *`classname`*`()`. When we make one of these class calls (for example, `Die()`), two things will happpen.

- A new object is created. This object has a reference to its class definition.
- The object's initializer method, `__init__`, is called. We'll look at how you define this method function in th next section.

Let's create two instances of our `Die` class.

```
>>> d1= Die()
>>> d2= Die()
>>> print d1.roll(), d2.roll()
1 4
>>> print d1.value, d2.value
1 4
>>> print d1, d2
<__main__.Die instance at 0x30427c> <__main__.Die instance at 0x30477c>
>>> d1.roll()
3
>>> d2.roll()
6
```

We use the `Die` class object to create two variables, `d1`, and `d2`; both are new objects, instances of `Die`.

We evaluate the `roll` method of `d1`; we also evaluate the `roll` method of `d2`. Each of these calls sets the object's `value` variable to a unique, random number. There's a pretty good chance (1 in 6) that both values might happen to be the same. If they are, simply call `d1.roll` and `d2.roll` again to get new values.

We print the `value` variable of each object. The results aren't too surprising, since the `value` attribute was set by the `roll` method. This attribute will be changed the next time we call the `roll` method.

We also ask for a representation of each object. If we provide a method named `__str__` in our class, that method is used; otherwise Python shows the memory address associated with the object. All we can see is that the numbers are different, indicating that these instances are distinct objects.

Special Method Names

There are several special methods that are essential to the implementation of a class. Each of them has a name that begins and ends with double underscores. These method names are used implicitly by Python. Section 3.3 of the *Python Language Reference* provides the complete list of these special method names.

We'll look at the special method names in depth in Chapter 24, *Creating or Extending Data Types*. Until then, we'll look at a few special method names that are used heavily.

__init__

> The `__init__` method of a class is called by Python to initialize a newly-created object

__str__

> The `__str__` method of a class is called whenever Python prints an object. This is the method used by the `str` built-in function.

__repr__

The __repr__ method of a class is used when we want to see the details of an object's values. This method is used by the repr function.

__cmp__

When we sort a list of objects, the cmp function uses the __cmp__ method of each object.

Initializing an Object with __init__. When you create an object, Python will both create the object and also call the object's __init__ method. This function can create the object's instance variables and perform any other one-time initialization. There are, typically, two kinds of instance variables that are created by the __init__ method: variables based on parameters and variables that are independent of any parameters.

Here's an example of a company description that might be suitable for evaluating stock performance. In this example, all of the instance variables (self.name, self.symbol, self.price) are based on parameters to the __init__ method.

```
class Company( object ):
    def __init__( self, name, symbol, stockPrice ):
        self.name= name
        self.symbol= symbol
        self.price= stockPrice
    def valueOf( self, shares ):
        return shares * self.price
```

When we create an instance of Company, we use code like this.

```
c1= Company( "General Electric", "GE", 30.125 )
```

This will provide three values to the parameters of __init__.

String value of an object with __str__. The __str__ method function is called whenever an instance of a class needs to be converted to a string. Typically, this occus when we use the str function on an object. Impicitly, when we reference object in a **print** statement, the str function is evaluated. The other example is wConsider this definition of the class Card.

```
class Card( object ):
    def __init__( self, rank, suit ):
        self.rank= rank
        self.suit= suit
        self.points= rank
    def hard( self ):
        return self.points
    def soft( self ):
        return self.points
```

When we try to print an instance of the class, we get something like the following.

```
>>> c= Card( 3, "D" )
>>> print c

<__main__.Card object at 0x2e5f6c>
```

This is the default behavior for the __str__ method. We can, however, override this with a function that produces a more useful-looking result.

```
    def __str__( self ):
        return "%2d%s" % (self.rank, self.suit)
```

Adding this method function converts the current value of the die to a string and returns

this. Now we get something much more useful.

```
>>> d= Card( 4, "D" )
>>> print d

 4D
```

Representation details with __repr__. While the __str__ method produces a human-readable string, we sometimes want the nitty-gritty details. The __repr__ method function is evaluated whenever an instance of a class must have its detailed representation shown. This is usually done in response to evaluating the repr function. Examples include the following:

```
>>> print repr(c)

<__main__.Card object at 0x2f639c>
```

If we would like to produce a more useful result, we can override the __repr__ function. The objective is to produce a piece of Python programming that would reconstruct the original object.

```
    def __repr__( self ):
        return "Card(%d,%r)" % (self.rank,self.suit)
```

We use __repr__ to produce a clear definition of how to recreate the given object.

```
>>> f= Card(5,"D")
>>> print repr(f)

Card(5,'D')
```

Sorting and Comparing with __cmp__. One other useful special method is __cmp__. We use the __cmp__ to provide the results used to sort and compare objects. The built-in cmp function, generally, uses this method. If you don't make other arrangements, then this method is also used for <, <=, >, >=, == and !=.

```
    def __cmp__( self, other ):
        return cmp(self.rank, other.rank) or cmp(self.suit, self.suit)
```

Once we've added the __cmp__ method we can compare cards to check their relative rank.

```
>>> cmp(c,d)
-1
>>> c < d
True
>>> c >= d
False
```

Special Attribute Names. In addition to the special method names, each object has a number of special attributes. These are documented in section 2.3.10 of the *Python Library Reference*. There are __dict__, __class__ and __bases__.

The attribute variables of a class instance are kept in a special dictionary object named __dict__. As a consequence, when you say self.attribute= value, this has almost identical meaning to self.__dict__['attribute']= value.

Combined with the % string formatting operation, this feature is handy for writing __str__ and __repr__ functions.

```
    def __str__( self  ):
```

```
        return "%(rank)2s%(suit)s" % self.__dict__
    def __repr__( self ):
        return "Card(%(rank)r,%(suit)r)" % self.__dict__
```

Some Examples

We'll look at two examples of class definitions. In the both examples, we'll write a script which defines a class and then uses the class.

Example 21.1. die.py

```
#!/usr/bin/env python
"""Define a Die and simulate rolling it a dozen times."""
import random
class Die(object):
    """Simulate a generic die."""
    def __init__( self ):
        self.sides= 6
        self.roll()
    def roll( self ):
        """roll() -> number
        Updates the die with a random roll."""
        self.value= 1+random.randrange(self.sides)
        return self.value
    def getValue( self ):
        """getValue() -> number
        Return the last value set by roll()."""
        retur self.value

def main():
    d1, d2 = Die(), Die()
    for n in range(12):
         print d1.roll(), d2.roll()

main()
```

❶ This version of the `Die` class contains a doc string and three methods: `__init__`, `roll` and `getValue`.

❷ The `__init__` method, called a *constructor*, is called automatically when the object is created. We provide a body that sets two instance variables of a `Die` object. It sets the number of sides, `sides` to 6 and it then rolls the die a first time to set a value.

❸ The `roll` method, called a *manipulator*, generates a random number, updating the `value` instance variable.

❹ The `getValue` method, called a *getter* or an *accessor*, returns the value of the value instance variable, `value`. Why write this kind of function? Why not simply use the instance variable? We'll address this in the FAQ's at the end of this chapter.

❺ The `main` function is outside the `Die` class, and makes use of the class definition. This function creates two `Die`, `d1` and `d2`, and then rolls those two `Die` a dozen times.

❻ This is the top-level script in this file. It executes the `main` function, which — in turn — then creates `Die` objects.

The `__init__` method can accept arguments. This allows us to correctly initialize an object while creating it. For example:

Example 21.2. point.py - part 1

```
#!/usr/bin/env python
"""Define a geometric point and a few common manipulations."""
class Point( object ):
    """A 2-D geometric point."""
    def __init__( self, x, y ):
```

```
        """Create a point at (x,y)."""
        self.x, self.y = x, y
    def offset( self, xo, yo ):
        """Offset the point by xo parallel to the x-axis
        and yo parallel to the y-axis."""
        self.x += xo
        self.y += yo
    def offset2( self, val ):
        """Offset the point by val parallel to both axes."""
        self.offset( val, val )
    def __str__( self ):
        """Return a pleasant representation."""
        return "(%g,%g)" % ( self.x, self.y )
```

❶ This class, `Point`, initializes each point object with the x and y coordinate values of the point. It also provides a few member functions to manipulate the point.

❷ The `__init__` method requires two argument values. A client program would use `Point( 640, 480 )` to create a `Point` object and provide arguments to the `__init__` method function.

❸ The `offset` method requires two argument values. This is a manipulator which changes the state of the point. It moves the point to a new location based on the offset values.

❹ The `offset2` method requires one argument value. This method makes use of the `offset` method. This kind of reuse assures that both methods are perfectly consistent.

❺ We've added a `__str__` method, which returns the string representation of the object. When we print any object, the **print** statement automatically calls the `str` built-in function. The `str` function uses the `__str__` method of an object to get a string representation of the object.

Example 21.3. point.py - part 2

```
def main():
    obj1_corner = Point( 12, 24 )
    obj2_corner = Point( 8, 16 )
    obj1_corner.offset( -4, -8 )
    print obj1_corner
    print obj2_corner

main()
```

❶ We construct a `Point`, named `obj1_corner`.

❷ We manipulate the `obj1_corner` `Point` to move it a few pixels left and up.

❸ We access the `obj1_corner` object by printing it. This will call the `str` function, which will use the `__str__` method to get a string representation of the `Point`.

The `self` Variable. These examples should drive home the ubiquirty of the self variable. Within a class, we must be sure to use `self.` in front of the function names as well as attribute names. For example, our `offset2` function accepts a single value and calls the object's `offset` function using `self.offset( val, val )`.

The self variable is so important, we'll highlight it.

> **The `self` Variable**
>
> In Python, the `self` qualifier is simply required all the time.
>
> Programmers experienced in Java or C++ may object to seeing the explicit `self.` in front of all variable names and method function names. In Java and C++, there is a `this.` qualifier which is assumed by the compiler. Sometimes this qualifier is required to disambiguate names, other times the compiler can work out what you meant.

> Some programmers complain that `self` is too much typing, and use another variable name like `my`. This is unusual, generally described as a bad policy, but it is not unheard of.

An object is a namespace; it contains the attributes. We can call the attributes *instance variables* to distinguish them from global variables and free variables.

Instance Variables

> Instance variables are part of an object's namespace. Within the method functions of a class, these variables are qualified by `self.`. Outside the method functions of the class, these variables are qualified by the object's name. In Example 21.1, "die.py", the `main` function would refer to `d1.value` to get the value attribute of object `d1`.

Global Variables

> Global variables are pare of the special global namespace. The **global** statement creates the variable name in the global namespace instead of the local namespace. See the section called "The **global** Statement" for more information.

Free Variables

> Within a method function, a variable that is not qualified by `self.`, nor marked by **global** is a free variable. Python checks the local namespace, then the global namespace for this variable. This ambiguity is, generally, not a good idea.

Object Collaboration

Object-oriented programming helps us by encapsulating data and processing into a tidy class definition. This encapsulation assures us that our data is processed correctly. It also helps us understand what a program does by allowing us to ignore the details of an object's implementation.

When we combine multiple objects into a collaboration, we exploit the power of ecapsulation. We'll look at a simple example of creating a composite object, which has a number of detailed objects inside it.

Defining Collaboration. Defining a collaboration means that we are creating a class which depends on one or more other classes. Here's a new class, `Dice`, which uses instances of our `Die` class. We can now work with a `Dice` collection, and not worry about the details of the individual `Die` objects.

Example 21.4. dice.py - part 1

```
#!/usr/bin/env python
"""Define a Die, and Dice and simulate a dozen rolls."""

class Die( object ):
See the definition in Example 21.1, "die.py".

class Dice( object ):
    """Simulate a pair of dice."""
    def __init__( self ):
        "Create the two Die objects."
        self.myDice = ( Die(), Die() )
    def roll( self ):
        "Return a random roll of the dice."
        for d in self.myDice:
            d.roll()
    def getTotal( self ):
        "Return the total of two dice."
```

```
        t= 0
        for d in self.myDice:
            t += d.getValue()
        return t
    def getTuple( self ):
        "Return a tuple of the dice values."
        return tuple( [d.getValue() for d in self.myDice] )
    def hardways( self ):
        "Return True if this is a hardways roll."
        return self.myDice[0].getValue() == self.myDice[1].getValue()
```

❶ This is the definition of a single `Die`, from the `die.py` example. We didn't repeat it here to save some space in the example.

❷ This class, `Dice`, defines a pair of `Die` instances.

❸ The `__init__` method creates an instance variable, `myDice`, which has a tuple of two instances of the `Die` class.

❹ The `roll` method changes the overall state of a given `Dice` object by changing the two individual `Die` objects it contains. This manipulator uses a **for** loop to assign each of the internal `Die` objects to `d`. In the loop it calls the `roll` method of the `Die` object, `d`. This technique is called *delegation*: a `Dice` object delegates the work to two individual `Die` objects. We don't know, or care, how each `Die` computes it's next value.

❺ The `getTotal` method computes a sum of all of the `Die` objects. It uses a **for** loop to assign each of the internal `Die` objects to `d`. It then uses the `getValue` method of `d`. This is the official interface method; by using it, we can remain blissfully unaware of how `Die` saves it's state.

❻ The `getTuple` method returns the values of each `Die` object. It uses a list comprehension to create a `list` of the `value` instance variables of each `Die` object. The built-in function `tuple` converts the `list` into an immutable `tuple`.

❼ The `harways` method examines the value of each `Die` objec to see if they are the same. If they are, the total was made "the hard way."

The `getTotal` and `getTuple` methods return basic attribute information about the state of the object. These kinds of methods are often called *getters* because their names start with “get”.

Collaborating Objects. The following function exercises an instance this class to roll a `Dice` object a dozen times and print the results.

```
def test2():
    x= Dice()
    for i in range(12):
        x.roll()
        print x.getTotal(), x.getTuple()
```

This function creates an instance of `Dice`, called `x`. It then enters a loop to perform a suite of statements 12 times. The suite of statements first manipulates the `Dice` object using its `roll` method. Then it accesses the `Dice` object using `getTotal` and `getTuple` method.

Here's another function which uses a `Dice` object. This function rolls the dice 1000 times, and counts the number of hardways rolls as compared with the number of other rolls. The fraction of rolls which are hardways is ideally 1/6, 16.6%.

```
def test3():
    x= Dice()
    hard= 0
    soft= 0
    for i in range(1000):
        x.roll()
        if x.hardways(): hard += 1
        else: soft += 1
    print hard/1000., soft/1000.
```

Independence. One point of object collaboration is to allow us to modify one class definition without breaking the entire program. As long as we make changes to Die that don't change the interface that Die uses, we can alter the implementation of Die all we want. Similarly, we can change the implementation of Dice, as long as the basic set of methods are still present, we are free to provide any alternative implementation we choose.

We can, for example, rework the definition of `Die` confident that we won't disturb `Dice` or the functions that use `Dice` (`test2` and `test3`). Let's change the way it represents the value rolled on the die. Here's an alternate implemetation of Die. In this case, the private instance variable, `value`, will have a value in the range `0<=value<=5`. When getValue adds 1, the value is in the usual range for a single die, $1 \le n \le 6$.

```
class Die(object):
    """Simulate a 6-sided die."""
    def __init__( self ):
        self.roll()
    def roll( self ):
        self.value= random.randint(0,5)
        retuen self.value
    def getValue( self ):
        return 1+self.value
```

Since this version of `Die` has the same interface as other versions of `Die` in this chapter, it is isomorphic to them. There could be performance differences, depending on the performance of `randint` and `randrange` functions. Since `randint` has a slightly simpler definition, it may process more quickly.

Similarly, we can replace `Die` with the following alternative. Depending on the performance of `choice`, this may be faster or slower than other versions of `Die`.

```
class Die(object):
    """Simulate a 6-sided die."""
    def __init__( self ):
        self.domain= range(1,7)
    def roll( self ):
        self.value= random.choice(self.domain)
        return self.value
    def getValue( self ):
        return self.value
```

Class Definition Exercises

These exercises are considerably more sophisticated then the exercises in previous parts. Each of these sections describes a small project that requires you to create a number of distinct classes which must collaborate to produce a useful result.

Stock Valuation

A `Block` of stock has a number of attributes, including a purchase price, purchase date, and number of shares. Commonly, methods are needed to compute the total spent to buy the stock, and the current value of the stock. A `Position` is the current ownership of a company reflected by all of the blocks of stock. A `Portfolio` is a collection of `Positions`; it has methods to compute the total value of all `Blocks` of stock.

When we purchase stocks a little at a time, each `Block` has a different price. We want to compute the total value of the entire set of `Blocks`, plus an average purchase price for the set of `Blocks`.

The StockBlock class. First, define a `StockBlock` class which has the purchase date, price per share and number of shares. Here are the method functions this class should

have.

`__init__`

> The `__init__` method will populate the individual fields of date, price and number of shares. Don't include the company name or ticker symbol; this is information which is part of the `Position`, not the individual blocks.

`__str__`

> The `__str__` method must return a nicely formatted string that shows the date, price and shares.

`getPurchValue`

> The `getPurchValue` method should compute the value as purchase price per share × shares.

`getSaleValue(`*`salePrice`*`)`

> The `getSaleValue` method requires a *`salePrice`*; it computes the value as sale price per share × shares.

`getROI(`*`salePrice`*`)`

> The `getROI` method requires a *`salePrice`*; it computes the return on investment as (sale value - purchase value) ÷ purchase value.

We can load a simple database with a piece of code the looks like the following. The first statement will create a sequence with four blocks of stock. We chose variable name that would remind us that the ticker symbols for all four is 'GM'. The second statement will create another sequence with four blocks.

```
blocksGM = [
StockBlock( purchDate='25-Jan-2001', purchPrice=44.89, shares=17 ),
StockBlock( purchDate='25-Apr-2001', purchPrice=46.12, shares=17 ),
StockBlock( purchDate='25-Jul-2001', purchPrice=52.79, shares=15 ),
StockBlock( purchDate='25-Oct-2001', purchPrice=37.73, shares=21 ),
]
blocksEK = [
StockBlock( purchDate='25-Jan-2001', purchPrice=35.86, shares=22 ),
StockBlock( purchDate='25-Apr-2001', purchPrice=37.66, shares=21 ),
StockBlock( purchDate='25-Jul-2001', purchPrice=38.57, shares=20 ),
StockBlock( purchDate='25-Oct-2001', purchPrice=27.61, shares=28 ),
]
```

The Position class. A separate class, `Position`, will have an the name, symbol and a sequence of `StockBlocks` for a given company. Here are some of the method functions this class should have.

`__init__`

> The `__init__` method should accept the company name, ticker symbol and a collection of `StockBlock` instances.

`__str__`

> The `__str__` method should return a string that contains the symbol, the total number of shares in all blocks and the total purchse price for all blocks.

`getPurchValue`

> The `getPurchValue` method sums the purchase values for all of the `StockBlocks`

in this `Position`. It delegates the hard part of the work to each `StockBlock`'s `getPurchValue` method.

`getSaleValue(`*`salePrice`*`)`

The `getSaleValue` method requires a *`salePrice`*; it sums the sale values for all of the `StockBlocks` in this `Position`. It delegates the hard part of the work to each `StockBlock`'s `getSaleValue` method.

`getROI`

The `getROI` method requires a *`salePrice`*; it computes the return on investment as (sale value - purchase value) ÷ purchase value. This is an ROI based on an overall yield.

We can create our `Position` objects with the following kind of initializer. This creates a sequence of three individual `Position` objects; one has a sequence of GM blocks, one has a sequence of EK blocks and the third has a single CAT block.

```
portfolio= [
    Position( "General Motors", "GM", blocksGM ),
    Position( "Eastman Kodak", "EK", blocksEK )
    Position( "Caterpillar", "CAT",
        [ StockBlock( purchDate='25-Oct-2001',
            purchPrice=42.84, shares=18 ) ] )
]
```

An Analysis Program. You can now write a main program that writes some simple reports on each `Position` object in the `portfolio`. One report should display the individual blocks purchased, and the purchase value of the block. This requires iterating through the `Positions` in the `portfolio`, and then delegating the detailed reporting to the individual `StockBlocks` within each `Position`.

Another report should summarize each position with the symbol, the total number of shares and the total value of the stock purchased. The overall average price paid is the total value divided by the total number of shares.

In addition to the collection of `StockBlocks` that make up a `Position`, one additional piece of information that is useful is the current trading price for the `Position`. First, add a `currentPrice` attribute, and a method to set that attribute. Then, add a `getCurrentValue` method which computes a sum of the `getSaleValue` method of each `StockBlock`. using the trading price of the `Position`.

Annualized Return on Investment. In order to compare portfolios, we might want to compute an annualized ROI. This is ROI as if the stock were held for eactly one year. In this case, since each block has different ownership period, the annualized ROI of each block has to be computed. Then we return an average of each annual ROI weighted by the sale value.

The annualization requires computing the duration of stock ownership. This requires use of the `time` module. We'll cover that in depth in Chapter 32, *Dates and Times: the `time` and `datetime` Modules*. The essential feature, however, is to parse the date string to create a time object and then get the number of days between two time objects. Here's a code snippet that does most of what we want.

```
>>> import time
>>> dt1= "25-JAN-2001"
>>> timeObj1= time.strptime( dt1, "%d-%b-%Y" )
>>> dayNumb1= int(time.mktime( timeObj1 ))/24/60/60
>>> dt2= "25-JUN-2001"
>>> timeObj2= time.strptime( dt2, "%d-%b-%Y" )
>>> dayNumb2= int(time.mktime( timeObj2 ))/24/60/60
```

```
>>> dayNumb2 - dayNumb1
151
```

In this example, `timeObj1` and `timeObj2` are time structures with details parsed from the date string by `time.strptime`. The `dayNumb1` and `dayNumb2` are a day number that corresponds to this time. Time is measured in seconds after an epoch; typically January 1, 1970. The exact value doesn't matter, what matters is that the epoch is applied consistently by `mktime`. We divide this by 24 hours per day, 60 minutes per hour and 60 seconds per minute to get days after the epoch instead of seconds. Given two day numbers, the difference is the number of days between the two dates. In this case, there are 151 days between the two dates.

All of this processing must be encapsulated into a method that computes the ownership duration.

```
def ownedFor(self, saleDate):
```

This method computes the days the stock was owned.

```
def annualizedROI(self, salePrice, saleDate):
```

We would need to add an `annualizedROI` method to the `StockBlock` that divides the gross ROI by the duration in years to return the annualized ROI. Similarly, we would add a method to the `Position` to use the `annualizedROI` to compute the a weighted average which is the annualized ROI for the entire position.

Dive Logging and Surface Air Consumption Rate

The Surface Air Consumption Rate is used by SCUBA divers to predict air used at a particular depth. If we have a sequence of `Dive` objects with the details of each dive, we can do some simple calculations to get averages and ranges for our air consumption rate.

For each dive, we convert our air consumption at that dive's depth to a normalized air consumption at the surface. Given depth (in feet), d, starting tank pressure (psi), s, final tank pressure (psi), f, and time (in minutes) of t, the SACR, c, is given by the following formula.

$$c = \frac{33(s-f)}{t(d+33)}$$

Typically, you will average the SACR over a number of similar dives.

The `Dive` Class. You will want to create a `Dive` class that contains attributes which include start pressure, finish pressure, time and depth. Typical values are a starting pressure of 3000, ending pressure of 700, depth of 30 to 80 feet and times of 30 minutes (at 80 feet) to 60 minutes (at 30 feet). SACR's are typically between 10 and 20. Your `Dive` class should have a function named `getSACR` which returns the SACR for that dive.

To make life a little simpler putting the data in, we'll treat time as string of "HH:MM", and use string functions to pick this apart into hours and minutes. We can save this as tuple of two intgers: hours and minutes. To compute the duration of a dive, we need to normalize our times to minutes past midnight, by doing `hh*60+mm`. Once we have our times in minutes past midnight, we can easily subtract to get the number of minutes of duration for the dive. You'll want to create a function `getDuration` to do just this computation for each dive.

`__init__`

> The `__init__` method will initialize a `Dive` with the start and finish pressure in PSI, the in and out time as a string, and the depth as an integer. This method

should parse both the `in` string and `out` string and normalize each to be minutes after midnight so that it can compute the duration of the dive. Note that a practical dive log would have additional information like the date, the location, the air and water temperature, sea state, equipment used and other comments on the dive.

`__str__`

The `__str__` method should return a nice string representation of the dive information.

`getSACR`

The `getSACR` method can then compute the SACR value from the starting pressure, final pressure, time and depth information.

The `Dive` Log. We'll want to initialize our dive log as follows:

```
log = [
    Dive( start=3100, finish=1300, in="11:52", out="12:45", depth=35 ),
    Dive( start=2700, finish=1000, in="11:16", out="12:06", depth=40 ),
    Dive( start=2800, finish=1200, in="11:26", out="12:06", depth=60 ),
    Dive( start=2800, finish=1150, in="11:54", out="12:16", depth=95 ),
]
```

Rather than use a simple sequence of `Dives`, you can create a `DiveLog` class which has a sequence of `Dives` plus a `getAvgSACR` method. Your `DiveLog` method can be initiatlized with a sequence of dives, and can have an append method to put another dive into the sequence.

Exercising the Dive and DiveLog Classes. Here's how the final application could look. Note that we're using an arbitrary number of argument values to the `__init__` function, therefore, it has to be declared as `def __init__( self, *listOfDives )`

```
log= DiveLog(
    Dive( start=3100, finish=1300, in="11:52", out="12:45", depth=35 ),
    Dive( start=2700, finish=1000, in="11:16", out="12:06", depth=40 ),
    Dive( start=2800, finish=1200, in="11:26", out="12:06", depth=60 ),
    Dive( start=2800, finish=1150, in="11:54", out="12:16", depth=95 ),
)
print log.getAvgSACR()
for d in log.dives:
    print d
```

Multi-Dice

If we want to simulate multi-dice games like Yacht, Kismet, Yatzee, Zilch, Zork, Greed or Ten Thousand, we'll need a collection that holds more than two dice. The most common configuration is a five-dice collection. In order to be flexible, we'll need to define a `Dice` object which will use a `tuple`, `list` or `Set` of individual `Die` instances. Since the number of dice in a game rarely varies, we can also use a `FrozenSet`.

Once you have a `Dice` class which can hold a collection of dice, you can gather some statistics on various multi-dice games. These games fall into two types. In both cases, the player's turn starts with rolling all the dice, the player can then elect to re-roll or preserve selected dice.

- **Scorecard Games.** In Yacht, Kismet and Yatzee, five dice are used. The first step in a player's turn is a roll of all five dice. This can be followed by up to two additional steps in which the player decides which dice to preserve and which dice to roll. The player is trying to make a scoring hand. A typical scorecard for these games lists a dozen or more "hands" with associated point values. The player attempts to make one of each of the various kinds of hands listed on the

scorecard. Each turn must fill in one line of the scorecard; if the dice match a hand which has not been scored, the player enters a score. If a turn does not result in a hand that matches an unscored hand, then a score of zero is entered.

- **Point Games.** In Zilch, Zork, Green or Ten Thousand, five dice are typical, but there are some variations. The player in this game has no limit on the number of steps in their turn. The first step is to roll all the dice and determine a score. Their turn ends when they perceive the risk of another step to be too high, or they've made a roll which gives them a score of zero (or zilch) for the turn. Typically, if the newly rolled dice are non-scoring, their turn is over with a score of zero. At each step, the player is looking at newly rolled dice which improve their score. A straight scores 1000. Three-of-a-kind scores 100× the die's value (except three ones is 1000 points). After removing any three-of-a-kinds, each die showing 1 scores 100, each die showing 5 scores 50. Additionally, some folks will score 1000× the die's value for five-of-a-kind.

Our `MultiDice` class will be based on the example of `Dice` in this chapter. In addition to a collection of `Die` instances (a sequence, `Set` or `FrozenSet`), the class will have the following methods.

`__init__`

When initializing an instance of `MultiDice`, you'll create a collection of five individual `Die` instances. You can use a sequence of some kind, a `Set` or a `FrozenSet`.

`roll`

The `roll` method will roll all dice in the sequence or Set. Note that your choice of collection doesn't materially alter this method. That's a cool feature of Python.

`getDice`

This method returns the collection of dice so that a client class can examine them and potentialy re-roll some or all of the dice.

score

This method will score the hand, returning a `list` of two-tuples. Each two-tuple will have the name of the hand and the point value for the particular game. In some cases, there will be multiple ways to score a hand, and the `list` will reflect all possible scorings of the hand, in order from most valuable to least valuable. In other cases, the `list` will only have a single element.

It isn't practical to attempt to write a universal `MultiDice` class that covers all variations of dice games. Rather than write a gigantic does-everything class, the better policy is to create a family of classes that build on each other using inheritance. We'll look at this in the section called “Inheritance”. For this exercise, you'll have to pick one of the two families of games and compute the score for that particular game. Later, we'll see how to create an inheritance hierarchy that can cover the two-dice game of Craps as well as these multi-dice games.

For the scorecard games (Yacht, Kismet, Yatzee), we want to know if this set of dice matches any of the scorecard hands. In many cases, a set of dice can match a number of potential hands. A hand of all five dice showing the same value (e.g, a 6) is matches the sixes, three of a kind, four of a kind, five of a kind and wild-card rows on most game score-sheets. A sequence of five dice will match both a long straight and a short straight.

Common Scoring Methods. No matter which family of games you elect to pursue, you'll need some common method functions to help score a hand. The following

methods will help to evaluate a set of dice to see which hand it might be.

- `matchDie`. This function will take a `Die` and a `Dice` set. It uses `matchValue` to partition the dice based on the value of the given `Die`.

- `matchValue`. This function is like `matchDie`, but it uses a numeric value instead of a `Die`. It partitions the dice into two sets: the dice in the `Dice` set which have a value that matches the given `Die`, and the remaining `Die` which do not match the value.

- `threeOfAKind`, `fourOfAKind`, `fiveOfAKind`. These three functions will compute the `matchDie` for each `Die` in the `Dice` set. If any given `Die` has a `matchDie` with 3 (or 4 or 5) matching dice, the hand as a whole matches the template.

- `largeStraight`. This function must establish that all five dice form a sequence of values from 1 to 5 or 2 to 6. There must be no gaps and no duplicated values.

- `smallStraight`. This function must establish that four of the five dice form a sequence of values. There are a variety of ways of approaching this; it is actually a challenging algorithm. If we create a sequence of dice, and sort them into order, we're looking for an ascending sequence with one "irrelevant" die in it: this could be a gap before or after the sequence (1, 3, 4, 5, 6; 1, 2, 3, 4, 6) or a duplicated value (1, 2, 2, 3, 4, 5) within the sequence.

- `chance`. The chance hand is simply the sum of the dice values. It is a number between 5 and 30.

This isn't necessarily the best way to do this. In many cases, a better way is to define a series of classes for the various kinds of hands, following the Strategy design pattern. The Dice would then have a collection of Strategy objects, each of which has a `match` method that compares the actual roll to a kind of hand. The Strategy objects would have a `score` method as well as a `name` method. This is something we'll look at in the section called “Strategy”.

Scoring Yacht, Kismet and Yatzee. For scoring these hands, you'll use the common scoring method functions. Your overall `score` method function will step through the candidate hands in a specific order. Generally, you'll want to check for `fiveOfAKind` first, since `fourOfAKind` and `threeOfAKind` will also be true for this hand. Similarly, you'll have to check for `largeStraight` before `smallStraight`.

Your `score` method will evaluate each of the scoring methods. If the method matches, your method will append a two-tuple with the name and points to the list of scores.

Scoring Zilch, Zork and 10,000. For scoring these dice throws, you'll need to expand on the basic `threeOfAKind` method. Your `score` method will make use of the two results sets created by the `threeOfAKind` method.

Note that the hand's description can be relatively complex. For example, you may have a hand with three 2's, a 1 and a 5. This is worth 350. The description has two parts: the three-of-a-kind and the extra 1's and 5's. Here are the steps for scoring this game.

- Evaluate the `largeStraight` method. If the hand matches, then return a `list` with an appropriate 2-tuple.

- If you're building a game variation where five of a kind is a scoring hand, then evaluate `fiveOfAKind`. If the hand matches, then return a list with an appropriate 2-tuple.

- 3K. Evaluate the `threeOfAKind` method. This will create the first part of the hand's description.

    - If a `Die` created a matching set with exactly three dice, then the set of unmatched dice must be examined for additional 1's and 5's. The first part of the hand's description string is three-of-a-kind.

    - If a `Die` created a matching with four or five dice, then one or two dice must be popped from the matching set and added to the non-matching set. The set of unmatched dice must be examined for addtional 1's and 5's. The first part of the hand's description string is three-of-a-kind.

    - If there was no set of three matching dice, then all the dice are in the non-matching set, which is checked for 1's and 5's. The string which describes the hand has no first part, since there was no three-of-a-kind.

- 1-5's. Any non-matching dice from the `threeOfAKind` test are then checked using `matchValue` to see if there are 1's or 5's. If there are any, this is the second part of the hand's description. If there are none, then there's no second part of the description.

- The final step is to assemble the description. There are four cases: nothing, 3K with no 1-5's, 1-5's with no 3K, and 3K plus 1-5's. In the nothing case, this is a non-scoring hand. In the other cases, it is a scoring hand.

Exercising The Dice. Your main script should create a `Dice` set, execute an initial roll and score the result. It should then pick three dice to re-roll and score the result. Finally, it should pick one die, re-roll this die and score the result. This doesn't make sophisticated strategic decisions, but it does exercise your `Dice` and `Die` objects thoroughly.

When playing a scorecard game, the list of potential hands is examined to fill in another line on the scorecard. When playing a points game, each throw must result in a higher score than the previous throw or the turn is over.

Strategy. When playing these games, a person will be able to glance at the dice, form a pattern, and decide if the dice are "close" to one of the given hands. This is a challenging judgement, and requires some fairly sophisticated software to make a proper odd-based judgement of possible outcomes. Given that there are only 7,776 possible ways to roll 5 dice, it's a matter of exhaustively enumerating all of the potential outcomes of the various kinds of rerolls. This is an interesting, but quite advanced exercise.

Rational Numbers

A Rational number is a ratio of two integers. Examples include 1/2, 2/3, 22/7, 355/113, etc. We can do arithmetic operations on rational numbers. We can display them as proper fractions (`3 1/7`), improper fractions (`22/7`) or decimal expansions (`3.1428571428571428`).

The essence of this class is to perform arithmetic operations. We'll start by defining methods to add and multiply two rational values. Later, we'll delve into the additional methods you'd need to write to create a robust, complete implementation.

Your `add` and `mul` methods will perform their processing with two `Rational` values: `self` and `other`. In both cases, the variable `other` has to be another `Rational` number instance. You can check this by using the `type` function: if `type(self) != type(other)`, you should raise a `TypeException`.

It's also important to note that all arithmetic operations will create a new `Rational` number computed from the inputs.

A Rational class has two attributes: the numerator and the denominator of the value. These are both integers. Here are the various methods you should created.

`__init__`

The `__init__` constructor accepts the numerator and denominator values. It can have a default value for the denominator of 1. This gives us two constructors: `Rational(2,3)` and `Rational(4)`. The first creates the fraction 2/3. The second creates the fraction 4/1.

This method should call the `reduce` method to assure that the fraction is properly reduced. For example, `Rational(8,4)` should automatically reduce to a numerator of 2 and a denominator of 1.

`__str__`

The `__str__` method returns a nice string representation for the rational number, typically as an improper fraction. This gives you the most direct view of your `Rational` number.

You should provide a separate method to provide a proper fraction string with a whole number and a fraction. This other method would do additional processing to extract a whole name and remainder.

`__float__`

If you provide a method named `__float__`, this can return the floating-point value for the fraction. This method is called when a program does `float(` *`rationalValue`* `)`.

`add(`*`self`*, *`other`*`)`

The `add` method creates and returns a new `Rational` number. This new fraction that has a numerator of (self.numerator × other.denominator + other.numerator × self.denominator), and a denominator of (self.denominator × other.denominator).

Equation 21.1. Adding Fractions

$$\frac{x_n}{x_d} + \frac{y_n}{y_d} = \frac{x_n y_d + y_n x_d}{x_d y_d}$$

Example: 3/5 + 7/11 = (33 + 35)/55 = 71/55.

`mul(`*`self`*, *`other`*`)`

The `mul` method creates and returns a new `Rational` number. This new fraction that has a numerator of (self.numerator × other.numerator), and a denominator of (self.denominator × other.denominator).

Equation 21.2. Multiplying Fractions

$$\frac{x_n}{x_d} \times \frac{y_n}{y_d} = \frac{x_n y_n}{x_d y_d}$$

Example: 3/5 × 7/11 = 21/55.

`reduce`

In addition to adding and multiplying two `Rational` numbers, you'll also need to provide a `reduce` method which will reduce a fraction by removing the greatest common divisor from the numerator and the denominator. This should be called by `__init__` to assure that all fractions are reduced.

To implement `reduce`, we find the greatest common divisor between the

numerator and denominator and then divide both by this divisor. For example 8/4 has a GCD of 4, and reduces to 2/1. The Greatest Common Divisor (GCD) algorithm is given in Greatest Common Divisor and Greatest Common Divisor. If the GCD of `self.numerator` and `self.denominator` is 1, the `reduced` function can return `self`. Otherwise, `reduced` must create a new `Rational` with the reduced fraction.

Playing Cards and Decks

Standard playing cards have a rank (ace, two through ten, jack, queen and king) and suit (clubs, diamonds, hearts, spades). These form a nifty `Card` object with two simple attributes. We can add a few generally useful functions.

Here are the methods for your `Card` class.

`__init__`(*`self`*, *`rank`*)

The `__init__` method sets the rank and suit of the card. The suits can be coded with a single character ("C", "D", "H", "S"), and the ranks can be coded with a number from 1 to 13. The number 1 is an ace. The numbers 11, 12, 13 are Jack, Queen and King, respectively. These are the ranks, not the point values.

`__str__`

The `__str__` method can return the rank and suit in the form "2C" or "AS" or "JD". A rank of 1 would become "A", a rank of 11, 12 or 13 would become "J", "Q" or "K", respectively.

`__cmp__`(*`self`*, *`other`*)

If you define a `__cmp__` method, this will used by the `cmp` function; the `cmp` function is used by the `sort` method of a `list` unless you provide an overriding function used for comparison. By providing a `__cmp__` method in your class you can assure that cards are sorted by rank in preference to suit. You can also use `<`, `>`, `>=` and `<=` operations among cards.

Sometime as simple as `cmp(self.rank,other.rank) or cmp(self.suit,other.suit)` works surprisingly well.

Dealing and Decks. `Card`s are dealt from a `Deck`; a collection of `Card`s that includes some methods for shuffling and dealing. Here are the methods that comprise a `Deck`.

`__init__`

The `__init__` method creates all 52 cards. It can use two loops to iterate through the sequence of suits `("C", "D", "H", "S")` and iterate through the ranks `range(1,14)`. After creating each `Card`, it can append each `Card` to a sequence of `Card`s.

`deal`

The `deal` method should do two things: iterate through the sequence, exchanging each card with a randomly selected card. It turns out the `random` module has a `shuffle` function which does precisely this.

Dealing is best done with a generator method function. The `deal` method function should have a simple for-loop that yields each individual `Card`; this can be used by a client application to generate hands. The presence of the **yield** statement will make this method function usable by a **for** statement in a client application script.

Basic Testing. You should do some basic tests of your Card objects to be sure that they respond appropriately to comparison operations. For example,

```
>>> x1= Card(11,"C")
>>> x1
 JC
>>> x2= Card(12,"D")
>>> x1 < x2
True
```

You can write a simple test script which can the do the following to deal `Cards` from a `Deck`. In this example, the variable `dealer` will be the iterator object that the **for** statement uses internally.

```
d= Deck()
dealer= d.deal()
c1= dealer.next()
c2= dealer.next()
```

Hands. Many card games involve collecting a hand of cards. A Hand is a collection of `Cards` plus some addition methods to score the hand in way that's appropriate to the given game. We have a number of collection classes that we can use: `list`, `tuple`, `dictionary` and `set`.

Consider Blackjack. The `Hand` will have two `Cards` assigned initially; it will be scored. Then the player must choose among accepting another card (a hit), using this hand against the dealer (standing), doubling the bet and taking one more card, or splitting the hand into two hands. Ignoring the split option for now, it's clear that the collection of `Cards` has to grow and then get scored again. What are the pros and cons of `list`, `tuple`, `set` and `dictionary`?

Consider Poker. There are innumerable variations on poker; we'll look at simple five-card draw poker. Games like seven-card stud require you to score potential hands given only two cards, and as many as 21 alternative five-card hands made from seven cards. Texas Hold-Em has from three to five common cards plus two private cards, making the scoring rather complex. For five-card draw, the `Hand` will have five cards assigned initially, and it will be scored. Then some cards can be removed and replaced, and the hand scored again. Since a valid poker hand is an ascending sequence of cards, called a straight, it is handy to sort the collection of cards. What are the pros and cons of `list`, `tuple`, `set` and `dictionary`?

Blackjack Hands

Changes to the Card class. We'll extend our `Card` class to score hands in Blackjack, where the rank is used to determine the hand that is held. When used in Blackjack, a `Card` has a point value in addition to a rank and suit. Aces are either 1 or 11; two through ten are worth 2-10; the face cards are all worth 10 points. When an ace is counted as 1 point, the total is called the hard total. When an ace is counted as 11 points, the total is called a soft total.

You can add a point attribute to your card class. This can be set as part of `__init__` processing. In that case, the following methods simple return the point value.

As an alternative, you can compute the point value each time it is requested. This has the obvious disadvantage of being slower. However, it is considerably simpler to add methods to a class without revising the existing `__init__` method.

Here are the methods you'll need to add to your `Card` class in order to handle Blackjack hands.

`getHardValue`

> The `getHardValue` method returns the rank, with the following exceptions: ranks of 11, 12 and 13 return a point value of 10.

`getSoftValue`

> The `getSoftValue` method returns the rank, with the following exceptions: ranks of 11, 12 and 13 return a point value of 10; a rank of 1 returns a point value of 11.

As a teaser for the next chapter, we'll note that these methods should be part of a Blackjack-specific subclass of the generic `Card` class. For now, however, we'll just update the `Card` class definition.When we look at inheritance in the section called "Inheritance", we'll see that a class hierarchy can be simpler than the if-statements in the `getHardValue` and `getSoftValue` methods.

Scoring Blackjack Hands. The objective of Blackjack is to accumulate a `Hand` with a total point value that is less than or equal to 21. Since an ace can count as 1 or 11, it's clear that only one of the aces in a hand can have a value of 11, and any other aces must have a value of 1.

Each `Card` produces a hard and soft point total. The `Hand` as a whole also has hard and soft point totals. Often, both hard and soft total are equal. When there is an ace, however, the hard and soft totals for the hand will be different. We have to look at two cases.

- No Aces. The hard and soft total of the hand will be the same; it's the total of the hard value of each card. If the hard total is less than 21 the hand is in play. If it is equal to 21, it is a potential winner. If it is over 21, the hand has gone bust. Both totals will be computed as the hard value of all cards.

- One or more Aces. The hard and soft total of the hand are different. The hard total for the hand is the sum of the hard point values of all cards. The soft total for the hand is the soft value of one ace plus the hard total of the rest of the cards. If the hard or soft total is 21, the hand is a potential winner. If the hard total is less than 21 the hand is in play. If the hard total is over 21, the hand has gone bust.

The `Hand` class has a collection of `Cards`, usually a sequence, but a `Set` will also work. Here are the methods of the `Hand` class.

`__init__`

> The `__init__` method should be given two instances of `Card` to represent the initial deal. It should create a sequence or `Set` with these two initial cards.

`__str__`

> The `__str__` method a string with all of the individual cards. A construct like the following works out well: `",".join( map(str,self.cards)`. This gets the string representation of each card in the `self.cards` collection, and then uses the `string`'s `join` method to assemble the final display of cards.

`hardTotal`

> The `hardTotal` method sums the hard value of each `Card`.

`softTotal`

> The `softTotal` method is more complex. It needs to partition the cards into two sets. If there are any cards with a different hard and soft point value (this will be an ace), then one of these cards forms the `softSet`. The remaining cards form the `hardSet`. It's entirely possible that the `softSet` will be empty. It's also entirely possible that there are multiple cards which could be part of the `softSet`. The value of this function is the total of the hard values for all of the cards in the `hardSet` plus the soft value of the card in the `softSet`.

`add`

The `add` method will add another `Card` to the `Hand`.

Exercising Card, Deck and Hand. Once you have the `Card`, `Deck` and `Hand` classes, you can exercise these with a simple function to play one hand of blackjack. This program will create a `Deck` and a `Hand`; it will deal two `Cards` into the `Hand`. While the `Hand`'s total is soft 16 or less, it will add `Cards`. Finally, it will print the resulting `Hand`.

There are two sets of rules for how to fill a `Hand`. The dealer is tightly constrained, but players are more free to make their own decisions. Note that the player's hands which go bust are settled immediately, irrespective of what happens to the dealer. On the other hand, the player's hands which total 21 aren't resolved until the dealer finishes taking cards.

The dealer must add cards to a hand with a soft 16 or less. If the dealer has a soft total between 17 and 21, they stop. If the dealer has a soft total which is over 21, but a hard total of 16 or less, they will take cards. If the dealer has a hard total of 17 or more, they will stop.

A player may add cards freely until their hard total is 21 or more. Typically, a player will stop at a soft 21; other than that, almost anything is possible.

Additional Plays. We've avoided discussing the options to split a hand or double the bet. These are more advanced topics that don't have much bearing on the basics of defining `Card`, `Deck` and `Hand`. Splitting simply creates additional `Hands`. Doubling down changes the bet and gets just one additional card.

Poker Hands

We'll extend the `Card` class we created in the section called "Playing Cards and Decks" to score hands in Poker, where both the rank and suit are used to determine the hand that is held.

Poker hands are ranked in the following order, from most desirable (and least likely) down to least desirable (and all too common).

1. **Straight Flush.** Five cards of adjacent ranks, all of the same suit.
2. **Four of a Kind.** Four cards of the same rank, plus another card.
3. **Full House.** Three cards of the same rank, plus two cards of the same rank.
4. **Flush.** Five cards of the same suit.
5. **Straight.** Five cards of adjacent ranks. In this case, Ace can be above King or below 2.
6. **Three of a Kind.** Three cards of the same rank, plus two cards of other ranks.
7. **Two Pair.** Two cards of one rank, plus two cards of another rank, plus one card of a third rank.
8. **Pair.** Two cards of one rank, plus three cards of other ranks.
9. **High Card.** The highest ranking card in the hand.

Note that a straight flush is both a straight and a flush; four of a kind is also two pair as well as one pair; a full house is also two pair, as well as a one pair. It is important, then, to evaluate poker hands in decreasing order of importance in order to find the best hand possible.

In order to distinguish between two straights or two full-houses, it is important to also

record the highest scoring card. A straight with a high card of a Queen, beats a straight with a high card of a 10. Similarly, a full house or two pair is described as “queens over threes”, meaning there are three queens and two threes comprising the hand. We'll need a numeric ranking that includes the hand's rank from 9 down to 1, plus the cards in order of “importance” to the scoring of the hand.

The importance of a card depends on the hand. For a straight or straight flush, the most important card is the highest-ranking card. For a full house, the most important cards are the three-of-a kind cards, followed by the pair of cards. For two pair, however, the most important cards are the high-ranking pair, followed by the low-ranking pair. This allows us to compare “two pair 10's and 4's” against “two pair 10's and 9s'”. Both hands have a pair of 10's, meaning we need to look at the third card in order of importance to determine the winner.

Scoring Poker Hands. The `Hand` class should look like the following. This definition provides a number of methods to check for straight, flush and the patterns of matching cards. These functions are used by the `score` method, shown below.

```
class PokerHand:
    def __init__( self, cards ):
        self.cards= cards
        self.rankCount= {}
    def straight( self ):
        all in sequence
    def straight( self ):
        all of one suit
    def matches( self ):
        tuple with counts of each rank in the hand
    def sortByRank( self ):
        sort into rank order
    def sortByMatch( self ):
        sort into order by count of each rank, then rank
```

This function to score a hand checks each of the poker hand rules in descending order.

```
    def score( self ):
        if self.straight() and self.flush():
            self.sortByRank()
            return 9
        elif self.matches() == ( 4, 1 ):
            self.sortByMatch()
            return 8
        elif self.matches() == ( 3, 2 ):
            self.sortByMatch()
            return 7
        elif self.flush():
            self.sortByRank()
            return 6
        elif self.straight():
            self.sortByRank()
            return 5
        elif self.matches() == ( 3, 1, 1 ):
            self.sortByMatch()
            return 4
        elif self.matches() == ( 2, 2, 1 ):
            self.sortByMatchAndRank()
            return 3
        elif self.matches() == ( 2, 1, 1, 1 ):
            self.sortByMatch()
            return 2
        else:
            self.sortByRank()
            return 1
```

You'll need to add the following methods to the PokerHand class.

- `straight` returns True if the cards form a straight. This can be tackled easily by sorting the cards into descending order by rank and then checking to see if the ranks all differ by exactly one.
- `flush` returns True if all cards have the same suit.
- `matches` returns a tuple of the counts of cards grouped by rank. This can be done iterating through each card, using the card's rank as a key to the `self.rankCount` dictionary; the value for that dictionary entry is the count of the number of times that rank has been seen. The values of the dictionary can be sorted, and form six distinct patterns, five of which are shown above. The sixth is simply `(1, 1, 1, 1, 1)`, which means no two cards had the same rank.
- `sortByRank` sorts the cards by rank.
- `sortByMatch` uses the counts in the `self.rankCount` dictionary to update each card with its match count, and then sorts the cards by match count.
- `sortByMatchAndRank` uses the counts in the `self.rankCount` dictionary to update each card with its match count, and then sorts the cards by match count and rank as two separate keys.

Exercising Card, Deck and Hand. Once you have the `Card`, `Deck` and `Hand` classes, you can exercise these with a simple function to play one hand of poker. This program will create a `Deck` and a `Hand`; it will deal five `Cards` into the `Hand`. It can score the hand. It can replace from zero to three cards and score the resulting hand.

Chapter 22. Advanced Class Definition

Table of Contents

This section builds up some additional class definition techniques. We describe the basics of inheritance in the section called “Inheritance”. We turn to a specific inheritance technique, polymorphism in the section called “Polymorphism”. There are some class-related functions, which we describe in the section called “Built-in Functions”. We'll look at some specific class initializer technique in the section called “Initializer Techniques”. We include a digression on design approaches in the section called “Design Approaches”. In the section called “Class Variables” we provide information on class-level variables, different from instance variables. We conclude this chapter with some style notes in the section called “Style Notes”.

Inheritance

One of the four important features of class definition is inheritance. You can create a subclass which inherits all of the features of a superclass. The subclass can add or replace method functions of the superclass. This is typically used by defining a general-purpose superclass and creating specialized subclasses that all inherit the general-purpose features but add special-purposes features of their own.

We do this by specifying the parent class when we create a subclass.

class *`subclass`* (*`superclass`*): *`suite`*

All of the methods of the superclass are, by definition, also part of the subclass. Often the suite of method functions will add to or override the definition of a parent method.

If we omit providing a superclass, we create a *classical* class definition, where the Python type is `instance`; we have to do additional processing to determine the actual type. When we use `object` as the superclass, the Python type is reported more simply as the appropriate `class` object. As a general principle, every class definition should be a subclass of `object`, either directly or indirectly.

Extending a Class. There are two trivial subclassing techniques. One defines a subclass which adds new methods to the superclass. The other overrides a superclass method. The overriding technique leads to two classes which are polymorphic because they have the same interface. We'll return to polymorphism in the section called “Polymorphism”.

Here's a revised version of our basic `Dice` class and a subclass to create `CrapsDice`.

Example 22.1. crapsdice.py

```
#!/usr/bin/env python
"""Define a Die, Dice and CrapsDice."""

class Die( object ):
See the definition in Example 21.1, “die.py”.

class Dice( object ):
    """Simulate a pair of dice."""
    def __init__( self ):
        "Create the two Die objects."
        self.myDice = ( Die(), Die() )
    def roll( self ):
        "Return a random roll of the dice."
        for d in self.myDice:
            d.roll()
    def getTotal( self ):
        "Return the total of two dice."
        return self.myDice[0].value + self.myDice[1].value
    def getTuple( self ):
        "Return a tuple of the dice."
        return self.myDice

class CrapsDice( Dice ):
    """Extends Dice to add features specific to Craps."""
    def hardways( self ):
        """Returns True if this was a hardways roll?"""
        return self.myDice[0].value == self.myDice[1].value
    def isPoint( self, value ):
        """Returns True if this roll has the given total"""
        return self.getTotal() == value
```

The `CrapsDice` class contains all the features of `Dice` as well as the additional features we added in the class declaration. We can write applications which create a CrapsDice instance. We can, for example, evaluate the `roll` and `hardways` methods of `CrapsDice`. The `roll` method is inherited from `Dice`, but the `hardways` method is a direct part of

`CrapsDice`.

Adding Instance Variables. Adding new instance variables requires that we extend the `__init__` method. In this case we want an `__init__` function that starts out doing everything the superclass `__init__` function does, and then creates a few more attributes. Python provides us the `super` function to help us do this. We can use `super` to distinguish between method functions with the same name defined in the superclass and extended in a subclass.

`super(`*`type`*`,` *`variable`*`)`

> This will do two things: locate the superclass of the given type, and it will assure that the given variable is an appropriate object of the superclass. This is often used to call a superclass method from within a subclass: `super(`*`MyClass`*`,self).`*`aMethod`*`(`*`args`*`)`.

Here's a template that shows how a subclass `__init__` method uses `super` to evaluate the superclass `__init__` method.

```
class Subclass( Superclass ):
    def __init__( self ):
        super(Subclass,self)__init__()
        Subclass-specific stuff
```

This will provide the original `self` variable to parent class method function so that it gets the superclass initialization. After that, we can add our subclass initialization.

We'll look at additional techniques for creating very flexible `__init__` methods in the section called “Initializer Techniques”.

Various Kinds of Cards. Let's look clisely at the problem of cards in Blackjack. All cards have several general features: they have a rank and a suit. All cards have a point value. However, some cards use their rank for point value, other cards use 10 for their point value and the aces can be either 1 or 11, depending on the the rest of the cards in the hand. We looked at this in the the section called “Playing Cards and Decks” exercise in Chapter 21, *Classes*.

We can model this very accurately by creating a `Card` class that encapsulates the generic features of rank, suit and point value. Our class will have instance variables for these attribites. The class will also have two functions to return the hard value and soft value of this card. In the case of ordinary non-face, non-ace cards, the point value is always the rank. We can use this `Card` class for the number cards, which are most common.

```
class Card( object ):
    """A standard playing card for Blackjack."""
    def __init__( self, r, s ):
        self.rank, self.suit = r, s
        self.pval= r
    def __str__( self ):
        return "%2d%s" % ( self.rank, self.suit )
    def getHardValue( self ):
        return self.pval
    def getSoftValue( self ):
        return self.pval
```

We can create a subclass of `Card` which is specialized to handle the face cards. This subclass simply overrides the value of `self.pval`, using 10 instead of the rank value. In this case we want a `FaceCard.__init__` method that uses the parent's `Card.__init__` method, and then does additional processing. The existing definitions of `getHardValue` and `getSoftValue` method functions, however, work fine for this subclass. Since `Card` is a subclass of `object`, so is `FaceCard`.

Additionally, we'd like to report the card ranks using letters (J, Q, K) instead of numbers. We can override the `__str__` method function to do this translation from rank to label.

```
class FaceCard( Card ):
    """A 10-point face card: J, Q, K."""
    def __init__( self, r, s ):
        super(FaceCard,self).__init__( r, s )
        self.pval= 10
    def __str__( self ):
        label= ("J","Q","K")[self.rank-11]
        return "%2s%s" % ( label, self.suit )
```

We can also create a subclass of `Card` for Aces. This subclass inherits the parent class `__init__` function, since the work done there is suitable for aces. The `Ace` class, however, provides a more complex algorithms for the `getHardValue` and `getSoftValue` method functions. The hard value is 1, the soft value is 11.

```
class Ace( Card ):
    """An Ace: either 1 or 11 points."""
    def __str__( self ):
        return "%2s%s" % ( "A", self.suit )
    def getHardValue( self ):
        return 1
    def getSoftValue( self ):
        return 11
```

Deck and Shoe as Collections of Cards. In a casino, we can see cards handled in a number of different kinds of collections. Dealers will work with a single deck of 52 cards or a multi-deck container called a shoe. We can also see the dealer putting cards on the table for the various player's hands, as well as a dealer's hand.

Each of these collections has some common features, but each also has unique features. Sometimes it's difficult to reason about the various classes and discern the common features. In these cases, it's easier to define a few classes and then refactor the common features to create a superclass with elements that have been removed from the subclasses. We'll do that with Decks and Shoes.

We can define a `Deck` as a sequence of `Cards`. The `__init__` method function of `Deck` creates appropriate `Cards` of each subclass; `Card` objects in the range 2 to 10, `FaceCard` obejcts with ranks of 11 to 13, and `Ace` objects with a rank of 1.

```
class Deck( object ):
    """A deck of cards."""
    def __init__( self ):
        self.cards= []
        for suit in ( "C", "D", "H", "S" ):
            self.cards+= [Card(r,suit) for r in range(2,11)]
            self.cards+= [TenCard(r,suit) for r in range(11,14)]
            self.cards+= [Ace(1,suit)]
    def deal( self ):
        for c in self.cards:
            yield c
```

In this example, we created a single instance variable `self.cards` within each `Deck` instance. For dealing cards, we've provided a generator function which yields the `Cards` in a random order. We've omitted the randomization from the `deal` function; we'll return to it in the exercises.

For each suit, we created the `Cards` of that suit in three steps.

1. We created the number cards with a list comprehension to generate all ranks in the

range 2 through 10.

2. We created the face cards with a similar process, except we use the `TenCard` class constructor, since blackjack face cards all count as having ten points.

3. Finally, we created a singleton list of an `Ace` instance for the given suit.

We can use `Deck` objects to create an multi-deck *shoe*; a shoe is what dealers use in casinos to handle several decks of slippery playing cards. The `Shoe` class will create six separate decks, and then merge all 312 cards into a single sequence.

```
class Shoe( object ):
    """Model a multi-deck shoe of cards."""
    def __init__( self, decks=6 ):
        self.cards= []
        for i in range(decks):
            d= Deck()
            self.cards += d.cards
    def deal( self ):
        for c in self.cards:
            yield c
```

For dealing cards, we've provided a generator function which yields the `Cards` in a random order. We've omitted the randomization from the `deal` function; we'll return to it in the exercises.

Factoring Out Common Features. When we compare Deck and Shoe, we see two obviously common features: they both have a collection of `Cards`, called `self.cards`; they both have a `deal` method which yields the set of cards.

We also see things which are different. The most obvious differences are details of initializing `self.cards`. It turns out that the usual procedure for dealing from a shoe involves shuffling all of the cards, but dealing from only four or five of the six available decks. This is done by inserting a marker one or two decks in from the end of the shoe.

In factoring out the common features, we have a number of strategies.

- One of our existing classes is already generic-enough to be the superclass. In the `Card` example, we used the generic `Card` class as superclass for other cards as well as the class used to implement the number cards. In this case we will make concrete object instances from the superclass.

- We may need to create a superclass out of our subclasses. Often, the superclass isn't useful by itself; only the subclasses are really suitable for making concrete object instances. In this case, the superclass is really just an abstraction, it isn't meant to be used by itself.

Here's an abstract `CardDealer` from which we can subclass `Deck` and `Shoe`. Note that it does not create any cards. Each subclass must do that. Similarly, it can't deal properly because it doesn't have a proper `shuffle` method defined.

```
class CardDealer( object ):
    def __init__( self ):
        self.cards= []
    def deal( self ):
        for c in self.shuffle():
            yield c
    def shuffle( self ):
        return NotImplemented
```

Python does not have a formal notation for abstract or concrete superclasses. When creating an abstract superclass it is common to return `NotImplemented` or raise `NotImplementedError` to indicate that a method must be overridden by a subclass.

We can now rewrite `Deck` as subclasses of `CardDealer`.

```
class Deck( CardDealer ):
    def __init__( self ):
        super(Deck,self).__init__()
        for s in ("C","D","H","S"):
            Build the three varieties of cards

    def shuffle( self ):
        Randomize all cards, return all cards
```

We can also rewrite `Shoe` as subclasses of `CardDealer`.

```
class Shoe( CardDealer ):
    def __init__( self, decks=6 ):
        CardDealer.__init__( self )
        for i in range(decks):
            d= Deck()
            self.cards += d.cards

    def shuffle( self ):
        Randomize all cards, return a subset of the cards
```

The benefit of this is to assure that `Deck` and `Shoe` actually share common features. This is not "cut and paste" sharing. This is "by definition" sharing. A change to `CardDealer` will change both Deck and Shoe, assuring complete consistency.

Polymorphism

One of the four important features of class definition is polymorphism. Polymorphism exists when you define a number of subclasses which have commonly named methods. A function can use objects of any of the polymorphic classes without being aware that the classes are distinct.

In some languages, it is essential that the polymorphic classes have the same interface (or be subinterfaces of a common parent interface), or be subclasses of a common superclass. This is sometimes called "strong, hierarchical typing", since the type rules are very rigid and follow the subclass/subinterface hierarchy.

Python implements something that is less rigid, often called "duck typing". The phrase follows from a quote attributed to James Whitcomb Riley: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." In short, two objects are effectively of the class `Duck` if they have a common collection of methods (walk, swim and quack, for example.)

When we look at the examples for `Card`, `FaceCard`, `Ace` in the section called "Inheritance", we see that all three classes have the same method names, but have different implementations for some of these methods. These three classes are polymorphic. A client class like `Hand` can contain individual objects of any of the subclasses of `Card`. A function can evaluate these polymorphic methods without knowing which specific subclass is being invoked.

In our example, both `FaceCard` and `Ace` were subclasses of `Card`. This subclass relationship isn't necesary for polymorphism to work correctly in Python. However, the subclass relationship is often an essential ingredient in an overall design that relies on polymorphic classes.

What's the Benefit? If we treat all of the various subclasses of `Card` in a uniform way, we effectively delegate any special-case processing into the relevant subclass. We concentrate the implementation of a special case in exactly one place.

The alternative is to include **if**-statements all over our program to enforce special-case processing rules. This diffusing of special-case processing means that many components wind up with an implicit relationship. For example, all portions of a program that deal with `Cards` would need multiple **if**-statements to separate the number card points, face card points and ace points.

By making our design polymorphic, all of our subclasses of `Card` have ranks and suits, as well as hard and soft point values. We we can design the `Deck` and `Shoe` classes to deal cards in a uniform way. We can also design a `Hand` class to total points without knowing which specific class to which a `Card` object belongs.

Similarly, we made our design for `Deck` and `Shoe` classes polymorphic. This allows us to model one-deck blackjack or multi-deck blackjack with no other changes to our application.

The Hand of Cards. In order to completely model Blackjack, we'll need a class for keeping the player and dealer's hands. There are some differences between the two hands: the dealer, for example, only reveals their first card, and the dealer can't split. There are, however, some important similarities. Every kind of `Hand` must determine the hard and soft point totals of the cards.

The hard point total for a hand is simply the hard total of all the cards. The soft total, on the other hand, is not simply the soft total of all cards. Only the `Ace` cards have different soft totals, and only one Ace can meaningfully contribute it's soft total of 11. Generally, all cards provide the same hard and soft point contributions. Of the cards where the hard and soft values differ, only one such card needs to be considered.

Note that we are using the values of the `getHardValue` and `getSoftValue` methods. Since this test applies to all classes of cards, we preserve polymorphism by checking this property of every card. We'll preserving just one of the cards with a soft value that is different from the hard value. At no time do use investigate the class of a `Card` to determine if the card is of the class `Ace`. Examining the class of each object needlessly constrains our algorithm. Using the polymorphic methods means that we can make changes to the class structure without breaking the processing of the `Hand` class.

> **Pretty Poor Polymorphism**
>
> The most common indicator of poor use polymorphism is using the `type`, `isinstance` and `issubclass` functions to determine the class of an object. These should used rarely, if at all. All processing should be focused on what is different about the objects, not the class to which an object belongs.

We have a number of ways to represent the presence of a `Card` with a distinct hard and soft value.

- An attribute with the point difference (usually 10).
- A collection of all `Cards` except for one `Card` with a point difference, and a single attribute for the extra card.

We'll choose the first implementation. We can use use a sequence to hold the cards. When cards are added to the hand, the first card that returns distinct values for the hard value and soft value will be used to set a variable has keeps the hard vs. soft point difference.

Example 22.2. hand.py

```
class Hand( object ):
    """Model a player's hand."""
    def __init__( self ):
        self.cards = [ ]
        self.softDiff= 0
    def addCard( self, aCard ):
        self.cards.append( aCard )
        if aCard.getHardValue() != aCard.getSoftValue():
            if self.softDiff == 0:
                self.softDiff= aCard.getSoftValue()-aCard.getHardValue()
    def points( self ):
        """Compute the total points of cards held."""
        p= 0
        for c in self.cards:
            p += c.getHardValue()
        if p + self.softDiff <= 21:
            return p + self.softDiff
        else:
            return p
```

❶ The `__init__` special function creates the instance variable, `self.cards`, which we will use to accumulate the `Card` objects that comprise the hand. This also sets `self.softDiff` which is the difference in points between hard and soft hands. Until we have an Ace, the difference is zero. When we get an Ace, the difference will be 10.

❷ We provide an `addCard` method that places an additional card into the hand. At this time, we examine the `Card` to see if the soft value is different from the hard value. If so, and we have not set the `self.softDiff` yet, we save this difference.

❸ The `points` method evaluates the hand. It initializes the point count, `p`, to zero. We start a **for**-loop to assign each card object to `c`. We could, as an alternative, use a `reduce` function to perform this operation: `reduce( lambda a,b:a+b.getSoftValue(), self.cards, 0 )`.

If the total with the `self.softDiff` is 21 or under, we have a soft hand, and these are the total points. If the total with the `self.softDiff` is over 21, we have a hard hand. The hard hand may total more than 21, in which case, the hand is bust.

Built-in Functions

There are two built in functions of some importance to object oriented programming. These are used to determine the class of an object, as well as the inheritance hierarchy among classes.

`isinstance(`*`object`*`,` *`type`*`)` → boolean

> True if *`object`* is an instance of the given *`type`* or any of the subclasses of *`type`*.

`issubclass(`*`class`*`,` *`base`*`)` → boolean

> True if class *`class`* is a subclass of class *`base`*. This question is usually moot, because most programs are designed to provide the expected classes of objects. There are some occasions for deep paranoia; when working with untrusted software, your classes may need to be sure that other programmers are following the rules. In Java and C++, the compiler can check these situations. In Python, the compiler doesn't check this, so we may want to include run-time checks.

`super(`*`type`*`)`

> This will return the superclass of the given type.

All of the basic factory functions (`str`, `int`, `float`, `long`, `complex`, `unicode`, `tuple`, `list`, `dict`, `set`) are effectively class names. You can, therefore, use a test like `isinstance(`*`myParam`*`,int)` to confirm that the argument value provided to this

parameter is an integer. An additional class, `basestring` is the parent class of both `str` and `unicode`.

The following example uses the `isinstance` function to validate the type of argument values. First, we'll define a Roulette wheel class, `Wheel`, and two subclasses, `Wheel1` with a single zero and `Wheel2` with zero and double zero.

Example 22.3. wheel.py

```
import random
class Wheel( object ):
    def value( self ):
        return NotImplemented

class Wheel1( Wheel ):
    def value( self ):
        spin= random.randrange(37)
        return str(spin)

class Wheel2( Wheel ):
    def __init__( self ):
        self.values= ['00'] + map( str, range(37) )
    def value( self ):
        return random.randchoice( self.values )
```

❶ The `Wheel` class defines the interface for Roulette wheels. The actual class definition does nothing except show what the expected method functions should be. We could call this an abstract definition of a `Wheel`.

❷ The `Wheel1` subclass uses a simple algorithm for creating the spin of a wheel. The `value` method chooses a number between 0 and 36. It returns a string representation of the number. This has only a single zero.

❸ The `Wheel2` subclass creates an instance variable, `values`, to contain all possible results. This includes the 37 values from 0 to 36, plus an additional '00' value. The `value` method chooses one of these possible results.

The following function expects that its parameter, `w`, is one of the subclasses of `Wheel`.

```
def simulate( w ):
    if not isinstance( w, Wheel ):
        raise TypeError( "Must be a subclass of Wheel" )
    for i in range(10):
        print w.value()
```

In this case, the simulate function checks its argument, `w` to be sure that it is a subclass of `Wheel`. If not, the function raises the built in `TypeError`.

Collaborating with `max`, `min` and `sort`

The `min` and `max` functions can interact with our classes in relatively simple ways. Similarly, the `sort` method of a list can also interact with our new class definitions. In all three cases, a keyword parameter of *`key`* can be used to control which attributes are used for determining minimum, maximum or sort order.

The *`key`* parameter must be given a function, and that function is evaluated on each item that is being compared. Here's a quick example.

```
class Boat( object ):
    def __init__( self, name, loa ):
        self.name= name
        self.loa= loa

def byName( aBoat ):
    return aBoat.name
```

```
def byLOA( aBoat ):
    return aBoat.loa

fleet = [ Boat("KaDiMa", 18 ), Boat( "Emomo", 21 ), Boat("Leprechaun", 30 ) ]

first= min( fleet, key=byName )
print "Alphabetically First:", first
longest= max( fleet, key=byLOA )
print "Longest:", longest
```

As `min`, `max` or sort traverse the sequence doing comparisons among the objects, they evaluate the `key` function we provided. In this example, the provided function simply selects an attribute. Clearly the functions could do calculations or other operations on the objects.

Initializer Techniques

When we define a subclass, we are often extending some superclass to add features. One common design pattern is to do this by defining a subclass to use parameters in addition to those expected by the superclass. We must reuse the superclass constructor properly in our subclass.

Referring back to our `Card` and `FaceCard` example in the section called "Inheritance", we wrote an initializer in `FaceCard` that referred to `Card`. The `__init__` function evaluates `super(FaceCard,self).__init__`, passing the same arguments to the `Card`'s `__init__` function.

As an aside, in older programs, you'll often see an alternative to the `super` function. Some classes will have an explicit call to `Card.__init__( self, r, s )`. Normally, we evaluate `object.method()`, and the *`object`* is implicitly the *`self`* parameter. We can, it turns out, also evaluate `Class.method( object )`. This provides the `object` as the `self` parameter.

We can make use of the techniques covered in the section called "Advanced Parameter Handling For Functions" to simplify our subclass initializer.

```
class FaceCard( Card ):
    """Model a 10-point face card: J, Q, K."""
    def __init__( self, *args ):
        super(FaceCard,self).__init__( *args )
        self.label= ("J","Q","K")[self.rank-11]
        self.pval= 10
    def __str__( self ):
        return "%2s%s" % ( self.label, self.suit )
```

In this case we use the `def __init__( self, *args )` to capture all of the positional parameters in a single list, `args`. We then give that entire list of positional parameters to `Card.__init__`. By using the * operator, we tell Python to explode the list into individual positional parameters.

Let's look at a slightly more sophisticated example.

Example 22.4. boat.py

```
class Boat( object ):
    def __init__( self, name, loa ):
        """Create a new Boat( name, loa )"""
        self.name= name
        self.loa= loa
class Catboat( Boat )
    def __init__( self, sailarea, *args ):
        """Create a new Catboat( sailarea, name, loa )"""
```

```
        super(Catboat,self).__init__( *args )
        self.mainarea= sailarea
class Sloop( CatBoat ):
    def __init__( self, jibarea, *args );
        """Create a new Sloop( jibarea, mainarea, name, loa )"""
        super(Sloop,self).__init__( *args )
        self.jibarea= jibarea
```

❶ The Boat class defines essential attributes for all kinds of boats: the name and the length overall (LOA).

❷ In the case of a `Catboat`, we add a single sail area to be base definition of `Boat`. We use the superclass initialization to prepare the basic name and length overall attributes. Then our subclass adds the sailarea for the single sail on a catboat.

❸ In the case of a `Sloop`, we add another sail to the definition of a `Catboat`. We add the new parameter first in the list, and the remaining parameters are simply given to the superclass for its initialization.

Class Variables

The notion of object depends on having instance variables (or "attributes") which have unique values for each object. We can extend this concept to include variables that are not unique to each instance, but shared by every instance of the class. Class level variables are created in the class definition itself; instance variables are created in the individual class method functions (usually `__init__`).

Class level variables are usually "variables" with values that don't change; these are sometimes called manifest constants or named constants. In Python, there's no formal declaration for a named constant.

A class level variable that changes will be altered for all instances of the class. This use of class-level variables is often confusing to readers of your program. Class-level variables with state changes need a complete explanation.

This is an example of the more usual approach with class-level constants. These are variables whose values don't vary; instead, they exist to clarify and name certain values or codes.

Example 22.5. wheel.py

```
import random
class Wheel( object ):
    """Simulate a roulette wheel."""
    green, red, black= 0, 1, 2
    redSet= [1,3,5,7,9,12,14,16,18,19,21,23,25,27,30,32, 34,36]
    def __init__( self ):
        self.lastSpin= ( None, None )
    def spin( self ):
        """spin() -> ( number, color )

        Spin a roulette wheel, return the number and color."""
        n= random.randrange(38)
        if n in [ 0, 37 ]: n, color= 0, Wheel.green
        elif n in Wheel.redSet: color= Wheel.red
        else: color= Wheel.black
        self.lastSpin= ( n, color )
        return self.lastSpin
```

❶ Part of definition of the class `Wheel` includes some class variables. These variables are used by all instances of the class. By defining three variables, `green`, `red` and `black`, we can make our programs somewhat more clear. Other parts of our program that use the `Wheel` class can then reference the colors by name, instead of by an obscure numeric code. A program would use `Wheel.green` to refer to the code for green within the `Wheel` class.

❷ The `Wheel` class also creates a class-level variable called `redSet`. This is the set

of red positions on the Roulette wheel. This is defined at the class level because it does not change, and there is no benefit to having a unique copy within each instance of `Wheel`.

❸ The `__init__` method creates an instance variable called `lastSpin`. If we had multiple wheel objects, each would have a unique value for `lastSpin`. They all would all, however, share a common definition of `green`, `red`, `black` and `redSet`.

❹ The `spin` method updates the state of the wheel. Notice that the class level variables are referenced with the class name: `Wheel.green`. The instance level variables are referenced with the instance parameter: `self.lastSpin`. The class level variables are also available using the instance parameter, `Wheel.green` is the same object as `self.green`.

The `spin` method determines a random number between 0 and 37. The numbers 0 and 37 are treated as 0 and 00, with a color of green; a number in the `Wheel.redSet` is red, and any other number is black. We also update the state of the `Wheel` by setting `self.lastSpin`. Finally, the `spin` method returns a `tuple` with the number and the code for the color. Note that we can't easily tell 0 from 00 with this particular class definition.

The following program uses this `Wheel` class definition. It uses the class-level variables `red` and `black` to clarify the color code that is returned by `spin`.

```
w= Wheel()
n,c= w.spin()
if c == Wheel.red: print n, "red"
elif c == Wheel.black: print n, "black"
else: print n
```

Our sample program creates an instance of `Wheel`, called `w`. The program calls the `spin` method of `Wheel`, which updates `w.lastSpin` and returns the tuple that contains the number and color. We use multiple assignment to separate the two parts of the tuple. We can then use the class-level variables to decode the color. If the color is `Wheel.red`, we can print `"red"`.

Static Methods and Class Method

In a few cases, our class may have methods which depend on only the argument values, or only on class variables. In this case, the `self` variable isn't terribly useful, since the method doesn't depend on any attributes of the instance. Objects which depend on argument values instead of internal status are called **Lightweight** or **Flyweight** objects.

A method which doesn't use the `self` variable is called a static method. These are defined using a built-in function named `staticmethod`. Python has a handy syntax, called a decorator, to make it easier to apply the `staticmethod` function to our method function definition. We'll return to decorators in Chapter 26, *Decorators*.

Here's the syntax for using the `staticmethod` decorator.

```
@staticmethod

def name(args...): suite
```

To evaluate a static method function, we simply reference the method of the class: `Class.method()`.

Example of Static Method. Here's an example of a class which has a static method. We've defined a deck shuffler. It doesn't have any attributes of its own. Instead, it applies it's `shuffle` algorithm to a `Deck` object.

```
class Shuffler( object ):
```

```
    @staticmethod
    def shuffle( aDeck ):
        for i in range(len(aDeck)):
            card= aDeck.get( random.randrange(len(aDeck)) )
            aDeck.put( i, card )

d1= Deck()
Shuffler.shuffle( d1 )
```

Class Method. The notion of a class method is relatively specialized. A class method applies to the class itself, not an instance of the class. A class method is generally used for "introspection" on the structure or definition of the class. It is commonly defined by a superclass so that all subclasses inherit the necessary introspection capability.

Generally, class methods are defined as part of sophisticated, dynamic frameworks. For our gambling examples, however, we do have some potential use for class methods. We might want to provide a base `Player` class who interacts with a particular `Game` to make betting decisions. Our superclass for all players can define methods that would be used in a subclass.

Design Approaches

When we consider class design, we have often have a built-in or library class which does some of the job we want. For example, we want to be able to accumulate a list of values and then determine the average: this is a very list-like behavior, extended with a new feature.

There are two overall approaches for extending a class: *wrapping* and *inheritance*.

- Wrap an existing class (for example, a tuple, list, set or map) in a new class which adds features. This allows you to redefine the interface to the existing class, which often involves removing features.

- Inherit from an existing class, adding features as part of the more specialized subclass. This may require you to read more of the original class documentation to see a little of how it works internally.

Both techniques work extremely well; there isn't a profound reason for making a particular choice. When wrapping a collection, you can provide a new, focused interface on the original collection; this allows you to narrow the choices for the user of the class. When subclassing, however, you often have a lot of capabilities in the original class you are extending.

"Duck" Typing. In the section called “Polymorphism”, we mentioned "Duck" Typing. In Python, two classes are practically polymorphic if they have the same inteface methods. They do not have to be subclasses of the same class or interface (which is the rule in Java.)

This principle means that the distinction between wrapping and inheritance is more subtle in Python than in other languages. If you provide all of the appropriate interface methods to a class, it behaves as if it was a proper subclass. It may be a class that is wrapped by another class that provides the same interface.

For example, say we have a class Dice, which models a set of individual Die objects.

```
class Dice( object ):
    def __init__( self ):
        self.theDice= [ Die(), Die() ]
    def roll( self ):
        for d in self.theDice:
            d.roll()
        return self.theDice
```

In essence, our class is a wrapper around a list of dice, named `theDice`. However, we don't provide any of the interface methods that are parts of the built-in `list` class.

Even though this class is a wrapper around a `list` object, we can add method names based on the built-in `list` class: `append`, `extend`, `count`, `insert`, etc. We'll look closely at this in Chapter 24, *Creating or Extending Data Types*.

```
class Dice( object ):
    def __init__( self ):
        self.theDice= [ Die(), Die() ]
    def roll( self ):
        for d in self.theDice:
            d.roll()
        return self.theDice
    def append( self, aDie ):
        self.theDice.append( aDie )
    def __len__( self ):
        return len( self.theDice )
```

Once we've defined these list-like functions we have an ambiguous situation.

- We could have a subclass of `list`, which initializes itself to two `Die` objects and has a `roll` method.
- We could have a distinct `Dice` class, which provides a `roll` method and a number of other methods that make it look like a `list`.

For people who will read your Python, clarity is the most important feature of the program. In making design decisions, one of your first questions has to be "what is the real thing that I'm modeling?" Since many alternatives will work, your design should reflect something that clarifies the problem you're solving.

Advanced Class Definition Exercises

`Sample` Class with Statistical Methods

We can create a `Samples` class which holds a collection of sample values. This class can have functions for common statistics on the object's samples. For additional details on these algorithms, see the exercises in Chapter 13, *Tuples* and the section called "Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`".

We'll look at subclassing the built-in `list` class, by creating a class, `Samples`, which extends `list`. You'll need to implement the following methods in your new class.

`__init__`(*`*args`*)

> The `__init__` method saves a sequence of samples. It could, at this time, also precompute a number of useful values, like the sum, count, min and max of this set of data. When no data is provided, these values would be set to `None`.

__str__

> The `__str__` method should return string with a summary of the data. An example is a string like `"%d values, min %g, max %g, mean %g"` with the number of data elements, the minimum, the maximum and the mean. The superclass, `list`, `__repr__` function will return the raw data.

`mean`

> The `mean` method returns the sum divided by the count.

`mode`

The `mode` returns the most popular of the sample values. Below we've provided an algorithm that can be used to locate the mode of a sequence of samples.

`median`

The median is the value in the middle of the sequence. First, sort the sequence. If there is an odd number of elements, pick the middle-most element. If there is an even number of elements, average the two elements that are mid-most.

`variance`

The `variance` is a more complex calculation than the simple mean. For each sample, compute the difference between the sample and the mean, square this value, and sum these squares. The number of samples minus 1 is the degrees of freedom. The sum, divided by the degrees of freedom is the variance. Note that you need two samples to meaningfully compute a variance.

`stdev`

The `stdev` is the square root of the variance.

Note that the `list` superclass already works correctly with the built-in `min` and `max` functions. In this case, this consequence of using inheritance instead of wrapping turns out to be an advantage.

Procedure 22.1. Computing the Mode

1. **Initialize.** Create an empty dictionary, `freq`, for frequency distribution.
2. **For Each Value.** For each value, `v`, in the sequence of sample values.
    a. **Unknown?** If `v` is not a key in `freq`, then add `v` to the dictionary with a value of `0`.
    b. **Increment Frequency.** Increment the frequency count associated with `v` in the dictionary.
3. **Extract.** Extract the dictionary items as a sequence of tuples (*value*, *frequency*).
4. **Sort.** Sort the sequence of tuples in ascending order by frequency.
5. **Result.** The last value in the sorted sequence is a tuple with the most common sample value and the frequency count.

Shuffling Method for the `Deck` class

Shuffling is a matter of taking existing cards and putting them into other positions. There are a many ways of doing this. We'll need to try both to see which is faster. In essence, we need to create a polymorphic family of classes that we can use to measure performance.

Procedure 22.2. Shuffling Variation 1 - Exchange

- For `i` in range 0 to the number of cards
    a. Generate a random number `r` in the range 0 to the number of cards.
    b. Use Multiple Assignement to swap cards at position `i` and `r`.

Procedure 22.3. Shuffling Variation 2 - Build

1. Create an empty result sequence, `s`.

2. While there are cards in the source `self.cards` sequence.

    a. Generate a random number `r` in the range 0 to the number of cards.

    b. Append card `r` to the result sequence; delete object `r` from the source `self.cards` sequence. The `pop` method of a sequence can return a selected element and delete it from a sequence nicely.

3. Replace `self.cards` with the result sequence, `s`.

Procedure 22.4. Shuffling Variation 3 - Sort

1. Create a function which returns a random value selected from the sequence `( -1, 0, 1 )`. This is similar to the built-in `cmp` function.

2. Use the `sort` method of a `list` with this random comparison-like function.

```
self.cards.sort( lambda a,b: random.choice( (-1,0,1) ) )
```

Procedure 22.5. Shuffling Variation 4 - random.shuffle

- The `random` module has a `shuffle` method which can be used as follows.

```
random.shuffle( self.cards )
```

Of these four algorithms, which is fastest? The best way to test these is to create four separate subclasses of `Deck`, each of which provides a different implementation of the `shuffle` method. A main program can then create an instance of each variation on `Deck` and do several hundred shuffles.

We can create a timer using the `time` module. The `time.clock` function will provide an accurate time stamp. The difference between two calls to `time.clock` is the elapsed time.

Because all of our variations on `Deck` are polymorphic, our main program should look something like the following.

```
d1= DeckExch()
d2= DeckBuild()
d3= DeckSortRandom()
d4= DeckShuffle()
for deck in ( d1, d2, d3, d4 ):
    start= time.clock()
    for i in range(100):
        d.shuffle()
    finish= time.clock()
```

Encapsulation

The previous exercise built several alternate solutions to a problem. We are free to implement an algorithm with no change to the interface of `Deck`. This is a important effect of the principal of encapsulation: a class and the clients that use that class are only coupled together by an interface defined by method functions.

There are a variety of possible dependencies between a class and its clients.

- **Interface Method Functions.** A client can depend on method functions specifically designated as an interface to a class. In Python, we can define internal methods by prefixing their names with _. Other names (without the leading _) define the public interface.

- **All Method Functions.** A client can depend on all method functions of a class.

This removes the complexity of hidden, internal methods.

- **Instance Variables.** A client can depend on instance variables in addition to method functions. This can remove the need to write method functions that simply return the value of an instance variable.

- **Global Variables.** Both classes share global variables. The Python **global** statement is one way to accomplish this.

- **Implementation.** A client can depend on the specific algorithm being executed by a class. A client method can have expectations of how a class is implemented.

What are the advantages and disadvantages of each kind of dependency?

Class Responsibilities

Assigning responsibility to class can be challenging. A number of reasons can be used to justify the functions and instance variables that are combined in a single class.

- **Convenience.** A class is defined to do things because — well — it's convenient to write the program that way.

- **Similar Operations.** A class is defined because it does all input, all output, or all calculations.

- **Similar Time.** A class is defined to handle all initialization, all processing or all final cleanup.

- **Sequence.** We identify some operations which are performed in a simple sequence and bundle these into a single class.

- **Common Data.** A class is defined because it has the operations which isolate a data structure or algorithm.

What are the possible differences between theses? What are the advantages and disadvantages of each?

Style Notes

Classes are perhaps the most important organizational tool for Python programming. Python software is often designed as a set of interacting classes. There are several conventions for naming and documenting class definitions.

It is important to note that the suite within a class definition is typically indented four spaces. It is often best to set your text editor with tab stops every four spaces. This will usually yield the right kind of layout. Each function's suite is similarly indented four spaces, as are the suites within compound statements.

Blank lines are used sparingly; most typically a single blank line will separate each function definition within the class. A lengthy class definition, with a number of one-liner set-get accessor functions may group the accessors together without any intervening blank lines.

Class names are typically MixedCase with a leading uppercase letter. Members of the class (method functions and attributes) typically begin with a lowercase letter. Class names are also, typically singular nouns. We don't define `People`, we define `Person`. A collection might be a PersonList or PersonSet.

Note that the following naming conventions are honored by Python:

single_trailing_underscore_. Used to make a variable names different from a similar

Python reserved word. For example: range_ is a legal variable name.

_single_leading_underscore. Used to make variable or method names hidden. This conceals them from the `dir` function.

__double_leading_underscore. Class-private names. Use this to assure that a method function is not used directly by clients of a class.

__double_leading_and_trailing_underscore__. These are essentialy reserved by Python for its own internals.

The first line of a class body is the docstring; this provides an overview of the class. It should summarize the responsibilities and collaborators of the class. It should summarize the public methods, instance variables and particulary the `__init__` function used to construct instances of the class. Individual method functions are each documented in their own docstrings.

When defining a subclass, be sure to mention the specific features added (or removed) by the subclass. There are two basic cases: overriding and extending. When overriding a superclass method function, the subclass has replaced the superclass function. When extending a superclass function, the subclass method will call the superclass function to perform some of the work. The override-extend distinctions must be made clear in the docstring.

When initializing instance variables in the `__init__` function, a string placed after the assignment statement can serve as a definition of the variable.

```
class Dice( object ):
    """Model two dice used for craps.  Relies on Die class.

    theDice -- tuple with two Die instances
    Dice() -- initialize two dice
    roll() -- roll dice and return total
    """

    def __init__(self):
        self.theDice = ( Die(), Die() ) """The two simulated dice."""

    def roll(self):
        """Roll two dice and return the total."""
        map( lambda d: d.roll(), self.theDice )
        t = reduce( lambda d: d.face(), self.theDice, 0 )
        return t
```

Generally, we have been omitting a complete docstring header on each class in the interest of saving some space for the kind of small examples presented in the text.

Chapter 23. Some Design Patterns

Table of Contents

The best way to learn object-oriented design is to look at patterns for common solutions to ubiquitous problems. These patterns are often described with a synopsis that gives you several essential features. The writer of a pattern will describe a programming context, the specific problem, the forces that lead to various kinds of solutions, a solution that optimizes the competing forces, and any consequences of choosing this solution.

There are a number of outstanding books on patterns. We'll pick a few key patterns from one of the books, and develop representative classes in some depth. The idea is to add a few additional Python programming techniques, along with a number of class design techniques.

Most of these patterns come from the "Gang of Four" design patterns book, *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma99]. We'll look at a few design patterns that illustrate some useful Python programming techniques.

Factory Method

> This is a pattern for designing a class which is used as a factory for a family of other classes. This allows a client program to use a very flexible and extensible Factory to create necessary objects.

State

> This is a pattern for desiging a hierarchy of classes that describes states (or status) and state-specific processing or data.

Strategy

> This is a pattern that helps create a class that supports an extension in the form of alternative strategies for implementing methods.

Factory Method

When we add subclasses to a class hierarchy, we may also need to rearrange the statements where objects are created. To provide a flexible implementation, it is generally a good idea to centralize all of the object creation statements into a single method of class. When we extend the subclass hierarchy we can also create a relevant subclass of the centralized object creation class.

The design pattern for this kind of centralized object creator can be called a Factory. It contains the details for creating an instance of each of the various subclasses.

In the next section, Part IV, "Components, Modules and Packages", we'll look at how to package a class hierarchy in a module. Often the classes and the factory object are bundled into one seamless module. Further, as the module evolves and improves, we need to preserve the factory which creates instances of other classes in the module. Creating a class with a factory method helps us control the evolution of a module. If we omit the **Factory Method**, then everyone who uses our module has to rewrite their programs when we change our class hierarchy.

Extending the Card Class Hierarchy. We'll extend the `Card` class hierarchy, introduced in the section called "Inheritance". That original design had three classes: `Card`, `FaceCard` and `AceCard`.

While this seems complete for basic Blackjack, we may need to extend these classes. For example, if we are going to simulate a common card counting technique, we'll need to separate 2-6 from 7-9, leading to two more subclasses. Adding subclasses can easily ripple through an application, leading to numerous additional, sometimes complex changes. We would have to look for each place where the various subclasses of cards were created. The **Factory** design pattern, however, provides a handy solution to this

problem.

An object of a class based on the **Factory** pattern creates instances of other classes. This saves having to place creation decisions throughout a complex program. Instead, all of the creation decision-making is centralized in the factory class.

For our card example, we can define a `CardFactory` that creates new instances of `Card` (or the appropriate subclass.)

```
class CardFactory( object ):
    def newCard( self, rank, suit ):
        if rank == 1:
            return Ace( rank, suit )
        elif rank in [ 11, 12, 13 ]:
            return FaceCard( rank, suit )
        else:
            return Card( rank, suit )
```

We can simplify our version of `Deck` using this factory.

```
class Deck( object ):
    def __init__( self ):
        factory= CardFactory()
        self.cards = [ factory.newCard( rank+1, suit )
            for suit in range(4)
                for rank in range(13) ]
    Rest of the class is the same
```

Centralized Object Creation

While it may seem like overhead to centralize object creation in factory objects, it has a number of benefits.

First, and foremost, centralizing object creation makes it easy to locate the one place where objects are constructed, and fix the constructor. Having object construction scattered around an application means that time is spent searching for and fixing things that are, in a way, redundant.

Additionally, centralized object creation is the norm for larger applications. When we break down an application into the data model, the view objects and the control objects, we find at least two kinds of factories. The data model elements are often created by fetching from a database, or parsing an input file. The control objects are part of our application that are created during initialization, based on configuration parameters, or created as the program runs based on user inputs.

Finally, it makes evolution of the application possible when we are creating a new version of a factory rather than tracking down numerous creators scattered around an application. We can assure ourselves that the old factory is still available and still passes all the unit tests. The new factory creates the new objects for the new features of the application software.

Exetending the Factory. By using this kind of Factory method design pattern, we can more easily create new subclasses of Card. When we create new subclasses, we do three things:

1. Extend the `Card` class hierarchy to define the additional subclasses.

2. Extend the `CardFactory` creation rules to create instances of the new subclasses. This is usually done by creating a new subclass of the factory.

3. Extend or update `Deck` to use the new factory. We can either create a new subclass of `Deck`, or make the factory object a parameter to `Deck`.

Let's create some new subclasses of `Card` for card counting. These will subdivide the number cards into low, neutral and high ranges. We'll also need to subclass our existing `FaceCard` and `Ace` classes to add this new method.

```
class CardHi( Card ):
    """Used for 10."""
    def count( self ): return -1

class CardLo( Card ):
    """Used for 3, 4, 5, 6, 7."""
    def count( self ): return +1

class CardNeutral( Card ):
    """Used for 2, 8, 9."""
    def count( self ): return 0

class FaceCount( FaceCard ):
    """Used for J, Q and K"""
    def count( self ): return -1

class AceCount( Ace ):
    """Used for A"""
    def count( self ): return -1
```

A counting subclass of `Hand` can sum the `count` values of all `Card` instances to get the count of the deck so far.

Once we have our new subclasses, we can create a subclass of `CardFactory` to include these new subclasses of `Card`. We'll call this new class `HiLoCountFactory`. This new subclass will define a new version of the `newCard` method that creates appropriate objects.

By using default values for parameters, we can make this factory option transparent. We can design `Deck` to use the original `CardFactory` by default. We can also design `Deck` to accept an optional `CardFactory` object, which would tailor the `Deck` for a particular player strategy.

```
class Deck( object ):
    def __init__( self, factory=CardFactory() ):
        self.cards = [ factory.newCard( rank+1, suit )
            for suit in range(4)
                for rank in range(13) ]
    Rest of the class is the same
```

The Overall Main Program. Now we can have main programs that look something like the following.

```
d1 = Deck()
d2 = Deck(HiLoCountFactory())
```

In this case, `d1` is a `Deck` using the original definitions, ignoring the subclasses for card counting. The `d2` `Deck` is built using a different factory and has cards that include a particular card counting strategy.

We can now introduce variant card-counting schemes by introducing further subclasses of `Card` and `CardFactory`. To pick a particular set of card definitions, the application creates an instance of one of the available subclasses of `CardFactory`. Since all

subclasses have the same `newCard` method, the various objects are interchangeable. Any `CardFactory` object can be used by `Deck` to produce a valid deck of cards.

This evolution of a design via new subclasses is a very important technique of object-oriented programming. If we add features via subclasses, we are sure that the original definitions have not been disturbed. We can be completely confident that adding a new feature to a program will not break old features.

State

Objects have state changes. Often the processing that an object performs depends on the state. In non-object-oriented programming languages, this state-specific processing is accomplished with long, and sometimes complex series of **if** statements. The **State** design pattern gives us an alternative design.

As an example, the game of Craps has two states. A player's first dice roll is called a *come out* roll. Depending on the number rolled, the player immediately wins, immediately loses, or the game transitions to a *point* roll. The game stays in the point roll state until the player makes their point or crap out with a seven. The following table provides a complete picture of the state changes and the dice rolls that cause those changes.

Table 23.1. Craps Game States

State	Roll	Bet Resolution	Next State
Point Off; the Come Out Roll; only Pass and Don't Pass bets allowed.	2, 3, 12	"Craps": Pass bets lose, Don't Pass bets win.	Point Off
	7, 11	"Winner": Pass bets win, Don't Pass bets lose.	Point Off
	4, 5, 6, 8, 9, 10	No Resolution	Point On the number rolled, *p*.
Point On; any additional bets may be placed.	2, 3, 12	No Resolution	Point still on
	11	No Resolution	Point still on
	7	"Loser": all bets lose. The table is cleared.	Point Off
	Point, *p*	"Winner": point is made, Pass bets win, Don't Pass bets lose.	Point Off
	Non-*p* number	Nothing; Come bets are activated	Point still on

The **State** design pattern is essential to almost all kinds of programs. The root cause of the hideous complexity that characterizes many programs is the failure to properly use the **State** design pattern.

The `Craps` Class. The overall game of craps can be represented in an object of class `Craps`. A `Craps` object would have a `play1Round` function to initialize the game in the come out roll state, roll dice, pay off bets, and possibly change states.

Following the **State** design pattern, we will delegate state-specific processing to an object that represents just attributes and behaviors unique to each state of the game. We pan to create a `CrapsState` class with two subclasses: `CrapsStateComeOutRoll` and `CrapsStatePointRoll`.

The overall `Craps` object will pass the dice roll to the `CrapsState` object for evaluation. The `CrapsState` object calls methods in the original `Craps` object to pay or collect when there is a win or loss. The `CrapsState` object can also return an object for the next state. Additionally, the `CrapsState` object will have to indicate then the game

actually ends.

We'll look at the `Craps` object to see the context in which the various subclasses of `CrapsState` must operate.

Example 23.1. craps.py

```
import dice
class Craps( object ):
    """Simple game of craps."""
    def __init__( self ):
        self.state= None
        self.dice= dice.Dice()
        self.playing= False
    def play1Round( self ):
        """Play one round of craps until win or lose."""
        self.state= CrapsStateComeOutRoll()
        self.playing= True
        while self.playing:
           self.dice.roll()
           self.state= self.state.evaluate( self, self.dice )
    def win( self ):
        """Used by CrapsState when the roll was a winner."""
        print "winner"
        self.playing= False
    def lose( self ):
        """Used by CrapsState when the roll was a loser."""
        print "loser"
        self.playing= False
```

❶ The `Craps` class constructor, `__init__`, creates three instance variables: `state`, `dice` and `playing`. The `state` variable will contain an instance of `CrapsState`, either a `CrapsStateComeOutRoll` or a `CrapsStatePointRoll`. The `dice` variable contains an instance of the class `Dice`, defined in the section called "Class Definition: the **class** Statement". The `playing` variable is a simple switch that is `True` while we the game is playing and `False` when the game is over.

❷ The `play1Round` method sets the `state` to `CrapsStateComeOutRoll`, and sets the `playing` variable to indicate that the game is in progress. The basic loop is to roll the dice and the evaluate the dice.

This method calls the state-specific `evaluate` function of the current `CrapsState` object. We give this method a reference to overall game, via the `Craps` object. That reference allows the `CrapsState` to call the `win` or `lose` method in the `Craps` object. The `evaluate` function of `CrapsState` is also given the `Dice` object, so it can get the number rolled from the dice. Some propositions (called "hardways") require that both dice be equal; for this reason we pass the actual dice to `evaluate`, not just the total.

❸ ❹ When the `win` or `lose` method is called, the game ends. These methods can be called by the the `evaluate` function of the current `CrapsState`. The `playing` variable is set to `False` so that the game's loop will end.

The `CrapsState` Class Hierarchy. Each subclass of `CrapsState` has a different version of the `evaluate` operation. Each version embodies one specific set of rules. This generally leads to a nice simplification of those rules; the rules can be stripped down to simple **if** statements that evaluate the dice in one state only. No additional **if** statements are required to determine what state the game is in.

```
class CrapsState( object ):
    """Superclass for states of a craps game."""
    def evaluate( self, crapsGame, dice ):
        raise NotImplementedError
    def __str__( self ):
        return self.__doc__
```

The `CrapsState` superclass defines any features that are common to all the states. One common feature is the definition of the `evaluate` method. The body of the method is uniquely defined by each subclass. We provide a definition here as a formal place-holder for each subclass to override. In Java, we would declare the class and this function as abstract. Python lacks this formalism, but it is still good practice to include a placeholder.

Subclasses for Each State. The following two classes define the unique evaluation rules for the two game states. These are subclasses of `CrapsState` and inherit the common operations from the superclass.

```
class CrapsStateComeOutRoll ( CrapsState ):
    """Come out roll rules."""
    def evaluate( self, crapsGame, dice ):
        if dice.total() in [ 7, 11 ]:
            crapsGame.win()
            return self
        elif dice.total() in [ 2, 3, 12 ]:
            crapsGame.lose()
            return self
        return CrapsStatePointRoll( dice.total() )
```

The `CrapsStateComeOutRoll` provides an `evaluate` function that defines the come out roll rules. If the roll is an immediate win (7 or 11), it calls back to the `Craps` object to use the `win` method. If the roll is an immediate loss (2, 3 or 12), it calls back to the `Craps` object to use the `lose` method. In all cases, it returns an object which is the next state; this might be the same instance of `CrapsStateComeOutRoll` or a new instance of `CrapsStatePointRoll`.

```
class CrapsStatePointRoll ( CrapsState ):
    """Point roll rules."""
    def __init__( self, point ):
        self.point= point
    def evaluate( self, crapsGame, dice ):
        if dice.total() == 7:
            crapsGame.lose()
            return None
        if dice.total() == self.point:
            crapsGame.win()
            return None
        return self
```

The `CrapsStatePointRoll` provides an `evaluate` function that defines the point roll rules. If a seven was rolled, the game is a loss, and this method calls back to the `Craps` object to use the `lose` method, which end the game. If the point was rolled, the game is a winner, and this method calls back to the `Craps` object to use the `win` method. In all cases, it returns an object which is the next state. This might be the same instance of `CrapsStatePointRoll` or a new instance of `CrapsStateComeOutRoll`.

Extending the State Design. While the game of craps doesn't have any more states, we can see how additional states are added. First, a new state subclass is defined. Then, the main object class and the other states are updated to use the new state.

An additional feature of the state pattern is its ability to handle state-specific conditions as well as state-specific processing. Continuing the example of craps, the only bets allowed on the come out roll are pass and don't pass bets. All other bets are allowed on the point rolls.

We can implement this state-specific condition by adding a `validBet` method to the `Craps` class. This will return `True` if the bet is valid for the given game state. It will return `False` if the bet is not valid. Since this is a state-specific condition, the actual processing must be delegated to the `CrapsState` subclasses.

Strategy

Objects can often have variant algorithms. The usual textbook example is an object that has two choices for an algorithm, one of which is slow, but uses little memory, and the other is fast, but requires a lot of storage for all that speed. In our examples, we can use the **Strategy** pattern to isolate the details of a betting strategy from the rest of a casino game simulation. This will allow us to freely add new betting strategies without disrupting the simulation.

One strategy in Roulette is to always bet on black. Another strategy is to wait, counting red spins and bet on black after we've seen six or more reds in a row. These are two alternate player strategies. We can separate these betting decision algorithms from other features of player.

We don't want to create an entire subclass of player to reflect this choice of algorithms. The Strategy design pattern helps us break something rather complex, like a Player, into separate pieces. The essential features are in one object, and the algorithm(s) that might change are in separate strategy object(s). The essential features are defined in the core class, the other features are strategies that are used by the core class. We can then create many alternate algorithms as subclasses of the plug-in Strategy class. At run time, we decide which strategy object to plug into the core object.

The Two Approaches. As mentioned in the section called “Design Approaches”, we have two approaches for extending an existing class: wrapping and inheritance. From an overall view of the collection of classes, the Strategy design emphasizes wrapping. Our core class is a kind of wrapper around the plug-in strategy object. The strategy alternatives, however, usually form a proper class hierarchy and are all polymorphic.

Let's look at a contrived, but simple example. We have two variant algorithms for simulating the roll of two dice. One is quick and dirty and the other more flexible, but slower.

First, we create the basic `Dice` class, leaving out the details of the algorithm. Another object, the strategy object, will hold the algorithm

```
class Dice( object ):
    def __init__( self, strategy ):
        self.strategy= strategy
        self.lastRoll= None
    def roll( self ):
        self.lastRoll= self.strategy.roll()
        return self.lastRoll
    def total( self ):
        return reduce( lambda a,b:a+b, self.lastRoll, 0 )
```

The `Dice` class rolls the dice, and saves the roll in an instance variable, `lastRoll`, so that a client object can examine the last roll. The `total` method computes the total rolled on the dice, irrespective of the actual strategy used.

The Strategy Class Hierarchy. When an instance of the `Dice` class is created, it must be given a strategy object to which we have delegated the detailed algorithm. A strategy object must have the expected interface. The easiest way to be sure it has the proper interface is to make each alternative a subclass of a strategy superclass.

```
import random
class DiceStrategy( object ):
    def roll( self ):
        raise NotImplementedError
```

The `DiceStrategy` class is the superclass for all dice strategies. It shows the basic method function that all subclasses must override. We'll define two subclasses that provide alternate strategies for rolling dice.

The first, `DiceStrategy1` is simple.

```
class DiceStrategy1( DiceStrategy ):
    def roll( self ):
        return ( random.randrange(6)+1, random.randrange(6)+1 )
```

This `DiceStrategy1` class simply uses the `random` module to create a tuple of two numbers in the proper range and with the proper distribution.

The second alternate strategy, `DiceStrategy2`, is quite complex.

```
class DiceStrategy2( DiceStrategy ):
    class Die:
        def __init__( self, sides=6 ):
            self.sides= sides
        def roll( self ):
            return random.randrange(self.sides)+1
    def __init__( self, set=2, faces=6 ):
        self.dice = tuple( DiceStrategy2.Die(faces) for d in range(set) )
    def roll( self ):
        return tuple( x.roll() for x in self.dice )
```

This `DiceStrategy2` class has an internal class definition, `Die` that simulates a single die with an arbitrary number of faces. An instance variable, `sides` shows the number of sides for the die; the default number of sides is six. The `roll` method returns are random number in the correct range.

The `DiceStrategy2` class creates a number of instances of `Die` objects in the instance variable `dice`. The default is to create two instances of `Die` objects that have six faces, giving us the standard set of dice for craps. The `roll` function creates a tuple by applying a `roll` function to each of the `Die` objects in `self.dice`.

Creating Dice with a Plug-In Strategy. We can now create a set of dice with either of these strategies.

```
dice1= Dice( DiceStrategy1() )
dice2 = Dice( DiceStrategy2() )
```

The `dice1` instance of Dice uses an instance of the `DiceStrategy1` class. This strategy object is used to constuct the instance of `Dice`. The `dice2` variable is created in a similar manner, using an instance of the `DiceStrategy2` class.

Both `dice1` and `dice2` are of the same class, `Dice`, but use different algorithms to achieve their results. This technique gives us tremendous flexibility in designing a program.

Multiple Patterns. Construction of objects using the strategy pattern works well with a **Factory Method** pattern, touched on in the section called “Factory Method”. We could, for instance, use a Factory Method to decode input parameters or command-line options. This give us something like the following.

```
class MakeDice( object ):
    def newDice( self, strategyChoice ):
        if strategyChoice == 1:
            strat= DiceStrategy1()
        else:
            strat= DiceStrategy2()
        return Dice( strat )
```

This allows a program to create the Dice with something like the following.

```
dice = MakeDice().newDice( someInputOption )
```

When we add new strategies, we can also subclass the `MakeDice` class to include those new strategy alternatives.

Design Pattern Exercises

Alternate Counting Strategy

A simple card counting strategy in Blackjack is to score +1 for cards of rank 3 to 7, 0 for cards of rank 2, 8 and 9 and -1 for cards 10 to King and Ace. The updates to the `Card` class hierarchy are shown in the text.

Create a subclass of `CardFactory`, which replaces `newCard` with a version that creates the correct subclass of `Card`, based on the rank.

Six Reds

A common strategy in Roulette is to wait until six reds in a row are spun and then start betting on only black. There are three player betting states: waiting, counting and betting.

A full simulation will require a `RouletteGame` class to spin the wheel and resolve bets, a `Wheel` object to produce a sequence of random spins, and a `Table` to hold the individual bets. We'd also need a class to represent the `Bets`. We'll skim over the full game and try to focus on the player and player states.

A `Player` has a *stake* which is their current pool of money. The `RouletteGame` offers the `Player` an opportunity to bet, and informs the player of the resulting spin of the wheel. The `Player` uses a `SixRedsState` to count reds and bet on black.

The various `SixRedsState` classes have two methods, a `bet` method decides to bet or not bet, and an `outcome` method that records the outcome of the previous spin. Each class implements these methods differently, because each class represents a different state of the player's betting policy.

counting

> In the counting state, the player is counting the number of reds in a row. If a red was spun and the count is < 6, add one to a red counter and stay in this state. If a red is spun and the count is = 6, add one to a red counter and transition to the betting state. If black or green is spun, reset the count to zero.

betting

> In the betting state, the player is betting on black. In a more advanced version, the player would also increase their bet for each red count over six. If a red was spun, add one to a red counter and stay in the betting state. If black was spun, transition to the counting state. If green was spun, transition to the counting state.

> **Caution**
>
> We'll focus on the `SixRedsState` design. We won't spend time on the actual betting or payouts. For now, we can simply log wins and losses with a **print** statement.

First, build a simple `Player` class, that has the following methods.

__init__(self, stake)

> Sets the player's initial stake. For now, we won't do much with this. In other

player strategies, however, this may influence the betting.

More importantly, this will set the initial betting state of Counting.

__str__(self)

Returns a string that includes the current stake and state information for the player.

getBet(self)

This will use the current state to determine what bet (if any) to place.

outcome(self, spin)

This will provide the color information to the current state. It will also update the player's betting state with a state object returned the current state. Generally, each state will simple return a copy of itself. However, the counting state object will return a betting state object when six reds have been seen in a row.

Second, create a rudimentary `RouletteGame` that looks something like the following.

__init__(self, player)

A `RouletteGame` is given a `Player` instance when it is constructed.

__str__(self)

It's not clear what we'd display. Maybe the player?

play1Round(self)

The `play1Round` method gets a bet from the `Player` object, spins the wheel, and reports the spin back to the `Player` object. A more complete simulation would also resolve the bets, and increase the player's stake with any winnings.

Note that calling the `Player`'s `outcome` method does two things. First, it provides the spin to the player

playRounds(self, rounds=12)

The `playRounds` is a simple loop that calls `self.play1Round` as many times as required.

For guidance on designing the `Wheel` used by the `RouletteGame`, see the section called "Class Variables" and the section called "Function Definition: The **def** and **return** Statements".

State Class Hierarchy. The best approach is to get the essential features of `RouletteGame`, `Wheel` and `Player` to work. Rather than write a complete version of the player's `getBet` and `outcome` methods, we can use simple place-holder methods that simply print out the status information. Once we have these objects collaborating, then the three states can be introduced.

The superclass, `SixRedsState`, as well as the two subclasses, would all be similar to the following.

__init__(self, player)

The superclass initialization saves the player object. Some subclasses will initialize a count to zero.

__str__(self)

The superclass `__str__` method should return a `NotImplemented` value, to indicate that the superclass was used improperly.

getBet(self)

The `getBet` either returns `None` in the waiting and counting states, or returns a bet on red in the betting state. The superclass can return `None`, since that's a handy default behavior.

outcome(self, spin)

The `outcome` method is given a tuple with a number and a color. Based on the rules given above, each subclass of `SixRedsState` will do slightly different things. The superclass can return `NotImplemented`.

We need to create two subclasses of `SixRedState`:

SixRedCounting

In this state, the `getBet` method returns `None`; this behavior is defined by the superclass, so we don't need to implement this method. The `outcome` method checks the spin. If it is Red, this object increments the count by one. If it is black it resets the count to zero. If the count is six, return an instance of `SixRedBetting`. Otherwise, return `self` as the next state.

SixRedBetting

In this state, the `getBet` method returns a bet on Black; for now, this can be the string `"Black"`. The `outcome` method checks the spin. If it is Red, this object increments the count by one and returns `self`. If the spin is black it returns an instance of `SixRedCounting`. This will stop the betting and start counting.

Once we have the various states designed, the `Player` can then be revised to initialize itself with an instance of the wating class, and then delegate the `getBet` request from the game to the state object, and delegate the `outcome` information from the game to the state object.

Roulette Wheel Alternatives

There are several possible implementations of the basic Roulette wheel. One variation simply uses `random.randrange` to generate numbers in the range of 0 to 37, and treats 37 as double zero. To separate double zero from zero, it's best to use character string results.

Another strategy is to create a sequence of 38 strings ('00', '0', '1', etc.) and use `random.choice` to pick a number from the sequence.

Create a superclass for `WheelStrategy` and two subclasses with these variant algorithms.

Create a class for `Wheel` which uses an instance of `WheelStrategy` to get the basic number. This class for `Wheel` should also determine whether the number is red, black or green. The `Wheel` class `spin` function should return a tuple with the number and the color.

Create a simple test program to create an instance of `Wheel` with an instance of `WheelStrategy`. Collect 1000 spins and print the frequency distribution.

Shuffling Alternatives

Shuffling rearranges a deck or shoe of multiple decks; there are many possible algorithms. First, you will need a `Card` class to keep basic rank and suit information. Next, you will need a basic `Deck` class to hold cards. See the section called "Playing Cards and Decks" for additional details.

We looked at shuffling in an earlier exercise, but packaged it as part of the `Deck` class, not as a separate strategy. See the section called "Advanced Class Definition Exercises". We can rework those exercises to separate shuffing from `Deck`.

The `Deck` class must have a `shuffle` function; but this should simply call a method of the shuffle strategy object. Because the `Deck` is a collection of `Cards`, the `Deck` object should be passed to the shuffle strategy. The call would like something like this:

```
class Deck( object ):
    Other parts of Deck
    def shuffle( self ):
        self.shuffleStrategy.shuffle( self )
```

Create a superclass for shuffle strategies. Create a subclass for each of the following algorithms:

- For each card position in the deck, exchange it with a card position selected randomly
- For even-numbered card position (positions 0, 2, 4, 6, etc.) exchange it with an odd-numbered card position, selected randomly (random.choice from 1, 3, 5, 7, 9, etc.)
- Swap two randomly-selected positions; do this 52 times

Create a simple test program that creates a `Deck` with each of the available a `Shuffle` strategy objects.

Shuffling Quality

An open issue in the shuffling exercise is the statistical quality of the shuffling actually performed. Statistical tests of random sequences are subtle, and more advanced than we can cover in this set of exercises. What we want to test is that each card is equally likely to land in each position of the deck.

We can create a dictionary, with the key of each card, and the item associated with that key is a list of positions in which the card occured. We can evaluate a shuffle algorithm as follows.

Procedure 23.1. Test A Shuffle

1. **Setup**

   Create a `Deck`.

   Create an empty dictionary, `positions` for recording card positions.

   For each `Card` in the `Deck`, insert the `Card` in the `positions` dictionary; the value associated with the `Card` is a unique empty `list` used to record the positions at which this `Card` is found.

2. **For Each Strategy.** Perform the following evaluation for an instance of each `Shuffle` class, `s`.

   a. **Create Deck.** Set the `Deck`'s current shuffle strategy to `s`.

b. **Shuffle.** Shuffle the `Deck`.

c. **Record Positions.** For each card in the deck.

i. **Record Position.** Locate the `Card`'s position list in the `positions` dictionary; append the position of this `Card` to the list in the `positions` dictionary.

d. **Chi-Squared.** The chi-squared statistical test can be used to compare the actual frequency histogram to the expected frequency histogram. If you shuffle each deck 520 times, a given card should appear in each of the positions approximately 10 times. Ideally, the distribution is close to flat, but not exactly.

The chi-squared test compares sequence of actual frequencies, a, and a sequence of expected frequencies, e. It returns the chi-squared metric for the comparison of these two sequences. Both sequences must be the same length and represent frequencies in the same order.

Equation 23.1. Chi-Squared

$$\chi^2 = \sum_{0 \le i < n} \frac{(a_i - e_i)^2}{e_i}$$

We can use the built-in `zip` function to interleave the two lists, creating a sequence of tuples of `( actual, expected )`. This sequence of tuples can be used with the multiple-assignment **for** loop to assign a value from actual to one variable, and a value from expected to another variable. This allows a simple, elegant **for** statement to drive the basic comparison function.

Chapter 24. Creating or Extending Data Types

Table of Contents

When we use an operator, like + or *, what happens depends on the types of the objects involved. When we say `c*2`, the value depends on the type of `c`. If `c` is numeric, then 2 may have to be converted to the same type as `c`, and the answer will be a number. If, however, `c` is a sequence, the result is a new sequence.

```
>>> c=8.0
>>> c*2
16.0
>>> c="8.0"
>>> c*2
```

```
'8.08.0'
>>> c=(8,0)
>>> c*2
(8, 0, 8, 0)
```

The selection of appropriate behavior is accomplished by the relatively simple mechanism of "special method names" within Python. Each class of objects, either built-in or created by a programmer, can provide the required special method names to create the intimate relationship between the class, the built-in functions and the mathematical operators.

If you provide special methods, you can make your class behave like a built-in class. Your class can participate seamlessly with built-in Python functions like `str`, `len`, `repr`. Your class can also participate with the usual mathematical operators like `+` and `*`. Additionally, your class could also use the collection operators in a manner similar to a `map` or `list`.

The special method names are similar in principle to operator overloading provided in C++. Similar considerations apply: you should make the operator behave in way that parallels the mathematical sense of the symbol.

Additionally, we can extend built-in classes. We do this by extending some of the special methods to do additional or different things.

Semantics of Special Methods

Python has a number of language features that interact with the built-in data types. For example, objects of all built-in types can be converted to strings. You can use the built-in `str` function to perform these conversions. The `str` function invokes the `__str__` special method of the given object. In effect, `str(a)` is evaluated as `a.__str__()`.

When you create your own class, however, a built-in function like `str` or `cmp` can't easily determine how to perform these functions on your new class. Your class must supply the specially-named method function that the built-in `str` function can use to successfully convert your classes values to strings.

In the section called "Special Method Names" we introduced a few special method names. We looked at `__init__`, which is evaluated implicitly when an object is created. We looked at `__str__`, which is used by the `str` function and `__cmp__`, which is evaluated by the `cmp` function.

A huge number of Python features work through these special method names. When you provide appropriate special methods for your class, it behaves more like a built-in class.

You may be suspicious that the special method name `__str__` matches the built-in function `str`. There is no simple, obvious rule. Many of the built-in functions invoke specially-named methods of the class that are similar. The operators and other special symbols, however, can't have a simple rule for pairing operators with special methods. You'll have to actually read the documentation for built-in functions (*Library Reference*, section 2.1) and special method names (*Language Reference*, section 3.3) to understand all of the relationships.

Categories of Special Method Names. The special methods fall into several broad categories. The categories are defined by the kind of behavior your class should exhibit.

Basic Object Behaviors

> A number of special method names make your object behave like oher built-in objects. These special methods make your class respond to `str`, `repr`, `cmp` and comparison operators. This also includes methods that allow your object to respond to the `hash` function, which allows instances of your class to be a key to

a mapping.

Numeric Behaviors

These special methods allow your class to respond to the artithmetic operators: +, -, *, /, %, **, <<, >>, **and**, **or** and **not**. When you implement these special methods, your class will behave like the built-in numeric types.

Container Behaviors

If your new class is a container or collection, there are a number of methods required so that your class can behave like the built-in collection types (sequence, set, mapping).

Iterator Behavior

An iterator has a unique protcol. The **for** statement requires an `__iter__` method to locate an iterator for an object. It then requires a `next` method on the iterator.

Attribute Handling Behavior

Some special methods customize how your class responds to the `.` operator for manpulating attributes. For example, when you evaluate `object.attr`. This is commonly used when attribute manipulation is more complex than simply locating an attribute that was defined by `__init__`.

Function Behavior

You can make your object behave like a function. When you define the method `__call__`, your object is *callable*, and can be used as if it was a function.

Statement Interaction

There are a few special methods required by statements. The **for** statement requires an `__iter__` method to locate an iterator for an object. It then requires a `next` method on the iterator. The **with** statement requires `__enter__` and `__exit__` methods.

Basic Special Methods

In addition to `__init__` and `__str__` there are a number of methods which are appropriate for classes of all kinds.

`__init__(` *`self`*`,` *`args...`* `)`

Called when a new instance of the is created. Not that this overrides any superclass `__init__` method; to to superclass initialization first, you must evaluate the superclass `__init__` like this: `super(`*`Class`*`,self).__init__(`*`args...`*`)`. The `super` function identifies the superclass of your class, *`Class`*.

`__del__(` *`self`* `)`

Called when the this object is no longer referenced anywhere in the running program; the object is about to be removed by garbage collection. This is rarely used. Note that this is called as part of Python garbage collection; it is not called by the **del** statement.

`__repr__(` *`self`*`)` → String

Called by the `repr` built-in function. Typically, the string returned by this will look like a valid Python expression to reconstruct the object.

`__str__`(*`self`*) → String

Called by the `str` built-in function. This is called implicitly by the **print** statement to convert an object to a convenient, "pretty" string representation.

`__cmp__`(*`self`*, *`other`*) → int

Called by all comparison operations, including the `cmp` function. This must return a negative integer if `self < other`, zero if `self == other`, a positive integer if `self > other`. If no `__cmp__` operation is defined, class instances are compared by object identity.

In addition to the basic `__cmp__`, there are additional methods which can be used to define the behaviors of individual comparison operators. These include `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, and `__ge__` which match the operators `<`, `<=`, `==`, `!=`, `>` and `>=`.

`__hash__`(*`self`*) → int

Called during dictionary operations, and by the built-in function `hash` to transform an object to a unique 32-bit integer hash value. Objects which compare equal (`__cmp__` returns 0) should also have the same hash value. If a class does not define a `__cmp__` method it should not define a `__hash__` operation either. Classes with mutable objects can define `__cmp__` but should not define `__hash__`, or objects would move around in the dictionary.

`__nonzero__`(*`self`*) → boolean

Called during truth value testing; must return 0 or 1. If this method is not defined, and `__len__` is defined, then `__len__` is called based on the assumption that this is a collection. If neither function is defined, all values are considered `True`.

Special Attribute Names

As part of creating a class definition, Python adds a number of special attributes. These are informational in nature, and cannot not be easily be changed except by redefining the class or function, or reimporting the module.

`__name__`

The class name.

`__module__`

The module in which the class was defined.

`__dict__`

The dictionary which contains the object's attributes and methods.

`__bases__`

The base classes for this class. These are also called superclasses.

`__doc__`

The documentation string. This is part of the response produced by the `help` function.

Here's an example of how the class docstring is used to produce the `help` results for a class.

```
import random
```

```
print random.Random.__doc__
help(random.Random)
```

Numeric Type Special Methods

When creating a new numeric data type, you must provide definitions for the essential mathematical and logical operators. When we write an expression using the usual +, -, *, and /, Python transforms this to method function calls. Consider the following:

```
v1= MyClass(10,20)
v2= MyClass(20,40)
x = v1 + v2
```

In this case, Python will evaluate line 3 as if you had written:

```
x = v1.__add__( v2 )
```

Every arithmetic operator is transformed into a method function call. By defining the numeric special methods, your class willwork with the built-in arithmetic operators. There are, however, some subtleties to this.

Forward, Reverse and In-Place Method Functions. First, there are as many as three variant methods required to implement each operation. For example, * is implemented by `__mul__`, `__rmul__` and `__imul__`. There are forward and reverse special methods so that you can assure that your operator is properly commutative. There is an in-place special method so that you can implement augmented assignment efficiently (see the section called “Augmented Assignment”).

You don't need to implement all three versions. If you implement just the forward version, and your program does nothing too odd or unusual, everything will work out well. The reverse name is used for special situations that involve objects of multiple classes.

Python makes two attempts to locate an appropriate method function for an operator. First, it tries a class based on the left-hand operand using the "forward" name. If no suitable special method is found, it tries the the right-hand operand, using the "reverse" name.

Consider the following:

```
v1= MyClass(10,20)
x = v1 * 14
y = 28 * v1
```

Both lines 2 and 3 require conversions between the built-in integer type and `MyClass`. For line 2, the forward name is used. The expression `v1*14` is evaluated as if it was

```
x = v1.__mul__( 14 )
```

For line 3, the reverse name is used. The expression `28*v1` is evaluated as if it was

```
y = v1.__rmul__( 28 )
```

The Operator Algorithm. The algorithm for determing what happens with ***x* op *y*** is approximately as follows. Historically, as Python has evolved, so have the ins and outs of argument coercion. In the future, beginning with Python 3.0, the older notion of type coercion and the `coerce` function will be dropped altogether, so we'll focus on the enduring features that will be preserved. Section 3.4.8 of the *Python Language Reference* covers this in more detail; along with the caveat that the rules have gotten too complex.

Note than a special method function can return the value `NotImplemented`. This indicates that the operation can't work directly only the values, and another operation

should be chosen. The rules provide for a number of alternative operations, this allows a class to be designed in a way that will cooperate successfully with potential future subclasses.

1. The expression ***string % anything*** is a special case and is handled first. This assures us that the value of *anything* is left untouched by any other rules. Generally, it is a tuple or a dictionary, and should be left as such.

2. If this is an augmented assignment statement (known as an in-place operator, e.g., ***a+=b***) where the left operand implements `__iop__`, then the `__iop__` special method is invoked without any coercion. These in-place operators permit you to do an efficient udpate the left operand object instead of creating a new object.

3. As a special case, if an operator's left operand is an object of a superclass of the right operand's class, the right operand's `__rop__` (subclass) method is tried first. If this is not implemented or returns `NotImplemented`, then the the left operand's `__op__` (superclass) method is used. This is done so that a subclass can completely override binary operators, even for built-in types.

4. Generally, `x.__op__(`*y*`)` is tried first. If this is not implemented or returns `NotImplemented`, `y.__rop__(`*x*`)` is tried second. In the case of

The following functions are the "forward" operations, used to implement the associated expressions.

method function	original expression
`__add__(`*self*, *other*`)`	***self* + *other***
`__sub__(`*self*, *other*`)`	***self* - *other***
`__mul__(`*self*, *other*`)`	***self* * *other***
`__div__(`*self*, *other*`)`	***self* / *other***
`__mod__(`*self*, *other*`)`	***self* % *other***
`__divmod__(`*self*, *other*`)`	`divmod (`*self*, *other* `)`
`__pow__(`*self*, *other*, [*modulo*] `)`	***self* ** *other*** or `pow (`*self*, *other*, [*modulo*] `)`
`__lshift__(`*self*, *other*`)`	***self* << *other***
`__rshift__(`*self*, *other*`)`	***self* >> *other***
`__and__(`*self*, *other*`)`	***self* and *other***
`__xor__(`*self*, *other*`)`	***self* xor *other***
`__or__(`*self*, *other*`)`	***self* or *other***

The method functions in this group are used to resolve operators using by attempting them using a reversed sense.

method function	original expression
`__radd__(`*self*, *other*`)`	***other* + *self***
`__rsub__(`*self*, *other*`)`	***other* - *self***
`__rmul__(`*self*, *other*`)`	***other* * *self***
`__rdiv__(`*self*, *other*`)`	***other* / *self***
`__rmod__(`*self*, *other*`)`	***other* % *self***
`__rdivmod__(`*self*, *other*`)`	`divmod (`*other*, *self* `)`
`__rpow__(`*self*, *other*`)`	***other* ** *self*** or `pow (`*other*, *self* `)`
`__rlshift__(`*self*, *other*`)`	***other* << *self***
`__rrshift__(`*self*, *other*`)`	***other* >> *self***
`__rand__(`*self*, *other*`)`	***other* and *self***
`__rxor__(`*self*, *other*`)`	***other* xor *self***
`__ror__(`*self*, *other*`)`	***other* or *self***

The method functions in the following group implement the basic unary operators.

method function	original expression
`__neg__(self)`	`- self`
`__pos__(self)`	`+ self`
`__abs__(self)`	`abs (self)`
`__invert__(self)`	`~ self`
`__complex__(self)`	`complex (self)`
`__int__(self)`	`int (self)`
`__long__(self)`	`long (self)`
`__float__(self)`	`float (self)`
`__oct__(self)`	`oct (self)`
`__hex__(self)`	`hex (self)`

Rational Number Example. Consider a small example of a number-like class. The the section called "Rational Numbers" exercise in Chapter 21, *Classes* describes the basic structure of a class to handle rational math, where every number is represented as a fraction. We'll add some of the special methods required to make this a proper numeric type. We'll finish this in the exercises.

```
class Rational( object ):
    def __init__( self, num, denom= 1L ):
        self.n= long(num)
        self.d= long(denom)
    def __add__( self, other ):
        return Rational( self.n*other.d + other.n*self.d,
        self.d*other.d )
    def __str__( self ):
        return "%d/%d" % ( self.n, self.d )
```

This class has enough methods defined to allow us to add fractions as follows:

```
>>> x = Rational( 3, 4 )
>>> y = Rational( 1, 3 )
>>> print x+y
7/12
>>>
```

In order to complete this class, we would need to provide most of the rest of the basic special method names (there is no need to provide a definition for `__del__`). We would also complete the numeric special method names.

Additionally, we would have to provide correct algorithms that reduced fractions, plus an additional conversion to respond with a mixed number instead of an improper fraction. We'll revisit this in the exercises.

Conversions From Other Types. For your class to be used successfully, your new numeric type should work in conjunction with existing Python types. You will need to use the `isinstance` function to examine the arguments and make appropriate conversions.

Consider the following expressions:

```
x = Rational( 22, 7 )
y = x+3
z = x+0.5
```

Variables `y` and `z` should be created as `Rational` fractions. However, our initial `__add__` function assumed that the *`other`* object is a `Rational` object. Generally, numeric classes must be implemented with tests for various other data types and appropriate conversions.

We have to use the `isinstance` function to perform checks like the following: `isinstance( other, int )`. This allows us to detect the various Python built-in types.

> **Function Reference vs. Function Call**
>
> In this case, we are using a reference to the `int` function; we are not evaluating the `int` function. If we incorrectly said `isinstance( other, int() )`, we would be attempting to evaluate the `int` function without providing an argument; this is clearly illegal.

If the result of `isinstance(`*`other`*`,` *`factory`*`)` is true in any of the following cases, some type of simple conversion should be done, if possible.

- **isinstance(other, complex).** You may want to raise an exception here, since it's hard to see how to make rational fractions and complex numbers conformable. If this is a common situation in your application, you might need to write an even more sophisticated class that implements complex numbers as a kind of rational fraction. Another choice is to write a version of the `abs` function of the complex number, which creates a proper rational fraction for the complex magnitude of the given value.

- **isinstance(other, float).** One choice is to truncate the value of `other` to `long`, using the built-in `long` function and treat it as a whole number, the other choice is to determine a fraction that approximates the floating point value.

- **isinstance(other, (int,long)).** Any of these means that the `other` value is clearly the numerator of a fraction, with a denominator of 1.

- **isinstance(other, str) or isinstance(other, unicode) or isinstance(other, basestring).** Any of these might convert the `other` value to a `long` using the built-in `long` function. If the conversion fails, an exception will be thrown, which will make the error obvious. The `basestring` type, by the way, is the superclass for ASCII strings (`str`) and Unicode strings (`unicode`).

- **isinstance(other, Rational).** This indicates that the `other` value is an instance of our `Rational` class; we can do the processing as expected, knowing that the object has all the methods and attributes we need.

Here is a version of `__sub__` with an example of type checking. If the *`other`* argument is an instance of the class `Rational`, we can perform the subtract operation. Otherwise, we attempt to convert the *`other`* argument to an instance of `Rational` and attempt the subtraction between two `Rationals`.

```
def __sub__( self, other ):
    if isinstance(other,Rational):
        return Rational( self.n*other.d - other.n*self.d,
        self.d*other.d )
    else:
        return self - Rational(long(other))
```

An alternative to the last line of code is the following.

```
        return Rational( self.n-long(other)*self.d, self.d )
```

While this second version performs somewhat quicker, it expresses the basic rational addition algorithm twice, once in the **if:** suite and again in the **else:** suite. A principle of object oriented programming is to maximize reuse and minimize restating an algorithm. My preference is to state the algorithm exactly once and reuse it as much as possible.

Reverse Operators. In many cases, Python will reverse the two operands, and use a

function like __rsub__ or __rdiv__. For example:

```
def __rsub__( self, other ):
    if isinstance(other,Rational):
       return Rational( other.n*self.d - self.n*other.d,
       self.d*other.d )
    else:
        return Rational(long(other)) - self
```

You can explore this behavior with short test programs like the following:

```
x = Rational( 3,4 )
print x-5
print 5-x
```

Container Special Methods

When designing a class that behaves like a container, it helps to provide definitions for the essential operators that are common to sequences, sets and mappings. Depending on whether your collection is more like a set (with no index), a list (indexed by simple integers) or a mapping (indexed by other objects), there are a number of additional methods you should provide.

Additionally, the container classes have numerous method names which aren't special. For example, sequence types also provide method with ordinary names like `append`, `count`, `index`, `insert`, `pop`, `remove`, `reverse` and `sort`. To provide a complete implemetation, you'll use special method names and ordinary method names.

Generally, we don't create our own containers from scratch. Most often we subclass an existing container, adding our new methods to the containers methods. We would use something like `class MyList( list ):` to extend the built-in list class. We looked at this approach in the section called "`Sample` Class with Statistical Methods".

One might want to create a new container class that has a sophisticated binary-tree implementation. This class behaves generally like a sequence, except that data is stored in order, and searches can be done much more rapidly than a simple sequence. It could be thought of as an implementation that extends the built-in sequence type.

By providing the following special methods, your class behaves like the built-in sequence, set or dictionary classes.

method function	original expression
__len__(*self*) → integer	`len`. Note that if an object lacks a __nonzero__ method and the __len__ method returns zero, the object is considered to be `False` in a Boolean context.
__getitem__(*self*, *key*) → object	***self*[*key*]**. For sequence types, the accepted keys should be integers or `slice` objects. The special interpretation of negative indexes (if the class wishes to emulate a sequence type) must be supported by the __getitem__ method. If *key* is of an inappropriate type, `TypeError` may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. The **for** statement requires an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.
__setitem__(*self*, *key*, *value*)	***self*[*key*] = *expression***. See `getitem`. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the `getitem` method.

__delitem__(*self*, *key*)	***self*[*key*]**. See getitem. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the getitem method.
__contains__(*self*, *item*)	***item* in *self*** operator. Return True if *item* is in *self*.
__missing__(*self*, *key*)	This is invoked by __getitem__ if the *key* does not exist. It can return an appropriate default value.

The __getitem__, __setitem__, __delitem__ method functions should be prepared for the *key* to be either a simple integer, a slice object, or a tuple of slice objects. A slice is a simple object with three attributes: *start*, *stop* and *step*. The following examples show common slice situations.

- The expression someSequence[1:5] is transformed to someSequence.__getitem__(slice(1,5)). The slice object is assigned to the *key* parameter of the __getitem__ function. This slice object has the following attribute values: key.start = 1, key.stop = 5, key.step = None.
- The expression someSequence[2:8:2] is transformed to someSequence.__getitem__(slice(2,8,2)). The slice object is assigned to the *key* parameter has the following attribute values: key.start = 2, key.stop = 8, key.step = 2.
- The expression someSequence[1:3,5:8] is transformed into someSequence.__getitem__((slice(1,3), slice(5,8))). With a tuple of slice objects.

Sequence types should also provide the methods append, count, index, insert, pop, remove, reverse and sort, with the same basic meanings as those for the built-in Python lists.

Additionally, sequence types should also implement concatenation (via the + operator) and repetition (via the * operator) by defining __add__, __radd__, __iadd__, __mul__, __rmul__, and __imul__. These were described in the section called "Numeric Type Special Methods" in a numeric context. The same methods, used in a container context, implement different behaviors.

Mappings should also provide the methods keys, values, items, has_key, get, clear, copy, and update with the same meanings as those for built-in Python mappings.

Iterator Special Method Names

An iterator is the object responsible for controlling iteration through a collection of objects or range of values. The **for** statement works by calling the next method of an iterator until the iterator raises an exception. The **yield** statement makes a function (or method) into an iterator by implicitly creating an object with a next method. We looked at this closely in Chapter 18, *Generators and the **yield** Statement*. The techniques there (principally, using the **yield** statement) are somewhat simpler than creating an explicit iterator object.

Generally, we provide a sequence object to the **for** statement. The sequence object responds to the **for** statement's request by creating the required iterator. Clearly, a statement like **for *var* in *object*** statement evaluates the *object*(.iter) method function to get the necessary iterator object.

The built-in sequence types (list, tuple, string) all produce iterator objects for use by the **for** statement. The set and frozenset classes also produces an iterator. In the case of a mapping, there are several choices for the target of the iteration: the iterator

could iterate over the keys, the values or the items (which are `(key,value)` pairs).

In addition to defining ordinary generator methods by using the **yield** statement, your classes can also produce iterator objects. This can make a program slightly simpler to read by assuring that loops are simple, obvious **for** statements.

Creating an Iterator. Generally, an iterator is an object that helps a program with with a more complex container. Consequently, the container will often contain a factory method which creates iterators. The special method `__iter__` usually handles this in a container. The for statement uses the `iter` built-in function. The `iter` function looks for the `__iter__` method.

An iterator object is created by a collection object when requested by the **for** statement. To make your collection play well with the **for** statement, implement the following method.

`__iter__`(*`self`*) → Iterator

> Returns an iterator in response to the `iter` function. The `iter` function is implicitly evaluated by a **for** statement.

An iterator controls the operation of the **for** statement, so it is clearly stateful. In addition to at least one internal variable, an iterator is usually created with a reference to the more complex object with which it works.

When we evaluate `iter( someList )`, we get an iterator object ready to be used with a **for** statement. The iter function uses `__iter__` method function of `someList`. The `__iter__` function creates the object to be used as an iterator.

A complex object will usually provide it's `self` variable to each iterator that it creates.

Methods of an Iterator. An iterator object has a simple interface definition: it provides a `next` method, which either returns the next value or raises the `StopIteration` exception. Further, an iterator needs an `__init__` method which will accept the complex object over which the iterator works, and allows the iterator to create appropriate initial values.

`__init__`(*`self`*, *`complexObject`*)

> This will initialize the iterator. Generally, a complex object will create an iterator from it's own `__iter__` method, providing it's `self` variable as the argument. It will do something like this: `return MyIterator( self )`.

`next`(*`self`*) → Object

> This will advance the iterator to the next value or element. If there is no next value, it will raise the `StopIteration` exception. If there is a next value, it will return this value.

There's little more than these two methods to an iterator. Often an iterator will also provide a definition of the __iter__ special method name. This will simply return the iterator. This prevents small problems with redundant calls to the `iter` built-in function.

Example: Non-Zero Iterator. In the following example classes, we'll create a class which wraps a `list` and provides and a specialized iterator that yields only non-zero values of the collection.

```
class DataSamples( object ):
    def __init__( self, aList=None ):
        self.values= aList or []
    def __iter__( self ):
        return NonZeroIter( self )
```

```
    def __len__( self ):
        return len( self.values )
    def __getitem__( self, index ):
        return self.values[index]
```

❶ When we initialize a `DataSamples` instance, we save any provided sequence of values. This class behaves like a collection. We haven't provided all of the methods, however, in order to keep the example short. Clearly, to be `list`-like, we'll need to provide an `append` method.

❷ When we evaluate the `iter` function for a `DataSamples` object, the `DataSamples` object will create a new, initialized `NonZeroIter`. Note that we provide the `DataSamples` object to the new `NonZeroIter`, this allows the iterator to process the collection properly.

```
class NonZeroIter( object ):
    def __init__( self, aDataSamples ):
        self.ds= aDataSamples
        self.pos= -1
    def next( self ):
        while self.pos+1 != len(self.ds) and self.ds[self.pos+1] == 0:
            self.pos += 1
        if self.pos+1 == len( self.ds ):
            raise StopIteration
        self.pos += 1
        return self.ds[self.pos]
    def __iter__( self ):
        return self
```

❶ When initialized, the `NonZeroIter` saves the collection that it works with. It also sets it's current state; in this instance, we have `pos` set to `-1`, just prior to the element we'll return.

❷ The `next` function of the iterator locates the next non-zero value. If there is no next value or no next non-zero value, it raises `StopIteration` to notify the **for** statement. Otherwise, it returns the next non-zero value. It updates its state to reflect the value just returned.

❸ The `__iter__` function of the iterator typically returns `self`.

We can make use of this iterator as follows.

```
ds = DataSamples( [0,1,2,0,3,0] )
for value in ds:
    print value
```

The **for** statement calls `iter(ds)` implicitly, which calls `ds.__iter__()`, which creates the `NonZeroIter` instance. The **for** statement then calls the `next` method of this iterator object to get the non-zero values from the `DataSamples` object. When the iterator finally raises the `StopIteration` exception, the **for** statement finishes normally.

Attribute Handling Special Method Names

When we reference an attribute of an object with something like `someObject.name`, Python uses several special methods to get the `someAttr` attribute of the object. Since we can say things like `myObject.anAttr= 5`, and `print myObject.anAttr`, there are actually three different implicit methods related to attribute access.

When the attribute reference occurs on the left side of an assignment statement, we are setting the attribute. When the attribute occurs almost anywhere else were are getting the value of the attribute. The final operation is deleting an attribute with the **del** statement.

While most attributes are simply instance variables, we have to make a firm distinction between an *attribute* and an *instance variable*.

- An attribute is a name, qualified by an object. It is a syntactic construction.

Generally, the attribute name is treated as a key to access the object's internal collection of instance variables, `__dict__`. However, we can change the behavior of an attribute reference.

- An instance variable is stored in the `__dict__` of an object. Generally, we access instance variables using attribute syntax. The attribute name is simply the key of the instance variable in the object's `__dict__`.

When we say `someObj.name`, the default behavior is effectively `someObj.__dict__['name']`.

There are several ways that we can tap into Python's internal mechanisms for getting and setting attribute values.

- The most accessible technique is to use the `property` function to define get, set and delete methods associated with an attribute name. The `property` function builds descriptors for you. We'll look at this in the section called "Properties".

- A slightly less accessible, but more extensible and reusable technique is to define *descriptor classes* yourself. This allows you considerable flexibility. You do this by creating a class which defines get, set and delete methods, and you associate this descriptor class with an attribute name. We'll look at this in the section called "Descriptors".

- You can tap into Python's low-level special methods for attribute access. There are three methods which plug into the standard algorithm. The fourth method, `__getattribute__`, allows you to change attribute access in a fundamental way.

> **Warning**
>
> Changing attribute access can interfere with how people understand the operation of your classes and objects. The default assumption is that an attribute is an instance variable. While we can fundamentally alter the meaning of a Python attribute, we need to be cautious about violating the default assumptions of people reading our software.

Attribute Access Special Methods. Fundamentally, attribute access works through a few special method names. Python has a default approach: it checks the object for an instance variable that has the attribute's name before using these attribute handling methods. Because Python uses these methods when an attribute isn't an instance variable, you can easily create infinite recursion. This can happen if you try to get an instance variable using a simple `self.someAttr` in the `__getattr__` method, or set the value of an instance variable with a simple `self.someAttr` in the `__setattr__` method. Within `__getattr__` and `__setattr__`, you have to use the internal `__dict__` explicitly.

These are the low-level attribute access methods.

`__getattr__`(*`self`*, *`name`*) → value

Returns a value for an attibute when the name is not an instance attribute nor is it found in any of the parent classes. *`name`* is the attribute name. This method returns the attribute value or raises an `AttributeError` exception.

`__setattr__`(*`self`*, *`name`*, *`value`*)

Assigns a value to an attribute. *`name`* is the attribute name, *`value`* is the value to assign to it. Note that if you naively do self.name= value in this method, you will have an infinite recursion of `__setattr__` calls. If you want to access the internal

dictionary of attributes, `__dict__`, you have to use the following: *`self`*`.__dict__[`*`name`*`] = `*`value`*.

`__delattr__`(*`self`*, *`name`*)

Delete the named attribute from the object. *`name`* is the attribute name.

`__getattribute__`(*`self`*, *`name`*) → value

Low-level access to a named attribute. If you provide this, it replaces the default approach of searching for an attribute and then using `__getattr__` if the named attribute isn't an instance variable of the class. To provide the default approach, this method must explicitly evaluate the superclass `__getattribute__` method with `super(`*`Class`*`,self).__getattribute__(` *`name`* `)`. This only works for classes which are derived from `object`.

Extending Built-In Classes

We can extend all of Python's built-in classes. This allows us to add or modify features of the data types that come with Python. This may save us from having to build a program from scratch. We'll look at an extended example of creating a specialized dictionary.

A common database technique is to create an index to a set of objects. The index contains key fields; each key is associated with a list of objects that share that key.

Let's say we have a `StockBlock` class, which includes the ticker symbol. We may have many small blocks of the same stock in a simple sequence. Our index mapping has a key of the ticker symbol; the value is a sequence of `StockBlock` objects that share the ticker symbol.

```
class StockBlock( object ):
    def __init__( self, ticker, price, shares ):
        ...
```

We'd like to do something like the following.

```
index = Index()
for block in portfolio:
    index[block.ticker].append( block )
```

As written, this won't work. What happens when we evaluate index['CTG'] before 'CTG' is a key in the dictionary?

Here's our Index class definition; it extends the built-in `dict` class. We use the `super` function to refer to the original dict implementation to which we adding features. In this case, we only want to extend the `__getitem__` method to provide a handy default value.

```
class Index( dict ):
    def __getitem__( self, key ):
        if not self.has_key( key ):
            super(Index,self).__setitem__( key, [] )
        return super(Index,self).__getitem__( key )
```

Since our subclass is based on `dict`, it does everyting the built-in class does.

This is similar to the `defaultdict` class in the `collections` module. This can also be accomplished by defining the `__missing__` special method of a dictionary subclass.

Special Method Name Exercises

Geometric Points

A 2-dimensional point is a coordinate pair, an x and y value. If we limit the range to the range 0 to 2**16, we can do a few extra operations quickly.

Develop the basic routines for `__init__`, `__repr__`, `__str__`. The `__cmp__` function should compute the point's distance from the origin, $r = \sqrt{x^2 + y^2}$. when comparing two points, or a point and a real number. The `__hash__` function can simply combine x and y via x<<16+y.

Develop a test routine that creates a sequence of points and then sorts the points. Also, be sure to develop a test that uses points as keys in a dictionary.

Rational Numbers

Finish the Rational number class by adding all of the required special methods. The the section called "Rational Numbers" exercise in Chapter 21, *Classes* describes the basic structure of a class to handle rational math, where every number is represented as a fraction.

Currency and the Cash Drawer

Currency comes in denominations. For instance, US currency comes in $100, $50, $20, $10, $5, $1, $.50, $.25, $.10, $.05, and $.01 denominations. Parker Brothers Monopoly™ game has currency in 500, 100, 50, 20, 10, 5 and 1. Prior to 1971, English currency had £50, £20, £10, £5, £1, shillings (1/12 of a pound) and pence (1/20 of a shilling). An amount of money can be represented as an appropriate tuple of integers, each of which represents the specific numbers of each denomination. For instance, one representation for $12.98 in US currency is (0, 0, 0, 1, 0, 2, 0, 3, 2, 0, 3).

Each subclass of `Currency` has a specific mix of denominations. We might define subclasses for US currency, Monopoly currency or old English currency. These classes would differ in the list of currencies.

An object of class currency would be created with a specific mix of denominatioons. The superclass should include operations to add and subtract `Currency` objects. An `__iadd__`(*`currency`*) method, for example would add the denominations in *`currency`* to this object's various denominations. An `__isub__`(*`currency`*) method, for example would subtract the denominations in *`currency`* to this object's various denominations; in the event of attempting to subtract more than is available, the object would raise an exception.

Be sure to define the various conversions to float, int and long so that the total value of the collection of bills and coins can be reported easily.

An interesting problem is to translate a decimal amount into appropriate currency. Note that numbers like 0.10 don't have a precise floating-point representation; floating point numbers are based on powers of 2, and 0.10 can only be approximated by a finite-precision binary fraction. For US currency, it's best to work in pennies, representing $1.00 as `100`.

Develop a method which will translate a given integer into an appropriate mixture of currency denominations. In this case, we can iterate through the denominations from largest to smallest, determining the largest number of that denomination ≤ the target amount. This version doesn't depend on the current value of the `Currency` object.

A more advanced version is to create a `Currency` object with a given value; this would represent the money in a cash drawer, for example. A method of this object would make an amount of money from only the available currency, or raise an exception if it could not be done. In this case, we iterate through the denominations from largest to smallest, determining the largest number ≤ the target amount, consistent with available money in the cash drawer. If we don't have enough of a given denomination, it means that we will

be using more of the smaller denominations.

One basic test case is to create a currency object with a large amount of money available for making change.

In the following example, we create a cash drawer with $804.55. We accept a payment of $10 as 1 $5, 4 $1, 3 $.25, 2 $.10 and 1 $.05. Then we accept a payment of $20, for a bill of $15.24, meaning we need to pay out $4.76 in change.

```
drawer = USCurrency( (5,2,6,5,5,5,5,5,5,5,5) )
drawer += USCurrency((0,0,0,0,1,4,0,3,2,1,0))
drawer += USCurrency((0,0,1,0,0,0,0,0,0,0,0))
drawer.payMoney( 4.76 )
```

Interestingly, if you have $186.91 (one of each bill and coin) you can find it almost impossible to make change. Confronted with impossible situations, this class should raise an `UnableToMakeChange` exception.

Each subclass of `Currency` has a specific mix of denominations. We might define subclasses for US currency, Monopoly currency or old English currency. These classes would differ in the list of currencies.

An object of class currency would be created with a specific mix of denominatioons. The superclass should include operations to add and subtract `Currency` objects. An `__iadd__`(*`currency`*) method, for example would add the denominations in *`currency`* to this object's various denominations. An `__isub__`(*`currency`*) method, for example would subtract the denominations in *`currency`* to this object's various denominations; in the event of attempting to subtract more than is available, the object would raise an exception.

Be sure to define the various conversions to float, int and long so that the total value of the collection of bills and coins can be reported easily.

An interesting problem is to translate a decimal amount into appropriate currency. Note that numbers like 0.10 don't have a precise floating-point representation; floating point numbers are based on powers of 2, and 0.10 can only be approximated by a finite-precision binary fraction. For US currency, it's best to work in pennies, representing $1.00 as `100`.

Develop a method which will translate a given integer into an appropriate mixture of currency denominations. In this case, we can iterate through the denominations from largest to smallest, determining the largest number of that denomination ≤ the target amount. This version doesn't depend on the current value of the `Currency` object.

A more advanced version is to create a `Currency` object with a given value; this would represent the money in a cash drawer, for example. A method of this object would make an amount of money from only the available currency, or raise an exception if it could not be done. In this case, we iterate through the denominations from largest to smallest, determining the largest number ≤ the target amount, consistent with available money in the cash drawer. If we don't have enough of a given denomination, it means that we will be using more of the smaller denominations.

One basic test case is to create a currency object with a large amount of money available for making change.

In the following example, we create a cash drawer with $804.55. We accept a payment of $10 as 1 $5, 4 $1, 3 $.25, 2 $.10 and 1 $.05. Then we accept a payment of $20, for a bill of $15.24, meaning we need to pay out $4.76 in change.

```
drawer = USCurrency( (5,2,6,5,5,5,5,5,5,5,5) )
drawer += USCurrency((0,0,0,0,1,4,0,3,2,1,0))
```

```
drawer += USCurrency((0,0,1,0,0,0,0,0,0,0,0))
drawer.payMoney( 4.76 )
```

Interestingly, if you have $186.91 (one of each bill and coin) you can find it almost impossible to make change. Confronted with impossible situations, this class should raise an `UnableToMakeChange` exception.

Sequences with Statistical Methods

Create a sequence class, `StatSeq` that can hold a sequence of data values. This class should define all of the usual sequence operators, including `__add__`, `__radd__`, `__iadd__`, but not `__mul__`. The `__init__` function should accept a sequence to initialize the collection. The various `__add__` functions should append the values from a `StatSeq` can instance as well as from ordinary sequences like `list` and `tuple`.

Most importantly, this class shold define the usual statistical functions like mean and standard deviation, described in the exercises after Chapter 13, *Tuples*, the section called "Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`" and the section called "`Sample` Class with Statistical Methods" in Chapter 21, *Classes*.

Since this class can be used everywhere a sequence is used, interface should match that of built-in sequences, but extra features are now readily available. For a test, something like the following should be used:

```
import random

samples = StatSeq( [ random.randrange(6) for i in range(100) ] )
print samples.mean()

s2= StatSeq()
for i in range(100):
    ss.append( random.randrange(6) )
    # Also allow s2 += [ random.randrange(6) ]
print s2.mean()
```

There are two approaches to this, both of which have pros and cons.

- Define a subclass of `list` with a few additional methods. This will be defined as `class StatSeq( list ):`.
- Define a new class (a subclass of `object`) that contains an internal list, and provides all of the sequence special methods. Some (like append) will be delegated to the internal list object. Others (like mean) will be performed by the StatSeq class itself. This will be defined as `class StatSeq( object ):`.

Note that the value of `mean` does not have to be computed when it is requested. It is possible to simply track the changing sum of the sequence and length of the sequence during changes to the values of the sequence. The sum and length are both set to zero by `__init__`. The sum and length are incremented by every `__add__`, `append`, `insert`, `pop`, `remove`, `__setitem__` and `__delitem__`. This way the calculation of mean is simply a division operation.

Keeping track of sums and counts can also optimize mode and standard deviation. A similar optimization for median is particularly interesting, as it requires that the sample data is retained by this class in sorted order. This means that each insert must preserve the sorted data set so that the median value can be retrieved without first sorting the entire sequence of data. You can use the `bisect` module to do this.

There are a number of algorithms for maintaining the data set in sorted order. You can refer to Knuth's *The Art of Computer Programming*[Knuth73], and Cormen, Leiserson, Rivest *Introduction to Algorithms*[Cormen90] which cover this topic completely.

Chessboard Locations

A chessboard can be thought of as a mapping from location names to pieces. There are two common indexing schemes from chessboards: algebraic and descriptive. In algebraic notation the locations have a rank-file address of a number and a letter. In descriptive notation the file is given by the starting piece's file, rank and player's color.

See Chapter 42, *Chess Game Notation* for an extension to this exercise.

The algebraic description of the chess board has *files* from a-h going from white's left to right. It has *ranks* from 1-8 going from white's side (1) to black's side (8). Board's are almost always shown with position a1 in the lower left-hand corner and h8 in the upper right, white starts at the bottom of the picture and black starts at the top.

In addition to the simplified algebraic notation, there is also a descriptive notation, which reflects each player's unique point of view. The descriptive board has a queen's side (white's left, files a-d) and a king's side (white's right, files e-h). Each rank is numbered starting from the player. White has ranks 1-8 going from white to black. Black, at the same time as ranks 1-8 going back toward white. Each of the 64 spaces on the board has two names, one from white's point of view and one from black's.

Translation from descriptive to algebraic is straight-forward. Given the player's color and a descriptive location, it can be translated to an algebraic location. The files translate through a relatively simple lookup to transform QR to a, QKt to b, QB to c, Q to d, K to e, KB to f, KKt to g, KR to h. The ranks translate through a simple calculation: white's ranks are already in algebraic notation; for black's rank of *r*, 9-*r* is the algebraic location.

Create a class to represent a chess board. You'll need to support the special function names to make this a kind of mapping. The `__getitem__` function will locate the contents of a space on the board. The `__setitem__` function will place a piece at a space on the board. If the *`key`* to either function is algebraic (2 characters, lower case file from a-h and digit rank from 1-8), locate the position on the board. If the *`key`* is not algebraic, it should be translated to algebraic.

The codes for pieces include the piece name and color. Piece names are traditionally "p" or nothing for pawns, "R" for rook, "N" for knight, "B" for bishop, "Q" for queen and "K" for king. Pawns would be simply the color code "w" or "b". Other pieces would have two-character names: "Rb" for a black rook, "Qw" for the white queen.

The `__init__` function should set the board in the standard starting position:

piece	algebraic	descriptive	piece code
white rooks	a1 and h1	wQR1, wKR1	Rw
white knights	b1 and g1	wQKt1, wKKt1	Nw
white bishop	c1 and f1	wQB1, wKB1	Bw
white queen	d1	wQ1	Qw
white king	e1	wK1	Kw
white pawns	a2-h2	wQR2-wKR2	w
black rooks	a8 and h8	bQR1, bKR1	Rb
black knights	b8 and g8	bQKt1, bKKt1	Nb
black bishops	c8 and f8	bQB1, bKB1	Bb
black queen	d8	bQ1	Qb
black king	e8	bK1	Kb
black pawns	a7-a7	bQR2-bKR2	b

Here's a sample five-turn game. It includes a full description of each move, and includes the abbreviated chess game notation.

1. white pawn from e2 to e4; K2 to K5

   black pawn from e7 to e5; K2 to K5

2. white knight from g1 to f3; KKt1 to KB3

   black pawn from d7 to d6; Q2 to Q3

3. white pawn from d2 to d4; Q2 to Q4

   black bishop from c8 to g4; QB1 to KKt5

4. white pawn at d4 takes pawn at e5; Q4 to K5

   black bishop at g4 takes knight at f3; KKt5 to KB6

5. white Q at d1 takes bishop at f3; Q1 to KB3

   black pawn at d6 takes e5; Q3 to K4

The main program should be able to place and remove pieces with something like the following:

```
chess= Board()
# move pawn from white King 2 to King 5
chess['wK5']= chess['wK2']; chess['wK2']= ''
# move pawn from black King 2 to King 5
chess['bK5']= chess['bK2']; chess['bK2']= ''

# algebraic notation to print the board
for rank in [ '8', '7', '6', '5', '4', '3', '2', '1']:
    for file in [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']:
        print "%5s" % board[file+rank],
    print
```

The algebraic output can be changed to the following, which some people find simpler.

```
for rank in ('8','7','6','5','4','3','2','1'):
    print "".join(
    [ "%5s" % board[file+rank]
        for file in ('a','b','c','d','e','f','g','h') ] )
```

You should also write a `move` function to simplify creating the test game. A move typically consists of the piece name, the from position, the to position, plus optional notes regarding check and pawn promotions.

Relative Positions on a Chess Board

When decoding a log of a chess game in Short Algebraic Notation (SAN), it is often necessary to search for a piece that made a given move. We'll look at this problem in detail in Chapter 42, *Chess Game Notation*. There are actually a number of search algorithms, each constrained by the rules for moving a particular piece. For example, the knight makes a short “L”-shaped move and there are only 8 positions on the board from which a knight can start to end up at a given spot. The queen, on the other hand, moves horizontally, vertically or diagonally any distance, and there are as many as 24 starting positions for the queen to end up at a given spot.

This search is simplified by having iterators that know a few rules of chess and an give us a sequence of appropriate rank and file values. We'd like to be able to say something like the following.

```
piece, move, toPos = ( "Q", "x", "f3" )
```

```
for fromPos in aBoard.queenIter( toPos ):
    if aBoard[fromPos] == 'Q':
        print "Queen from", fromPos, \
        "takes", aBoard[toPos], "at", toPos
```

We'll review a few chess definitions for this problem. You can also see the section called “Chessboard Locations” in the section called “Container Special Methods” for some additional background.

The algebraic description of the chess board has *files* from a-h going from white's left to right. It has *ranks* from 1-8 going from white's side (1) to black's side (8). Board's are almost always shown with position a1 in the lower left-hand corner and h8 in the upper right, white starts at the bottom of the picture and black starts at the top.

We need the following collection of special-purpose iterators.

- The `kingIter` method has to enumerate the eight positions that surround the king.
- The `queenIter` method has to enumerate all the positions in the same rank, the same file, and on the diagonals. Each of these must be examined from the queen's position moving toward the edge of the board. This search from the queen outward allows us to locate blocking pieces that would prevent the queen from making a particular move.
- The `bishopIter` method has to enumerate all the positions on the diagonals. Each of these must be examined from the bishop's position moving toward the edge of the board.
- The `knightIter` method has to enumerate the eight positions that surround the knight, reflecting the knight's peculiar move of 2 spaces on one axis and 1 space on the other axis. There are four combinations of two ranks and one file and four more combinations of two files and one rank from the ending position. As with the king, no piece can block a knight's move, so order doesn't matter.
- The `rookIter` method has to enumerate all the positions in the same rank and the same file. Each of these must be examined from the rook's position moving toward the edge of the board.
- The `pawnIter` method has to enumerate a fairly complex set of positions. Most pawn moves are limited to going forward one rank in the same file. Since we need to know which direction is forward, we need to know the color of the pawn. For white pawns, forward means the ranks increase from 2 to 8. For black pawns, then, forward means the ranks decrease from 7 down to 1. Pawn captures involve going forward one rank in an adjacent file. Further complicating the analysis is the ability for a pawn's first move to be two ranks instead of one.

We note that the queen's iterator is really a combination of the bishop and the rook. We'll look at the rook's iterator, because it is can be adapted to be a bishop iterator, and then those two combined to create the queen iterator.

Given a starting position with a rank of `r` and a file of `f`, we'll need to examine all ranks starting from `r` and moving toward the edge of the board. These are `r`−1, `r`+1, `r`−2, `r`+2, `r`−3, `r`+3, …. Similarly, we need to examine all of the files starting from `f` and moving toward the edge of the board. These are `f`−1, `f`+1, `f`−2, `f`+2, `f`−3, `f`+3, ….

Before doing an comparison, we need to filter the file and rank combinations to assure that they are legal positions. Additionally, we need to stop looking when we've encountered a piece of our own color or an opposing piece that isn't the one we're searching for. These intervnening pieces "block" the intentended move.

Chapter 25. Properties and Descriptors

Table of Contents

Python provides some advanced control over an object's attributes. In the section called "Attribute Handling Special Method Names" we explicitly separated attribute and instance variable. The default rule is that an attribute name gives us access to an instance variable.

- An attribute is a name that appears after an object name. For example, `someObj.name`.
- An instance variable is a name in the `__dict__` of an object.

When we say `someObj.name`, the default behavior is effectively `someObj.__dict__['name']`. However, we can alter this by making the name into a property or associating the name with a descriptor object.

There are several ways that we can tap into Python's internal mechanisms for getting and setting attribute values.

- The most accessible technique is to use the `property` function to define get, set and delete methods associated with an attribute name. The `property` function builds descriptors for you. We'll look at this in the section called "Properties".
- A slightly less accessible, but more extensible and reusable technique is to define *descriptor classes* yourself. This allows you considerable flexibility. You do this by creating a class which defines get, set and delete methods, and you associate your descriptor class with an attribute name. We'll look at this in the section called "Descriptors".
- You can tap into Python's low-level special methods for attribute access. This is covered in the section called "Attribute Handling Special Method Names".

Semantics of Attributes

Fundamentally, an object encapsulates data and processing via it's instance variables and method functions. Because of this encapsulation, we can think of a class definition as providing a interface definition and an implementation that fits the defined interface. The method names and public instance variables which begin with _ are treated as part of the private implementation of the class; the remaming elements form the public interface.

In Python, this distinction between interface and implementation is not heavily emphasized in the syntax, since it can often lead to wordy, complex programs. Most will-designed classes, however, tend to have a set of interface methods that form the interface for collaborating with objects of that class.

There are several commonly-used design patterns for an object's interface.

- **Getters and Setters.** We can encapsulate each instance variable with method functions that get and set the value of that instance variable. To be sure that the instace variables aren't accessed except via the method functions, we make each instance variable private, using _ as the first character of the variable name. When we do this, each access to an attribute of the object is via an explicit function: `anObject.setPrice( someValue ); anObject.getValue()`.
- **Properties.** We can bind getter, setter (and deleter) functions with an attribute

name, using the built-in `property` function. When we do this, each reference to an attribute looks like simple, direct access, but invokes the appropriate function of the object. For example, `anObject.price= someValue; anObject.value`.

- **Descriptors.** We can bind getter, setter (and deleter) functions into a separate class. We then assign an object of this class to the attribute name. When we do this, each reference to an attribute looks like simple, direct access, but invokes an appropriate function of the Descriptor object. For example, `anObject.price= someValue; anObject.value`.

The **Getter and Setter** design pattern is does not have a Pythonic look. This design is common in C++; it is required in Java. Python programmers find the use of getter and setter functions to be wordy and prefer to access attributes directly.

Descriptors

A Descriptor is a class which provides detailed get, set and delete control over an attribute of another object. This allows you to define attributes which are fairly complex objects in their own right. The idea is that we can use simple attribute references in a program, but those simple references are actually method functions of a descriptor object.

This allows us to create programs that look like the following example.

```
>>> oven= Temperature()
>>> oven.farenheit= 450
>>> oven.celsius
232.22222222222223
>>> oven.celsius= 175
>>> oven.farenheit
347.0
```

In this example, we set one attribute and the value of another attribute changes to mirror it precisely.

A common use for descriptors is in an object-oriented database (or an object-relational mapping). In a database context, getting an attribute value may require fetching data objects from the file system; which may involve creating and executing a query in a database.

Descriptor Design Pattern. The Descriptor design pattern has two parts: the **Owner** and the attribute **Descriptor**. The Owner is usually a relatively complex object that uses one or more Descriptors for it's attributes. Each Descriptor class defines get, set and delete methods for a specific attribute of the Owner.

Note that Desciptors can easily be written as reusable, generic types of attributes. The Owner can have multiple instances of each Descriptor. Each use of a Desciptor class is a unique instance of a Descriptor object, bound to an attribute name when the Owner class is defined.

To be recognized as a Descriptor, a class must implement some combination of the following three methods.

`__get__(`*`self`*`, `*`instance`*`, `*`owner`*`)`

> The *`instance`* argument is the `self` variable of the owning class; the *`owner`* argument is the owning class object. This method of the descriptor must return this attribute's value. If this descriptor implements a class level variable, the instance parameter can be ignored.

`__set__(`*`self`*`, `*`instance`*`, `*`value`*`)`

The *instance* argument is the self variable of the owning class. This method of the descriptor must set this attribute's value.

`__delete__`(*self*, *instance*)

The *instance* argument is the self variable of the owning class. This method of the descriptor must delete this attribute's value.

Sometimes, a descriptor class will also need an `__init__` method function to initialize the descriptor's internal state. Less commonly, the descriptor may also need `__str__` or `__repr__` method functions to display the instance variable correctly.

You must also make a design decision when defining a descriptor. You must determine where the underlying instance variable is contained. You have two choices.

- The Descriptor object has the instance variable.
- The Owner object contains the instance variable. In this case, the descriptor class must use the `instance` parameter to reference values in the owning object.

Descriptor Example. Here's a simple example of an object with two attributes defined by descriptors. One descriptor (`Celsius`) contains it's own value. The other desriptor (`Farenheit`), depends on the `Celsius` value, showing how attributes can be "linked" so that a change to one directly changes the other.

Example 25.1. descriptor.py

```
class Celsius( object ):
    def __init__( self, value=0.0 ):
        self.value= float(value)
    def __get__( self, instance, owner ):
        return self.value
    def __set__( self, instance, value ):
        self.value= float(value)

class Farenheit( object ):
    def __get__( self, instance, owner ):
        return instance.celsius * 9 / 5 + 32
    def __set__( self, instance, value ):
        instance.celsius= (float(value)-32) * 5 / 9

class Temperature( object ):
    celsius= Celsius()
    farenheit= Farenheit()
```

❶ We've defined a `Celsius` descriptor. The Celsius descriptor has an `__init__` method which defines the attribute's value. The Celsius descriptor implements the `__get__` method to return the current value of the attribute, and a `__set__` method to change the value of this attribute.

❷ The `Farenheit` descriptor implements a number of conversions based on the value of the celsius attribute. The `__get__` method converts the internal value from Celsius to Farenheit. The `__set__` method converts the supplied value (in Farenheit) to Celsius.

❸ The owner class, `Temperature` has two attributes, both of which are managed by descriptors. One attribute, `celsius`, uses an instance of the `Celsius` descriptor. The other attribute, `farenheit`, uses an instance of the `Fareheit` descriptor. When we use one of these attributes in an assignment statement, the descriptor's `__set__` method is used. When we use one of these attributes in an expression, the descriptor's `__get__` method is used. We didn't show a `__delete__` method; this would be used when the attribute is used in a **del** statement.

Let's look at what happens when we set an attribute value, for example, using `oven.farenheit= 450`. In this case, the `farenheit` attribute is a Descriptor with a

__set__ method. This __set__ method is evaluated with instance set to the object which is being modified (the oven variable) and owner set to the Temperature class. The __set__ method computes the celsius value, and provides that to the celsius attribute of the instance. The Celsius descriptor simply saves the value.

When we get an attribute value, for example, using oven.celsius, the following happens. Since celsius is a Descriptor with a __get__ method, this method is evaluated, and returns the celsius temperature.

Properties

The property function gives us a handy way to implement a simple descriptor without defining a separate class. Rather than create a complete class definition, we can write getter and setter method functions, and then bind these functions to an attribute name.

This allows us to create programs that look like the following example.

```
>>> oven= Temperature()
>>> oven.farenheit= 450
>>> oven.celsius
232.22222222222223
>>> oven.celsius= 175
>>> oven.farenheit
347.0
```

In this example, we set one attribute and the value of another attribute changes to mirror it precisely. We can do this by defining some method functions and binding them to attribute names.

Property Design Pattern. The Property design pattern has a number of method functions which are bound together with a single property name. The method functions can include any combination of a getter, a setter and a deleter.

To create a property, we define the instance variable and the method functions for some combination of getting, setting and deleting an attribute value. This is identical with the **Getter and Setter** design pattern. To make a property, we provide these method functions to the property function to bind the various methods to an attribute name.

Here's the definition of the property function.

property(*fget*, *fset*, *fdel*, *doc*) → property attribute

> Binds the given method functions into a property definition. This builds a descriptor object. Usually the result value is assigned to an attribute of a class.

Property Example. The following example shows a class definition with four method functions that are used to define two properties.

Example 25.2. property.py

```
class Temperature( object ):
    def fget( self ):
        return self.celsius * 9 / 5 + 32
    def fset( self, value ):
        self.celsius= (float(value)-32) * 5 / 9
    farenheit= property( fget, fset )
    def cset( self, value ):
        self.cTemp= float(value)
    def cget( self ):
        return self.cTemp
    celsius= property( cget, cset, doc="Celsius temperature" )
```

❶ We create the farenheit property from the fget and fset method functions.

When we use the `farenheit` attribute on the left side of an assignment statement, Python will use the setter method. When we use this attribute in an expression, Python will use the getter method. We don't show a deleter method; it would be used when the attribute is used in a **del** statement.

❷ We create the `celsius` property from the `cget` and `cset` method functions. When we use the `celsius` attribute on the left side of an assignment statement, Python will use the setter method. When we use this attribute in an expression, Python will use the getter method.

The doc string provided for the `celsius` attribute is available as `Temperature.celsius.__doc__`.

Attribute Access Exercises

1. **Rework Previous Exercises.** Refer to exercises for previous chapters (the section called "Class Definition Exercises", the section called "Advanced Class Definition Exercises", the section called "Design Pattern Exercises", the section called "Special Method Name Exercises"). Rework these exercises to manage attributes with getters and setters. Use the property function to bind a pair of getter and setter functions to an attribute name. The following examples show the "before" and "after" of this kind of transformation.

```
class SomeClass( object ):
    def __init__( self, someValue ):
        self.myValue= someValue
```

When we introduce the getter and setter method functions, we should also rename the original attribute to make it private. When we define the property, we can use the original attribute's name. In effect, this set of transformations leaves the class interface unchanged. We have added the ability to do additional processing around attribute get and set operations.

```
class SomeClass( object ):
    def __init__( self, someValue ):
        self._myValue= someValue
    def getMyValue( self ):
        return self._myValue
    def setMyvalue( self, someValue ):
        self._myValue= someValue
    myValue= property( getMyValue, setMyValue )
```

The class interface should not change when you replace an attibute with a property. The original unit tests should still work perfectly.

2. **Rework Previous Exercises.** Refer to exercises for previous chapters (the section called "Class Definition Exercises", the section called "Advanced Class Definition Exercises", the section called "Design Pattern Exercises", the section called "Special Method Name Exercises"). Rework these exercises to manage attributes with Descriptors. Define a Desciptor class with `__get__` and `__set__` methods for an attribute. Replace the attribute with an instance of the Descriptor.

When we introduce a descriptor, our class should look something like the following.

```
class ValueDescr( object ):
    def __set__( self, instance, value ):
        instance.value= value
    def __get__( self, instance, owner ):
        return instance.value

class SomeClass( object ):
    def __init__( self, someValue ):
```

```
        self.myValue= ValueDescr()
```

The class interface should not change when you replace an attibute with a descriptor. The original unit tests should still work perfectly.

3. **Tradeoffs and Design Decisions.** What is the advantage of Python's preference for referring to attributes directly instead of through getter and setter method functions?

   What is the advantage of having an attribute bound to a property or descriptor instead of an instance variable?

   What are the potential problems with the indirection created by properties or descriptors?

Chapter 26. Decorators

Table of Contents

In addition to object-oriented programming, Python also supports an approach called *Aspect-Oriented Programming*. Object-oriented programming focuses on structure and behavior of individual objects. Aspect-oriented programming refines object design techniques by defining aspects which are common across a number of classes or methods.

The focus of aspect-oriented programming is consistency. Toward this end Python allows us to define "decorators" which we can apply to class definitions and method definitions and create consistency.

We have to note that decorators can easily be overused. The issue is to strike a balance between the obvious programming in the class definition and the not-obvious programming in the decorator. Generally, decorators should be transparently simple and so obvious that they hardly bear explanation.

Semantics of Decorators

Essentially, a decorator is a function. The purpose of a decorator function is to transform one function definition (the argument function) into another function definition. When you apply a decorator to a function definition, Python creates the argument function, then applies the decorator function to the argument function. The object returned by the decorator is the net effect of a decorated definition. It should be a function or object that behaves like a function.

When we say

```
@theDecorator
def someFunction( anArg ):
    pass # some function body
```

We are doing the following:

1. We define an argument function, `someFunction`.

2. We modify the argument function with the decorator. Python will apply the decorator function, `theDecorator`, to the argument function. The decorator

returns a value; this should be some kind of callable object, either a class with a `__call__` method or a function.

3. Python binds the result of the decorator to the original function name, `someFunction`.

Generally, decorators fall into a number of common categories.

- **Simplifying Class Definitions.** In some cases, we want to create a method function which applies to the class-level attributes, not the instance variables. We described class-level variables in the section called "Class Variables". We introduced the built-in `@staticmethod` decorator in the section called "Static Methods and Class Method".

  Additionally, we may want to create a class function which applies to the class as a whole. To declare this kind of method function, the built-in `@classmethod` decorator can be used.

  If you look at the Python Wiki page for decorators (http://wiki.python.org/moin/PythonDecoratorLibrary), you can find several examples of decorators that help define properties for managing attributes.

- **Debugging.** There are several popular decorators to help with debugging. Decorators can be used to automatically log function arguments, function entrance and exit. The idea is that the decorator "wraps" your method function with additional statements to record details of the method function.

  One of the more interesting uses for decorators is to introduce some elements of type safety into Python. The Python Wiki page shows decorators which can provide some type checking for method functions where this is essential.

  Additionally, Python borrows the concept of *deprecation* from Java. A deprecated function is one that will be removed in a future version of the module, class or framework. We can define a decorator that uses the Python `warnings` module to create warning messages when the deprecated function is used.

- **Handling Database Transactions.** In some frameworks, like Django (http://www.djangoproject.org), decorators are used to simplify definition of database transactions. Rather than write explicit statements to begin and end a transaction, you can provide a decorator which wraps your method function with the necessary additional processing.

- **Authorization.** Web Security stands on several legs; two of those legs are authentication and authorization. Authentication is a serious problem involving transmission and validation of usernames and passwords or other credentials. It's beyond the scope of this book. Once we know who the user is, the next question is what are they authorized to do? Decorators are commonly used web frameworks to specify the authorization required for each function.

Built-in Decorators

Python has two built-in decorators.

`@staticmethod`

The `staticmethod` decorator modifies a method function so that it does not use the `self` variable. The method function will not have access to a specific instance of the class.

For an example of a static method, see the section called "Static Methods and Class Method".

`@classmethod`

> The `classmethod` decorator modifies a method function so that it receives the class object as the first parameter instead of an instance of the class. This method function wil have access to the class object itself.

The `@classmethod` decorator is used to create singleton classes. This is a Python technique for defining an object which is also a unique class. The class definition is also the one and only instance. This gives us a very handy, easy-to-read way to segregate attributes into a separate part of a class declaration. This is a technique used heavily by Python frameworks.

Generally, a function decorated with `@classmethod` is used for *introspection* of a class. An introspection method looks at the structure or features of the class, not the values of the specific instance.

Here's a contrived example of using introspection to display some features of a object's class.

Example 26.1. introspection.py

```
import types

class SelfDocumenting( object ):
    @classmethod
    def getMethods( aClass ):
        return [ (n,v.__doc__) for n,v in aClass.__dict__.items()
                 if type(v) == types.FunctionType ]
    def help( self ):
        """Part of the self-documenting framework"""
        print self.getMethods()

class SomeClass( SelfDocumenting ):
    attr= "Some class Value"
    def __init__( self ):
        """Create a new Instance"""
        self.instVar= "some instance value"
    def __str__( self ):
        """Display an instance"""
        return "%s %s" % ( self.attr, self.instVar )
```

❶ We import the `types` module to help us distinguish among the various elements of a class definition.

❷ We define a superclass that includes two methods. The classmethod, `getMethods`, introspects a class, looking for the method functions. The ordinary instance method, `help`, uses the introspection to print a list of functions defined by a class.

❸ We use the `@classmethod` decorator to modify the `getMethods` function. Making the `getMethods` into a class method means that the first argument will be the class object itself, not an instance.

❹ Every subclass of `SelfDocumenting` can print a list of method functions using a `help` method.

Here's an example of creating a class and calling the help method we defined. The result of the `getMethods` method function is a list of tuples with method function names and docstrings.

```
>>> ac= SomeClass()
>>> ac.help()
[('__str__', 'Display an instance'), ('__init__', 'Create a new Instance')]
```

Defining Decorators

A decorator is a function which accepts a function and returns a new function. Consequently, most decorators include a function definition and a **return** statement. A common alternative is to include a class definition as well as a **return** statement. If a class definition is used, it must define a callable object by including a definition for the `__call__` method.

There are two kinds of decorators, decorators without arguments and decorators with arguments. In the first case, the operation of the decorator is very simple. In the second case, the definition of the decorator is rather obscure. We'll look at the simple decorators first. We'll defer more complex decorators to the next section.

A simple decorator has the following outline:

```
def myDecorator( argumentFunction ):
   def resultFunction( *args, **keywords ):
       enhanced processing
       including a call to argumentFunction
   resultFunction.__doc__= argumentFunction.__doc__
   return resultFunction
```

In some cases, we may replace the result function definition with a result class definition to create a callable class.

Here's a simple decorator that we can use for debugging. This will log function entry, exit and exceptions.

Example 26.2. trace.py

```
def trace( aFunc ):
    """Trace entry, exit and exceptions."""
    def loggedFunc( *args, **kw ):
        print "enter", aFunc.__name__
        try:
            result= aFunc( *args, **kw )
        except Exception, e:
            print "exception", aFunc.__name__, e
            raise
        print "exit", aFunc.__name__
        return result
    loggedFunc.__name__= aFunc.__name__
    loggedFunc.__doc__= aFunc.__doc__
    return loggedFunc
```

❶ The result function, `loggedFunc`, is built when the decorator executes. This creates a fresh, new function for each use of the decorator.

❷ Within the result function, we evaluate the original function. Note that we simply pass the argument values from the evaluation of the result function to the original function.

❸ We move the original function's docstring and name to the result function. This assures us that the result function looks like the original function.

Here's a class which uses our `@trace` decorator.

Example 26.3. trace_user.py

```
class MyClass( object ):
    @trace
    def __init__( self, someValue ):
        """Create a MyClass instance."""
        self.value= someValue
    @trace
    def doSomething( self, anotherValue ):
        """Update a value."""
```

```
        self.value += anotherValue
```

Our class definition includes two traced function definitions. Here's an example of using this class with the traced functions. When we evaulate one of the traced methods it logs the entry and exit events for us. Additionally, our decorated function usees the original method function of the class to do the real work.

```
>>> mc= MyClass( 23 )
enter __init__
exit __init__

>>> mc.doSomething( 15 )
enter doSomething
exit doSomething

>>> mc.value
38
```

Defining Complex Decorators

A decorator transforms an argument function definition into a result function definition. In addition to a function, we can also provide argument values to a decorator. These more complex decorators involve a two-step dance that creates an intermediate function as well as the final result function.

The first step evaluates the abstract decorator to create a concrete decorator. The second step applies the concrete decorator to the argument function. This second step is what a simple decorator does.

Assume we have some qualified decorator, for example `@debug( flag )`, where flag can be `True` to enable debugging and `False` to disable debugging. Assume we provide the following function definition.

```
debugOption= True
class MyClass( object ):
    @debug( debugOption )
    def someMethod( self, args ):
        real work
```

Here's what happens when Python creates the definition of the `someMethod` function.

1. Define the argument function, `someMethod`.
2. Evaluate the abstract decorator `debug( debugOption )` to create a concrete decorator based on the argument value.
3. Apply the concrete decorator the the argument function, `someMethod`.
4. The result of the concrete decorator is the result function, which is given the name `someMethod`.

Here's an example of one of these more complex decorators. Note that these complex decorators work by creating and return a concrete decorators. Python then applies the concrete decorators to the argument function; this does the work of transforming the argument function to the result function.

Example 26.4. debug.py

```
def debug( theSetting ):
    def concreteDescriptor( aFunc ):
        if theSetting:
            def debugFunc( *args, **kw ):
                print "enter", aFunc.__name__
```

```
                return aFunc( *args, **kw )
            debugFunc.__name__= aFunc.__name__
            debugFunc.__doc__= aFunc.__doc__
            return debugFunc
        else:
            return aFunc
    return concreteDescriptor
```

❶ This is the concrete decorators, which is created from the argument, `theSetting`.

❷ If `theSetting` is `True`, the concrete decorator will create the result function named `debugFunc`, which prints a message and then uses the argument function.

❸ If `theSetting` is `False`, the concrete descriptor will simply return the argument function without any overhead.

Decorator Exercises

1. **Merge the `@trace` and `@debug` decorators.** Combine the features of the `@trace` decorator with the parameterization of the `@debug` decorator. This should create a better `@trace` decorator which can be enabled or disabled simply.

2. **Create a `@timing` decorator.** Similar to the parameterized `@debug` decorator, the `@timing` decorator can be turned on or off with a single parameter. This decorator prints a small timing summary.

Chapter 27. Managing Contexts: the with Statement

Table of Contents

Many objects manage resources, and must impose a rigid protocol on use of that resource. A file object in Python acquires and releases OS files, which may be associated with devices or network interfaces. A program may acquire and release database connections. In some cases, there may be a nested context of a database connection and one or more cursors.

In Python 2.6, the **with** statement interacts with certain types of context management objects to assure that the protocol for acquiring and releasing the object is followed irrespective of any exceptions that may be raised.

In Python 2.5, we must enable the **with** statement by using the following statement.

```
from __future__ import with_statement
```

In this section, we'll look at ways in which the new **with** statement will simplify file processing or database processing. We will look at the kinds of object design considerations which are required to create your own objects that work well with the **with** statement.

Semantics of a Context

While most use of the **with** statement involve acquiring and releasing specific resources, the statement can be applied somewhat more generally. To make the statement more widely applicable, Python works with a *context*. A context is not limited to acquiring and releasing a file or database connection. A context could be a web transaction, a user's logged-in session, a particular transaction or any other stateful condition.

Generally, a context is a state which must endure for one or more statements, has a specific method for entering the state and has a specific method for exiting the state. Further, a context's exit must be done with the defined method irrespective of any

exceptions that might occur within the context.

Database operations often center on transactions which must either be completed (to move the database to a new, iternally consistent state,) or rolled back to reset the database to a prior consistent state. In this case, exceptions must be tolerated so that the database server can be instructed to commit the transaction or roll it back.

We'll also use a context to be sure that a file is closed, or a lock is released. We can also use a context to be sure that the user interface is reset properly when a user switches their focus or an error occurs in a complex interaction.

The design pattern has two elements: a **Context Manager** and a **Working Object**. The Context Manager is used by the **with** statement to enter and exit the context. One thing that can happen when entering a context is that a Working Object is created as part of the entry process. The Working Object is often used for files and databases where we interact with the context. The Working Object isn't always necessary; for example acquiring and releasing locks is done entirely by the Context Manager.

Using a Context

There are a few Python library classes which provide context information that is used by the **with** statement. The most commonly-used class is the `file` class.

There are two forms of the **with** statement. In the first, the context object does not provide a context-specific object to work with. In the second, the context provides us an object to be used within the context.

`with` *`context`* : *`suite`*

`with` *`context`* as *`variable`* : *`suite`*

We'll look at the second form, since that is how the `file` class works. A `file` object is a kind of context manager, and responds to the protocol defined by the **with** statement.

When we open a file for processing, we are creating a context. When we leave that context, we want to be sure that the file is properly closed. Here's the standard example of how this is used.

```
with file('someData.txt','r') as theFile:
   for aLine in theFile:
       print aLine
# the file was closed by the context manager
```

❶ We create the file, which can be used as a context manager. The **with** statement enters the context, which returns a file object that we can use for input and output purposes. The **as** clause specifies that the working object is assigned to `theFile`.

❷ This is a pretty typical **for** statement that reads each line of a file.

❸ The **with** statement also exits the context, irrespective of the presence or absence of exceptions. In the case of a `file` context manager, this will close the file.

In the previous example, we saw that the file factory function is used to create a context manager. This is possible because a file has several interfaces: it is a context manager as well as being a working file object. This is potentially confusing because it conflate file context manager with the working file object. However, it also

Thas the advantage of making the **with** statement optional. In many applications, improperly closed files have few real consequences, and the carefully managed context of a with statement isn't necessary.

Defining a Context Manager Class

While the file example in the previous section shows an object which is both the Context Manager and the Working Object. We'll show the two as separate class definitions in order to clearly separate the two elements of the design pattern.

A Context Manager must implement two methods to collaborate properly with the **with** statement.

`__enter__`

> This method is called on entry to the **with** statement. The value returned by this method function will be the value assigned to the **as** variable.

`__exit__`(*`type`*, *`value`*, *`traceback`*)

> This method is called on exit from the **with** statement. If the *`type`*, *`value`* or *`traceback`* parameters have values other than `None`, then an exception occured. If the *`type`*, *`value`* or *`traceback`* parameters have values of `None`, then this is a normal conclusion of the **with** statement.

Example Context Manager. Let's assume we must produce a file which has a proper final line, irrespective of errors encountered during the file's production. For example, we could be producing a file with a secure hash like an MD5 digest of the contents, This secure digest can be used to detect processing errors or attempted tampering with the file.

We'll look at our working object first, then we'll look at a Context Manager for that object. This is the working object, `SecureLog`. This is a class which writes to a file, as well as accumulate a secure digest using the MD5 algorithm.

```
import hashlib
class SecureLog( object ):
    def __init__( self, someFile, marker="-----HASH-----" ):
        self.theFile= someFile
        self.marker= marker
        self.hash= hashlib.md5()
    def write( self, aLine ):
        self.theFile.write( aLine )
        self.hash.update(aLine)
    def finalize( self ):
        self.theFile.write( "%s\n%s\n" % ( self.marker, self.hash.hexdigest(), ) )
    def close( self ):
        pass
```

If this class is used correctly, the file will end with a marker line and the MD5 digest value. Other programs use the same MD5 algorithm when reading the file to confirm that the expected digest is the actual digest.

Here is a Context Manager, named `SecureLogManager`, which incorporates the `SecureLog` class. To be a context manager, this class implements the required `__enter__` and `__exit__` methods.

```
class SecureLogManager( object ):
    def __init__( self, someFile ):
        self.theFile= someFile
    def __enter__( self ):
        self.transLog= SecureLog( self.theFile )
        return self.transLog
    def __exit__( self, type, value, tb ):
        if type is not None:
            pass # Exception occurred
        self.transLog.finalize()
        self.transLog.close()
        self.theFile.close()
```

❶ The `__enter__` method creates the `SecureLog` and returns it so that the **as** clause will assign the log to a variable.

❷ The `__exit__` method checks to see if it is ending with an exception. In this case, we don't do any special processing for exceptions raised within the **with** statement. The `__exit__` method, however, uses the `SecureLog`'s `finalize` method to be absolutely sure that a proper digest is written to the log file before it is closed.

The overall main program can have the following structure. We don't need to make special arrangements in the main program to be sure that the log is finalized correctly. We have delegated those special arrangements to the context manager object, leaving us with an uncluttered main program.

```
from __future__ import with_statement

result= open( 'log.log', 'w' )
with SecureLogManager( result ) as log:
    log.write( "Some Configuration\n" )
    source= open('source.dat','rU')
    for line in source:
        log.write( line )
    source.close()
```

❶ Until Python 2.6 arrives, we must add the **with** statement to Python with the `from __future__ import` statement. This statement must be one of the first statements in the program.

❷ The **with** statement does several things. First, it creates an instance of `SecureLogManager`, which is our context manager. Then the `__enter__` method is evaluated, which returns an instance of `SecureLog` for use by the suite inside the **with** statement.

After this initialization, the suite of statements is executed. Once the suite of statements finishes, the `SecureLogManager`'s `__exit__` method is evaluated. This final step will finalize and close the log file.

Components, Modules and Packages

Delivering Components as Modules and Packages

The basic Python language is rich with features. These include several sophisticated built-in data types (Part II, "Data Structures"), numerous basic statments (Part I, "Language Basics"), a variety of common arithmetic operators and a library of built-in functions. In order to keep the basic Python kernel small, relatively feature features are built-in. A small kernel means that Python interpreters can be provided in a variety of software application, extending functionality of the application without bloating due to a large and complex command language.

The more powerful and sophisticated features of Python are separated into extension modules. There are several advantages to this. First, it allows each program to load only the relevant modules, speeding start-up. Second, it allows additional modules to be added easily. Third, it allows a module to be replaced, allowing you to choose among competing solutions to a problem.

The second point above, easily adding modules, is something that needs to be emphasized. In the Python community, this is called the *batteries included* principle. The ideal is to make Python directly applicable to just about any practical problem you may have.

Some modules have already been covered in other chapters. In the section called "The `math` Module" we covered `math` and `random` modules. In Chapter 12, *Strings* we covered the `string` module.

Overview of this part. This part will cover selected features of a few modules. The

objective is to introduce some of the power of key Python modules and show how the modules are used to support software development. This isn't a reference, or even a complete guide to these modules. The standard Python Library documentation and other books describe all available modules in detail. Remember that Python is an open-source project: in some cases, you'll have to read the module's source to see what it really does and how it works.

This part provides a general overview of how to create Python modules in Chapter 28, *Modules*. It also covers how to create packages in Chapter 30, *The Python Library*. An overview of the Python library is the focus of Chapter 30, *The Python Library*.

Module Details. We cover several essential modules in some detail.

- Chapter 31, *Complex Strings: the `re` Module* covers *regular expressions*, which you can use to do string matching and parsing.
- Chapter 32, *Dates and Times: the `time` and `datetime` Modules* covers how to handle the vagaries of our calendar with the `time` and `datetime` modules.
- We'll cover the basics of file handling in Chapter 33, *File Handling Modules*; this includes modules like: `sys`, `glob`, `fnmatch`, `fileinput`, `os`, `os.path`, `tempfile`, and `shutil`.
- We'll also look at modules for reading and writing files in various formats in Chapter 34, *File Formats: CSV, Tab, XML, Logs and Others*.
    - the section called “Comma-Separated Values: The `csv` Module” will cover Comma Separated Values (CSV) files.
    - Tab-delimited files, however, are a simpler problem, and don't require a separate module.
    - the section called “Property Files and Configuration (or `.INI`) Files: The `ConfigParser` Module” will cover parsing Configuration files, sometimes called `.INI` files. An `.INI` file is not the best way to handle configurations, but this technique is common enough that we need to show it.
    - the section called “Fixed Format Files, A COBOL Legacy: The `codecs` Module” will cover ways to handle COBOL files which are in a "fixed length" format, using EBCDIC data instead of the more common ASCII or Unicode.
    - the section called “XML Files: The `xml.minidom` and `xml.sax` Modules” will cover the techniques for parsing XML files.

Programs -- The Ultimate Modules. In a sense a top-level program is a module that does something useful. It's important understand "programs" as being reusable modules. Eventually most really useful programs get rewritten and merged into larger, more sophisticated programs.

In Chapter 35, *Programs: Standing Alone* this part covers modules essential for creating polished, complete stand-alone programs. This includes the `getopt` and `optparse` modules.

The final chapter covers integration among programs using the client-server programming model. This includes a number of modules that are essential for creating networked programs.

- We can use the HTTP protocol with a number of modules covered in the section called “Web Servers and the HTTP protocol”.

- We can use the XML-RPC standards to create or use web services. This is covered in the section called "Web Services: The `xmlrpclib` Module".
- Using the section called "Mid-Level Protocols: The `urllib2` Module", we can leverage a number of protocols to read a file located anywhere on the World-Wide Web via their Uniform Resource Locator (URL).
- If none of these protocols are suitable, we can invent our own, using the low-level socket module, covered in the section called "Socket Programming".

Table of Contents

Chapter 28. Modules

Table of Contents

A *module* allows us to group Python classes, functions and global variables. Modules are one level of composition of a large program from discrete components. The modules we design in one context can often be reused to solve other problems.

In the section called "Module Semantics" we describe the basic semantics of modules. In the section called "Module Definition" we describe how to define a module. We'll show how to use a module with the **import** statement in the section called "Module Use: The **import** Statement". A module must be found on the search path, we'll briefly talk about ways to control this in the section called "Finding Modules: The Path".

There are number of variations on the **import** statement; we'll look at these in the section called "Variations on An **import** Theme". We'll also look at the **exec** statement in the section called "The **exec** Statement". This chapter ends with some style notes in the section called "Style Notes".

Module Semantics

A *module* is a file that contains Python programming. A module can be brought into another program via the **import** statement, or it can be executed directly as the main script of an application program. There are two purposes for modules; some files may do both.

- A *library module* is expected to contain definitions of classes, functions and module variables. If it does anything beyond this, it is generally hard to understand and harder to use properly.
- A *script* (or application or "main" module) does the useful work of an application. It will have up to three disctinct elements: imports of any modules on which it depeds, main function definitions, a script that does the real work. It generally uses library modules.

Modules give us a larger-scale structure to our programs. We see the following levels of Python programming:

- The individual statement. A statement makes a specific state change by changing the value of a variable. State changes are what advance our program from its initial state to the desired ending state.

- Multiple statements are combined in a function. Functions are designed to be an indivisible, atomic unit of work. Functions can be easily combined with other functions, but never taken apart. Functions may be as simple as a single statement. While functions could be complex, it's important that they be easily understood, and this limits their complexity. A well-chosen function name helps to clarify the processing.

- Multiple functions and related data are used to define a class of objects. To be useful, a class must have a narrowly defined set of responsibilities. These responsibilities are characterized by the class attributes and behaviors.

- Multiple closely-related classes, functions and variables are combined into modules. The module name is the file name. A module should provide a closely related set of classes and functions. Sometimes, a module will package a set of closely related functions.

- Optionally, modules can be combined into packages. The directory structure defines the packages and their constituent modules. Additionally, packages contain some additional files that Python uses to locate all of the elements of the package.

- The user-oriented application program depend on modules (or packages).

The application — the functionality that the user perceives — is usually the top-most "executable" script that does the useful work. The relationship between shell commands (or desktop icons, or web links) that a user sees and the packaging of components that implement those commands can be murky. For example, a single application file may have multiple aliases, making it appear like independed commands. A single script can process multiple command-line options.

The application-level view, since it is presented to the user, must focused on usability: the shell commands or icons the user sees. The design of modules and packages sould be focused on maintenance and adaptability. The modules are the files that you, the developer, must use to keep the software understandable.

Components: Class and Module. A class is a container for attributes, method functions, and nested class definitions. Similarly, a module can also contain attributes, functions and class definitions.

A module is different from a class in several ways. First, a module is a physical file; it is the unit of software construction and configuration management. A class is defined within a file. Additionally, a class definition allows us to create a number of class instances, or objects. A module, on other hand, can only have a single instance. Python will only import a module one time; any additional requests to import a module have no effect. Any variables defined by the module, similarly, will only have a single instance.

Beyond this technical distinction, we generally understand modules to be the big, easy-to-understand components out of which are applications are built. A class is a finer-grained piece of functionality, which usually captures a small, very tightly bound collection of attribute and operations.

Module Definition

A module is a file; the name of the module is the file name. The `.py` extension on the

file name is required by the operating system, and gracefully ignored by Python. We can create a file named `roulette.py`, include numerous definitions of classes and functions related to the game of roulette.

Note that a module name must be a valid Python name. Operating system file names don't have the same restrictions on their names. The rules for variable names are in the section called “Variables”. A module's name is limited to letters, digits and _'s.

The first line of a module is usually a `#!` comment; this is typically `#!/usr/bin/env python`. The next few lines are a triple-quoted module doc string that defines the contents of the module file. As with other Python doc strings, the first line of the string is a summary of the module. This is followed by a more complete definition of the module's contents, purpose and usage.

For example, we can create the following module, called `dicedefs.py`

Example 28.1. dice.py

```
#!/usr/bin/env python
"""dice - basic definitions for Die and Dice.
Die - a single die
Dice - a collection of one or more dice
roll - a function to roll a pair of dice"""
from random import *

class Die( object ):
    """Simulate a 6-sided die."""
    def __init__( self ):
        self.value= None
    def roll( self ):
        self.value= randrange(6)+1
    def total( self ):
        return self.value

class Dice:
    """Simulate a pair of 6-sided dice."""
    def __init__( self ):
        self.value = ( Die(), Die() )
    def roll( self ):
        map( lambda d: d.roll(), self.value )
    def dice( self ):
        return tuple( [d.value for d in self.value] )

pair= Dice()

def roll():
    pair.roll()
    return pair.dice()
```

❶ A "main" script file must include the shell escape to run nicely in a Posix or Mac OS environment. Other files, even if they aren't main scripts, can include this to mark them as Python.

❷ Our docstring is a minimal summary. Well-written docstrings provide more information on the classes, variables and functions that are defined by the module.

❸ Many modules depends on other modules. Note that Python optimizes these imports; if some other module has already imported a given module, it is simply made available to our module. If the module has not been imported already, it is imported for use by our module.

❹ As is typical of many modules, this module provides some class definitions.

❺ This module defines a module-global variable. This variable is part of the module; it appears global to all classes and functions within this module. It is also available to every client of this module. Since this variable is part of the module, every client is sharing a single variable.

❻ This module defines a handy function, `roll`, which uses the module global variable.

Conspicuous by its absence is any main script. This module is a pure library module.

Module Use: The import Statement

Since a module is just a Python file, there are two ways to use a module. We can **import** the module, to make use of it's definitions, or we can execute it as a script file to have it do useful work. We started looking at execution of scripts back in the section called "Script Mode", and have been using it heavily.

We looked briefly at the **import** statement in the section called "Using Modules". There are several variations on this statement that we'll look at in the next section. In this section, we'll look at more features of the **import** statement.

The essential import statement has the following syntax:

`import` *`module`*

The module name is the Python file name with the "`.py`" file extension removed.

Python does the following.

1. Search the global namespace for the module. If the module exists, it had already been imported; for the basic **import**, nothing more needs to be done.

2. If the module doesn't exist, search the Python path for a file; the file name is the module name plus the "`.py`" extension. The search path has a default value, and can be modified by command-line arguments and by environment variables. If the module name can't be found anywhere on the path, an `ImportError` exception is raised.

3. If the file was found, create the module's new, unique namespace; this is the container in which the module's definitions and module-level variables will be created. Execute the statements in the module, using the module's namespace to store the variables and definitions. We'll look close at namespaces below.

The most important effect of importing a module is that the Python definitions from the module are now part of the running Python environment. Each class, function or variable defined in the module is available for use. Since these objects are contained in the module's namespace, The names of those elements must be qualified by the module's name.

In the following example, we import the `dice` module. Python will search for module `dice`, then for the file `dice.py` from which to create the module. After importing, create an instance of the `Dice` class and called that instance `craps`. We qualified the class name with the module name: `dice.Dice`.

```
>>> import dice
>>> craps= dice.Dice()
>>> craps.roll()
>>> craps.dice()
(3, 5)
```

Namespaces. Python maintains a number of local namespaces and one global namespace. A unique local namespace is used when evaluating each function or method function. In effect, a variable created in a function's namespace is private to that function; it only exists while the function executes.

Typically, when a module is imported, the module's namespace is the only thing created

in the global namespace. All of the module's objects are inside the module's namespace.

You can explore this by using the built-in `dir` function. Do the following sequence of steps.

1. Create a small module file (like the `dice.py` example, above).
2. Start a fresh command-line Python interpreter in the same directory as your module file. Starting the intepreter in the same directory is the simplest way to be sure that your module will be found by the **import** statement.
3. Evalute the `dir` function to see what is in the initial global namespace.
4. Import your module.
5. Evaluate the `dir` function to see what got added to the namespace.

Scripts and Modules. There are two ways to use a Python file. We can execute it as a script or we can import it as a library module. We need to keep this distinction clear when we create our Python applications. The file that a user executes will do useful work, and must be a script of some kind. This script can be an icon that the user double-clicked, or a command that the user typed at a command prompt; in either case, a single Python script initiates the processing.

A file that is imported will provide definitions. We'll emphasize this distinction.

> **Bad Behavior**
>
> The standard expectation is that a library module will contain only definitions. Some modules create module global variables; this must be fully documented. It is bad behavior for an imported module to attempt to do any real work beyond creating definitions. Any real work that a library module does makes reuse of that module nearly impossible.

Importing a module means the module file is executed. This creates is an inherent, but important ambiguity. A given file can be used as a script or used as a library; any file can be used either way. Here's the complete set of alternatives.

A top-level script.

> You execute a script with the Run Module menu item in IDLE. You can also execute a script from your operating system command prompt. For example, **python *`file`*.py** will execute the given file as a script. Also, you can set up most operating systems so that entering **file.py** at the command prompt will execute the file as a script. Also, you can set up most GUI's so that double-clicking the `file.py` icon will launch Python and execute the given file as a script.

import

> You can import a library module. As described above, Python will not import a module more than once. If the module was not previously imported, Python creates a namespace and executes the file. The namespace is saved.

exec

> Python's **exec** statement is similar to the **import** statement, with an important difference: The **exec** statement executes a file in the current namespace. The **exec** statement doesn't create a new namespace. We'll look at this in the section called

"The **exec** Statement".

The Main-Import Switch. Since a file can be used as script or library, we can intentionally create files that are both. We can create a script which can also be used as a library. And we can create a library which has a useful behavior when used as a script. This promotes reuse of libraries.

Python provides a global variable that helps to differentiate between a main program script and a module library module. The global `__name__` variable is equal to `"__main__"` when the initial (or "top-level" or "outermost") file is being processed. For example, when you have an executable script, and you run that script from the command line, that script sees `__name__` equal to `"__main__"`. However, when an import is in process, the `__name__` variable is the name of the module being imported.

As an example, we can make use of this to provide stand-alone unit testing for a library module. When we write a module that is primarily definitional, we can have it execute it's own unit tests when it is used as a main program. This makes testing a library module simple: we import it and it runs its unit test. We do this by examining the `__name__` variable.

```
if __name__ == "__main__":
    unittest.main()
```

When some other script imports a module (for example, named `dice.py`), the `__name__` variable is `"dice"` and nothing special is done. When testing, however, we can execute the module by itself; in this case the `__name__` variable is `"__main__"` and the test function is executed.

Finding Modules: The Path

For modules to be available for use, the Python interpreter must be able to locate the module file. Python has a set of directories in which it looks for module files. This set of directories is called the *search path*, and is analogous to the `PATH` environment variable used by an operating system to locate an executable file.

Python's search path is built from a number of sources:

- `PYTHONHOME` is used to define directories that are part of the Python installation. If this environment variable is not defined, then a standard directory structure is used. For Windows, the standard location is based on the directory into which Python is installed. For most Linux environments, Python is installed under `/usr/local`, and the libraries can be found there. For Mac OS, the home directory is under `/Library/Frameworks/Python.framework`.

- `PYTHONPATH` is used to add directories to the path. This environment variable is formatted like the OS `PATH` variable, with a series of filenames separated by `:`'s (or `;`'s for Windows).

- Script Directory. If you run a Python script, that script's directory is placed first on the search path so that locally-defined moules will be used instead of built-in modules of the same name.

- The `site` module's locations are also added. (This can be disabled by starting Python with the `-S` option.) The `site` module will use the `PYTHONHOME` location(s) to create up to four additional directories. Generally, the most interesting one is the `site-packages` directory. This directory is a handy place to put additional modules you've downloaded. Additionally, this directory can contain `.PTH` files. The `site` module reads `.PTH` files and puts the named directories onto the search path.

The search path is defined by the `path` variable in the `sys` module. If we `import sys`,

we can display `sys.path`. This is very handy for debugging. When debugging shell scripts, it can help to run '`python -c 'import sys; print sys.path'` just to see parts of the Python environment settings.

Installing a module, then, is a matter of assuring that the module appears on the search path. There are four central methods for doing this.

- Some packages will suggest you create a directory and place the package in that directory. This may be done by downloading and unzipping a file. It may be done by using Subversion and sychronizing your subversion copy with the copy on a server. Either way, you will likely only need to create an operating system link to this directory and place that link in `site-packages` directory.

- Some packages will suggest you download (or use subversion) to create a temporary copy. They will provide you with a script — typically based on `setup.py` — which moves files into the correct locations. This is called the `distutils` distribution. This will generally copy the module files to the `site-packages` directory.

- Some packages will rely on `setuptools`. This is a package from the Python Enterprise Application Kit that extends `distuils` to further automates download and installation. This tool, also, works by moving the working library modules to the `site-packages` directory.

- Extending the search path. Either set the `PYTHONPATH` environment variable, or put `.PTH` files in the site-packages directory.

> **Windows Environment**
>
> In the Windows environment, the `Python_Path` symbol in the Windows registry is used to locate modules.

Variations on An import Theme

There are several variations on the **import** statement. We looked at these briefly in the section called "The `math` Module". In this section, we'll cover the variations available on the **import** statement.

- Basic Import. This is covered in the section called "Module Use: The **import** Statement".

- Import As. This allows us to import a module, and assign it a new name.

- From Module Import Names. This allows us to import a module, making some names part of the global namespace.

- Combined From and As import.

Import As

A useful variation in the **import** statement is to rename a module using the **as** clause.

`import` *`module`* as *`name`*

This module renaming is used in two situations.

- We have two or more interchangable versions of a module.

- The module name is rather long and painful to type.

There are number of situations where we have interchangable versions of a module. One example is the built-in `os` module. This module gives us a number of functions that behave identically on most major operating systems. The way this is done is to create a number of variant implementations of these functions, and then use as appropriate **as** clause to give them a platform-neutral name.

Here's a summary of how the `os` module uses **import as**.

```
if 'posix' in _names:
    import posixpath as path
elif 'nt' in _names:
    import ntpath as path
elif 'mac' in _names:
    import macpath as path
```

After this **if**-statement, one of the various platform-specific modules will have been imported, and it will have the platform-independent name of `os.path`.

In the case of some modules, the name is rather long. For example, `sqlalchemy` is long and easy to misspell. It's somewhat simpler to use the following technique.

```
import sqlalchemy as sa
db= sa.create_engine('sqlite:///file.db')
```

This allows us to use `sa` as the module name.

From Module Import Names

Two other variations on the **import** statement introduce selected names from the module into the local namespace. One form picks specific names to make global.

`from` *`module`* import *`name`*,...

This version of **import** adds a step after the module is imported. It adds the given list of names into the local namespace, making them available without using the module name as a qualifier.

For example:

```
from math import sin, cos, tan
print dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos',
'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp',
'log', 'log10', 'modf', 'pi', 'pow', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']

print locals()
{'math': <module 'math' (built-in)>, '__doc__': None,
'__version__': '1.0',
'__file__': 'Macintosh HD:SWdev:Jack:Python:Mac:Tools:IDE:PythonIDE.py',
'__name__': '__main__',
'__builtins__': <module '__builtin__' (built-in)>,
'inspect': <function inspect at 0x0d084310>,
'sin': <built-in function sin>, 'cos': <built-in function cos>,
'tan': <built-in function tan>}
```

In this example, the `locals` value shows that the `sin`, `cos` and `tan` functions are now directly part of the namespace. We can use these functions without referring to the math module. We can evaluate `sin(0.7854)`, rather than having to say `math.sin(0.7854)`.

This is discouraged because it tends to conceal the origin of objects.

Another variation on **import** makes all names in the module part of the local namespace.

This import has the form:

from *`module`* import *

This makes all names from the module available in the local namespace.

Import and Rename

Finally, we can combine the **from** and **as** options to both import selected items and provide more understandable names for them.

We can say things like:

`from` *`module`* import *`name`* as *`name`*

In this case, we're both concealing the source of the item and it's original name. We'd best have a very good reason for this. Think of the confusion that can be caused by

```
from math import sqrt as sin
```

This must be used cautiously to prevent creating more problems than it appears to solve.

The exec Statement

The **import** statement, in effect, executes the module file. Typically, the files we import are defined as sequences of definitions. Since our main program often begins with a series of **import** statements, these modules are imported into the global namespace. Python also optimizes the modules brought in by the **import** statement so that they are only imported once.

The **exec** statement can execute a file, a string of Python code, as well as a `code` created by the `compile` function. Unlike the **import** statement, it doesn't optimize module definitions or create and save a new namespace.

`exec` *`expression`*

The functions `eval` and `execfile` do essentially the same thing.

> **Fundamental Assumptions**
>
> The **exec** statement, `eval` function and `execfile` functions are dangerous tools. These break one of the Fundamental Assumptions: the source you are reading is the source that is being executed. A program that uses the **exec** statement or `eval` function is incorprating other source statements into the program dynamically. This can be hard to follow, maintain or enhance.
>
> Generally, the **exec** statement is something that must be used with some care. The most common use is to bring in a set of configuration parameters written as simple Python assignment statements. For example, we might use a file like the following as the configuration paramaters for a program.
>
> ```
> db_server= "dbs_prod_01"
> db_port= "3306"
> db_name= "PROD"
> ```

Module Exercises

Refactor a Script

A very common situation is to take a script file apart and create a formal module of the definitions and a separate module of the script. If you refer back to your previous exercise scripts, you'll see that many of your files have definitions followed by a "main" script which demonstrates that your definitions actually work. When refactoring these, you'll need to separate the definitions from the test script.

Let's assume you have the following kind of script as the result of a previous exercise.

```
# Some Part 3 Exercise.
class X( object ):
    does something
class Y( X ):
    does something a little different

x1= X()
x1.someMethod()
y2= Y()
y2.someOtherMethod()
```

You'll need to create two files from this. The module will be the simplest to prepare, assume the file name is `myModule.py`

```
#!/usr/bin/env python
class X( object ):
    does something
class Y( X ):
    does something a little different
```

Your new new demonstation application will look like this because you will have to qualify the class and function names that are created by the module.

```
#!/usr/bin/env python
import myModule
x1= myModule.X()
x1.someMethod()
y2= myModule.Y()
y2.someOtherMethod()
```

Your original test script had an implicit assumption that the definitions and the test script were all in the same namespace. This will no longer be true. While you can finesse this by using `from myNewModule import *`, this is not the best programming style. It is better to rewrite the test script to explicitly qualify names with the module name.

There are a number of related class definitions in previous exercises that can be used to create modules.

- Any of the exercises in the section called “Class Definition Exercises” contains a number of related classes.
- The exercise in the section called “Shuffling Method for the `Deck` class” has two parts: definitions of `Deck` and related material, and a procedure for comparing different shuffling algorithms. This should be repackaged to separate the performance measurement script from the basic definitions. If you reworked this exercise in the section called “Shuffling Method for the `Deck` class” you will have `Deck` and the various shuffling strategies in a module separate from the performance measurement script.
- The simulation built in the section called “State” can be formalized into two modules. The lowest-level module defines the basic game of Roulette including the `Wheel` and `RouletteGame`. Another module imports this and defines the `Player` and the states. Finally, the main script imports the game, the player and runs the simulation to produce a log of wins and losses.
- The rational number class, built in the section called “Numeric Type Special

Methods" can be formalized into a module. A script can import this module and demonstrate the various operations on rational numbers.
- The sequence with additional statistical methods, built in the section called "`Sample` Class with Statistical Methods" can be formalized into a module. A script can import this module and demonstrate the various statistical operations on sample data.

Install a New Module

Create a simple module file with some definitions. Preferrably, this is a solution to the section called "Refactor a Script". Install the definitional part into the PYTHONPATH. Be sure to rename or remove the local version of this file. Be sure to use each installation method.

1. Move the module file to the site-packages directory. Be sure that it is removed (or renamed) in your local directory.

2. Move the module file to another directory and create a hard link (using the Linux ln or equivalent Windows utility) from site-packages to the other directory you created.

3. Remove the hard link and put a `.PTH` file in the site-packages directory.

4. Remove the `.PTH` file and update the PYTHONPATH environment variable to reference the new directory.

Planning for Maintenance and Upgrades

There are a number of module installation scenarios; each of these will require a different technique. Compare and contrast these techniques from several points of view: cost to deploy, security of the deployment, ease of debugging, and control over what the user experiences.

- You have to provide modules and an application on a number of desktop PC's. Python must be installed on each individual desktop. However, the application that uses Python could be put on a shared network drive. What are the pros and cons of installing a Python-based application locally versus on a network drive?
    - How would you handle the initial setup?
    - How would you handle an upgrade to Python itself? For example, how would you install Python 2.5 so as to preserve your modules and application?
    - How would you control an upgrade to a Python-based application? For example, you have a new module file that needs to be made available to all users.
- You have to provide modules and an application on a server, shared by a number of users. Python is installed on the server, as is the Python-based application. What security considerations should be put into place?
    - How would you handle initial installation of Python and your server-based application?
    - How would you handle an upgrade to Python on this shared server?
    - How would you control an upgrade to the Python-based application on this shared server?

Style Notes

Modules are a critical organizational tool for final delivery of Python programming. Python software is delivered as a set of module files. Often a large application will have one or more module files plus a main script that initiates the application. There are several conventions for naming and documenting module files.

Module names are python identifiers as well as file names. Consequently, they can only use "_" as a punctuation mark. Most modules have names that are mixedCase, beginning with lowercase letters. This conforms to the useage on most file systems. MacOS users would do well to keep their module names to a single word, and end with `.py`. This promotes portability to operating systems where file names are more typically single words. Windows, in particular, can have trouble with spaces in file names.

Some Python modules are a wrapper for a C-language module. In this case the C/C++ module is named in all lowercase and has a leading underscore (e.g. `_socket`).

A module's contents start with a docstring. After the docstring comes any version control information. The bulk of a module is typically a series of definitions for classes, exceptions and functions.

A module's docstring should begin with a one-line pithy summary of the module. This is usually followed by in inventory of the public classes, exceptions and functions this module creates. Detailed change history does not belong here, but in a separate block of comments or an additional docstring.

If you use CVS to track the versions of your source files, following style is recommended. This makes the version information available as a string, and visible in the `.py` source as well as any `.pyc` working files.

```
"""My Demo Module.

This module is just a demonstration of some common styles."""

__version__ = "$Revision: 1.3 "
```

Note that the module's name will qualift everything created in the module, it is never necessary to have a prefix in front of each name inside the module to show its origin. For example, consider a module that contains classes and functions related to statistical analysis, called `stats.py`. The `stats` module might contain a class for tracking individual samples.

Poor Names. We don't include extra name prefixes like `statsSample` or `stats_sample`.

Better Names. We would call our internal sample class `Sample`. A client application that contains an **import stats** statement, would refer to the class as `stats.Sample`.

This needless over-qualification of names sometimes devolves to sillines, with class names beginning with `c_`, function names beginning with `f_`, the expected data type indicated with a letter, and the scope (global variables, local variables and function parameters) all identified with various leading and trailing letters. This is not done in Python programming. Class names begin with uppercase letters, functions begin with lowercase. Global variables are identified explicitly in **global** statements. Most functions are kept short enough that that the parameter names are quite obvious.

Any element of a module with a name that begins with `_single_leading_underscore` is not created in the namespace of the client module. When we use **from stats import ***, these names that begin with _ are not inserted in the global namespace. While usable within the module, these names are not visible to client modules, making them the equivalent of Java's `private` declaration.

A common feature of modules is to create a module-wide exception class. The usuall approach looks like the following. Within a module, you would define an `Error` class as

follows:

```
class Error( Exception ): pass
```

You can then raise your module-specific exception with the following.

```
raise Error, "additional notes"
```

A client module or program can then reference the module's exception in a **try** statement as `module.Error`. For example:

```
import aModule

try:
    aModule.aFunction()
except: aModule.Error, ex:
    print "problem", ex
    raise
```

With this style, the origin of the error is shown clearly.

Chapter 29. Packages

Table of Contents

A *package* is a collection of Python modules. Packages allow us to structure a collection of modules. In the section called “Package Semantics” we describe the basic semantics of packages. In the section called “Package Definition” we describe how to define a package. We'll look at using a package in the section called “Package Use”.

Package Semantics

A *package* is a directory that contains modules. Having a directory of modules allows us to have modules contained within other modules. This allows us to use qualified module names, clarifying the organization of our software.

We can, for example, have several simulations of casino games. Rather than pile all of our various files into a single, flat directory, we might have the following kind of directory structure. (This isn't technically complete, it needs a few additional files.)

```
casino/
    craps/
        dice.py
        game.py
        player.py
    roulette/
        wheel.py
        game.py
        player.py
    blackjack/
        cards.py
        game.py
        player.py
    srategy/
        basic.py
        martingale.py
        bet1326.py
```

```
        cancellation.py
```

Given this directory structure, our overall simulation might include statements like the following.

```
import craps.game, craps.player
import strategy.basic as betting

class MyPlayer( craps.player.Player ):
    def __init__( self, stake, turns ):
        betting.initialize(self)
```

We imported the `game` and `player` modules from the `craps` package. We imported the `basic` module from the `strategy` package. We defined a new player based on a class named `Player` in the `craps.player` package.

We have a number of alternative betting strategies, all collected under the `strategy` package. When we import a particular betting strategy, we name the module `betting`. We can then change to a different betting strategy by changing the **import** statement.

There are two reasons for using a package of modules.

- There are a lot of modules, and the package structure clarifies the relationships among the modules. If we have several modules related to the game of craps, we might have the urge to create a craps_game.py module and a craps_player.py module. As soon as we start structuring the module names to show a relationship, we can use a package instead.

- There are alternative implementations, and the package contains polymorphic modules. In this case, we will often use an `import` *`package.alternative`* `as` *`interface`* kind of **import** statement. This is often used for interfaces and drivers to isolate the interface details and provide a uniform API to the rest of the Python application.

It is possible to go overboard in package structuring. The general rule is to keep the package structure relatively flat. Having only one module at the bottom of deeply-nested packages isn't really very informative or helpful.

Package Definition

In order for Python to make use of a directory as package, the directory must have a name that is a valid Python identifier and contain a special module named `__init__`. Valid Python names are composed of letters, digits and underscores. See the section called “Variables” for more informaiton.

The `__init__` module is often an empty file, `__init__.py` in the package directory. Nothing else is required to make a directory into a package. The `__init__.py` file, however, is essential. Without it, you'll get an `ImportError`.

For example, consider a number of modules related to the definition of cards. We might have the following files.

```
cards/
    __init__.py
    standard.py
    blackjack.py
    poker.py
```

The `cards.standard` module would provide the base definition of card as an object with suit and rank. The `cards.blackjack` module would provide the subclasses of cards that we looked at in the section called “Blackjack Hands”. The `cards.poker` module would provided the subclasses of cards that we looked at in the section called “Poker

Hands".

The cards.blackjack module and the cards.poker module should both import the cards.standard module to get the base definition for the `Card` and `Deck` classes.

The `__init__` module. The `__init__` module is the "initialization" module in a package. It is processed the first time that a package name is encountered in an **import** statement. In effect, it initializes Python's understanding of the package. Since it is always loaded, it is also effectively the default module in a package. There are a number of consequences to this.

- We can import the package, without naming a specific module. In this case we've imported just the initialization module, `__init__`. If this is part of our design, we'll put some kind of default or top-level definitions in the `__init__` module.

- We can import a specific module from the package. In this case, we also import the initialization module along with the requested module. In this case, the `__init__` module can provide additional definitions; or it can simply be empty.

In our cards example, above, we would do well to make the `__init__` module define the basic `Card` and `Deck` classes. If we import the package, `cards`, we get the default module, `__init__`, which gives us `cards.Card` and `cards.Deck`. If we import a specific module like `cards.blackjack`, we get the `__init__` module plus the named module within the package.

Package Use

If the `__init__` module in a package is empty, the package is little more than a collection of module files. In this case, we don't generally make direct use of a package. We merely mention it in an import statement: `import cards.poker`.

On other hand, if the `__init__` module has some definitions, we can import the package itself. Importing a package just imports the `__init__` module from the package directory. In this case, we mention the package in an import statement: `import cards`.

Even if the `__init__` module has some definitions in it, we can always import a specific module from within the package. Indeed, it is possible for the `__init__` module in a package is to do things like adjust the search path prior to locating individual module files.

Package Exercises

1. **Create a Package.** In the previous chapter's exercises (see the section called "Refactor a Script") are some suggestions for creating a simple module. The modules decribed in that exercise are so small that adding modules does not seem necessary. However, it's the best example of how packages arise when solving practical problems.

   Pick a module that you've already created. Add a second file with a few simple classes that do little real work. These are best described as "Hello World" classes, since they don't do anything more useful than provide a simple response to indicate that the module was imported correctly.

   Create a packge directory with the necessary __init__.py file.

   Create a demonstration script which imports and exercises both modules from this package.

Style Notes

Since a package, like a module, is both a file system location and a Python construct,

the name must be a valid Python name, using just letters, numbers and _'s. Additionally, some file systems are cavalier about maintaining the original case of the filename. The old Mac OS (pre Mac OS X), and many of the old Windows variants would casually alter the case of filenames.

Consequently, package and module names should be all lower case. This way, there is no ambiguity about the intended case of the module name.

Package structures should be relatively flat. The general rule is to keep the package structure relatively flat. Having only one module at the bottom of deeply-nested packages isn't really very informative or helpful. While it may seem like `import casino.games.definitions.tablegames.dicegames.craps` leaves lots of room for expansion, there just aren't enough casino games to make this immensly deep structure usable.

Packages are generally used for two things:

- Collecting related modules together in a directory to simplify installation, maintenenace and documentation.
- Defining alternative implementations as simply as possible.

If all of your modules are more-or-less unique, then a package structure isn't going to help. Similarly, if you don't have alternate implementations of some driver or interface, a package structure isn't useful.

Chapter 30. The Python Library

Table of Contents

Consistent with the Pythonic “Batteries Included” philopsophy, there are hundreds of extension modules. It can be difficult to match a programming need with a specific module. The *Python Library Reference* document can be hard to pick through to locate an appropriate module. We'll start at the top of the library organization and work our way down to a useful subset of the tremendous wealth that is Python.

In the section called “Overview of the Python Library” we'll take a very high level overview of what's in the Python library. We'll closely at the 50 or so most useful modules in the section called “Most Useful Library Sections”.

Overview of the Python Library

The *Python Library Reference* organizes modules into the following sections. The current version of the Library documentation strives to prsent the modules with the most useful near the front of the list. The first 23 chapters, plus chapter 26 are the most useful. From chapter 24 and below (except for chapter 26), the modules are too highly specialized to cover in this book.

1. Introduction
2. Built-in Objects. This chapter provides complete documentation of the built-in functions, exceptions and constants.
3. Built-in Types. All of the data types we've looked at are documented completely in this chapter of the library reference. Of course, there are additional types in the Python reference that we haven't looked at.

4. String Services. This chapter includes almost a dozen modules for various kinds of string and text handling. This includes regular expression pattern matching, Unicode codecs and other string-processing modules.

5. Data Types. This chapter has almost 20 modules providing additional data types, including `datetime`,

6. Numeric and Mathematical Modules. This chapter describes `math`, `decimal` and `random` modules.

7. Internet Data Handling. One secret behind the internet is the use of standardized sophisticated data objects, like email messages with attachments. This chapter covers over a dozen modules for handling data passed through the internet.

8. Structured Markup Processing Tools. XML, HTML and SGML are all markup languages. This chapter covers tools for parsing these languages to separate the content from the markup.

9. File Formats. This chapter covers modules for parsing files in format like Comma Separated Values (CSV).

10. Cryptographic Services. This chapter has modules which can be used to develop and compare secure message hashes.

11. File and Directory Access. This chapter of the Library Reference covers many of the modules we'll look at in Chapter 33, *File Handling Modules*.

12. Data Compression and Archiving. This chapter describes modules for reading and writing zip files, tar files and BZ2 files. We'll cover these modules in Chapter 33, *File Handling Modules*, also.

13. Data Persistence. Objects can be written to files, sockets or databases so that they can persist beyond the processing of one specific program. This chapter covers a number of packages for pickling objects so they are preserved. The SQLite 3 relational database is also described in this module.

14. Generic Operating System Services. An Operating System provides a number of services to our application programs, including access to devices and files, consistent notions of time, ways to handle command-line options, logging, and handling operating system errors. We'll look some of these modules in Chapter 35, *Programs: Standing Alone*.

15. Optional Operating System Services. This section includes operating system services that are common to most Linux variants, but not always available in Windows.

16. Unix Specific Services. There are a number of Unix and Linux-specific features provided by these modules.

17. Interprocess Communication and Networking. Larger and more complex application programs often consist of multiple, cooperating components. The World Wide Web, specifically, is based on the interaction between client and server programs. This chapter describes modules that provide a basis for communcating among the OS processes that execute our programs.

18. Internet Protocols and Support. This chapter describes over two dozen modules that process a wide variety of internet-related data structures. This varies from the relatively simple processing of URL's to the relatively complex processing of XML-based Remote Procedure Calls (XML-RPC).

19. Multimedia Services. Multimedia includes sounds and images; these modules can

be used to manipulate sound or image files.

20. Graphical User Interfaces with Tk. The Tkinter module is one way to build a graphical desktop application. The GTK libraries are also widely used to build richly interactive desktop applications; to make use of them, you'll need to download the pyGTK package.

21. Internationalization. These packages help you separating your message strings from the rest of your application program. You can then translate your messages and provide language-specific variants of your software.

22. Program Frameworks. These are modules to help build command-line applications.

23. Development Tools. These modules are essential to creating polished, high-quality software: they support the creation of usable documents and reliable tests for Python programs.

With the exception of chapter 26, the remaining chapters aren't of general interest to most programmers.

24. The Python Debugger

25. The Python Profilers

26. Python Runtime Services. This chapter describes the `sys` module, which provides a number of useful objects.

27. Custom Python Interpreters

28. Restricted Execution

29. Importing Modules

30. Python Language Services

31. Python compiler package

32. Abstract Syntax Trees

33. Miscellaneous Services

34. SGI IRIX Specific Services

35. SunOS Specific Services

36. MS Windows Specific Services

Most Useful Library Sections

This section will overview about 50 of the most useful libary modules. These modules are proven technology, widely used, heavily tested and constantly improved. The time spent learning these modules will reduce the time it takes you to build an application that does useful work.

We'll dig more deeply into just a few of these modules in subsequent chapters.

> **Lessons Learned**
>
> As a consultant, we've seen far too many programmers writing modules which overlap these. There are two causes: ignorance and

hubris. In this section, we hope to tackle the ignorance cause.

Python includes a large number of pre-built modules. The more you know about these, the less programming you have to do.

Hubris sometimes comes from the feeling that the library module doesn't fit our unique problem well-enough to justify studying the library module. In many cases you can't read the library module to see what it *really* does. In Python, the documentation is only an introduction; you're encouraged to actually read the library module.

We find that hubris is most closely associated with calendrical calcuations. It isn't clear why programmers invest so much time and effort writing buggy calendrical calculations. Python provides many modules for dealing with times, dates and the calendar.

4. String Services. The String Services modules contains string-related functions or classes. See Chapter 12, *Strings* for more information on strings.

re

The `re` module is the core of text pattern recognition and processing. A *regular expression* is a formula that specifies how to recognize and parse strings. The `re` module is described in detail in Chapter 31, *Complex Strings: the `re` Module*.

struct

The avowed purpose of the `struct` module is to allow a Python program to access C-language API's; it packs and unpacks C-language struct object. It turns out that this module can also help you deal with files in packed binary formats.

difflib

The `difflib` module contains the essential algorithms for comparing two sequences, usually sequences of lines of text. This has algorithms similar to those used by the Unix **diff** command (the Window **COMP** command).

StringIO, cStringIO

There are two variations on `StringIO` which provide file-like objects that read from or write to a string buffer. The `StringIO` module defines the class `StringIO`, from which subclasses can be derived. The `cStringIO` module provides a high-speed C-language implementation that can't be subclassed.

Note that these modules have atypical mixed-case names.

textwrap

This is a module to format plain text. While the word-wrapping task is sometimes handled by word processors, you may need this in other kinds of programs. Plain text files are still the most portable, standard way to provide a document.

codecs

This module has hundreds of text encodings. This includes the vast array of Windows code pages and the Macintosh code pages. The most commonly used are the various Unicode schemes (utf-16 and utf-8). However, there are also a number of codecs for translating between strings of text and arrays of bytes. These schemes include base-64, zip compression, bz2 compression, various quoting rules, and even the simple rot_13 substitution cipher.

5. Data Types. The Data Types modules implement a number of widely-used data structures. These aren't as useful as sequences, dictionaries or strings -- which are built-in to the language. These data types include dates, general collections, arrays, and schedule events. This module includes modules for searching lists, copying structures or producing a nicely formatted output for a complex structure.

datetime

> The `datetime` handles details of the calendar, including dates and times. Additionally, the `time` module provides some more basic functions for time and date processing. We'll cover both modules in detail in Chapter 32, *Dates and Times: the `time` and `datetime` Modules*.
>
> These modules mean that you never need to attempt your own calendrical calculations. One of the important lessons learned in the late 90's was that many programmers love to tackle calendrical calculations, but their efforts had to be tested and reworked prior to January 1, 2000, because of innumerable small problems.

calendar

> This module contains routines for displaying and working with the calendar. This can help you determine the day of the week on which a month starts and ends; it can count leap days in an interval of years, etc.

collections

> This package contains two data types, and is likely to grow with future releases of Python. One tye is the `deque` -- a "double-ended queue" -- that can be used as stack (LIFO) or queue (FIFO). The other class is a specialized dictionary, `defaultdict`, which can return a default value instead of raising an exception for missing keys.

bisect

> The `bisect` module contains the `bisect` function to search a sorted list for a specific value. It also contains the `insort` fucntion to insert an item into a list maintaining the sorted order. This module performs faster than simply appending values to a list and calling the `sort` method of a list. This module's source is instructive as a lesson in well-crafted algorithms.

array

> The `array` module gives you a high-performance, highly compact collection of values. It isn't as flexible as a list or a tuple, but it is fast and takes up relatively little memory. This is helpful for processing media like image or sound files.

sched

> The `sched` module contains the definition for the `scheduler` class that builds a simple task scheduler. When a scheduler is contructed, it is given two user-supplied functions: one returns the “time” and the other executes a “delay” waiting for the time to arrive. For real-time scheduling, the `time` module `time` and `sleep` functions can be used. The scheduler has a main loop that calls the supplied time function and compares the current time with the time for scheduled tasks; it then calls the supplied a delay function for the difference in time. It runs the scheduled task, and calls the delay function with a duration of zero to release any resources.
>
> Clearly, this simple algorithm is very versatile. By supplying custom time functions that work in minutes instead of seconds, and a delay function that does

additional background processing while waiting for the scheduled time, a flexible task manager can be constructed.

copy

The `copy` module contains functions for making copies of complex objects. This module contains a function to make a *shallow copy* of an object, where any objects contained within the parent are not copied, but references are inserted in the parent. It also contains a function to make a *deep copy* of an object, where all objects contained within the parent object are duplicated.

Note that Python's simple assignment only creates a variable which is a label (or reference) to an object, not a duplicate copy. This module is the easiest way to create an independent copy.

pprint

The `pprint` module contains some useful functions like `pprint.pprint` for printing easy-to-read representations of nested lists and dictionaries. It also has a `PrettyPrinter` class from which you can make subclasses to customize the way in which lists or dictionaries or other objects are printed.

6. Numeric and Mathematical Modules. These modules include more specialized mathemathical functions and some additional numeric data types.

decimal

The decimal module provides decimal-based arithmetic which correctly handles significant digits, rounding and other features common to currency amounts.

math

The `math` module was covered in the section called “The `math` Module”. It contains the math functions like sine, cosine and square root.

random

The `random` module was covered in the section called “The `math` Module”.

7. Internet Data Handling. The Internet Data Handling modules contain a number of handy algorithms. A great deal of data is defined by the Internet Request for Comments (RFCs). Since these effectively standardize data on the Internet, it helps to have modules already in place to process this standardized data. Most of these modules are specialized, but a few have much wider application.

mimify, base64, binascii, binhex, quopri, uu

These modules all provide various kinds of conversions, ecapes or quoting so that binary data can be manipulated as safe, universal ASCII text. The number of these modules reflects the number of different clever solutions to the problem of packing binary data into ordinary email messages.

8. Structured Markup Processing Tools. The following modules contain algorithms for working with structured markup: Standard General Markup Lanaguage (SGML), Hypertext Markup Language (HTML) and Extensible Markup Language (XML). These modules simplify the parsing and analysis of complex documents. In addition to these modules, you may also need to use the CSV module for processing files; that's in chapter 9, File Formats.

htmllib

Ordinary HTML documents can be examined with the `htmllib` module. This module based on the `sgmllib` module. The basic `HTMLParser` class definition is a superclass; you will typically override the various functions to do the appropriate processing for your application.

One problem with parsing HTML is that browsers — in order to conform with the applicable standards — must accept incorrect HTML. This means that many web sites publish HTML which is tolerated by browsers, but can't easily be parsed by `htmllib`. When confronted with serious horrows, consider downloading the Beautiful Soup module. This handles erroneous HTML more gracefully than `htmllib`.

xml.sax, xml.dom, xml.dom.minidom

The `xml.sax` and `xml.dom` modules provide the classes necessary to conveniently read and process XML documents. A SAX parser separates the various types of content and passes a series of events the handler objects attached to the parser. A DOM parser decomposes the document into the Document Object Model (DOM).

The `xml.dom` module contains the classes which define an XML document's structure. The `xml.dom.minidom` module contains a parser which creates a DOM object.

Additionally, there is a Miscellaneous Module (in chapter 33) that goes along with these.

formatter

The `formatter` module can be used in conjunction with the HTML and XML parsers. A formatter instance depends on a writer instance that produces the final (formatted) output. It can also be used on its own to format text in different ways.

9. File Formats. These are modules for reading and writing files in a few of the amazing variety of file formats that are in common use. In addition to these common formats, modules in chapter 8, Structured Markup Processig Tools are also important.

csv

The `csv` module helps you parse and create Comma-Separated Value (CSV) data files. This helps you exchange data with many desktop tools that produce or consume CSV files. We'll look at this in the section called “Comma-Separated Values: The `csv` Module”.

ConfigParser

Configuration files can take a number of forms. The simplest approach is to use a Python module as the configuration for a large, complex program. Sometimes configurations are encoded in XML. Many Windows legacy programs use `.INI` files. The ConfigParser can gracefully parse these files. We'll look at this in the section called “Property Files and Configuration (or `.INI`) Files: The `ConfigParser` Module”.

10. Cryptographic Services. These modules aren't specifically encryption modules. Many popular encryption algorithms are protected by patents. Often, encryption requires compiled modules for performance reasons. These modules compute secure digests of messages using a variety of algorithms.

hashlib, hmac, md5, sha

Compute a secure hash or digest of a message to ensure that it was not tampered with. MD5, for example, is often used for validating that a downloaded file was recieved correctly and completely.

11. File and Directory Access. We'll look at many of these modules in Chapter 33, *File Handling Modules*. These are the modules which are essential for handling data files.

os, os.path

The `os` and `os.path` modules are critical for creating portable Python programs. The popular operating systems (Linux, Windows and MacOS) each have different approaches to the common services provided by an operating system. A Python program can depend on `os` and `os.path` modules behaving consistently in all environments.

One of the most obvious differences among operating systems is the way that files are named. In particular, the *path separator* can be either the POSIX standard `/`, or the windows `\`. Additionally, the Mac OS Classic mode can also use `:`. Rather than make each program aware of the operating system rules for path construction, Python provides the `os.path` module to make all of the common filename manipulations completely consistent.

> Programmers are faced with a dilemma between writing a "simple" hack to strip paths or extensions from file names and using the `os.path` module. Some programmers argue that the `os.path` module is too much overhead for such a simple problem as removing the `.html` from a file name. Other programmers recognize that most hacks are a false economy: in the long run they do not save time, but rather lead to costly maintenance when the program is expanded or modified.

fileinput

The `fileinput` module helps your progam process a large number of files smoothly and simply.

glob, fnmatch

The `glob` and `fnmatch` modules help a Windows program handle wild-card file names in a manner consistent with other operating systems.

shutil

The `shutil` module provides shell-like utilities for file copy, file rename, directory moves, etc. This module lets you write short, effective Pytthon programs that do things that are typically done by shell scripts.

Why use Python instead of the shell? Python is far easier to read, far more efficient, and far more capable of writing moderately sophisticated programs. Using Python saves you from having to write long, painful shell scripts.

12. Data Compression and Archiving. These modules handle the various file compression algorithms that are available. We'll look at these modules in Chapter 33, *File Handling Modules*.

tarfile, zipfile

These two modules create archive files, which contain a number of files that are bound together. The TAR format is not compressed, where the ZIP format is compressed. Often a TAR archive is compressed using GZIP to create a .tar.gz archive.

zlib, gzip, bz2

These modules are different compression algorithms. They all have similar

features to compress or uncompress files.

13. Data Persistence. There are several issues related to making objects persistent. In Chapter 9 of the Python Reference, there are several modules that help deal with files in various kinds of formats. We'll talk about these modules in detail in Chapter 34, *File Formats: CSV, Tab, XML, Logs and Others*.

There are several additional techniques for managing persistence. We can "pickle" or "shelve" an object. In this case, we don't define our file format in detail, instead we leave it to Python to persist our objects.

We can map our objects to a relational database. In this case, we'll use the SQL language to define our storage, create and retrieve our objects.

pickle, shelve

> The `pickle` and `shelve` modules are used to create persistent objects; objects that persist beyond the one-time execution of a Python program. The `pickle` module produces a serial text representation of any object, however complex; this can reconstitute an object from its text representation. The `shelve` module uses a `dbm` database to store and retrieve objects. The `shelve` module is not a complete object-oriented database, as it lacks any transaction management capabilities.

sqlite3

> This module provides access to the SQLite relational database. This database provides a significant subset of SQL language features, allowing us to build a relational database that's compatible with products like MySQL or Postgres.

14. Generic Operating System Services. The following modules contain basic features that are common to all operating systems. Most of this commonality is acheived by using the C standard libraries. By using this module, you can be assured that your Python application will be portable to almost any operating system.

os, os.path

> These modules provide access to a number of operating system features. The `os` module provides control over Processes, Files and Directories. We'll look at `os` and `os.path` in the section called "The `os` Module" and the section called "The `os.path` Module".

time

> The `time` module provides basic functions for time and date processing. Additionally `datetime` handles details of the calendar more gracefully than `time` does. We'll cover both modules in detail in Chapter 32, *Dates and Times: the `time` and `datetime` Modules*.
>
> Having modules like `datetime` and `time` mean that you never need to attempt your own calendrical calculations. One of the important lessons learned in the late 90's was that many programmers love to tackle calendrical calculations, but their efforts had to be tested and reworked because of innumerable small problems.

getopt, optparse

> A well-written program makes use of the command-line interface. It is configured through options and arguments, as well as properties files. We'll cover the `getopt`, `optparse` and `glob` modules in Chapter 35, *Programs: Standing Alone*.

logging

Often, you want a simple, standardized log for errors as well as debugging information. We'll look at logging in detail in the section called "Log Files: The `logging` Module".

18. Internet Protocols and Support. The following modules contain algorithms for responding the several of the most common Internet protocols. These modules greatly simplify developing applications based on these protocols.

cgi

The `cgi` module is used for web server applications invoked as CGI scripts. This allows you to put Python programming in the `cgi-bin` directory. When the web server invokes the CGI script, the Python interpreter is started and the Python script is executed.

urllib, urllib2, urlparse

These modules allow you to write relatively simple application programs which open a URL as if it were a standard Python file. The content can be read and perhaps parsed with the HTML or XML parser modules, described below. The `urllib` module depends on the `httplib`, `ftplib` and `gopherlib` modules. It will also open local files when the scheme of the URL is `file:`. The `urlparse` module includes the functions necessary to parse or assemble URL's. The `urllib2` module handles more complex situations where there is authentication or cookies involved.

httplib, ftplib, gopherlib

The `httplib`, `ftplib` and `gopherlib` modules include relatively complete support for building client applications that use these protocols. Between the `html` module and `httplib` module, a simple character-oriented web browser or web content crawler can be built.

poplib, imaplib

The `poplib` and `imaplib` modules allow you to build mail reader client applications. The `poplib` module is for mail clients using the Post-Office Protocol, POP3 (RFC 1725), to extract mail from a mail server. The `imaplib` module is for mail servers using the Internet Message Access Protocol, IMAP4 (RFC 2060) to manage mail on an IMAP server.

nntplib

The `nntplib` module allows you to build a network news reader. The newsgroups, like `comp.lang.python`, are processed by NNTP servers. You can build special-purpose news readers with this module.

SocketServer

The `SocketServer` module provides the relatively advanced programming required to create TCP/IP or UDP/IP server applications. This is typically the core of a stand-alone application server.

SimpleHTTPServer, CGIHTPPServer, BaseHTTPServer

The `SimpleHTTPServer` and `CGIHTTPServer` modules rely on the basic `BaseHTTPServer` and `SocketServer` modules to create a web server. The `SimpleHTTPServer` module provides the programming to handle basic URL requests. The `CGIHTTPServer` module adds the capability for running CGI scripts; it does this with the `fork` and `exec` functions of the `os` module, which are not necessarily supported on all platforms.

asyncore, asynchat

The `asyncore` (and `asynchat`) modules help to build a time-sharing application server. When client requests can be handled quickly by the server, complex multi-threading and multi-processing aren't really necessary. Instead, this module simply dispatches each client communication to an appropriate handler function.

22. Program Frameworks. We'll talk about a number of program-related issues in Chapter 35, *Programs: Standing Alone* and Chapter 36, *Programs: Clients, Servers, the Internet and the World Wide Web*. Much of this goes beyond the standard Python library. Within the library are two modules that can help you create large, sophisticated command-line application programs.

cmd

The `cmd` module contains a superclass useful for building the main command-reading loop of an interactive program. The standard features include printing a prompt, reading commands, providing help and providing a command history buffer. A subclass is expected to provide functions with names of the form `do_command`. When the user enters a line beginning with `command`, the appropriate `do_command` function is called.

shlex

The `shlex` module can be used to tokenize input in a simple language similar to the Linux shell languages. This module defines a basic `shlex` class with parsing methods that can separate words, quotes strings and comments, and return them to the requesting program.

26. Python Runtime Services. The Python Runtime Services modules are considered to support the Python runtime environment. These can be divided into two groups: those that are an interface into the Python interpreter, and those that are generally useful for programming. The interpreter interface allows us to peer under the hood at how Python works internally. The programming category is more generally useful, and includes `sys`, `pickle`, and `shelve`.

sys

The `sys` module contains execution context information. It has the command-line arguments (in `sys.argv`) used to start the Python interpreter. It has the standard input, output and error file definitions. It has functions for retrieving exception information. It defines the platform, byte order, module search path and other basic facts. This is typically used by a main program to get run-time environment information.

Library Exercises

1. **Why are there multiple versions of some packages?** Look at some places where there are two modules which clearly do the same or almost the same things. Examples include `time` and `datetime`, `urllib` and `urllib2`, `pickle` and `cPickle`, `StringIO` and `cStringIO`, `subprocess` and `popen2`, `getopt` and `optparse`. Why allow this duplication? Why not pick a "best" module and discard the others?

2. **Is it better to build an application around the library or simply design the application and ignore the library?** Assuming that we have some clear, detailed requirements, what is the benefit of time spent searching through the library? What if most library modules are a near-miss? Should we alter our design to leverage the library, or just write the program without considering the library?

3. **Which library modules are deprecated or disabled?** Why are these still documented in the library?

Chapter 31. Complex Strings: the `re` Module

Table of Contents

There are a number of related problems when processing strings. When we get strings as input from files, we need to recognize the input as meaningful. Once we're sure it's in the right form, we need to parse the inputs, sometimes we'll ahve to convert some parts into numbers (or other objects) for further use.

For example, a file may contain lines which are supposed to be like `"Birth Date: 3/8/85"`. We may need to determine if a given string has the right form. Then, we may need to break the string into individual elements for date processing.

We can accomplish these recognition, parsing and conversion operations with the `re` module in Python. A *regular expression* (RE) is a rule or pattern used for matching strings. It differs from the fairly simple "wild-card" rules used by many operating systems for naming files with a pattern. These simple operating system file-name matching rules are embodied in two simpler packages: `fnmatch` and `glob`.

We'll look at the semantics of a regular expression in the section called "Semantics". We'll look at the syntax for defining a RE in the section called "Creating a Regular Expression". In the section called "Using a Regular Expression" we'll put the regular expression to use.

Semantics

One way to look at regular expressions is as a production rule for constructing strings. In principle, such a rule could describe an infinite number of strings. The real purpose is not to enumerate all of the strings described by the production rule, but to match a candidate string against the production rule to see if the rule could have constructed the given string.

For example, a rule could be `"aba"`. All strings of the form `"aba"` would match this simple rule. This rule produces only a single string. Determining a match between a given string and the one string produced by this rule is pretty simple.

A more complex rule could be `"ab*a"`. The `b*` means zero or more copies of `b`. This rule produces an infinite set of strings including `"aa"`, `"aba"`, `"abba"`, etc. It's a little more complex to see if a given string could have been produced by this rule.

The Python `re` module includes Python constructs for creating regular expressions (REs), matching candidate strings against RE's, and examining the details of the substrings that match. There is a lot of power and subtlety to this package. A complete treatment is beyond the scope of this book.

Creating a Regular Expression

There are a lot of options and clauses that can be used to create regular expressions. We can't pretend to cover them all in a single chapter. Instead, we'll cover the basics of creating and using RE's. The full set of rules is given in section 4.2.1 Regular Expression Syntax of the *Python Library Reference* document. Additionally, there are many fine books devoted to this subject.

- **Any ordinary character, by itself, is an RE.** Example: `"a"` is an RE that matches the character `a` in the candidate string. While trivial, it is critical to know that each ordinary character is a stand-alone RE.

  Some characters have special meanings. We can *escape* that special meaning by using a `\` in front of them. For example, `*` is a special character, but `\*` escapes the special meaning and creates a single-character RE that matches the character `*`.

  Additionally, some ordinary characters can be made special with `\`. For instance `\d` is any digit, `\s` is any whitespace character. `\D` is any non-digit, `\S` is any non-whitespace character.

- **The character `.` is an RE that matches any single character.** Example: `"x.z"` is an RE that matches the strings like `xaz` or `xbz`, but doesn't match strings like `xabz`.

- **The brackets, `"[...]"`, create a RE that matches any character between the [ ]'s.** Example: `"x[abc]z"` matches any of `xaz`, `xbz` or `xcz`. A range of characters can be specified using a `-`, for example `"x[1-9]z"`. To include a `-`, it must be first or last. ^ cannot be first. Multiple ranges are allowed, for example `"x[A-Za-z]z"`. Here's a common RE that matches a letter followed by a letter, digit or _: `"[A-Za-z][A-Za-z1-9_]"`

- **The modified brackets, `"[^...]"`, create a regular expression that matches any character *except* those between the [ ]'s.** Example: `"a[^xyz]b"` matches strings like `a9b` and `a$b`, but don't match `axb`. As with [ ], a range can be specified and multiple ranges can be specified.

- **A regular expression can be formed from concatenating regular expressions.** Example: `"a.b"` is three regular expressions, the first matches `a`, the second matches any character, the third matches `b`.

- **A regular expression can be a group of regular expressions, formed with ()'s.** Example: `"(ab)c"` is a regular expression composed of two regular expressions: `"(ab)"` (which, in turn, is composed of two RE's) and `"c"`. ()'s also group RE's for extraction purposes. The elements matched within ()'s are remembered by the regular expression processor and set aside in a `match` object.

- **A regular expression can be repeated.** Several repeat constructs are available: `"x*"` repeats `"x"` zero or more times; `"x+"` repeats `"x"` 1 or more times; `"x?"` repeats `"x"` zero or once. Example: `"1(abc)*2"` matches `12` or `1abc2` or `1abcabc2`, etc. The first match, against `12`, is often surprising; but there are zero copies of `abc` between `1` and `2`.

- **The character `"^"` is an RE that only matches the beginning of the line, `"$"` is an RE that only matches the end of the line.** Example: `"^$"` matches a completely empty line.

Here are some examples.

`"[_A-Za-z][_A-Za-z1-9]*"`

Matches a Python identifier. This embodies the rule of starting with a letter or _, and containing any number of letters, digits or _'s. Note that any number includes 0 occurances, so a single letter or _ is a valid identifier.

`"^\s*import\s"`

Matches a simple **import** statement. It matches the beginning of the line with `^`, zero or more whitespace characters with `\s*`, the sequence of letters `import`; and one more whitespace character. This pattern will ignore the rest of the line.

`"^\s*from\s+[_A-Za-z][_A-Za-z1-9]*\s+import\s"`

> Matches a `from` *`module`* `import` statement. As with the simple import, it matches the beginning of the line (`^`), zero or more whitespace characters (`\s*`), the sequence of letters `from`, a Python module name, one or more whitespace characters (`\s+`), the sequence `import`, and one more whitespace character.

`"(\d+):(\d+):(\d+\.?\d*)"`

> Matches a one or more digits, a `:`, one or more digits, a `:`, and digits followed by optional `.` and zero or more other digits. For example `20:07:13.2` would match, as would `13:04:05` Further, the ()'s would allow separating the digit strings for conversion and further processing.

`"def\s+([_A-Za-z][_A-Za-z1-9]*)\s+\([^)]*\):"`

> Matches Python function definition lines. It matches the letters `def`; a string of 1 or more whitespace characters (`\s`); an identifier, surrounded by ()'s to capture the entire identifier as a match. It matches a `(`; we've used `\(` to escape the meaning of (and make it an ordinary character. It matches a string of non-`)` characters, which would be the parameter list. The parameter list ends with a `)`; we've used `\)` to make escape the meaning of `)` and make it an ordinary character. Finally, we need tyo see the `:`.

Using a Regular Expression

There are several methods which are commonly used with regular expressions. The most common first step is to compile the RE definition string to make an `Pattern` object. The resulting Pattern object can then be used to match or search candidate strings. A successful match returns a `Match` object with details of the matching substring.

The `re` module provides the `compile` function.

`re.compile (`*`expr`*`)` → `Pattern`

> Create a `Pattern` object from an RE string. The Pattern is used for all subsequent searching or matching operations. A Pattern has several methods, including `match` and `search`.

Generally, raw string notation (`r"pattern"`) is used to write a RE. This simplifies the `\`'s required. Without the raw notation, each `\` in the string would have to be escaped by a `\`, making it `\\`. This rapidly gets cumbersome. There are some other options available for `re.compile`, see the *Python Library Reference*, section 4.2, for more information.

The following methods are part of a compiled `Pattern`. We'll use the name *`pat`* to refer to some `Pattern` object created by the `re.compile` function.

`pat``.match (`*`string`*`)` → `Match`

> Match the candidate string against the compiled regular expression, *`pat`*. Matching means that the regular expression and the candidate string must match, starting at the beginning of the candidate string. A `Match` object is returned if there is match, otherwise `None` is returned.

`pat``.search (`*`string`*`)` → `Match`

> Search a candidate string for the compiled regular expression, *`pat`*. Searching means that the regular expression must be found somewhere in the candidate string. A `Match` object is returned if the pattern is found, otherwise `None` is returned.

If `search` or `match` finds the pattern in the candidate string, a `Match` object is created to describe the part of the candidate string which matched. The following methods are part of a `Match` object. We'll use the name *`match`* to refer to some `Match` object created by a successul search or match operation.

`match.group` (*`number`*) → string

> Retrieve the string that matched a particular () grouping in the regular expression. Group zero is a tuple of everything that matched. Group 1 is the material that matched the first set of ()'s.

Here's a more complete example.

```
>>> import re
>>> rawin= "20:07:13.2"
>>> hms_pat= re.compile( r'(\d+):(\d+):(\d+\.?\d*)' )
>>> hms_match= hms_pat.match( rawin )
>>> print hms_match.group( 0, 1, 2, 3 )
('20:07:13.2', '20', '07', '13.2')
>>> h,m,s= map( float, hms_match.group(1,2,3) )
>>> seconds= ((h*60)+m)*60+s
>>> print h, m, s, "=", seconds
20.0 7.0 13.2 = 72433.2
```

This sequence decodes a complex input value into fields and then computes a single result. The **import** statement incorporates the `re` module. The `rawin` variable is sample input, perhaps from a file, perhaps from `raw_input`. The `hms_pat` variable is the compiled regular expression object which matches three numbers, using `"(\d+)"`, separated by :'s.

The digit-sequence RE's are surround by ()'s so that the material that matched is returned as a group. This will lead to four groups: group 0 is everything that matched, groups 1, 2, and 3 are successive digit strings. The `hms_match` variable is a `Match` object that indicates success or failure in matching. If `hms_match` is `None`, no match occurred. Otherwise, the `hms_match.group` method will reveal the individually matched input items.

The statement that sets `h`, `m`, and `s` does three things. First is uses `hms_match.group` to create a tuple of requested items. Each item in the tuple will be a string, so the `map` function is used to apply the built-in `float` function against each string to create a tuple of three numbers. Finally, this statement relies on the multiple-assignment feature to set all three variables at once. Finally, `seconds` is computed as the number of seconds past midnight for the given time stamp.

Regular Expression Exercises

1. **Parse Stock prices.** Create a function that will decode the old-style fractional stock price. The price can be a simple floating point number or it can be a fraction, for example, `4 5/8`.

   Develop two patterns, one for numbers with optional decimal places and another for a number with a space and a fraction. Write a function that accepts a string and checks both patterns, returning the correct decimal price for whole numbers (e.g., 14), decimal prices (e.g., 5.28) and fractional prices (27 1/4).

2. **Parse Dates.** Create a function that will decode a few common American date formats. For example, 3/18/87 is March 18, 1987. You might want to do 18-Mar-87 as an alternative format. Stick to two or three common formats; otherwise, this can become quite complex.

   Develop the required patterns for the candidate date formats. Write a function that accepts a string and checks all patterns. It will return the date as a tuple of (year,

month, day).

Chapter 32. Dates and Times: the `time` and `datetime` Modules

Table of Contents

When processing dates and times, we have a number of problems. Most of these problems stem from the irregularities and special cases in the units we use to measure time. We generally measure time in a number of compatible as well as incompatible units. For example, weeks, days, hours, minutes and seconds are generally compatible, with the exception of leap-second handling. Months, and years, however are incompatible with days and require sophisticated conversion.

Problems which mix month-oriented dates and numbers of days are particularly difficult. The number of days between two dates, or a date which is 90 days in the future are notoriously difficult to compute correctly.

We need to represent a point in time, a date, a time of day or a date-time stamp. We need to be able to do arithmetic on this point in time. And, we need to represent this point in time as a properly-punctuated string.

The `time` module contains a number of portable functions needed to format times and dates. The `datetime` module builds on this to provide a representation that is slightly more convenient for some things. We'll look at the definition of a moment in time in the section called “Semantics: What is Time?”.

Semantics: What is Time?

The Gregorian calendar is extremely complex. Most of that complexity stems from trying to impose a fixed "year" on the wobbly, irregular orbit of our planet. There are several concesssions required to impose a calendar year with integer numbers of days that will match the astronomial year of approximately 365.2425 days. The Gregorian calendar's concession is the periodic addition of a leap day to approximate this fractional day. The error is just under .25, so one leap day each four years gets close to the actual duration of the year.

Clearly, we have several systems of units available to us for representing a point in time.

- Seconds are the least common denominator. We can easily derive hours and minutes from seconds. There are 24×60×60 = 86,400 seconds in a day. Astronomers will periodically add a leap second, so this is not absolutely true. We can use seconds as a simple representation for a point in time. We can pick some epoch and represent any other point in time as the number of seconds after the epoch. This makes arithmetic very simple. However, it's hard to read; what month contains second number 1,190,805,137?

- Days are another common denominator in the calendar. There are seven days in a week, and (usually) 86,400 seconds in day, so those conversions are simple. We can pick some epoch and represent any other point in time as the number of days after the epoch. This also makes arithmetic very simple. However, it's hard to read; what month contains day number 732,945?

- Months are the real root cause of our problems. If we work with the conventional date triple of year, month, and day, we can't compute intervals between dates very well at all. We can't locate a date 90 days in the future without a rather complex algorithm.

We also to have acknowledge the subtlety of local time and the potential differences between local standard time and local daylight time (or summer time). Since the clock shifts, some time numbers (1:30 AM, for example) will appear to repeat, this can require the timezone hint to decode the time number.

The more general solution is to do all work in UTC. Input is accepted and displayed in local time for the convenience of users. This has the advantage of being timezone neutral, and it makes time values monotonically increasing with no confusing repeats of a given time of day during the hour in which the clock is shifted.

The `time` Solution. The `time` module uses two different representations for a point in time, and provides numerous functions to help us convert back and forth between the two representations.

- A `float` seconds number. This is the UNIX internal representation for time. (The number is seconds past an epoch; the epoch happens to January 1st, 1970.) In this representation, a duration between points in time is also a `float` number.

- A `struct_time` object. This object has nine attributes for representing a point in time as a Gregorian calendar date and time. We'll look at this structure in detail below. In this representation, there is no representation for the duration between points in time. You need to convert back and forth between `struct_time` and seconds representations.

The `datetime` Solution. The `datetime` module contain all of the objects and methods required to correctly handle the sometimes obscure rules for the Gregorian calendar. Additionally, it is possible to use date information in the `datetime` to usefully conver among the world's calendars. For details on conversions between calendar systems, see *Calendrical Calculations* [Deshowitz97].

The `datetime` module has just one representation for a point in time. It assigns an ordinal number to each day. The numbers are based on an epochal date, and algorithms to derive the year, month and day information for that ordinal day number. Similarly, this class provides algorithms to convert a calendar date to an ordinal day number. (Note: the Gregorian calendar was not defined until 1582, all dates before the official adoption are termed *proleptic*. Further, the calendar was adopted at different times in different countries.)

There are four classes in this module that help us handle dates and times in a uniform and correct manner. We'll skip the more advanced topic of the `datetime.tzinfo` class.

- **`datetime.time`.** An instance of `datetime.time` has four attributes: hour, minute, second and microsecond. Additionally, it can also carry an instance of `tzinfo` which describes the time zone for this time.

- **`datetime.date`.** An instance of `datetime.date` has three attributes: year, month and day. There are a number of methods for creating `datetime.date`s, and converting `datetime.date`s to various other forms, like floating-point timestamps, 9-tuples for use with the `time` module, and ordinal day numbers.

- **`datetime.datetime`.** An instance of `datetime.datetime` combines `datetime.date` and `datetime.time`. There are a number of methods for creating `datetime.datetime`s, and converting `datetime.datetime`s to various other forms, like floating-point timestamps, 9-tuples for use with the `time` module, and ordinal day numbers.

- **`datetime.timedelta`.** A `datetime.timedelta` is the duration between two dates, times or datetimes. It has a value in days, seconds and microseconds. These can be added to or subtracted from dates, times or datetimes to compute new dates, times or datetimes.

Some Class Definitions

A `time.struct_time` object behaves like an object as well as a tuple. You can access the attributes of the structure by position as well as by name. Note that this class has no methods of it's own; you manipulate these objects using functions in the `time` module.

`tm_year`

The year. This will be a full four digit year, e.g. 1998.

`tm_mon`

The month. This will be in the range of 1 to 12.

`tm_mday`

The day of the month. This will be in the range of 1 to the number of days in the given month.

`tm_hour`

The hour of the day, in the range 0 to 23.

`tm_min`

The minutes of the hour, in the range 0 to 59.

`tm_sec`

The seconds of the minute, in the range 0 to 61 because leap seconds may be included. Not all platforms support leap seconds.

`tm_wday`

The day of the week. This will be in the range of 0 to 6. 0 is Monday, 6 is Sunday.

`tm_yday`

The day of the year, in the range 1 to 366.

`tm_isdst`

Is the time in local daylight savings time. 0 means that this is standard time; 1 means daylight time. If you are creating this object, you can provide -1; the `mktime` can then determine DST based on the date and time.

We'll focus on the `datetime.datetime` class, since it includes `datetime.date` and `datetime.time`. This class has the following attributes.

`MINYEAR, MAXYEAR`

These two attributes bracket the time span for which datetime works correctly. This is year 1 to year 9999, which covers the forseeable future as well as a past the predates the invention of the Gregorian calendar in 1582.

`min, max`

The earliest and laterst representable `datetimes`. In effect these are `datetime(MINYEAR, 1, 1, tzinfo=None)` and `(MAXYEAR, 12, 31, 23, 59,`

59, 999999, tzinfo=None).

`resolution`

The smallest differences between `datetimes`. This is typically equaly to `timedelta(microseconds=1)`.

`year`

The year. This will be a full four digit year, e.g. 1998. It will always be between `MINYEAR` and `MAXYEAR`, inclusive.

`month`

The month. This will be in the range 1 to 12.

`day`

The day. This will be in the range 1 to the number of days in the given month.

`hour`

The hour. This will be in the range 0 to 23.

`minute`

The minute. This will be in the range 0 to 59.

`second`

The second. This will be in the range 0 to 59.

`microsecond`

The microsecond (millionths of a second). This will in the range 0 to 999,999. Some platforms don't have a system clock which is this accurate. However, the SQL standard imposes this resolution on most date time values.

`tzinfo`

The `datetime.tzinfo` object that was provided to the initial `datetime.datetime` constructor. Otherwise it will be `None`.

Creating a Date-Time

There are two use cases for creating `date`, `time`, `struct_time` or `datetime` instances. In the simplest case, we're asking our operating system for the current date-time or the date-time associated with some resource or event. In the more complex case, we are asking a user for input (perhaps on an interactive GUI, a web form, or reading a file prepared by a person); we are parsing some user-supplied values to see if they are a valid date-time and using that value.

From The OS. We often get time from the OS when we want the current time, or we want one of the timestamps associated with a system resource like a file or directory. Here's a sampling of techniques for getting a date-time.

`time.time()` → float

Returns the current moment in time as a `float` seconds number. See Chapter 33, *File Handling Modules* for examples of getting file timestamps; these are always a `float` seconds value. We'll often need to convert this to a `struct_time` or `datetime` object so that we can provide formatted output for users.

The functions `time.localtime` or `time.gmtime` will convert this value to a `struct_time`. The class methods `datetime.datetime.fromtimestamp`, and `datetime.datetime.utcfromtimestamp` will create a `datetime` object from this time value.

Then, we can use `time.strftime` or `time.asctime` to format and display the time.

`time.ctime()` → string, `time.asctime()` → string

Returns a string representation of the current time. These values aren't terribly useful for further calculation, but they are handy, standardized timestamp strings.

`time.localtime()` → `struct_time`, `time.gmtime()` → `struct_time`

When these functions are evaluated with no argument value, they will create `struct_time` objects from the current time. Since we can't do arithmetic with these values, we often need to convert them to something more useful.

The `time.mktime` function will convert the `struct_time` to a `float` seconds time.

We have to use the `datetime.datetime` constructor to create a datetime from a `struct_time`. This can be long-winded, it will look like `datetime.date( ts.tm_year, ts.tm_month, ts.tm_day )`.

`datetime.datetime.today()` → `datetime`, `datetime.datetime.now()` → `datetime`, `datetime.datetime.utcnow()` → `datetime`

All of these are class methods of the `datetime` class; the create a `datetime` object. The `today` function uses the simple time.time() notion of the current moment and returns local time. The `now` function may use a higher-precision time, but it will be local time. The `utcnow` function uses high-precision time, and returns UTC time, not local time.

We can't directly get a `float` seconds number number from a datetime value. However, we can do arithmetic directly with `datetime` values, making the `float` seconds value superflous.

We can get the `struct_time` value from a `datetime`, using the `timetuple` or `utctimetuple` method functions of the datetime object.

Getting Time From A User. Human-readable time information generally has to be parsed from one or more string values. Human-readable time can include any of the endless variety of formats in common use. This will include some combination of years, days, months, hours, minutes and seconds, and timezone names.

There are two general approaches to parsing time. In most cases, it is simplest to use `datetime.strptime` to parse a string and create a datetime object. In other cases, we can use `time.strptime`. In the most extreme case, we have to either use the `re` module (xChapter 31, *Complex Strings: the `re` Module*), or some other string manipulation, and then create the date-time object directly.

`datetime.strptime(`*`string`*`,` ⟨*`format`*⟩`)` → `datetime`

This function will use the given format to attempt to parse the input string. If the value doesn't match the format, it will raise a `ValueError` exception. If the format is not a complete datetime, then defaults are filled in. The default year is 1900, the default month is 1 the default day is 1. The default time values are all zero.

We'll look at the format string under the `time.strftime` function, below.

`time.strptime(`*`string`*`,⟨`*`format`*`⟩) → struct_time`

This function will use the given format to attempt to parse the input string. If the value doesn't match the format, it will raise a `ValueError` exception. If the format is not a complete time, then defaults are filled in. The default year is 1900, the default month is 1 the default day is 1. The default time values are all zero.

We'll look at the format string under the `time.strftime` function, below.

`time.struct_time(`*`9-tuple`*`) → struct_time`

Creates a `struct_time` from a 9-valued tuple: (year, month, day, hour, minute, second, day of week, day of year, dst-flag). Generally, you can supply 0 for day of week and day of year. The dst flag is 0 for standard time, 1 for daylight (or summer) time, and -1 when the date itself will define if the time is standard or daylight.

This constructor does no validation; it will tolerate invalid values. If we use the `time.mktime` function to do a conversion, this may raise an `OverflowError` if the time value is invalid.

Typically, you'll build this 9-tuple from user-supplied inputs. We could parse a string using the `re` module, or we could be collecting input from fields in a GUI or the values entered in a web-based form. Then you attempt a `time.mktime` conversion to see if it is valid.

`datetime.datetime(`*`year`*`,`*`month`*`,`*`day`*`, ⟨`*`hour`*`⟩ ,⟨`*`minute`*`⟩`*`second`*`⟨`*`microsecond`*`⟩ ⟨`*`tzinfo`*`⟩) → datetime`

Creates a `datetime` from individual parameter values. Note that the time fields are optional; if omitted the time value is 0:00:00, which is midnight.

This constructor will not tolerate a bad date. It will raise a `ValueError` for an invalid date.

Date-Time Calculations and Manipulations

There are two classes of date-time calculations.

- Duration or interval calculations in days (or seconds), where the month, week and year boundaries don't matter. For example, the number of hours, days or weeks between two dates doesn't depend on months or year boundaries. Similarly, calculating a date 90 days in the future or past doesn't depen on month or year considerations. Even the difference between two times is properly a date-time calculation so that we can allow for rollover past midnight.

  We have two ways to do this.

  - We can use `float` seconds information, as produced by the `time` module. When we're using this representation, a day is `24*60*60` (86,400) seconds, and a week is `168*24*60*60` seconds. For the following examples, `t1` and `t2` and `float` seconds times.

    `(t2-t1)/3600` is the number of hours between two times.

    `(t2-t1)/86400` is the days between two dates.

    `t1+90*86400` is the date 90 days in the future.

  - We can also use `datetime` objects for this, since `datetime` objects correctly handle arithmetic operations. When we're using this representation, we'll

also work with `datetime.timedelta` objects; these have days, seconds and microseconds attributes. For the following examples, `t1` and `t2` and `datetime` objects.

In a relatively simple case, the hours between two datetimes is `(t2-t1).seconds/3600`. This works when we're sure that there sill be less than 24 hours between the two `datetimes`.

In the more general case, we have a two-part calculation: `td= (t2-t1); td.days*86400+td.seconds` is the seconds between two time periods, assuming that there may be more than a whole day between them.

`(t2-t1).days` will create the `timedelta` between two `datetimes`; it extracts the `days` attribute of that `timedelta` object.

- Calendar calculations where the month, week of month and day of week matter. For example, the number of months between two dates rarely involves the day of the month. A date that is 3 months in the future, will land on the same day of the month, except in unusual cases where it would be the 30th of February. For these situations, highly problem-specific rules have to be applied; there's no general principle.

  We have two ways to do this.

  - We can use `struct_time` objects as produced by the `time` module. We can replace any `struct_time` fields, and possibly create an invalid date. We may need to use `time.mktime` to validate the resulting `struct_time` object. In the following examples, `t1` is a `struct_time` object.

    Adding some `offset` in months, correctly allowing for year-end rollover, is done as follows.

    ```
    monthSequence= (t1.tm_year*12 + t1.tm_mon-1) + offset
    futureYear, futureMonth0 = divmod( monthSequence, 12 )
    t1.tm_year= futureYear
    t1.tm_month= futureMonth0 + 1
    ```

  - We can also use `datetime` objects for this. In this case, we we'll use the `replace` method to replace a value in a `datetime` with other values. In the following examples, `t1` is a `datetime` object.

    Adding some `offset` in months, correctly allowing for year-end rollover, is done as follows.

    ```
    monthSequence= (t1.year*12 + t1.month-1) + offset
    futureYear, futureMonth0 = divmod( monthSequence, 12 )
    t1= t1.replace( year=futureYear, month=futureMonth0+1 )
    ```

The following methods return information about a given `datetime` object. In the following definitions, `dt` is a `datetime` object.

`dt.date` → `date`

Return a new `date` object from the date fields of this `datetime` object.

`dt.time` → `time`

Return a new `time` object from the time fields of this `datetime` object.

`dt.replace` (*`year=`*, *`month=`*, *`day=`*, *`hour=`*, *`minute=`*, *`second=`*, *`microsecond=`*) → `datetime`

Return a new `datetime` object from the current `datetime` object after replacing any values provided by the keyword arguments.

`dt.toordinal` → int

Return the ordinal day number for this `datetime`.

`dt.weekday` → int

Return the day of the week. Monday = 0, Sunday = 6.

`dt.isoweekday` → int

Return the day of the week. Monday = 1, Sunday = 7.

`dt.isocalendar` → tuple

Return a tuple with (ISO year, ISO week, ISO week day).

Presenting a Date-Time

To format human-readable time, we have a number of functions in the `time` module, and methods of a `datetime` object. Here are the functions in the `time` module.

`time.strftime` (*`format`*, *`struct`*) → string

Convert a `struct_time` to a string according to a format specification. The specification rules are provided below.

This is an example of how to produce a timestamp with the fewest implicit assumptions.

```
time.strftime( "%x %X", time.localtime( time.time() ) )
```

This line of code shows a standardized and portable way to produce a time stamp. The `time.time` function produces the current time in UTC (Coordinated Universal Time). Time is represented as a floating point number of seconds after an epoch.

`time.asctime` (*`struct`*) → string

Convert a `struct_time` to a string, e.g. 'Sat Jun 06 16:26:11 1998'. This is the same as a the format string `"%a %b %d %H:%M:%S %Y"`.

`time.ctime` (*`seconds`*) → string

Convert a time in seconds since the Epoch to a string in local time. This is equivalent to `asctime(localtime(seconds))`.

A `datetime` object has the following methods for producting formatted output. In the following definitions, `dt` is a `datetime` object.

`dt.isoformat`(*`separator`*) → string

Return string representing this date in ISO 8601 standard format. The *`separator`* string is used between the date and the time.

`dt.ctime` → string

Return string representing this date and time. This is equivalent to `time.ctime(time.mktime(`*`dt`*`.timetuple()))`.

`dt.strftime(`*`format`*`)` → string

Return string representing this date and time, formatted using the given format string. This is equivalent to `time.strftime( `*`format`*`, time.mktime( `*`dt`*`.timetuple() ) )`.

The `strftime` and `strptime` functions use the following formatting symbols to convert between 9-tuples and strings. Formatting symbols like `%c`, `%x` and `%X` produce standard formats for whole date-time stamps, dates or time. Other symbols format parts of the date or time value. The following examples show a particular date (Satuday, August 4th) formatted with each of the formatting strings.

%a	Locale's 3-letter abbreviated weekday name.	"Sat"
%A	Locale's full weekday name.	"Saturday"
%b	Locale's 3-letter abbreviated month name.	"Aug"
%B	Locale's full month name.	"August"
%c	Locale's appropriate full date and time representation.	"Saturday August 04 17:11:20 2001"
%d	Day of the month as a 2-digit decimal number.	"04"
%H	Hour (24-hour clock) as a 2-digit decimal number.	"17"
%I	Hour (12-hour clock) as a 2-digit decimal number.	"05"
%j	Day of the year as a 3-digit decimal number.	"216"
%m	Month as a 2-digit decimal number.	"08"
%M	Minute as a 2-digit decimal number.	"11"
%p	Locale's equivalent of either AM or PM.	"pm"
%S	Second as a 2-digit decimal number.	"20"

%U	Week number of the year (Sunday as the first day of the week) as a 2-digit decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	"30"
%w	Weekday as a decimal number, 0 = Sunday.	"6"
%W	Week number of the year (Monday as the first day of the week) as a 2-digit decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	"31"
%x	Locale's appropriate date representation.	"Saturday August 04 2001"
%X	Locale's appropriate time representation.	"17:11:20"
%y	Year without century as a 2-digit decimal number.	"01"
%Y	Year with century as a decimal number.	"2001"
%Z	Time zone name (or '' if no time zone exists).	""
%%	A literal '%' character.	"%"

Time Exercises

1. **Return on Investment.** Return on investment (ROI) is often stated on an annual basis. If you buy and sell stock over shorter or longer periods of time, the ROI must be adjusted to be a full year's time period. The basic calculation is as follows:

   Given the sale date, purchase date, sale price, `sp`, and purchase price, `pp`.

   Compute the period the asset was held: use `time.mktime` to create floating point time values for sale date, `s`, and purchase date, `p`. The weighting, `w`, is computed as

   `w`= (86400*365.2425) / (`s`-`p`)

Write a program to compute ROI for some fictitious stock holdings. Be sure to include stocks held both more than one year and less than one year. See the section called "Stock Valuation" in Chapter 21, *Classes* for some additional information on this kind of calculation.

2. **Surface Air Consumption Rate.** When doing SACR calculations (see Surface Air Consumption Rate, and the section called "Dive Logging and Surface Air Consumption Rate") we've treated the time rather casually. In the event of a night dive that begins before midnight and ends after midnight the next day, our quick and dirty time processing doesn't work correctly.

   Revise your solution to use a more complete date-time stamp for the start and end time of the dive. Use the `time` module to parse those date-time stamps and compute the actual duration of the dive.

Additional `time` Module Features

Here are some additional functions in the `time` module.

`time.sleep (`*`seconds`*`,`*`tuple`*`)`

Delay execution for a given number of seconds. The argument may be a floating point number for subsecond precision.

`time.accept2dyear`

If non-zero, 2-digit years are accepted. 69-99 is treated as 1969 to 1999, 0 to 68 is treated as 2000 to 2068. This is 1 by default, unless the `PYTHONY2K` environment variable is set; then this variable will be zero.

`time.altzone`

Difference in seconds between UTC and local Daylight Savings time. Often a multiple of 3600 (all US time zones are in whole hours). For example, Eastern Daylight Time is 14400 (4 hours).

`time.daylight`

Non-zero if the locale uses daylight savings time. Zero if it does not.

`time.timezone`

Difference in seconds between UTC and local Standard time. Often a multiple of 3600 (all US timezones are in whole hours).

`time.tzname`

The name of the timezone.

Chapter 33. File Handling Modules

Table of Contents

There are a number of operations closely related to file processing. Deleting and renaming files are examples of operations that change the directory information that the operating system maintains to describe a file. Python provides numerous modules for these operating system operations.

We can't begin to cover all of the various ways in which Python supports file handling. However, we can identify the essential modules that may help you avoid reinventing the wheel. Further, these modules can provide you a view of the Pythonic way of working with data from files.

The following modules have features that are essential for supporting file processing. We'll cover selected features of each module that are directly relevant to file processing. We'll present these in the order you'd find them in the Python library documentation.

Chapter 11 - File and Directory Access. Chapter 11 of the Library reference covers many modules which are essential for reliable use of files and directories. We'll look closely at the following modules.

`os.path`

> Common pathname manipulations. Use this to split and join full directory path names. This is operating-system neutral, with a correct implementation for all operating systems.

`os`

> Miscellaneous OS interfaces. This includes parameters of the current process, additional file object creation, manipluations of file descriptors, managing directories and files, managing subprocesses, and additional details about the current operating system.

`fileinput`

> This module has functions which will iterate over lines from multiple input streams. This allows you to write a single, simple loop that processes lines from any number of input files.

`tempfile`

> Generate temporary files and temporary file names.

`glob`

> UNIX shell style pathname pattern expansion. Unix shells translate name patterns like `*.py` into a `list` of files. This is called *globbing*. The `glob` module implements this within Python, which allows this feature to work even in Windows where it isn't supported by the OS itself.

`fnmatch`

> UNIX shell style filename pattern matching. This implements the glob-style rules using *, ? and []. * matches any number of characters, ? matches any single character, [*chars*] encloses a `list` of allowed characters, [!*chars*] encloses a `list` of disallowed characters.

`shutil`

> High-level file operations, including copying and removal. The kinds of things that the shell handles with simple commands like **cp** or **rm** become available to a

Python program, and are just as simple in Python as they are in the shell.

Chapter 12 - Data Compression and Archiving. Data Compression is covered in Chapter 12 of the Library referece. We'll look closely at the following modules.

`tarfile`, `zipfile`

These modules helps you read and write archive files; files which are an archive of a complex directory structure. This includes GNU/Linux tape archive (`.tar`) files, compressed GZip tar files (`.tgz` files or `.tar.gz` files) sometimes called tarballs, and ZIP files.

`zlib`, `gzip`, `bz2`

These modules are all variations on a common theme of reading and writing files which are *compressed* to remove redundant bytes of data. The `zlib` and `bz2` modules have a more sophisticated interface, allowing you to use compression selectively within a more complex application. The `gzip` module has a different (and simpler) interface that only applies only to complete files.

Chapter 26 - Python Runtime Services. These modules described in Chapter 26 of the Library reference include some that are used for handling various kinds of files. We'll look closely as just one.

`sys`

This module has several system-specific parameters and functions, including definitions of the three standard files that are available to every program.

The `os.path` Module

The `os.path` module contains more useful functions for managing path and directory names. A serious mistake is to use ordinary `string` functions with literal `strings` for the path separators. A Windows program using `\` as the separator won't work anywhere else. A less serious mistake is to use `os.pathsep` instead of the routines in the `os.path` module.

The `os.path` module contains the following functions for completely portable path and filename manipulation.

`os.path.basename` (*`path`*) → fileName

Return the base filename, the second half of the result created by `os.path.split(` *`path`* `)`

`os.path.dirname` (*`path`*) → dirName

Return the directory name, the first half of the result created by `os.path.split(` *`path`* `)`

`os.path.exists` (*`path`*) → boolean

Return True if the pathname refers to an existing file or directory.

`os.path.getatime` (*`path`*) → time

Return the last access time of a file, reported by `os.stat`. See the `time` module for functions to process the time value.

`os.path.getmtime` (*`path`*) → time

Return the last modification time of a file, reported by `os.stat`. See the `time`

module for functions to process the time value.

`os.path.getsize` (*`path`*) → int

Return the size of a file, in bytes, reported by `os.stat`.

`os.path.isdir` (*`path`*) → boolean

Return True if the pathname refers to an existing directory.

`os.path.isfile` (*`path`*) → boolean

Return True if the pathname refers to an existing regular file.

`os.path.join` (*`string`*, `...`) → path

Join path components using the appropriate path separator.

`os.path.split` (*`path`*) → tuple

Split a pathname into two parts: the directory and the basename (the filename, without path separators, in that directory). The result (s, t) is such that `os.path.join`(*`s`*, *`t`*) yields the original path.

`os.path.splitdrive` (*`path`*) → tuple

Split a pathname into a drive specification and the rest of the path. Useful on DOS/Windows/NT.

`os.path.splitext` (*`path`*) → tuple

Split a path into root and extension. The extension is everything starting at the last dot in the last component of the pathname; the root is everything before that. The result (r, e) is such that `r+e` yields the original path.

The following example is typical of the manipulations done with `os.path`.

```
import sys, os.path
def process( oldName, newName ):
    Some Processing...

for oldFile in sys.argv[1:]:
    dir, fileext= os.path.split(oldFile)
    file, ext= os.path.splitext( fileext )
    if ext.upper() == '.RST':
        newFile= os.path.join( dir, file ) + '.HTML'
        print oldFile, '->', newFile
        process( oldFile, newFile )
```

❶ This program imports the `sys` and `os.path` modules.

❷ The `process` function does something interesting and useful to the input file. It is the real heart of the program.

❸ The **for** statement sets the variable `oldFile` to each `string` (after the first) in the sequence `sys.argv`.

❹ Each file name is split into the path name and the base name. The base name is further split to separate the file name from the extension. The `os.path` does this correctly for all operating systems, saving us having to write platform-specific code. For example, `splitext` correctly handles the situation where a linux file has multiple '.'s in the file name.

❺ The extension is tested to be '.RST'. A new file name is created from the path, base name and a new extension ('.HTML'). The old and new file names are printed and some processing, defined in the `process`, uses the `oldFile` and `newFile` names.

The os Module

The os module contains an interface to many operating system-specific functions to manipulate processes, files, file descriptors, directories and other "low level" features of the OS. Programs that import and use os stand a better chance of being portable between different platforms. Portable programs must depend only on functions that are supported for all platforms (e.g., unlink and opendir), and perform all pathname manipulation with os.path.

The os module exports the following variables that characterize your operating system.

os.name

> A name for the operating system, for example 'posix', 'nt', 'dos', 'os2', 'mac', or 'ce'. Note that Mac OS X has an os.name of 'posix'; but sys.platform is 'darwin'. Windows XP has an os.name of 'nt'.

os.curdir

> String representing the current directory ('.', generally).

os.pardir

> String representing the parent directory ('..', generally).

os.sep, os.altsep

> The (or a most common) pathname separator ('/' or ':' or '\') and the alternate pathname separator (None or '/'). Most of the Python library routines will translate '/' to the correct value for the operating system (typically, '\' on Windows. It is best to always use os.path rather than these low-level constants.

os.pathsep

> The component separator used in $PATH (':', generally).

os.linesep

> The line separator in text files ('\n' or '\015\012'). This is already part of the readlines function.

os.defpath

> The default search path for executables, for example, ':/bin:/usr/bin' or '.;C:\\bin'.

The os module has a large number of functions. Many of these are not directly related to file manipulation. However, a few are commonly used to create and remove files and directories. Beyond these basic manipulations, the shutil module supports a variety of file copy operations.

os.chdir*path*

> Change the current working directory to the given path. This is the directory which the OS uses to transform a relative file name into an absolute file name.

os.getcwd

> Returns the path to the current working directory. This is the directory which the OS use to transform a relative file name into an absolute file name.

os.listdir*path*

Returns a `list` of all entries in the given directory.

`os.mkdir`*`path [mode]`*

Create the given directory. In GU/Linux, the mode can be given to specify the permissions; usually this is an octal number. If not provided, the default of 0777 is used, after being updated by the OS umask value.

`os.rename`*`source destination`*

Rename the source filename to the destination filename. There are a number of errors that can occur if the source file doesn't exist or the destination file already exists or if the two paths are on different devices. Each OS handles the situations slightly differently.

`os.remove(`*`file`*`)`

Remove (also known as delete or unlink) the file. If you attempt to remove a directory, this will raise `OSError`. If the file is in use, the standard behavior is to remove the file when it is finally closed; Windows, however, will raise an exception.

`os.rmdir`*`path`*

Remove (also known as delete or unlink) the directory. if you attempt to remove an ordinary file, this will raise `OSError`.

Here's a short example showing some of the functions in the `os` module.

```
>>> import os
>>> os.chdir("/Users/slott")
>>> os.getcwd()
'/Users/slott'
>>> os.listdir(os.getcwd())
['.bash_history', '.bash_profile', '.CFUserTextEncoding', '.DS_Store',
'.fonts.cache-1', '.idlerc', '.jedit', '.leoRecentFiles.txt', '.lpoptions',
'.sqlite_history', '.ssh', '.subversion', '.Trash', '.viminfo', '.xxe',
'argo.user.properties', 'Desktop', 'Documents', 'idletest', 'Library',
'Movies', 'Music', 'Pictures', 'Public', 'Sites']
```

The `fileinput` Module

The `fileinput` module interacts with `sys.argv`. The `fileinput.input` function opens files based on all the values of `sys.argv[1:]`. It carefully skips `sys.argv[0]`, which is the name of the Python script file. For each file, it reads all of the lines as text, allowing a program to read and process multiple files, like many standard Unix utilities.

The typical use case is:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, with a default of `sys.stdin` if the `list` is empty. If a filename is `-` it is also replaced by `sys.stdin` at that position in the list of files. To specify an alternative list of filenames, pass it as the argument to `input`. A single file name is also allowed in addition to a list of file names.

While processing input, several functions are available in the `fileinput` module:

`fileinput.filename` → string

the filename of the line that has just been read.

`fileinput.lineno` → int

the cumulative line number of the line that has just been read.

`fileinput.filelineno` → int

the line number in the current file.

`fileinput.isfirstline` → int

true if the line just read is the first line of its file.

`fileinput.isstdin` → int true

if the line was read from `sys.stdin`.

`fileinput.nextfile`

close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count; the filename is not changed until after the first line of the next file has been read.

`fileinput.close`

closes the sequence.

All files are opened in text mode. If an I/O error occurs during opening or reading a file, the `IOError` exception is raised.

This makes it easy to write a Python version of the common Unix utility, **grep**. The **grep** utility searches a `list` of files for a given pattern.

Example 33.1. greppy.py

```
#!/usr/bin/env python
import sys, re, fileinput
pattern= re.compile( sys.argv[1] )
for line in fileinput.input(sys.argv[2:]):
    if pattern.match( line ):
        print fileinput.filename(), fileinput.filelineno(), line
```

This contains the essential features of the **grep**. For non-Unix users, the **grep** utility looks for the given regular expression in any number of files. The name grep is an acronym of Global Regular Expression Print.

The `re` module provides the pattern matching, and the `fileinput` module makes searching an arbitrary `list` of files simple. We cover the `re` module in more depth in Chapter 31, *Complex Strings: the `re` Module*.

The first command line argument (`sys.argv[0]`) is the name of the script, which this program ignores. This program uses the second command-line argument as the pattern that defines the target of the search. The remaining command-line arguments are given to `fileinput.input` so that all files will be examined. The pattern regular expression is matched against each individual input line. If `match` returns `None`, the line did not match. If `match` returns an object, the program prints the current file name, the current line number of the file and the actual input line that matched.

After we do a `chmod +x greppy.py`, we can use this program as follows. Note that we have to provide quotes to prevent the shell from doing globbing on our pattern `string`.

```
$ greppy.py 'import.*random' *.py
```

```
demorandom.py 2 import random

dice.py 1 import random

functions.py 2 import random
```

Windows users will be disappointed in attempting to use something like this for practical work. The GNU/Linux shell languages all handle file wild-card processing ("globbing") automatically. The shell uses the file-name patterns to create a complete list of file names that match the pattern to the application. Windows does not supply lists of file names that match patterns to application programs. Therefore, we have to use the `glob` module to transform `sys.argv[2:]` from a pattern to lists of files.

Also, Windows users will have to use `""` around the pattern, where Unix and Mac OS shell users will typically use `''`. This is a difference between the Unix shell quoting rules and the Windows quoting rules.

The `tempfile` Module

One common problem is to open a unique temporary file to hold intermediate results. There are two ways to do this: the `os` module and the `tempfile` module. The `os` module has a subtle security flaw, so it isn't recommended.

The `tempfile` module creates a temporary file in the most secure and reliable manner. The `tempfile` module includes an internal function, `mkstemp` which dioes the hard work of creating a unique temporary file.

`tempfile.TemporaryFile` ⟨*mode*⟩ ⟨*suffix*⟩ ⟨*prefix*⟩ ⟨*directory*⟩ → `file`

> This function creates a file which is automatically deleted when it is closed. All of the parameters are optional. By default, the mode is `'w+b'`, allowing updating. The keyword parameters *`suffix`*, *`prefix`* and *`directory`* provide some structure to the name assigned to the file. The *`suffix`* should include the dot, for example `suffix='.tmp'`.

`tempfile.NamedTemporaryFile` ⟨*mode*⟩ ⟨*suffix*⟩ ⟨*prefix*⟩ ⟨*directory*⟩ → `file`

> This function is similar to `TemporaryFile`; it creates a file which is automatically deleted when it is closed. The temporary file, however, is guaranteed to be visible on the file system while the file is open.

`tempfile.mkstemp` ⟨*suffix*⟩ ⟨*prefix*⟩ ⟨*directory*⟩ → *fd*, `file`

> This function does the essential job of creating a temporary file. It returns a file descriptor as well as the name of the file. The file is not automatically deleted. If necessary, the file created by this function can be explicitly deleted with `os.remove`.

```
import tempfile, os
fd, tempName = tempfile.mkstemp( '.d1' )
temp= open( tempName, 'w+' )

Some Processing...
```

This fragment will create a unique temporary file name with an extension of `.d1`. Since the name is guaranteed to be unique, this can be used without fear of damaging or overwriting any other file.

The `glob` and `fnmatch` Modules

The `glob` module adds a necessary Unix shell capability to Windows programmers. The `glob` module includes the following function

`glob.glob(`*`wildcard`*`)` → list

> Return a `list` of filenames that match the given wild-card pattern. The `fnmatch` module is used for the wild-card pattern matching.

A common use for `glob` is something like the following under Windows.

```
import glob, sys
for arg in sys.argv[1:]:
    for f in glob.glob(arg):
        process( f )
```

This makes Windows programs process command line arguments somewhat like Unix programs. Each argument is passed to `glob.glob` to expand any patterns into a `list` of matching files. If the argument is not a wild-card pattern, glob simply returns a `list` containing this one file name.

The `fnmatch` module has the algorithm for actually matching a wild-card pattern against specific file names. This module implements the Unix shell wild-card rules. These are not the same as the more sophisticated regular expression rules. The module contains the following function:

`fnmatch.fnmatch (`*`file`*`, `*`pattern`*`)` → boolean

> Return True if the file matches the pattern.

The patterns use `*` to match any number of characters, `?` to match any single character. `[letters]` matches any of these letters, and `[!letters]` matches any letter that is not in the given set of letters.

```
>>> import fnmatch
>>> fnmatch.fnmatch('greppy.py','*.py')
True
>>> fnmatch.fnmatch('README','*.py')
False
```

The `shutil` Module

The `shutil` module helps you automate copying files and directories. This saves the steps of opening, reading, writing and closing files when there is no actual processing, simply moving files.

`shutil.copy (`*`src`*`, `*`dest`*`)`

> Copy data and mode bits, basically the unix command `cp src dst`. If *`dest`* is a directory, a file with the same base name as *`src`* is created. If *`dest`* is a full file name, this is the destination file.

`shutil.copyfile (`*`src`*`, `*`dest`*`)`

> Copy data from *`src`* to *`dest`*. Both names must be files.

`shutil.copytree (`*`src`*`, `*`dest`*`)`

> Recursively copy the entire directory tree rooted at *`src`* to *`dest`*. *`dest`* must not already exist. Errors are reported to standard output.

`shutil.rmtree (`*`path`*`)`

Recursively delete a directory tree rooted at *path*.

Note that this allows us to build Python applications that are like shell scripts. There are a lot of advantages to writing Python programs rather than shell scripts to automate mundane tasks.

First, Python programs are easier to read than shell scripts. This is because the language did not evolve in way that emphasized tersness; the shell script languages use a minimum of punctuation, which make them hard to read.

Second, Python programs have a more sophisticated programming model, with class definitions, and numerous sophisticated data structures. The shell works with simple argument lists; it has to resort to running the **test** or **expr** programs to process `strings` or numbers.

Finally, Python programs have direct access to more of the operating system's features than the shell. Generally, many of the basic GNU/Linux API calls are provided via innumerable small programs. Rather than having the shell run a small program to make an API call, Python can simply make the API call.

The File Archive Modules: `tarfile` and `zipfile`

An archive file contains a complex, hierarchical file directory in a single sequential file. The archive file includes the original directory information as well as a the contents of all of the files in those directories. There are a number of archive file formats, Python directory supports two: tar and zip archives.

The tar (Tape Archive) format is widely used in the GNU/Linux world to distribute files. It is a POSIX standard, making it usable on a wide variety of operating systems. A tar file can also be compressed, often with the GZip utility, leading to `.tgz` or `.tar.gz` files which are compressed archives.

The Zip file format was invented by Phil Katz at PKWare as a way to archive a complex, hierarchical file directory into a compact sequential file. The Zip format is widely used but is not a POSIX standard. Zip file processing includes a choice of compression algorithms; the exact algorithm used is encoded in the header of the file, not in the name of file.

Creating a `TarFile` or a `ZipFile`. Since an archive file is still, essentially a file, it is opened with a variation on the `open` function. Since an archive file contains directory and file contents, it has a number of methods above and beyond what a simple file has.

`tarfile.open` ⟨*name*⟩ ⟨*mode*⟩ ⟨*fileobj*⟩ ⟨*buffersize*⟩ → `TarFile`

This module-level function opens the given tar file for processing. The *name* is a file name `string`; it is optional because the *fileobj* can be used instead. The *mode* is similar to the built-in `open` (or `file`) function; it has additional characters to specify the compression algorithms, if any. The *fileobject* is a conventional file object, which can be used instead of the *name*; it can be a standard file like `sys.stdin`. The *buffersize* is like the built-in `open` function.

`zipfile.(ZipFile`*name*, *mode*, *compression*)→ `ZipFile`

This class constructor opens the given zip file for processing. The *name* is a file name `string`. The mode is similar to the built-in `open` (or `file`) function. The *compression* is the compression code. It can be `zipfile.ZIP_STORED` or `zipfile.ZIP_DEFLATED`. A *compression* of `ZIP_STORED` uses no compression; a value of `ZIP_DEFLATED` uses the Zlib compression algorithms

The open function can be used to read or write the archive file. It can be used to process a simple disk file, using the filename. Or, more importantly, it can be used to process a

non-disk file: this includes tape devices and network sockets. In the non-disk case, a file object is given to tarfile.open.

For tar files, the mode information is rather complex because we can do more than simply read, write and append. The mode `string` adresses three issues: the kind of opening (reading, writing, appending), the kind of access (block or stream) and the kind of compression.

For zip files, however, the mode is simply the kind of opening that is done.

Opening - Both zip and tar files. A zip or tar file can be opened in any of three modes.

`r`

Open the file for reading.

`w`

Open the file for writing.

`a`

Open the file for appending.

Access - tar files only. A tar file can have either of two fundamentally different kinds of access. If a tar file is a disk file, which supports seek and tell operations, then you we access the tar file in block mode. If the tar file is a stream, network connection or a pipeline, which does not support seek or tell operations, then we must access the archive in stream mode.

`:`

Block mode. The tar file is an disk file, and seek and tell operations are supported. This is the assumed default, if neither `:` or `|` are specified.

`|`

Stream mode. The tar file is a stream, socket or pipeline, and cannot respond to seek or tell operations. Note that you cannot append to a stream, so the `'a|'` combination is illegal.

This access distinction isn't meaningful for zip files.

Compression - tar files only. A tar file may be compressed with GZip or BZip2 algorithms, or it may be uncompressed. Generally, you only need to select compression when writing. It doesn't make sense to attempt to select compression when appending to an existing file, or when reading a file.

(nothing)

The tar file will not be compressed.

`gz`

The tar file will be compressed with GZip.

`bz2`

The tar file will be compressed with BZip2.

This compression distinction isn't meaningful for zip files. Zip file compression is specified in the `zipfile.ZipFile` constructor.

Tar File Examples. The most common block modes for tar files are `r`, `a`, `w:`, `w:gz`, `w:bz2`. Note that read and append modes cannot meaningfully provide compression information, since it's obvious from the file if it was compressed, and which algorithm was used.

For stream modes, however, the compression information must be provided. The modes include all six combinations: `r|`, `r|gz`, `r|bz2`, `w|`, `w|gz`, `w|bz2`.

Directory Information. Each individual file in a tar archive is described with a `TarInfo` object. This has name, size, access mode, ownership and other OS information on the file. A number of methods will retrieve member information from an archive. In the following summaries, *`tf`* is a tar file, created with `tarfile.open`.

`tf``.getmember(`*`name`*`)` → `TarInfo`

> Reads through the archive index looking for the given member *`name`*. Returns a `TarInfo` object for the named member, or raises a `KeyError` exception.

`tf``.getmembers` → list of `TarInfo`

> Returns a `list` of `TarInfo` objects for all of the members in the archive.

`tf``.next` → `TarInfo`

> Returns a `TarInfo` object for the next member of the archive.

`tf``.getnames` → list of strings

> Returns a `list` of member names.

Each individual file in a zip archive is described with a `ZipInfo` object. This has name, size, access mode, ownership and other OS information on the file. A number of methods will retrieve member information from an archive. In the following summaries, *`zf`* is a zip file, created with `zipfile.ZipFile`.

`zf``.,` `(getinfo`*`name`*`)` → `ZipInfo`

> Locates information about the given member *`name`*. Returns a `ZipInfo` object for the named member, or raises a `KeyError` exception.

`zf``.,` `(infolist)` → list of `ZipInfo`

> Returns a `list` of `ZipInfo` objects for all of the members in the archive.

`zf``.namelist` → list of strings

> Returns a `list` of member names.

Extracting Files From an Archive. If a tar archive is opened with `r`, then you can read the archive and extract files from it. The following methods will extract member files from an archive. In these summaries, *`tf`* is a tar file, created with `tarfile.open`.

`tf``.extract(`*`member`*`,` ⟨*`path`*⟩`)`

> The *`member`* can be either a `string` member name or a `TarInfo` for a member. This will extract the file's contents and reconstruct the original file. If *`path`* is given, this is the new location for the file.

`tf``.extractfile(`*`member`*`)` → `file`

> The *`member`* can be either a `string` member name or a `TarInfo` for a member. This will open a simple file for access to this member's contents. The member

access file has only read-oriented methods, limited to `read`, `readline`, `readlines`, `seek`, `tell`.

If a zip archive is opened with r, then you can read the archive and extract the contents of a file from it. In these summaries, *`zf`* is a zip file, created with `zipfile.ZipFile`.

`zf``.read(`*`member`*`)` → `string`

The *`member`* is a `string` member name. This will extract the member's contents, decompress them if necessary, and return the bytes that consitute the member.

Creating or Extending an Archive. If a tar archive is opened with `w` or `a`, then you can add files to it. The following methods will add member files to an archive. In the following summaries, *`tf`* is a tar file, created with `tarfile.open`.

`tf``.add(`*`name`*`,` ⟨*`arcname`*⟩⟨*`recursive`*⟩`)`

Adds the file with the given *`name`* to the current archive file. If *`arcname`* is provided, this is the name the file will have in the archive; this allows you to build an archive which doesn't reflect the source structure. Generally, directories are expanded; using `recursive=False` prevents expanding directories.

`tf``.addfile(`*`tarinfo`*`,` *`fileobj`*`)`

Creates an entry in the archive. The description comes from the *`tarinfo`*, an instance of `TarInfo`, created with the `gettarinfo` function. The *`fileobj`* is an open file, from which the content is read. Note that the `TarInfo.size` field can override the actual size of the file. For a given filename, `fn`, this might look like the following: `tf.addfile( tf.gettarinfo(fn), open(fn,"r") )`.

`tf``.`(close)

Closes the archive. For archives being written or appended, this adds the block of zeroes that defines the end of the file.

`tf``.gettarinfo(`*`name`*`,` ⟨*`arcname`*⟩ `,` ⟨*`fileobj`*⟩ `)` → `TarInfo`

Creates a `TarInfo` object for a file based either on *`name`*, or the *`fileobj`*. If a *`name`* is given, this is a local filename. The *`arcname`* is the name that will be used in the archive, allowing you to modify local filesystem names. If the *`fileobj`* is given, this file is interrogated to gather required information.

If a zip archive is opened with `w` or `a`, then you can add files to it. The following methods will add member files to an archive. In the following summaries, *`zf`* is a zip file, created with `zipfile.ZipFile`.

`zf``.write(`*`filename`*`,` ⟨*`arcname`*⟩ `,` ⟨*`compress`*⟩ `)` → `string`

The *`filename`* is a `string` file name. This will read the file, compress it, and write it to the archive. If the *`arcname`* is given, this will be the name in the archive; otherwise it will use the original *`filename`*. The *`compress`* parameter overrides the default compression specified when the `ZipFile` was created.

`zf``.writestr(`*`arcname`*`,` *`bytes`*`)` → `string`

The *`arcname`* is a `string` file name or a `ZipInfo` object that will be used to create a new member in the archive. This will write the given bytes to the archive. The compression used is specified when the `ZipFile` is created.

A `tarfile` Example. Here's an example of a program to examine a tarfile, looking for documentation like `.html` files or `README` files. It will provide a list of `.html` files, and

actually show the contents of the README files.

Example 33.2. readtar.py

```
#!/usr/bin/env python
"""Scan a tarfile looking for *.html and a README."""
import tarfile
import fnmatch

archive= tarfile.open( "SQLAlchemy-0.3.5.tar.gz", "r" )
for mem in archive.getmembers():
    if fnmatch.fnmatch( mem.name, "*.html" ):
        print mem.name
    elif fnmatch.fnmatch( mem.name.upper(), "*README*" ):
        print mem.name
        docFile= archive.extractfile( mem )
        print docFile.read()
```

A zipfile Example. Here's an example of a program to create a zipfile based on the .xml files in a particular directory.

Example 33.3. writezip.py

```
import zipfile, os, fnmatch

bookDistro= zipfile.ZipFile( 'book.zip', 'w', zipfile.ZIP_DEFLATED )
for nm in os.listdir('..'):
    if fnmatch.fnmatch(nm,'*.xml'):
        full= os.path.join( '..', nm )
        bookDistro.write( full )
bookDistro.close()
```

The Data Compression Modules: zlib, gzip, bz2

The zlib, gzip and bz2 modules provide essential data and file compression tools. Data files are often built for speedy processing, and may contain characters which are meaningless spacing. This extraneous data can be reduced in size, or compressed.

For example, a .tar file is often compressed using GZip to create a .tar.gz file, sometimes called a .tgz file. In the case of the ZIP file archive, the compression algorithms are already part of the zipfile module.

These modules are very flexible and can be used in a variety of ways by an application program. The interface to zlib and bz2 are very similar. The interface to gzip is greatly simplified.

Data Compression and Decompression. The zlib and bz2 modules provide essential data compression and decompression algorithms. Each module provides a compress and decompress function which will compress a string into a sequence of bytes. For relatively short strings, there may be no reduction in size.

Note that the interfaces for the zlib and bz2 modules are designed to be nearly identical. The results, however, are not compatible at all; these are different algorithms for data compression.

zlib.compress(*string*, ⟨*level*⟩) → string

> Compress the given *string*. The *level* provides a speed vs. size tradeoff: a value of 1 is very fast, but may not compress very well; a value of 9 will be slower but provide the best compression. The default value for *level* is 6.

`zlib.decompress(`*`string`* → *`string`*`)`

Decompress the given *`string`*.

`bz2.compress → string`

Compress the given *`string`*. The *`level`* provides a speed vs. size tradeoff: a value of 1 is very fast, but may not compress very well; a value of 9 will be slower but provide the best compression. The default value for *`level`* is 9.

`bz2.decompress → string`

Decompress the given *`string`*.

In addition to simple compression and decompression, `zlib` provides some additional features for computing various kinds of checksums of the data.

`zlib.adler32(`*`string`*`) → number`

Compute the 32-bit Adler checksum of the given *`string`*.

`zlib.crc32(`*`string`* → *`number`*`)`

Compute the 32-bit CRC (Cyclic Redundancy Check) checksum the given *`string`*.

Simple File Compression and Decompression. The `zlib` and `bz2` modules provide functions for compression and decompression of files or streams. Each module provides a way to create a compressor object, as well as process a file while compressing or decompressing it.

The following methods will create a compressor object or a decompressor object. A compressor objects has methods which will compress a sequence of bytes incrementally. A decompressor object, conversely, has methods to uncomress a sequence of bytes. These objects can work with chunks of data rather than the complete contents of a file, making it possible to compress or uncompress large amounts of data.

`zlib.compressobj ⟨`*`level`*`⟩ → Compress`

Creates a ew compressor object, an instance of class `Compress`. A compressor object has two methods that can be used to incrementally compress a sequence of bytes. The *`level`* is the integer compression level, between 1 and 9. In the following summaries, *`c`* is a compressor.

`c``.compress(`*`bytes`*`)` → bytes

Compresses the given sequence of bytes; returns the next sequence of compressed bytes. The idea is to feed blocks of data into the compressor, getting blocks of compressed data out. Some data is retained as part of the compression, so `flush` must be called to finalize the compression and get the last of the bytes.

`c``.flush ⟨`*`mode`*`⟩ →` bytes

Finishes compression processing, returning the remaining bytes. While there are three values for *`mode`*, offering some data recovery capability, the default is `zlib.Z_FINISH`, which completely finishes all compression. Providing no *`mode`* is compatible with the `BZ2Compressor`.

`zlib.decompressobj → Decompress`

Creates a new decompressor object, an instance of class `Decompress`. A

decompressor object has two methods that can be used to decompress a sequence of bytes.

d., (decompress*bytes*) → bytes

Decompresses the given sequence of bytes; returns the next sequence of uncompressed bytes. The idea is to feed blocks of compressed data into the decompressor, getting blocks of uncompressed data out. Some data is retained as part of the compression, so `flush` must be called to finalize the decompression and get the last of the bytes.

d.(flush) → bytes

Finishes decompression processing, returning the remaining bytes.

`bz2.BZ2Compressor`*level*

Creates a new compressor object, an instance of class `BZ2Compressor`. A compressor object has two methods that can be used to incrementally compress a sequence of bytes. The *level* is the integer compression level, between 1 and 9. A `bz2.BZ2Compressor` has the same methods as a `zlib.Compress`, described above.

`bz2.BZ2Decompressor`

Creates a new decompressor object, an instance of class `BZ2Decompressor`. A decompressor object has two methods that can be used to decompress a sequence of bytes. A `bz2.BZ2Decompressor` has the same methods as a `zlib.Decompress`, described above.

Example 33.4. compdecomp.py

```
#/usr/bin/env python
"""Compress a file using a compressor object."""
import zlib, bz2, os

def compDecomp( compObj, srcName, dstName ):
    source= file( srcName, "r" )
    dest= file( dstName, "w" )
    block= source.read( 2048 )
    while block:
        cBlock= compObj.compress( block )
        dest.write(cBlock)
        block= source.read( 2048 )
    cBlock= compObj.flush()
    dest.write( cBlock )
    source.close()
    dest.close()

compObj1= zlib.compressobj()
compDecomp( compObj1, "../python.xml", "python.xml.gz" )
print "source", os.stat("../python.xml").st_size/1024, "k"
print "dest", os.stat("python.xml.gz").st_size/1024, "k"

compObj2= bz2.BZ2Compressor()
compDecomp( compObj2, "../python.xml", "python.xml.bz" )
print "source", os.stat("../python.xml").st_size/1024, "k"
print "dest", os.stat("python.xml.bz").st_size/1024, "k"
```

❶ We define a function, `compdecomp`, which applies a compressor object to a source file and produces a destination file. This reads the file in small blocks of 2048 bytes, compresses each block, and writes the compressed block to the destination file.

The final block will generally be less than 2048 bytes. The final call to

`compObj.flush` notifies the compressor object that there will be no more data; the value returned is the tail end of the compressed data.

❷ We create a compressor, `compObj1`, from the `zlib.compressobj` function. This object is used by our `compDecomp` function to compress a sample file. In this case, the sample file is 1,208K, the resulting file is 301K, about ¼ the original size.

❸ We create a compressor, `compObj2`, from the `bz2.BZ2Compressor` class. This object is used by our `compDecomp` function to compress a sample file. The resulting file is 228K, about ⅕ the original size.

Decompression uses a function nearly identical to the `compDecomp` function. The decompress version calls the `decompress` method of a `decompObj` instead of the `compress` method of a `compObj`.

Gzip File Handling. The `gzip` module provides function and class definitions that make it easy to handle simple Gzip files. These allow you to open a compressed file and read it as if it were already decompressed. They also allow you to open a file and write to it, having the data automatically compressed as you write.

`gzip.open(`*`filename`*`,` *`mode`*`,` *`level`*`,` *`fileobj`*`)` → `gzip.GzipFile`

> Open the file named *`filename`* with the given *`mode`* (`'r'`, `'w'` or `'a'`). The compression *`level`* is an integer that provides a preference for speed versus size. As an alternative, you can open a file or socket separately and provide a *`fileobj`*, for example, `f= open('somefile','r'); zf= gzip.open( fileobj=f )`.

Once this file is open, it can perform ordinary `read`, `readline`, `readlines`, `write`, `writeline`, and `writelines` operations. If you open the file with `'rb'` mode, the various read functions will decompress the file's contents as the file is read. If you open the file with `'wb'` or `'ab'` modes, the various write functions will compress the data as they write to the file.

The `sys` Module

The `sys` module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

The `sys` module also provides the three standard files used by Python.

`sys.stdin`

> Standard input file object; used by `raw_input` and `input`. Also available via `sys.stdin.read()` and related methods of the file object.

`sys.stdout`

> Standard output file object; used by the **print** statement. Also available via `sys.stdout.write()` and related methods of the file object.

`sys.stderr`

> Standard error object; used for error messages, typically unhandled exceptions. Available via `sys.stderr.write()` and related methods of the file object.

A program can assign another file object to one of these global variables. When you change the file for these globals, this will redirect all of the interpreter's I/O.

One important object made available by this module is the variable `sys.argv`. This variable has the command line arguments used to run this script. For example, if we had a python script called portfolio.py, and executed it with the following command:

```
python portfolio.py -xvb display.csv
```

Then the `sys.argv` `list` would be `["portfolio.py", "-xvb", "display.csv"]`. Sophisticated argument processing is done with the `getopt` or `optparse` modules.

A few other interesting objects in the `sys` module are the following variables.

`sys.version`

The version of this interpreter as a `string`. For example, `'2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)]'`

`sys.version_info`

Version information as a `tuple`, for example: `(2, 5, 1, 'final', 0)`.

`sys.hexversion`

Version information encoded as a single integer. Evaluating `hex(sys.hexversion)` yields `'0x20501f0'`.

`sys.copyright`

Copyright notice pertaining to this interpreter.

`sys.platform`

Platform identifier, for example, `'darwin'`, `'win32'` or `'linux2'`.

`sys.prefix`

Prefix used to find the Python library, for example `'/usr'`, `'/Library/Frameworks/Python.framework/Versions/2.5'`, `'c:\\Python25'`.

Additional File-Processing Modules

There are several other chapters of the Python Library Reference that cover with even more file formats. We'll identify them briefly here.

Chapter 7 - Internet Data Handling. Reading and processing files of Internet data types is very common. Internet data types have formal definitions governed by the internet standards, called Requests for Comment (RFC's). The following modules are for handling Internet data structures. These modules and the related standards are beyond the scope of this book.

email

Helps you handle email MIME attachments.

mailcap

Mailcap file handling.

mailbox

Read various mailbox formats.

mhlib

Manipulate MH mailboxes from Python.

mimetools

Tools for parsing MIME-style message bodies.

mimetypes

Mapping of filename extensions to MIME types.

MimeWriter

Generic MIME file writer.

mimify

Mimification and unmimification of mail messages.

multifile

Support for reading files which contain distinct parts, such as some MIME data.

rfc822

Parse RFC 822 style mail headers.

base64

Encode and decode files using the MIME base64 data.

binhex

Encode and decode files in binhex4 format.

binascii

Tools for converting between binary and various ASCII-encoded binary representations.

quopri

Encode and decode files using the MIME quoted-printable encoding.

uu

Encode and decode files in uuencode format.

Chapter 13 - Data Persistence. Many Python programs will also deal with Python objects that are exported from memory to external files or retrieved from files to memory. Since an external file is more persistent than the volatile working memory of a computer, this process makes an object persistent or retrieves a persistent object. One mechanism for creating a persistent object is called serialization, and is supported by several modules, which are beyond the scope of this book.

pickle

Convert Python objects to streams of bytes and back.

cPickle

Faster version of pickle, but not subclassable.

copy_reg

Register pickle support functions.

shelve

Python object persistence.

marshal

Convert Python objects to streams of bytes and back (with different constraints).

More complex file structures can be processed using the standard modules available with Python. The widely-used DBM database manager is available, plus additional modules are available on the web to provide ODBC access or to connect to a platform-specific database access routine. The following Python modules deal with these kinds of files. These modules are beyond the scope of this book.

anydbm

Generic interface to DBM-style database modules.

whichdb

Guess which DBM-style module created a given database.

dbm

The standard database interface, based on the ndbm library.

gdbm

GNU's reinterpretation of dbm.

dbhash

DBM-style interface to the BSD database library.

bsddb

Interface to Berkeley DB database library

dumbdbm

Portable implementation of the simple DBM interface.

sqlite3

A very pleasant, easy-to-use relational database (RDBMS).

File Module Exercises

1. **Source Lines of Code.** One measure of the complexity of an application is the count of the number of lines of source code. Often, this count discards comment lines. We'll write an application to read Python source files, discarding blank lines and lines beginning with `#`, and producing a count of source lines.

   We'll develop a function to process a single file. We'll use the `glob` module to locate all of the `*.py` files in a given directory.

   Develop a `fileLineCount(`*`name`*`)` which opens a file with the given *`name`* and examines all of the lines of the file. Each line should have `strip` applied to remove leading and trailing spaces. If the resulting line is of length zero, it was effectively blank, and can be skipped. If the resulting line begins with `#` the line is entirely a comment, and can be skipped. All remaining lines should be counted, and `fileLineCount(`*`name`*`)` returns this count.

Develop a `directoryLineCount(`*`path`*`)` function which uses the path with the `glob.glob` to expand all matching file names. Each file name is processed with `fileLineCount(`*`name`*`)` to get the number of non-comment source lines. Write this to a tab-delimited file; each line should have the form "*`filename \t lines`*".

For a sample application, look in your Python distribution for `Lib/idelib/*.py`.

2. **Summarize a Tab-Delimited File.** The previous exercise produced a file where each line has the form "*`filename \t lines`*". Read this tab-delimited file, producing a nicer-looking report that has column titles, file and line counts, and a total line count at the end of the report.

3. **File Processing Pipeline.** The previous two exercises produced programs which can be part of a processing pipeline. The first exercise should p should produce it's output on `sys.stdout`. The second exercise should gather it's input from `sys.stdin`. Once this capability is in place, the pipeline can be invoked using a command like the following:

```
$ python lineCounter.py | python lineSummary.py
```

Chapter 34. File Formats: CSV, Tab, XML, Logs and Others

Table of Contents

We looked at general features of the file system in Chapter 19, *Files*. In this chapter we'll look at Python techniques for handling files in a few of the innumeraable formats that are in common use. Most file formats are relatively easy to handle with Python techniques we've already seen. Comma-Separated Values (CSV) files, XML files and packed binary files, however, are a little more sophisticated.

This only the tip of the iceberg in the far larger problem called "persistence". In addition to simple file system persistence, we also have the possibility of object persistence using an object database. In this case, the databse processing lies between our program and the file system on which the database resides. This area also includes object-relational mapping, where our program relies on a mapper; the mapper uses to database, and the database manages the file system. We can't explore the whole persistence problem in this chapter.

In this chapter we'll present a conceptual overview of the various approaches to reading and writing files in the section called "Overview". We'll look at reading and writing CSV files in the section called "Comma-Separated Values: The `csv` Module", tab-delimited files in the section called "Tab Files: Nothing Special". We'll look reading property files in the section called "Property Files and Configuration (or `.INI`) Files: The `ConfigParser` Module". We'll look at the subleties of processing legacy COBOL files

in the section called "Fixed Format Files, A COBOL Legacy: The `codecs` Module". We'll cover the basics of reading XML files in the section called "XML Files: The `xml.minidom` and `xml.sax` Modules".

Most programs need a way to write sophisticated, easy-to-control log files what contain status and debugging information. For simple one-page programs, the **print** statement is fine. As soon as we have multiple modules, where we need more sophisticated debugging, we find a need for the `logging` module. Of course, any program that requires careful auditing will benefit from the `logging` module. We'll look at creating standard logs in the section called "Log Files: The `logging` Module".

Overview

When we introduced the concept of file we mentioned that we could look at a file on two levels.

- A file is a sequence of bytes. This is the OS's view of views, as it is the lowest-common denominator.
- A file is a sequence of data objects, represented as sequences of bytes.

A *file format* is the processing rules required to translate between usable Python objects and sequences of bytes. People have invented innumerable distinct file formats. We'll look at some techniques which should cover most of the bases.

We'll look at three broad families of files: text, binary and pickled objects. Each has some advantages and processing complexities.

- Text files are designed so that a person can easily read and write them. We'll look at several common text file formats, including CSV, XML, Tab-delimited, property-format, and fixed position. Since text files are intended for human consumption, they are difficult to update in place.
- Binary files are designed to optimize processing speed or the overall size of the file. Most databases use very complex binary file formats for speed. A JPEG file, on the other hand, uses a binary format to minimize the size of the file. A binary-format file will typically place data at known offsets, making it possible to do direct access to any particular byte using the `seek` method of a Python file object.
- Pickled Objects are produced by Python's `pickle` or `shelve` modules. There are several pickle protocols available, including text and binary alternatives. More importantly, a pickled file is not designed to be seen by people, nor have we spent any design effort optimizng performace or size. In a sense, a pickled object requires the least design effort.

Comma-Separated Values: The `csv` Module

Often, we have data that is in Comma-Separated Value (CSV) format. This used by many spreadsheets and is a widely-used standard for data files.

In the section called "Several Examples" we parsed CSV files using simple `string` manipulations. The `csv` module does a far better job at parsing and creating CSV files than the programming we showed in those examples.

About CSV Files

CSV files are text files organized around data that has rows and columns. This format is used to exchange data between spread-sheet programs or databases. A CSV file uses a number of punctuation rules to encode the data.

- Each row is delimited by a line-ending sequence of characters. This is usually the

ASCII sequence `\r\n`. Since this may not be the default way to process text files on your platform, you have to open files using the "rb" and "wb" modes.

- Within a row, columns are delimited by a `,`. To handle the situation where a column's data contains a `,`, the column data may be quoted; surrounded by `"`'s. If the column contains a ", there are two common rules used. One CSV dialect uses an escape character, usually `\` for `\"`. The other dialect uses double `""`'s.

In the ideal case, a CSV file will have the same number of columns in each row, and the first row will be column titles. Almost as pleasant is a file without column titles, but with a known sequence of columns. In the more complex cases, the number of columns per row varies.

The CSV Module

The CSV module provides you with readers or writers; these are objects which use an existing `file` object, created with the `file` or `open` function. A CSV reader will read a file, parsing the commas and quotes, delivering you the data elements of each row in a sequence or mapping. A CSV writer will create a file, adding the necessary commas and quotes to create a valid CSV file.

Module-Level Constructors. The following constructors within the `csv` module are used to create a `reader`, `DictReader`, `writer` or `DictWriter`.

`csv.reader(`*`csvfile`*`)` → `reader`

Creates a `reader` object which can parse the given file, returning a sequence of values for each line of the file. This can be used as follows: `rdr= csv.reader( open( "file.csv", "rb" ) )`. The `csvfile` can be any iterable object.

`csv.writer(`*`csvfile`*`)` → `writer`

Creates a `writer` object which can format a sequence of values and write them to a line of the file. This can be used as follows: `wtr= csv.writer( open( "file.csv", "wb" ) )`. The `csvfile` can be any object which supports `write`.

`csv.DictReader(`*`csvfile`*`,` ⟨*`fieldnames`*⟩`)` → `DictReader`

Creates a `DictReader` object which can parse the given file, returning a dictionary of values for each line of the file. The dictionary keys are typically the first line of the file. You can, optionally, provide the field names if they are not the first line of the file. The `csvfile` can be any iterable object.

`csv.DictWriter(`*`csvfile`*`,` *`fieldnames`*`)` → `DictWriter`

Creates a `DictWriter` object which can format a dictionary of values and write them to a line of the file. You must provide a sequence of field names which is used to format each individual dictionary entry. The `csvfile` can be any object which supports `write`.

Reader Functions. The following functions within a `reader` (or `DictReader`) object will read and parse the CSV file. In these function definitions `cr` is a `csv.reader` or `csv.DictReader`.

`cr``.(next)` → `sequence`

Reads the next line of the source file, parses it, and returns a sequence (for `reader`) or dictionary (for `DictReader`) of the individual column values.

`cr``.line_num` → `number`

Returns the line number of the source file.

Writer Functions. The following functions with a `writer` (or `DictWriter`) object will format and write a CSV file. In these function definitions `cw` is a `csv.writer` or `csv.DictWriter`.

cw.(writerow*row*)

Writes the next lines of the destination file from the given sequence (for `writer`) or dictionary (for `DictWriter`).

cw., (writerows*rowList*)

Writes the next lines of the destination file with each sequence (for `writer`) or dictionary (for `DictWriter`) from the list, `rowList`.

Basic CSV Reading

The basic CSV reader processing treats each line of the file as data. This is typical for files which lack column titles, or files which have such a complex format that special parsing and analysis is required. In some cases, a file has a simple, regular format with a single row of column titles, which can be processed by a special reader we'll look at below.

We'll revise the `readquotes.py` program from the section called "Reading a File as a Sequence of Strings". This will properly handle all of the quoting rules, eliminating a number of irritating problems with the example in the previous chapter.

Example 34.1. readquotes2.py

```
import csv
qFile= file( "quotes.csv", "rb" )
csvReader= csv.reader( qFile )
for q in csvReader:
    try:
        stock, price, date, time, change, opPrc, dHi, dLo, vol = q
        print stock, float(price), date, time, change, vol
    except ValueError:
        pass
qFile.close()
```

❶ We open our quotes file, `quotes.csv`, for reading, creating an object named `qFile`.

❷ We create a `csv` reader object which will parse this file for us, transforming each line into a sequence of individual column values.

❸ We use a **for** statement to iterate through the sequence of lines in the file.

❹ In the unlikely event of an invalid number for the price, we surround this with a **try** statement. The invalid number line will raise a `ValueError` exception, which is caught in the **except** clause and quietly ignored.

❺ Each stock quote, `q`, is a sequence of column values. We use multiple assignment to assign each field to a relevant variable. We don't need to strip whitespace, split the string, or handle quotes; the reader already did this.

❻ Since the price is a string, we use the `float` function to convert this string to a proper numeric value for further processing.

Consistent Columns as Dictionaries

In some cases, you have a simple, regular file with a single line of column titles. In this case, you can transform each line of the file into a dictionary. The key for each field is the column title. This can lead to programs which are more clear, and more flexible. The flexibility comes from not assuming a specific order to the columns.

We'll revise the `readportfolio.py` program from the section called "Reading "Records"". This will properly handle all of the quoting rules, eliminating a number of irritating problems with the example in the previous chapter. It will make use of the column titles in the file.

Example 34.2. readportfolio2.py

```
import csv
quotes=open( "display.csv", "rb" )
csvReader= csv.DictReader( quotes )
invest= 0
current= 0
for data in csvReader:
    print data
    invest += float(data["Purchase Price"])*float(data["# Shares"])
    current += float(data["Price"])*float(data["# Shares"])
print invest, current, (current-invest)/invest
```

❶ We open our portfolio file, `display.csv`, for reading, creating a file object named `quotes`.

❷ We create a `csv DictReader` object from our `quotes` file. This will read the first line of the file to get the column titles; each subsequent line will be parsed and transformed into a dictionary.

❸ We initialize two counters, `invest` and `current` to zero. These will accumulate our initial investment and the current value of this portfolio.

❹ We use a **for** statement to iterate through the lines in `quotes` file. Each line is parsed, and the column titles are used to create a dictionary, which is assigned to `data`.

❺ Each stock quote, `q`, is a `string`. We use the `strip` operation to remove excess whitespace characters; the `string` which is created then performs the `split ( ',' )` operation to separate the fields into a `list`. We assign this `list` to the variable `values`.

❺ We perform some simple calculations on each `dict`. In this case, we convert the purchase price to a number, convert the number of shares to a number and multiply to determine how much we spent on this stock. We accumulate the sum of these products into `invest`.

We also convert the current price to a number and multiply this by the number of shares to get the current value of this stock. We accumulate the sum of these products into `current`.

❻ When the loop has terminated, we can write out the two numbers, and compute the percent change.

Writing CSV Files

The most general case for writing CSV is shown in the following example. Assume we've got a list of objects, named `someList`. Further, let's assume that each object has three attributes: `this`, `that` and `aKey`.

```
import csv
myFile= open( "result", "wb" )
wtr= csv.writer( myFile )
for someObject in someList:
    aRow= [ someData.this, someData.that, someData.aKey, ]
    wtr.writerow( aRow )
myFile.close()
```

In this case, we assemble the list of values that becomes a row in the CSV file.

In some cases we can provide two methods to allow our classes to participate in CSV writing. We can define a `csvRow` method as well as a `csvHeading` method. These

methods will provide the necessary tuples of heading or data to be written to the CSV file.

For example, let's look at the following class definition for a small database of sailboats. This class shows how the `csvRow` and `csvHeading` methods might look.

```
class Boat( object ):
    csvHeading= [ "name", "rig", "sails" ]
    def __init__( aBoat, name, rig, sails ):
        self.name= name
        self.rig= rig
        self.sails= sails
    def __str__( self ):
        return "%s (%s, %r)" % ( self.name, self.rig, self.sails )
    def csvRow( self ):
        return [ self.name, self.rig, self.sails ]
```

Including these methods in our class definitions simplifies the loop that writes the objects to a CSV file. Instead of building each row as a list, we can do the following: `wtr.writerow( someData.csvRow() )`.

Here's an example that leverages each object's internal dictionary (`__dict__`) to dump objects to a CSV file.

```
db= [
    Boat( "KaDiMa", "sloop", ( "main", "jib" ) ),
    Boat( "Glinda", "sloop", ( "main", "jib", "spinnaker" ) ),
    Boat( "Eillean Glas", "sloop", ( "main", "genoa" ) ),
    ]

test= file( "boats.csv", "wb" )
wtr= csv.DictWriter( test, Boat.csvHeading )
wtr.writerow( dict( zip( Boat.csvHeading, Boat.csvHeading ) ) )
for d in db:
    wtr.writerow( d.__dict__ )
test.close()
```

Tab Files: Nothing Special

Tab-delimited files are text files organized around data that has rows and columns. This format is used to exchange data between spread-sheet programs or databases. A tab-delimited file uses just rwo punctuation rules to encode the data.

- Each row is delimited by an ordinary newline character. This is usually the standard `\n`. If you are exchanging files across platforms, you may need to open files for reading using the "rU" mode to get universal newline handling.
- Within a row, columns are delimited by a single character, often `\t`. The column punctuation character that is chosen is one that will never occur in the data. It is usually (but not always) an unprintable character like `\t`.

In the ideal cases, a CSV file will have the same number of columns in each row, and the first row will be column titles. Almost as pleasant is a file without column titles, but with a known sequence of columns. In the more complex cases, the number of columns per row varies.

When we have a single, standard punctuation mark, we can simply use two operations in the `string` and `list` classes to process files. We use the `split` method of a `string` to parse the rows. We use the `join` method of a `list` to assemble the rows.

We don't actually need a separate module to handle tab-delimited files. We looked at a related example in the section called “Reading a Text File”.

Reading. The most general case for reading Tab-delimited data is shown in the following example.

```
myFile= open( "somefile", "rU" )
for aRow in myFile:
    print aRow.split('\t')
myFile.close()
```

Each row will be a `list` of column values.

Writing. The writing case is the inverse of the reading case. Essentially, we use a `"\t".join( someList )` to create the tab-delimeted row. Here's our sailboat example, done as tab-delimited data.

```
test= file( "boats.tab", "w" )
test.write( "\t".join( Boat.csvHeading ) )
test.write( "\n" )
for d in db:
    test.write( "\t".join( map( str, d.csvRow() ) ) )
    test.write( "\n" )
test.close()
```

Note that some elements of our data objects aren't string values. In this case, the value for sails is a tuple, which needs to be converted to a proper string. The expression `map(str, someList)` applies the `str` function to each element of the original list, creating a new list which will have all string values. See the section called "Sequence Processing Functions: `map`, `filter`, `reduce` and `zip`".

Property Files and Configuration (or `.INI`) Files: The `ConfigParser` Module

A property file, also known as a configuration (or `.INI`) file defines property or configuration values. It is usually just a collection of settings. The essential property-file format has a simple row-oriented format with only two values in each row. A configuration (or `.INI`) file organizes a simple list of properties into one or more named sections.

A property file uses a few punctuation rules to encode the data.

- Lines begining with `#` or `;` are ignored. In some dialects the comments are `#` and `!`.
- Each property setting is delimited by an ordinary newline character. This is usually the standard `\n`. If you are exchanging files across platforms, you may need to open files for reading using the "rU" mode to get universal newline handling.
- Each property is a simple name and a value. The name is a string characters that does not use a separator character of `:` or `=`. The value is everything after the punctuation mark, with leading and trailing spaces removed. In some dialects space is also a separator character.

Some property file dialects allow a value to continue on to the next line. In this case, a line that ends with `\` (the cwo-character sequence `\ \n`) escapes the usual meaning of `\n`. Rather being the end of a line, `\\n` is just another whitespace character.

A property file is an extension to the basic tab-delimited file. It has just two columns per line, and some space-stripping is done. However, it doesn't have a consistent separator, so it is slightly more complex to parse.

The extra feature introduced in a configuration file is named sections.

- A line beginning with `[`, ending with `]`, is the beginning of a section. The `[]`'s

surround the section name. All of the lines from here to the next section header are collected together.

Reading a Simple Property File. Here's an example of reading the simplest kind of property file. In this case, we'll turn the entire file into a dictionary. Python doesn't provide a module for doing this. The processing is a sequence string manipulations to parse the file.

```
propFile= file( r"C:\Java\jdk1.5.0_06\jre\lib\logging.properties", "rU" )
propDict= dict()
for propLine in propFile:
    propDef= propLine.strip()
    if len(propDef) == 0:
        continue
    if propDef[0] in ( '!', '#' ):
        continue
    punctuation= [ propDef.find(c) for c in ':= ' ] + [ len(propDef) ]
    found= min( [ pos for pos in punctuation if pos != -1 ] )
    name= propDef[:found].rstrip()
    value= propDef[found:].lstrip(":= ").rstrip()
    propDict[name]= value
propFile.close()
print propDict
print propDict['handlers']
```

The input line is subject to a number of processing steps.

1. First the leading and trailing whitespace is removed. If the line is empty, nothing more needs to be done.

2. If the line begins with `!` or `#` (`;` in some dialects) it is ignored.

3. We find the location of all relevant puncuation marks. In some dialects, space is not permitted. Note that we through the length of the line on the end to permit a single word to be a valid property name, with an implicit value of a zero-length string.

4. By discarding punction positions of -1, we are only processing the positions of punctuation marks which actually occur in the string. The smallest of these positions is the left-most punctuation mark.

5. The name is everything before the punctuation mark with whitespace remove.

6. The value is everything after the punctuaion mark. Any additional separators are removed, and any trailing whitespace is also removed.

Reading a Config File. The `ConfigParser` module has a number of classes for processing configuration files. You initialize a ConfigParse object with default values. The object can the read one or more a configuration files. You can then use methods to determine what sections were present and what options were defined in a given section.

```
import ConfigParser
cp= ConfigParser.RawConfigParser( )
cp.read( r"C:\Program Files\Mozilla Firefox\updater.ini" )
print cp.sections()
print cp.options('Strings')
print cp.get('Strings','info')
```

Eschewing Obfuscation. While a property file is rather simple, it is possible to simplify property files further. The essential property definition syntax is so close to Python's own syntax that some applications use a simple file of Python variable settings. In this case, the settings file would look like this.

Example 34.3. settings.py

```
# Some Properties
TITLE = "The Title String"
INFO = """The information string.
Which uses Python's ordinary techniques
for long lines."""
```

This file can be introduced in your program with one statement: `import settings`. This statement will create module-level variables, `settings.TITLE` and `settings.INFO`.

Fixed Format Files, A COBOL Legacy: The `codecs` Module

Files that come from COBOL programs have three characteristic features:

- The file layout is defined positionally. There are no delimiters or separators on which to base file parsing. The file may not even have `\n` characters at the end of each record.
- They're usually encoded in EBCDIC, not ASCII or Unicode.
- They may include packed decimal fields; these are numeric values represented with two decimal digits (or a decimal digit and a sign) in each byte of the field.

The first problem requires figuring the starting position and size of each field. In some cases, there are no gaps (or filler) between fields; in this case the sizes of each field are all that are required. Once we have the position and size, however, we can use a string slice operation to pick those characters out of a record. The code is simply `aLine[start:start+size]`.

We can tackle the second problem using the `codecs` module to decode the EBCDIC characters. The result of `codecs.getdecoder('cp037')` is a function that you can use as an EBCDIC decoder.

The third problem requires that our program know the data type as well as the position and offset of each field. If we know the data type, then we can do EBCDIC conversion or packed decimal conversion as appropriate. This is a much more subtle algorithm, since we have two strategies for converting the data fields. See the section called "Strategy" for some reasons why we'd do it this way.

In order to mirror COBOL's largely decimal world-view, we will need to use the `decimal` module for all numbers and airthmetic.

We note that the presence of packed decimal data changes the file from text to binary. We'll begin with techniques for handling a text file with a fixed layout. However, since this often slides over to binary file processing, we'll move on to that topic, also.

Reading an All-Text File. If we ignore the EBCDIC and packed decimal problems, we can easily process a fixed-layout file. The way to do this is to define a handy structure that defines our record layout. We can use this structure to parse each record, transforming the record from a string into a dictionary that we can use for further processing.

In this example, we also use a generator function, `yieldRecords`, to break the file into individual records. We separate this functionality out so that our processing loop is a simple **for** statement, as it is with other kinds of files. In principle, this generator function can also check the length of `recBytes` before it yields it. If the block of data isn't the expected size, the file was damaged and an exception should be raised.

```
layout = [
```

```
    ( 'field1', 0, 12 ),
    ( 'field2', 12, 4 ),
    ( 'anotherField', 16, 20 ),
    ( 'lastField', 36, 8 ),
]
reclen= 44

def yieldRecords( aFile, recSize ):
    recBytes= aFile.read(recSize)
    while recBytes:
        yield recBytes
        recBytes= aFile.read(recSize)

cobolFile= file( 'my.cobol.file', 'rb' )
for recBytes in yieldRecords(cobolFile, reclen):
    record = dict()
    for name, start, size in layout:
        record[name]= recBytes[start:start+len]
```

Reading Mixed Data Types. If we have to tackle the complete EBCDIC and packed decimal problem, we have to use a slightly more sophisticated structure for our file layout definition. First, we need some data conversion functions, then we can use those functions as part of picking apart a record.

We may need several conversion functions, depending on the kind of data that's present in our file. Minimally, we'll need the following two functions.

`display`

> This function is used to get character data. In COBOL, this is called display data. It will be in EBCDIC if our files originated on a mainframe.

packed

> This function is used to get packed decimal data. In COBOL, this is called "comp-3" data. In our example, we have not dealt with the insert of the decimal point prior to the creation of a `decimal.Decimal` object.

```
import codecs
display = codecs.getdecoder('cp037')

def packed( bytes ):
    n= [ '' ]
    for b in bytes[:-1]:
        hi, lo = divmod( ord(b), 16 )
        n.append( str(hi) )
        n.append( str(lo) )
    digit, sign = divmod( ord(bytes[-1]), 16 )
    n.append( str(digit) )
    if sign in (0x0b, 0x0d ):
        n[0]= '-'
    else:
        n[0]= '+'
    return n
```

Given these two functions, we can expand our handy record layout structure.

```
layout = [
    ( 'field1', 0, 12, display ),
    ( 'field2', 12, 4, packed ),
    ( 'anotherField', 16, 20, display ),
    ( 'lastField', 36, 8, packed ),
]
reclen= 44
```

This changes our record decoding to the following.

```
cobolFile= file( 'my.cobol.file', 'rb' )
for recBytes in yieldRecords(cobolFile, reclen):
    record = dict()
    for name, start, size, convert in layout:
        record[name]= convert( recBytes[start:start+len] )
```

This example underscores some of the key values of Python. Simple things can be kept simple. The layout structure, which describes the data, is both easy to read, and written in Python itself. The evolution of this example shows how adding a sophisticated feature can be done simply and cleanly.

At some point, our record layout will have to evolve from a simple tuple to a proper class definition. We'll need to take this evolutionary step when we want to convert packed decimal numbers into values that we can use for further processing.

XML Files: The `xml.minidom` and `xml.sax` Modules

XML files are text files, intended for human consumption, that mix markup with content. The markup uses a number of relatively simple rules. Additionally, there are structural requirements that assure that an XML file has a minimal level of validity. There are additional rules (either a Document Type Defintion, DTD, or an XML Schema Definition, XSD) that provide additional structural rules.

There are three separate XML parsers available with Python. We'll ignore the `xml.expat` module (not for any good reason), and focus on the `xml.sax` and `xml.minidom` parsers.

`xml.sax` Parsing. The Standard API for XML (SAX) parser is described as an event parser. The parser recognizes different elements of an XML document and invokes methods in a handler which you provide. Your handler will be given pieces of the document, and can do appropriate processing with those pieces.

For most XML processing, your program will have the following outline: This parser will then use your `ContentHandler` as it parses.

1. Define a subclass of `xml.sax.ContentHandler`. The methods of this class will do your unique processing will happen.

2. Request the module to create an instance of an `xml.sax.Parser`.

3. Create an instance of your handler class. Provide this to the parser you created.

4. Set any features or options in the parser.

5. Invoke the parser on your document (or incoming stream of data from a network socket).

Here's a short example that shows the essentials of building a simple XML parser with the `xml.sax` module. This example defines a simple `ContentHandler` that prints the tags as well as counting the occurances of the `<informaltable>` tag.

```
import xml.sax

class DumpDetails( xml.sax.ContentHandler ):
    def __init__( self ):
        self.depth= 0
        self.tableCount= 0
    def startElement( self, aName, someAttrs ):
        print self.depth*' ' + aName
        self.depth += 1
```

```
        if aName == 'informaltable':
            self.tableCount += 1
    def endElement( self, aName ):
        self.depth -= 1
    def characters( self, content ):
        pass # ignore the actual data

p= xml.sax.make_parser()
myHandler= DumpDetails()
p.setContentHandler( myHandler )
p.parse( "../p5-projects.xml" )
print myHandler.tableCount, "tables"
```

Since the parsing is event-driven, your handler must accumulate any context required to determine where the individual tags occur. In some content models (like XHTML and DocBook) there are two levels of markup: structural and semantic. The structural markup includes books, parts, chapters, sections, lists and the like. The semantic markup is sometimes called "inline" markup, and it includes tags to identify function names, class names, exception names, variable names, and the like. When processing this kind of document, you're application must determine the which tag is which.

A `ContentHandler` Subclass. The heart of a SAX parser is the subclass of `ContentHandler` that you define in your application. There are a number of methods which you may want to override. Minimally, you'll override the `startElement` and `characters` methods. There are other methods of this class described in section 13.10.1 of the *Python Library Reference*.

`setDocumentLocator(`*`locator`*`)`

> The parser will call this method to provide an `xml.sax.Locator` object. This object has the XML document ID information, plus line and column information. The locator will be updated within the parser, so it should only be used within these handler methods.

`startDocument`

> The parser will call this method at the start of the document. It can be used for initialization and resetting any context information.

`endDocument`

> This method is paired with the `startDocument` method; it is called once by the parser at the end of the document.

`startElement(`*`name`*`,` *`attrs`*`)`

> The parser calls this method with each tag that is found, in non-namespace mode. The `name` is the string with the tag name. The `attrs` parameter is an `xml.sax.Attributes` object. This object is reused by the parser; your handler cannot save this object. The xml.sax.Attributes object behaves somewhat like a mapping. It doesn't support the `[]` operator for getting values, but does support `get`, `has_key`, `items`, `keys`, and `values` methods.

`endElement(`*`name`*`)`

> The parser calls this method with each tag that is found, in non-namespace mode. The `name` is the string with the tag name.

`startElementNS(`*`name`*`,` *`qname`*`,` *`attrs`*`)`

> The parser calls this method with each tag that is found, in namespace mode. You set namesace mode by using the parser's `p.setFeature(`

`xml.sax.handler.feature_namespaces, True )`. The `name` is a tuple with the URI for the namespace and the tag name. The `qname` is the fully qualified text name. The `attrs` parameter is an `xml.sax.Attributes` object. This object is reused by the parser; your handler cannot save this object. The xml.sax.Attributes object behaves somewhat like a mapping. It doesn't support the `[]` operator for getting values, but does support `get`, `has_key`, `items`, `keys`, and `values` methods.

`endElementNS(`*`name`*`,` *`qname`*`)`

The parser calls this method with each tag that is found, in namespace mode. The `name` is a tuple with the URI for the namespace and the tag name. The `qname` is the fully qualified text name.

`characters(`*`content`*`)`

The parser uses this method to provide character data to the `ContentHandler`. The parser may provide character data in a single chunk, or it may provide the characters in several chunks.

`ignorableWhitespace(`*`whitespace`*`)`

The parser will use this method to provide ignorable whitespace to the `ContentHandler`. This is whitespace between tags, usually line breaks and indentation. The parser may provide whitespace in a single chunk, or it may provide the characters in several chunks.

`processingInstructions(`*`target`*`,` *`data`*`)`

The parser will provide all `<?`*`target data`*`?>` processing instructions to this method. Note that the initial `<?xml version="1.0" encoding="UTF-8"?>` is not reported.

`xml.minidom` Parsing. The Document Object Model (DOM) parser creates a document object model from your XML document. The parser transforms the text of an XML document into a DOM object. Once your program has the DOM object, you can examine that object.

Here's a short example that shows the essentials of building a simple XML parser with the `xml.dom` module. This example defines a simple `ContentHandler` that prints the tags as well as counting the occurances of the `<informaltable>` tag.

We defined a `walkNode` function which does a recursive, depth-first traversal of the elements in the document structure. In many applications, the structure of the XML document is well known, and functions which are tied to the structure of the document can be used. In this example, we're reading a DocBook XML file, which has a complex, highly-nested structure.

```
import xml.dom.minidom

tables= []
def walkNode( n, depth=0 ):
    print depth*' ', n.tagName
    if n.tagName == "informaltable":
        tables.append( n )
    for d in n.childNodes:
        if d.nodeType == xml.dom.Node.ELEMENT_NODE:
            walkNode( d, depth+1 )

dom1 = xml.dom.minidom.parse("../p5-projects.xml")
walkNode( dom1.documentElement )
print tables
```

The DOM Object Model. The heart of a DOM parser is the DOM class hierarchy. Your program will work with a `xml.dom.Document` object. We'll look at a few essential classes of the DOM. There are other classes in this model, described in section 13.6.2 of the *Python Library Reference*. We'll focus on the most commonly-used classes.

The XML Document Object Model is a standard definition. The standard applies to both Java programs as well as Python. The `xml.dom` package provides definitions which meet this standard. The standard doesn't address how XML is parsed to create this structure. Consequently, the `xml.dom` package has no official parser. You could, for example, use a SAX parser to produce a DOM structure. Your handler would create objects from the classes defined in `xml.dom`.

The `xml.dom.minidom` package is an implementation of the DOM standard, which is slightly simplified. This implementation of the standard is extended to include a parser. The essential class definitions, however, come from `xml.dom`. We'll only look at methods used to get data from an XML document. We'll ignore the additional methods used by a parser to build a DOM object.

class Node

> The `Node` class is the superclass for all of the various DOM classes. It defines a number of attributes and methods which are common to all of the various subclasses. This class should be thought of as abstract: it is not used directly; it exists to provide common features to all of the subclasses.
>
> Here are the attributes which are common to all of the various kinds of `Node`s
>
> nodeType
>
> > This is an integer code that discriminates among the subclasses of `Node`. There are a number of helpful symbolic constants which are class variables in xml.dom.Node. These constants define the various types of Nodes. `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`.
>
> attributes
>
> > This is a map-like collection of attributes. It is an instance of `xml.dom.NamedNodeMap`. It has method functions including `get`, `getNamedItem`, `getNamedItemNS`, `has_key`, `item`, `items`, `itemsNS`, `keys`, `keysNS`, `length`, `removeNamedItem`, `removeNamedItemNS`, `setNamedItem`, `setNamedItemNS`, `values`. The `item` and `length` methods are defined by the standard and provided for Java compatibility.
>
> localName
>
> > If there is a namespace, then this is the portion of the name after the colon. If there is no namespace, this is the entire tag name.
>
> prefix
>
> > If there is a namespace, then this is the portion of the name before the colon. If there is no namespace, this is an empty string.
>
> namespaceURI
>
> > If there is a namespace, this is the URI for that namespace. If there is no namespace, this is `None`.
>
> parentNode

This is the parent of this Node. The Document Node will have None for this attribute, since it is the parent of all Nodes in the document. For all other Nodes, this is the context in which the Node appears.

previousSibling

Sibling Nodes share a common parent. This attribute of a Node is the Node which precedes it within a parent. If this is the first Node under a parent, the previousSibling will be None. Often, the preceeding Node will be a Text containing whitespace.

nextSibling

Sibling Nodes share a common parent. This attribute of a Node is the Node which follows it within a parent. If this is the last Node under a parent, the nextSibling will be None. Often, the following Node will be Text containing whitespace.

childNodes

The list of child Nodes under this Node. Generally, this will be a xml.dom.NodeList instance, not a simple Python list. A NodeList behaves like a list, but has two extra methods: item and length, which are defined by the standard and provided for Java compatibility.

firstChild

The first Node in the childNodes list, similar to childNodes[:1]. It will be None if the childNodes list is also empty.

lastChild

The last Node in the childNodes list, similar to childNodes[-1:]. It will be None if the childNodes list is also empty.

Here are some attributes which are overridden in each subclass of Node. They have slightly different meanings for each node type.

nodeName

A string with the "name" for this Node. For an Element, this will be the same as the tagName attribute. In some cases, it will be None.

nodeValue

A string with the "value" for this Node. For an Text, this will be the same as the data attribute. In some cases, it will be None.

Here are some methods of a Node.

hasAttributes

This function returns True if there are attributes associated with this Node.

hasChildNodes

This function returns True if there child Nodes associated with this Node.

class Document(Node)

This is the top-level document, the object returned by the parser. It is a subclass of Node, so it inherits all of those attributes and methods. The Document class adds

some attributes and method functions to the `Node` definition.

documentElement

This attribute refers to the top-most `Element` in the XML document. A `Document` may contain `DocumentType`, `ProcessingInstruction` and `Comment` `Nodes`, also. This attribute saves you having to dig through the `childNodes` list for the top `Element`.

`getElementsByTagName(`*`tagName`*`)`

This function returns a `NodeList` with each `Element` in this `Document` that has the given tag name.

`getElementsByTagNameNS(`*`namespaceURI`*`, `*`tagName`*`)`

This function returns a `NodeList` with each `Element` in this `Document` that has the given namespace URI and local tag name.

class Element(Node)

This is a specific element within an XML document. An element is surrounded by XML tags. In `<para id="sample">Text</para>`, the tag is `<para>`, which provides the name for the `Element`. Most `Elements` will have children, some will have `Attributes` as well as children. The `Element` class adds some attributes and method functions to the `Node` definition.

tagName

The full name for the tag. If there is a namesace, this will be the complete name, including colons. This will also be in `nodeValue`.

`getElementsByTagName(`*`tagName`*`)`

This function returns a `NodeList` with each `Element` in this `Element` that has the given tag name.

`getElementsByTagNameNS(`*`namespaceURI`*`, `*`tagName`*`)`

This function returns a `NodeList` with each `Element` in this `Element` that has the given namespace URI and local tag name.

`hasAttribute(`*`name`*`)`

Returns `True` if this `Element` has an `Attr` with the given name.

`hasAttributeNS(`*`namespaceURI`*`, `*`localName`*`)`

Returns `True` if this `Element` has an `Attr` with the given name based on the namespace and localName.

`getAttribute(`*`name`*`)`

Returns the string value of the `Attr` with the given name. If the attribute doesn't exist, this will return a zero-length string.

`getAttributeNS(`*`namespaceURI`*`, `*`localName`*`)`

Returns the string value of the `Attr` with the given name. If the attribute doesn't exist, this will return a zero-length string.

`getAttributeNode(`*`name`*`)`

Returns the `Attr` with the given name. If the named attribute doesn't exist, this method returns `None`.

`getAttributeNodeNS(`*`namespaceURI`*`, `*`localName`*`)`

Returns the `Attr` with the given name. If the named attribute doesn't exist, this method returns `None`.

class Attr(Node)

This is an attribute, within an Element. In `<para id="sample">Text</para>`, the tag is `<para>`; this tag has an attribute of `id` with a value of `sample`. Generally, the `nodeType`, `nodeName` and `nodeValue` attributes are all that are used. The `Attr` class adds some attributes to the `Node` definition.

name

The full name of the attribute, which may include colons. The `Node` class defines `localName`, `prefix` and `namespaceURI` which may be necessary for correctly processing this attribute.

value

The string value of the attribute. Also note that `nodeValue` will have a copy of the attribute's value.

class Text(Node) and class CDATASection(Node)

This is the text within an element. In `<para id="sample">Text</para>`, the text is `Text`. Note that end of line characters and indentation also count as `Text` nodes. Further, the parser may break up a large piece of text into a number of smaller `Text` nodes. The `Text` class adds an attribute to the `Node` definition.

data

The text. Also note that `nodeValue` will have a copy of the text.

class Comment(Node)

This is the text within a comment. The `<!--` and `-->` characters are not included. The `Comment` class adds an attribute to the `Node` definition.

data

The comment. Also note that `nodeValue` will have a copy of the comment.

Log Files: The `logging` Module

Most programs need a way to write sophisticated, easy-to-control log files what contain status and debugging information. Any program that requires careful auditing will benefit from using the `logging` module to create an easy-to-read permanent log. Also, when we have programs with multiple modules, and need more sophisticated debugging, we'll find a need for the `logging` module.

There are several closely related concepts that define a log. First, your program will have a hierarchical tree of `Loggers`. Each `Logger` is used to do two things. It creates `LogRecords` with your messages about errors, or debugging information. It provides these `LogRecords` to `Handlers` which ignore them, write them to files or insert them into databases. Each `Handler` can make use of a `Formatter` to provide a nice, readable version of each `LogRecord` message. Also, you can build sophisticated `Filters` if you need to handle complex situations.

The default configuration gives you a single `Logger`, named `""`, which uses a `StreamHandler` configured to write to standard error file, stderr.

While the logging module can appear complex, it gives us a number of distinct advatages.

- **Multiple Loggers.** We can easily create a large number of separate loggers. This helps us to manage large, complex programs. Each component of the program can have it's own, indepenent logger. We can configure the collection of loggers centrally, however, supporting sophisticated auditing and debugging which is independent of each individual component.

- **Hierarchy of Loggers.** Each `Logger` instance has a name, which is a `.`-separated string of names. For example, 'myapp.stock', 'myapp.portfolio'. This forms a natural hierarchy of `Loggers`. Each child inherits the configuration from its parent, which simplifies configuration. If, for example, we have a program which does stock portfolio analysis, we might have a component which does stock prices and another component which does overall portfolio calculations. Each component, then, could have a separate `Logger` which uses component name. Both of these `Loggers` are children of the `"" Logger`; the configuration for the top-most `Logger` would apply to both children.

  Some components define their own `Loggers`. For example SQLAlchemy, has a set of `Loggers` with `'sqlalchemy'` as the first part of their name. You can configure all of them by using that top-level name. For specific debugging, you might alter the configuration of just one `Logger`, for example, `'sqlalchemy.orm.sync'`.

- **Multiple Handlers.** Each `Logger` can feed a number of `Handlers`. This allows you to assure that a single important log messages can go to multiple destinations. A common setup is to have two `Handlers` for log messages: a `FileHandler` which records everything, and a `StreamHandler` which writes only severe error messages to stderr.

  For some kinds of applications, you may also want to add the `SysLogHandler` (in conjunction with a `Filter`) to send some messages to the operating system-maintained system log as well as the application's internal log. Another example is using the `SMTPHandler` to send selected log messages via email as well as to the application's log and stderr.

- **Level Numbers and Filters.** Each LogRecord includes a message level number, and a destination `Logger` name (as well as the text of the message and arguments with values to insert into the message). There are a number of predefined level numbers which are used for filtering. Additionally, a `Filter` object can be created to filter by destination `Logger` name, or any other criteria.

  The predefined levels are `CRITICAL`, `ERROR`, `WARNING`, `INFO`, and `DEBUG`; these are coded with numeric values from 50 to 10. Critical messages usually indicate a complete failure of the application, they are often the last message sent before it stops running; error messages indicate problems which are not fatal, but preclude the creation of usable results; warnings are questions or notes about the results being produced. The information messages are the standard messages to describe successful processing, and debug messages provide additional details.

  By default, all `Loggers` will show only messages which have a level number greater than or equal to `WARNING`, which is generally 30. When enabling debugging, we rarely want to debug an entire application. Instead, we usually enable debugging on specific modules. We do this by changing the level of a specific `Logger`.

  You can create additional level numbers or change the level numbers.

Programmers familiar with Java, for example, might want to change the levels to `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`, using level numbers from 70 through 10.

Module-Level Functions. The following module-level functions will get a `Logger` that can be used for logging. Additionally, there are functions can also be used to create `Handler`s, `Filter`s and `Formatter`s that can be used to configure a `Logger`.

`logging.getLogger(`*`name`*`)` → `Logger`

Returns a `Logger` with the given name. The name is a `.`-separated string of names (e.g., `"x.y.z"`) If the `Logger` already exists, it is returned. If the `Logger` did not exist, it is created and returned.

`logging.addLevelName(`*`lvl`*`, `*`levelName`*`)`

Defines (or redefines) a level number, proving a name that will be displayed for the given level number Generally, you will parallel these definitions with your own constants. For example,
```
CONFIG=20;
logging.addLevelName(CONFIG,"CONFIG")
```

`logging.basicConfig`

Configures the logging system. By default this creates a `StreamHandler` directed to stderr, and a default `Formatter`. Also, by default, all Loggers show only `WARNING` or higher messages. There are a number of keyword parameters that can be given to `basicConfig`.

`filename`

This keyword provides the filename used to create a `FileHandler` instead of a `StreamHandler`. The log will be written to the given file.

`filemode`

If a filename is given, this is the mode to open the file. By default, a file is opened with `'a'`, appending the log file.

`format`

This is the format string for the `Handler` that is created. A `Formatter` object has a `format` method which expects a dictionary of values; the format string uses `%(`*`key`*`)s` conversion specifications. See String Formatting with Dictionaries for more information. The dictionary provided to a `Formatter` is the `LogRecord`, which has a number of fields that can be interpolated into a log string.

`datefmt`

The date/time format to use for the asctime attribute of a `LogRecord`. This is a format string based on the time package `strftime` function. See Chapter 32, *Dates and Times: the `time` and `datetime` Modules* for more information on this format string.

`level`

This is the default message level for all loggers. The default is `WARNING`, 30.

`stream`

This is a stream that will be used to initialize a `StreamHandler` instead of a `FileHandler`. This is incompatible with *`filename`*. If both *`filename`* and *`stream`* are provided, stream is ignored.

Typically, you'll use this in the following form: `logging.basicConfig( level=logging.INFO )`.

`logging.fileConfig`

Configures the logging system. This will read a configuration file, which defines the loggers, handlers and formatters that will be built initially. Once the loggers are built by the configuration, then the `logging.getLogger` function will return one of these pre-built loggers.

`logging.shutdown`

Finishes logging by flushing all buffers and closing all handlers, which generally closes any internally created files and streams. An application must do this last to assure that all log messages are properly recorded in the log.

Logger Functions. The following functions are used to create a `LogRecord` in a `Logger`; a `LogRecord` is then processed by the `Handlers` associated with the `Logger`.

Many of these functions have essentially the same signature. They accept the text for a message as the first argument. This message can have string conversion specifications, which are filled in from the various arguments. In effect, the logger does *`message`* `%` `(` *`args`* `)` for you.

You can provide a number of argument values, or you can provide a single argument which is a dictionary. This gives us two principle methods for producing log messages.

- `log.info( "message %s, %d", "some string", 2 )`
- `log.info( "message %(part1)s, %(anotherpart)d", { "part1" : "some string", "anotherpart": 2 } )`

These functions also have an optional argument, *`exc_info`*, which can have either of two values. You can provide the keyword argument `exc_info= sys.exc_info()`. As an alternative, you can provide `exc_info=True`, in which case the logging module will call `sys.exc_info` for you.

In the following definitions, we'll assume that we've created a `Logger` named `log`.

`log.``,` (debug*`message`*, ⟨*`args`*⟩⟨*`exc_info`*⟩)

Creates a `LogRecord` with level `DEBUG`, then processes this `LogRecord` on this `Logger`. The *`message`* is the message test; the *`args`* are the arguments which are provided to the formatting operator, `%`. If the `exc_info` keyword argument is provided, then exception information will be added to the logging message. The value of exc_info can be an exception tuple (as provided by `sys.exc_info`); otherwise, `sys.exc_info` will be called to get the exception information.

`log.``,` (info*`message`*, ⟨*`args`*⟩⟨*`exc_info`*⟩)

Creates a `LogRecord` with level `INFO` on this logger. The positional arguments fill in the message; a single positional argument can be a dictionary. The exc_info keyword argument can provide exception information.

`log.``,` (warning*`message`*, ⟨*`args`*⟩⟨*`exc_info`*⟩)

Creates a `LogRecord` with level `WARNING` on this logger. The positional arguments fill in the message; a single positional argument can be a dictionary. The exc_info keyword argument can provide exception information.

`log.``,` (error*`message`*, ⟨*`args`*⟩⟨*`exc_info`*⟩)

Creates a LogRecord with level ERROR on this logger. The positional arguments fill in the message; a single positional argument can be a dictionary. The exc_info keyword argument can provide exception information.

log.(critical*message*, ⟨*args*⟩⟨*exc_info*⟩)

Creates a LogRecord with level CRITICAL on this logger. The positional arguments fill in the message; a single positional argument can be a dictionary. The exc_info keyword argument can provide exception information.

*log.*log*lvlmessage*⟨*args*⟩⟨*exc_info*⟩

Creates a LogRecord with the given *lvl* on this logger. The positional arguments fill in the message; a single positional argument can be a dictionary. The exc_info keyword argument can provide exception information.

log.(exception*message*, ⟨*args*⟩)

Creates a LogRecord with level ERROR on this logger. The positional arguments fill in the message; a single positional argument can be a dictionary. Exception info is added to the logging message, as if the keyword parameter exc_info=True. This method should only be called from an exception handler.

log.(isEnabledFor*level*)

Returns True if this Logger will handle messages of this level or higher. This can be handy to prevent creating really complex debugging output that would only get ignored by the logger. This is rarely needed, and is used in the following structure: `if log.isEnabledFor(logging.DEBUG): log.debug( "some complex message" ).`

These functions are used to configure a Logger. Generally, you'll configure Loggers using the module level basicConfig and fileConfig functions. However, in some specialized circumstances (like unit testing), you may want finer control without the overhead of a configuration file. In the following definitions, we'll assume that we've created a Logger named log.

*log.*propagate

When True, all the parents of a given Logger must also handle the message. This assures consistency for audit purposes. When False, the parents will not handle the message. A False value might be used for keeping debugging messages separate from other messages. By default this is a True value.

log.(setLevel*level*)

Sets the level for this Logger; messages less severe are ignored. Messages of this severity or higher are handled. The special value of logging.NOTSET indicates that this Logger inherits the setting from the parent. The root logger has a default value of logging.WARNING.

*log.*getEffectiveLevel → number

Gets the level for this Logger. If this Logger has a setting of logging.NOTSET (the default for all Loggers) then it inherits the level from its parent.

log.(addFilter*filter*)

Adds the given Filter object to this Logger.

log.(removeFilter*filter*)

Removes the given `Filter` object from this `Logger`.

log.addHandler

Adds the given `Handler` object to this `Logger`.

log.removeHander

Removes the given `Handler` object from this `Logger`.

There are also some functions which would be used if you were creating your own subclass of `Logger` for more specialized logging purposes. These methods include `log.filter`, `log.handle` and `log.findCaller`.

Using the Logger. Generally, there are a number of ways of using a `Logger`. In a module that is part of a larger application, we will get an instance of a `Logger`, and trust that it was configured correctly by the overall application. In the top-level application we may both configure and use a `Logger`.

This example shows a simple module file which uses a Logger.

Example 34.4. logmodule.py

```
import logging, sys

log= logging.getLogger('logmodule')

def someFunc( a, b ):
    log.debug( "someFunc( %d, %d )", a, b )
    try:
        return 2*int(a) + int(b)
    except ValueError, e:
        log.warning( "ValueError in someFunc( %r, %r )", a, b, exc_info=True )

def mainFunc( *args ):
    log.info( "Starting mainFunc" )
    z= someFunc( args[0], args[1] )
    print z
    log.info( "Ending mainFunc" )

if __name__ == "__main__":
    logging.fileConfig( "logmodule_log.init" )
    mainFunc( sys.argv[1:] )
```

❶ We import the `logging` module and the `sys` module.

❷ We ask the logging module to create a `Logger` with the given name. We do this through a factory function to assure that the logger is configured correctly. The logging module actually keeps a pool of `Loggers`, and will assure that there is only one instance of each named `Logger`.

❸ This function has a debugging message and a warning message. This is typical of most function definitions. Ordinarily, the debug message will not show up in the log; we can only see it if we provide a configuration which sets the log level to `DEBUG` for the root logger or the `logmodule` `Logger`.

❹ This function has a pair of informational messages. This is typical of "main" functions which drive an overall application program. Applications which have several logical steps might have informational messages for each step. Since informational messages are lower level than warnings, these don't show up by default; however, the main program that uses this module will often set the overall level to logging.INFO to enable informational messages.

File Format Exercises

1. **Create An Office Suite Result.** Back in the section called “Iteration Exercises”

we used the **for** statement to produce tabular displays of data. This included How Much Effort to Produce Software?, Wind Chill Table, Celsius to Fahrenheit Conversion Tables and Dive Planning Table. Update one of these programs to produce a CSV file. If you have a desktop office suite, be sure to load the CSV file into a spreadsheet program to be sure it looks correct.

2. **Proper File Parsing.** Back in the section called "File Exercises" we built a quick and dirty CSV parser. Fix these programs to use the CSV module properly.

3. **Configuration Processing.** In the section called "Stock Valuation", we looked at a program which processed blocks of stock. One of the specific programs was an analysis report which showed the value of the portfolio on a given date at a given price. We make this program more flexible by having it read a configuration file with the current date and stock prices.

4. **Office Suite Extraction.** Most office suite software can save files in XML format as well as their own proprietary format. The XML is complex, but you can examine it in pieces using Python programs. It helps to work with highly structured data, like an XML version of a spreadsheet. For example, your spreadsheet may use tags like `<Table>`, `<Row>` and `<Cell>` to organize the content of the spreadsheet.

   First, write a simple program to show the top-level elements of the document. It often helps to show the text within those elements so that you can correlate the XML structure with the original document contents.

   Once you can display the top-level elements, you can focus on the elements that have meaningful data. For example, if you are parsing spreadsheet XML, you can assembled the values of all of the `<Cell>`'s in a `<Row>` into a proper row of data, perhaps using a simple Python `list`.

Chapter 35. Programs: Standing Alone

Table of Contents

This chapter will cover additional aspects of creating some common kinds of programs in Python. We'll survey the landscape in the section called "Kinds of Programs". Then, in the section called "Command-Line Programs: Servers and Batch Processing" will the essence of program startup using command-line options and operands. In the `getopt` module and show how to properly control the startup of programs of every kind. In the section called "The `getopt` Module" we'll look at parsing command line options with `getopt`. In xthe section called "The `optparse` Module" we'll look at parsing command-line options with `optparse`.

Interactive graphical user interfaces are beyond the scope of this book. There are several handy graphic frameworks, including Tkinter and GTK that help you write graphical user interfaces. However, GUI programs are still started from the command line, so this section is relevant for those kinds of programs.

Kinds of Programs

There are many design patterns for our application programs. We can identify a number of features that distinguish different kinds of programs. We can create a taxonomy of

program designs based on how we interact with them. We could also create a taxonomy based on the program's internal structure or its interfaces.

We can look at a program based on the type of interaction that it has with a person. There's a spectrum of interaction.

- A program can be started from the command line and have no further interaction with the human user. We can call these batch programs because they usually process a batch of individual transactions. We can also call them command-line programs because our only interaction is at the command prompt. A large number of data analysis and business-oriented programs work with batches of data. Additionally, we can describe servers as being similar to batch programs. This is a focus for this chapter.

- A program can have very sophisticated interaction with the human user. The interaction may be character-oriented, or it can have a graphic user interface (GUI) and be started by double-clicking an icon. What's important is that the user drives the processing, not the batch of data. Typically, a program with rich user interaction will be a client of one or more services. These programs are beyond the scope of this book.

We can also look at programs based on their structure and how the interact with other programs.

- Some programs stand alone. They have an executable file which starts things off, and perhaps includes some libraries. Often a client program is a stand-alone program that runs on someone's desktop. This is a focus for this chapter.

- Some programs plug into a larger and more sophisticated frameworks. The framework is, essentially, a closely related collection of libraries and interfaces. Most web applications are built as programs which plug into a web server framework. There is a tremendous amount of very common processing in handling a web transaction. There's little value in repeating this programming, so we inherit it from the framework.

We can distinguish programs in how they interact with other programs to create a larger system. We'll turn to this topic in the next chapter, Chapter 36, *Programs: Clients, Servers, the Internet and the World Wide Web*.

- Some programs are clients. They rely on services provided by other programs. The service it relies on might be a web server or a database server. In some cases, the client program has rich user interaction and stands alone.

- Some programs are servers. They provide services to other programs. The service might be domain names, time, or any of a myriad of services that are an essential part of Linux and other operating systems.

- Some programs are both servers and clients of other services. Most servers have no interaction; they are command-line programs which are clients of other command-line programs. A web server typically has plug in web applications which use database servers. A database server may make use of other services within an operating system.

Some Subspecies. Stand-alone, command-line programs have a number of design patterns. Some programs are *filters* that read an input file, perform an extract or a calculation and produce a result file that is derived from the input. Programs can be *compilers*, performing extremely complex transformations from one or more input files to create an output file. Programs can be *interpreters*, where statements in a language are read and processed. Some programs, like the Unix awk utility, combine filtering and interpreting.

Stand-alone, iteractive programs allow a user to create and manipulate data objects. Interactive programs often have sophisticated graphics. All games are interactive progams. The software often characterized as the *office suite*, including word processors, spread sheets, graphics programs, schedule managers and contact managers are interactive programs.

Some programs are *clients* of services. For example, browsers take user queries, fetch and format data, and present the data to the user. An FTP client program may display contents of an FTP server, accepting user commands through a graphical user interface (GUI) and transferring files. An IMAP client program may display mailboxes on a mail server, accepting commands and transferring or displaying mail messages.

Many programs combine interaction with being a client of one or more services. Most browsers, like Firefox, are clients for servers which use a number of protocols, including HTTP, POP3, IMAP4, FTP, NNTP, and GOPHER. Besides being a client, a browser also provides graphics, handling numerous MIME data types for different kinds of images and sounds.

Yet another common type of program is a *server*. These programs are also interactive, but they interact with client programs, not a person through a GUI. An HTTP server like Apache, for instance, responds to browser requests for web pages. An FTP server responds to FTP client requests for file transfers. A server is often a kind of batch program, since it is left running for indefinite periods of time, and has no user interaction.

Command-Line Programs: Servers and Batch Processing

Many programs have minimal or no user interaction at all. They are run from a command-line prompt, perform their function, and exit gracefully. They may produce a log; they may return a status code to the operating system to indicate success for failure.

Almost all of the core Linux utilities (cp, rm, mv, ln, ls, df, du, etc.) are programs that decode command-line parameters, perform their processing function and return a status code. Except for a few explicitly interactive programs like editors (ex, vi, emacs, etc.), almost all of the core elements of Linux are filter-like programs.

There are two critical features that make a command-line program well-behaved. First, the program should accept the arguments in a standard manner. Second the program should generally limit output to the standard output and standard error files created by the environment. When any other files are written it must be by user request and possibly require interactive confirmation.

Command Line Options and Operands. The standard handling of command-line arguments is given as 13 rules for UNIX commands, as shown in the *intro* section of UNIX man pages. These rules describe the program names (rules 1-2), simple options (rules 3-5), options that take argument values (rules 6-8) and operands (rules 9 and 10) for the program.

1. The program name should be between two and nine characters. This is consistent with most file systems where the program name is a file name. In the Python environment, the program file must have extension of `.py`.

2. The program name should include only lower-case letters and digits. The objective is to keep names relatively simple and easy to type correctly. Mixed-case names and names with punctuation marks can introduce difficulties in typing the program name correctly. To be used as a module or package in Python, the program file name *must* be just letters, digits and _'s.

3. Option names should be one character long. This is difficult to achieve in complex programs. Often, options have two forms: a single-character short form and a

multi-character long form.

4. Single-character options are preceded by `-`. Multiple-character options are preceeded by `--`. All options have a flag that indicates that this is an option, not an operand. Single character options, again, are easier to type, but may be hard to remember for new users of a program.

5. Options with no arguments may be grouped after a single `-`. This allows a series of one-character options to be given in a simple cluster, for example `ls -ldai bin` clusters the `-l`, `-d`, `-a` and `-i` options.

6. Options that accept an argument value use a space separator. The option arguments are not run together with the option. Without this rule, it might be difficult to tell a option cluster from an option with arguments. Without this rule `cut -ds` could be an argument value of `s` for the `-d` option, or it could be clustered single-character options `-d` and `-s`.

7. Option-arguments cannot be optional. If an option requires an argument value, presence of the option means that an argument value will follow. If the presence of an option is somehow different from supplying a value for the option, two separate options must be used to specify these various conditions.

8. Groups of option-arguments following an option must be a single word; either separated by commas or quoted. For example: `-d "9,10,56"`. A space would mean another option or the beginning of the operands.

9. All options must precede any operands on the command line. This basic principle assures a simple, easy to understand uniformity to command processing.

10. The string `--` may be used to indicate the end of the options. This is particularly important when any of the operands begin with `-` and might be mistaken for an option.

11. The order of the options relative to one another should not matter. Generally, a program should absorb all of the options to set up the processing.

12. The relative order of the operands may be significant. This depends on what the operands mean and what the program does.

13. The operand `-` preceded and followed by a space character should only be used to mean standard input. This may be passed as an operand, to indicate that the standard input file is processed at this time. For example, `cat file1 - file2` will process file1, standard input and file2.

These rules are handled by the `getopt` (or `optparse`) module and the `sys.argv` variable in the `sys` module.

Output Control. A well-behaved program does not overwrite data without an explicit demand from a user. Programs with a assumed, default or implicit output file are a pronblem waiting to happen. A well-behaved program should work as follows.

1. A well-designed program has an obvious responsibility that is usually tied to creating one specific output. This can be a report, or a file of some kind. In a few cases we may find it necessary to optimize processing so that a number of unrelated outputs are produced by a single program.

2. The best policy for this output is to write the resulting file to standard output (`sys.stdout`, which is the destination for the **print** statement.) Any logging, status or error reporting is sent to `sys.stderr`. If this is done, then simple shell redirection operators can be used to collect this output in an obvious way.

```
python someProgram.py >this_file_gets_written
```

3. In some cases, there are actually two outputs: details and a useful summary. In this case, the summary should go to standard output, and an option specifies the destination of the details.

```
python aProgram.py -o details.dat >summary.txt
```

Program Startup and the Operating System Interface. The essential operating system interface to our programs is relatively simple. The operating system will start the Python program, providing it with the three standard files (stdin, stdout, stderr; see the section called "File Semantics" for more information), and the command line arguments. In response, Python provides a status code back to the operating system. Generally a status code of 0 means things worked perfectly. Status codes which are non-zero indicate some kind of problem or failure.

When we run something like

```
python casinosim.py -g craps
```

The operating system command processor (the Linux shell or Windows cmd.exe) breaks this line into a command (**python**) and a sequence of argument values. The shell finds the relevant executable file by searching it's `PATH`, and then starts the program, providing the rest of the command line as argument values to that program.

A Python program will see that the command line arguments are assigned to `sys.argv` as `["casinosim.py", "-g", "craps"]`. `argv[0]` is the name of the main module, the script Python is currently running.

When the script in `casinosym.py` finishes running, the Python interpreter also finishes, and returns a status code of 0 to the operating system.

To return a non-zero status code, use the `sys.exit` function.

Reuse and The Main-Import Switch. In the section called "Module Use: The **import** Statement" we talked about the Main-Import switch. The global `__name__` variable is essential for determing the context in which a module is used.

A well-written application module often includes numerous useful class and function definitions. When combining modules to create application programs, it may be desirable to take a module that had been originally designed as a stand-alone program and combine it with others to make a larger and more sophisticated program. In some cases, a module may be both a main program for some use cases and a library module for other use cases.

The `__name__` variable defines the context in which a module is being used. During evaluation of a file, when `__name__ == "__main__"`, this module is the *main* module, started by the Python interpreter. Otherwise, `__name__` will be the name of the file being imported. If `__name__` is not the string `"__main__"`, this module is being imported, and should take no action of any kind.

This test is done with the as follows:

```
if __name__ == "__main__":
    main()
```

This kind of reuse assures that programming is not duplicated. It is notoriously difficult to maintain two separate files that are supposed to contain the same program text. This kind of "cut and paste reuse" is a terrible burden on programmers. Python encourages reuse through both classes and modules. All modules can be cofigured as importable and reusable programming.

The `getopt` Module

The command line arguments from the operating system are put into the `sys.argv` variable as a sequence of strings. Looking at the syntax rules for command line options and operands in the previous section we can see that it can be challenging to parse this sequence of strings.

The `getopt` module helps us parse the options and operands that are provided to our program on the command line. This module has one very important function, also named `getopt`.

`getopt.getopt(`*`args`*`,` *`options`*`,` ⟨*`long_options`*⟩`)` → (options, operands)

> Decode the given sequence of arguments, *`args`*, using the given set of *`options`* and *`long_options`*. Returns a tuple with a sequence of normalized (option,value) pairs plus a sequence of the program's operand values.
>
> The *`args`* value should not include `sys.argv[0]`, the program name. Therefore, the argument value for *`args`* is almost always `sys.argv[1:]`.
>
> The *`options`* value is a string of the one-letter options. Any options which require argument values are followed by a `:`. For example, `"ab:c"` means that the program will accept `-a`, `-c`, `-ac`, `-b value` as options.
>
> The *`long_options`* value is optional, if present it is a list of the long options. If a long option requires a parameter value, it's name must end in `=`. For example, `("silent","debug","log=")` means that options like `--silent`, `--debug`, and `--log=myfile.log` are accepted as options.

There are two results of `getopt`: the options and the operands. The options is a list of (*`name, value`*) pairs. The operands is the list of names which follow the last option. In most cases, this list is a list of file names to be used as input.

There are several ways to handle the options list.

- We can iterate through this list, setting global variables, or configuring some processing object. This works well when we have both short and long option names for the same configuration setting.

```
options, operands = getopt.getopt( sys.argv[1:], ... )
for name, value in options:
    if name == "-X" or name == "--long":
        set some global variable
```

- We can define our configuration as a dictionary. We can then update this dictionary with the options. This forces the rest of our program to handle the `-x` or `--long` names for cofiguration parameters.

```
config = { "-X" : default, "--long": default }
options, operands = getopt.getopt( sys.argv[1:], ... )
config.update( dict(options) )
```

- We can define our configuration as a dictionary. We can initialize that configuration dictionary with the given options then fold in default values. While pleasantly obvious, it still makes the `-x` and `--long` options visible throughout our program.

```
options, operands = getopt.getopt( sys.argv[1:], ... )
config= dict(options)
config.setdefault( "-X", value )
config.setdefault( "--long", value )
```

One very adaptable and reusable structure is the following.

```
class MyApplication( object ):
    def __init__( self ):
        self.someProperty= default
    def process( aFileName ):
        """ The Real Work. """
        This is the real work of this applications

def main():
    theApp= MyApplication()
    options, operands = getopt.getopt( sys.argv[1:], "..." )
    for name, value in options:
        if name == "-X" or name == "--long":
            set properties in theApp
    for fileName in operaands:
        theApp.process( aFileName )
```

A Complete Example. Here's a more complete example of using `getopt`. Assume we have a program with the following synopsis.

`portfolio.py` ⟨-v⟩ ⟨-h⟩ ⟨-d *mm/dd/yy*⟩ ⟨-s *symbol*...⟩ file

This program has two single-letter options: `-v` and `-h`. It has two options which take argument values, `-d` and `-s`. Finally, it accepts an operand of a file name.

These options can be processed as follows:

```
"""portfolio.py -- examines a portfolio file
"""
import getopt
class Portfolio( object ):
    ...

def main():
    portfolio= Portfolio()
    opts, operands = getopt( sys.argv[1:], "vhd:s:" )
    for o,v in opts:
        if o == "-v": portfolio.verbose= True
        elif o == "-h":
            print __doc__
            return
        elif o == "-d": portfolio.date= v
        elif o == "-s": portfolio.symbol= v
    for f in operands:
        portfolio.process( f )

if __name__ == "__main__":
    main()
```

The program's options are coded as `"vhd:s:"`: the single-letter options (`-v` and `-h`) and the value options (`-d` and `-s`). The `getopt` function separates the the options from the arguments, and returns the options as a sequence of option flags and values. The `process` function performs the real work of the program, using the options and operands extracted from the command line.

The `optparse` Module

The command line arguments from the operating system are put into the `sys.argv` variable as a sequence of strings. Looking at the syntax rules for command line options and operands in the previous section we can see that it can be challenging to parse this sequence of strings.

The `optparse` module helps us parse the options and operands that are provided to our

program on the command line. This module has two very important class definitions: `OptionParser` and `Values`.

An `OptionParser` object does two things:

- It contais a complete map of your options, operands and any help strings or documentation. This module can, therefore, produce a complete, standard-looking command synopsis. The `-h` and `--help` options will do this by default.
- It parses the `sys.argv[1:]` list of strings and creates a `Values` object with the resulting option values.

The `OptionParser` has the following methods and attributes. There are a number of features which are used by applications which need to create a specialized subclass. We'll focus on the basic use case and ignore some of the features focused on extensibility.

`OptionParser`

The constructor for an `optparse.OptionParser` has a number of keyword arguments that can be used to define the program's options.

usage

This keyword parameter sets the usage summary that will print when the options cannot be parsed, or help is requested. If you don't provide this, then your program's name will be taken from `sys.argv[0]`. You can suppress the usage information by setting this to the special constant `optparse.SUPPRESS_USAGE`.

version

This keyword parameter provides a version string. It also adds the option of `-version` which displays this string. This string can contain the formatting characters `%prog` which will be replaced with the program's name.

description

A paragraph of text with an overview of your program. This is displayed in respose to a help request.

add_help_option

This is `True` by default; it adds the `-h` and `-help` options. You can set this to `False` to prevent adding the help options.

prog

The name of your program. Use this if you don't want `sys.argv[0]` to be used.

`add_option(`*`string`*`, `*`keywords`*`)`

This method of an `OptionParser` defines an option. The positional argument values are the various option strings for this option. There can be any mixture of short (`-x`) and long (`--long`) option strings. This is followed by any number of keyword arguments to provide additional details about this option. It is rare to have multiple short option strings for the same option.

For example, `add_option( "-o", "--output", "-w", dest="output_file", metavar="output" )` defines three different variations that set a single destination value, `output_file`. In the help test, the string `"output"` will be used

to identify the three alternative option strings.

action

This keyword parameter defines what to do when this option appears on the command line. The default action is `"store"`. Choices include `"store"`, `"store_const"`, `"store_true"`, `"store_false"`, `"append"`, `"count"`, `"callback"` and `"help"`. The store actions store the option's value. The append action accumulates a list of values. The count simply counts occurances of the option. Count is often used so that `-v` is verbose and `-vv` is even more verbose.

type

This keyword parameter defines what type of value this option uses. The default type is "string". Choices include `"string"`, `"int"`, `"long"`, `"choice"`, `"float"` and `"complex"`.

dest

This keyword parameter defines the attribute name in the `OptionParse` object that will have the final value. If you don't provide a value, then the first long option name will be used. If you didn't provide any long option names, then the first short option name will be used.

nargs

This keyword parameter defines how many values are permitted for this option. The default value is 1. If this value is more than 1, then a tuple is created to contain the sequence of values.

const

If the action was `"store_const"`, this keyword parameter provides the constant value which is stored.

choice

If the type was `"choice"`, this is a list of strings that contain the valid choices. If the option's value is not in this list, then this is a run-time error. This set of choices is displayed as the help for this option.

help

This keyword parameter provides the help text for this option.

metavar

This keyword parameter provides the option's name as shown to the user in the help documentation. This may be different than the abbreviations chosen for the option.

callback

If the action was `"callback"`, this is a callable (either a function or a class with a `__call__` method) that is called. This is called with four positional values: the `Option` object which requested the callback, the command line option string, the option's value string, and the overall `OptionParser` object.

callback_args, callback_kwargs

If the action was `"callback"`, these keyword parameters provide the additional arguments and keywords used when calling the given function or object.

`set_defaults`*`keywords`*

This method of an `OptionParser` provides an option's default value. Each keyword parameter is a destination name. These must match the *`dest`* names (or the the option string) for each option that you are providing a default value.

`parse_args(`*`args`*, ⟨*`values`*⟩) → (options, operands)

This method will parse the provided command-line argument strings and update a given `optparse.Values` object. By default, this will parse `sys.argv[1:]` so you don't need to provide a value for the *`args`* parameter. Also, this will create and populate a fresh `optparse.Values` object, so you don't need to provide a value for the *`values`* parameter.

The usual call form is `options, operands = myParser.parse_args()`.

A `Values` object is created by an `OptionParser`. It only has attribute values. The attributes are defined by the options seen during parsing and any default settings that were provided to the `OptionParser`.

A Complete Example. Here's a more complete example of using `optparse`. Assume we have a program with the following synopsis.

`portfolio.py` ⟨-v⟩ ⟨-h⟩ ⟨-d *`mm/dd/yy`*⟩ ⟨-s *`symbol`*...⟩ file

This program has two single-letter options: `-v` and `-h`. It has two options which take argument values, `-d` and `-s`. Finally, it accepts an operand of a file name.

These options can be processed as follows:

```
"""portfolio.py -- examines a portfolio file
"""
import optparse
class Portfolio( object ):
    ...

def main():
    portfolio= Portfolio()
    oparser= optparse.OptionParser( usage=__doc__ )
    oparser.add_option( "-v", action="count", dest="verbose" )
    oparser.add_option( "-d", dest="date" )
    oparser.add_option( "-s", dest="symbol" )
    oparser.set_defaults( verbose=0, date=None, symbol="GE" )
    options, operands = oparser.parse_args()
    portfolio.date= options.date
    portfolio.symbol= options.symbol
    for f in operands:
        portfolio.process( f )

if __name__ == "__main__":
    main()
```

The program's options are added to the parser. The default values, similarly are set in the parser. The `parse_args` function separates the the options from the arguments, and builds the options object with the defaults and the parsed options. The `process` function performs the real work of the program, using the options and operands extracted from the command line.

Command-Line Examples

Let's look at a simple, but complete program file. The program simulates several dice throws. We've decided that the command-line synopsis should be:

`dicesim.py` ⟨`-v`⟩ ⟨`-s` *`samples`*⟩

The `-v` option leads to *verbose* output, where every individual toss of the dice is shown. Without the `-v` option, only the summary statistics are shown. The `-s` option tells how many samples to create. If this is omitted, 100 samples are used.

Here is the entire file. This program has a five-part design pattern that we've grouped into three sections.

Example 35.1. dicesim.py

```
#!/usr/bin/env python
"""dicesim.py

Synopsis:
    dicesim.py [-v] [-s samples]
-v is for verbose output (show each sample)
-s is the number of samples (default 100)
"""

import dice, getopt, sys
```

❶ **Docstring.** The docstring provides the synopsis of the program, plus any other relevant documentation. This should be reasonably complete. Each element of the documentation is separated by blank lines. Several standard document extract utilities expect this kind of formatting.

❷ **Imports.** The imports line lists the other modules on which this program depends. Each of these modules might have the main-import switch and a separate main program. Our objective is to reuse the imported classes and functions, not the main function.

```
def dicesim( samples=100, verbose=0 ):
    d= dice.Dice()
    t= 0
    for s in range(samples):
        n= d.roll()
        if verbose: print n
        t += n
    print "%s samples, average is %s" % ( samples, t/float(samples) )
```

❶ **Actual processing in dicesym.** This is the actual heart of the program. It is a pure function with no dependencies on a particular operating system. It can be imported by some other program and reused.

```
def main():
    samples= 100
    verbose= 0
    opts,operands= getopt.getopt( sys.argv[1:], "vs:" )
    for o,v in opts:
        if o == "-v": verbose = 1
        elif o == "-s": samples= int(v)
    dicesim( samples, verbose )

if __name__ == "__main__":
    main()
```

❶ **Argument decoding in main.** This is the interface between the operating system that initiates this program and the actual work in `dicesym`. This does not have much reuse potential.

❷ **Program vs. import switch.** This makes the determination if this is a main program or an import. If it is an import, then `__name__` is not `"__main__"`, and no additional processing happens beyond the definitions. If it is the main program, the `__name__` is `"__main__"`; the arguments are parsed by `main`, which

calls `dicesym` to do the real work.

This is a typical layout for a complete Python main progam. There are two clear objecives. First, keep the `main` program focused; second, provide as many opportunities for reuse as possible.

Other Command-Line Features

Python, primarily, is a programming language. However, Python is also a family of related programs which interpret the Python language. While we can generally assume that the Python language is the same as the Python interpreter, there are some subtleties that are features of the interpreter, separate from the language.

Generally, the CPython interpreter is the baseline against which others are compared, and from which others are derived. Other interpreters include Jython, Iron Python and Python for .Net.

The Python interpreter has a fairly simple command-line interface. We looked at it briefly in the section called “Script Mode”. In non-Windows environments, you can use the man command to see the full set of command-line options. In all cases, you can run **python -h** or **python --help** to get a summary of the options.

Generally there are four kinds of command-line options.

- **Identify The Program (`-c`, `-m`, `-`, *`file`*).** The `-c` option provides the Python program on the command line as a quoted string. This isn't terribly useful. However, we can use it for things like the following.

  ```
  python -c 'import sys; print sys.version'
  ```

  Note the rarely-used `;` to terminate a statement.

  The `-m` option will locate a module on the `PYTHONPATH` and execute that module. This allows you to install a complete application in the Python library and execute the top-level "main program" script.

  As we noted in the section called “Script Mode”, the command-line argument to the Python interpreter is expected to be a Python program file. Additionally, we can provide a Python program on standard input and use **python -** to read and process that program.

- **Select the Division Operator Semantics (`-Q`).** As we noted in the section called “Division Operators”, there are two senses for division. You can control the meaning of `/` using `-Qnew` and `-Qold`. You can also debug problems with `-Qwarn` or `-Qwarnall`. Rather than rely on `-Qnew`, you should include `from __future__ import division` in every program that uses the new `//` operator and the new sense of the `/` operator.

- **Optimization (`-O`).** We can use `-O` and `-OO` to permit some optimization of the Python bytecode. This may lead to small performance improvements.

  Generally, there are two sources for performance improvements that are far more important than optimization. First, and most fundamentally, correct choices of data structures and algorithms have the most profound influence on performance. Second, modules can be written in C and use the Python API's. These C-language modules can dramatically improve performance, also.

- **Startup and Loading (`-S`, `-E`).** There are several ways to control the way Python starts and which modules it loads.

  The `-E` option ignores all environment variables (`PYTHONPATH` is the most

commonly used environment variable.)

Ordinarily Python executes an implicit `import site` when it starts executing. The `site` module populates `sys.path` with standard locations for packages and modules. The `-S` option will suppress this behavior.

- **Debugging (`-d`, `-i`, `-v`, `-u`).** Python has some additional debugging information that you can access with the `-d` option. The `-i` option will allow you to execute a script and then interact with the Python intrpreter. The `-v` option will display verbose information on the **import** processing.

  Sometimes it will help to remove the automatic buffering of standard output. If you use the `-u` option, mixed stderr and stdout streams may be easier to read.

- **Indentation Problem-Solving (`-t`, the TabNanny, and `-x`).** The `-t` option gives warning on inconsistent use of tabs and spaces. The `-tt` option makes these warnings into errors, stopping your program from running.

  The `-x` option skips the first line of a file. This can be used for situations where the first line of a Python file can't be a simple `#!` line. If the first line can't be a comment to Python, this will skip that line.

There are a number of environment variables that Python uses. We'll look at just a few.

`PYTHONPATH`

This defines the set of directories searched for modules. This is in addition to the directories placed on to `sys.path` by the `site` module.

`PYTHONSTARTUP`

This file is executed when you start Python for interactive use. You can use the script executed at startup time to import useful modules, define handy functions or alter your working environment in other ways.

Command-Line Exercises

1. **Create Programs.** Refer back to exercises in Part I, "Language Basics". See sections the section called "Numeric Types and Expressions", the section called "Condition Exercises", the section called "Iteration Exercises", the section called "Function Exercises". Modify these scripts to be stand-alone programs. In particular, they should get their input via `getopt` from the command line instead of `raw_input` or other mechanism.

2. **Larger Programs.** Refer back to exercises in Part II, "Data Structures". See sections the section called "String Exercises", the section called "Tuple Exercises", the section called "List Exercises", the section called "Dictionary Exercises", the section called "Exception Exercises". Modify these scripts to be stand-alone programs. In many cases, these programs will need input from files. The file names should be taken from the command line using `getopt`.

3. **Object-Oriented Programs.** Refer back to exercises in the section called "Class Definition Exercises", the section called "Advanced Class Definition Exercises". Modify these scripts to be stand-alone programs.

Chapter 36. Programs: Clients, Servers, the Internet and the World Wide Web

Table of Contents

The *World-Wide Web* is a metaphorical description for the sophisticated interactions among computers. The core technology that creates this phenomenon is the Internetworking Protocol suite, sometimes called *The Internet*. Fundamentally, the internetworking protocols define a relationship between pieces of software called the *client-server model*. In this case some programs (like browsers) are clients. Other programs (like web servers, databases, etc.) are servers.

This client-server model of programming is very powerful and adaptable. It is powerful because it makes giant, centralized servers available to large numbers of remote, widely distributed users. It is adaptable because we don't need to send software to everyone's computer to make a change to the centralized service.

Essentially, every client-server application involves a client application program, a server application, and a protocol for communication betweem the two processes. In most cases, these protocols are part of the popular and enduring suite of internetworking protocols based on TCP/IP. For more information in TCP/IP, see *Internetworking with TCP/IP* [Comer95].

We'll digress into the fundamentals of TCP/IP in the section called “About TCP/IP”. We'll look at what's involved in a web server in the section called “Web Servers and the HTTP protocol”. We'll look briefly at web services in the section called “Web Services: The xmlrpclib Module”. We'll look at slightly lower-level protocols in the section called “Mid-Level Protocols: The urllib2 Module”. Finally, we'll show how you can use low-level sockets in the section called “Socket Programming”. Generally, you can almost always leverage an existing protocol; but it's still relatively simple to invent your own.

About TCP/IP

The essence of TCP/IP is a multi-layered view of the world. This view separates the mechanics of operating a simple Local Area Network (LAN) from the interconnection between networks, called *internetworking*.

The lowest level of network services are provided by mechanisms like Ethernet (see the IEEE 802.3 standards) that covers wiring between computers. The Ethernet standards include things like 10BaseT (for twisted pairs of thin wires), 10Base2 (for thicker coaxial cabling). Network services may also be wireless, using the IEEE 802.11 standards. In all cases, though, these network services provide for simple naming of devices and moving bits from device to device. These services are limited by having to know the hardware name of the receiving device; usually called the MAC address.

When you buy a new network card for your computer, you change your computer's hardware name.

The TCP/IP standards put several layers of control on top of these data passing mechanisms. While these additional layers allow interconnection between networks, they also provide a standard library for using all of the various kinds of network hardware that is available. First, the Internet Protocol (IP) standard specifies addresses that are independent of the underlying hardware. The IP also breaks messages into packets and reassembles the packets in order to be independent of any network limitations on transmission lengths. The IP standard specifies how to handle errors. Additionally, the IP standard specifies how to route packets among networks, allowing packets to pass over bridges and routers between networks. Finally, IP provides a formal Network Interface Layer to divorce IP and all higher level protocols from the mechanics of the actual network.

The Transport Control Protocol (TCP) protocol relies on IP. It provides a reliable stream of bytes from one application process to another. It does this by breaking the data into packets and using IP to route those packets from source to receiver. It also uses IP to send status information and retry lost or corrupted packets. TCP keeps complete control so that the bytes that are sent are recieved exactly once and in the correct order.

Many applications, in turn, depend on the TCP/IP protocol capabilities. The Hypertext Transport Protocol (HTTP), used to view a web page, works by creating a TCP/IP connection (called a *socket*) between browser and web server. A request is sent from browser to web server. The web server responds to the browser request. When the web page content is complete, the socket is closed and the socket connection can be discarded.

Python provides a number of complete client protocols that are built on TCP/IP in the following modules: `urllib`, `httplib`, `ftplib`, `gopherlib`, `poplib`, `imaplib`, `nntplib`, `smtplib`, `telnetlib`. Each of these exploits one or more protocols in the TCP/IP family, including HTTP, FTP, GOPHER, POP, IMAP, NNTP, SMTP and Telnet. The `urllib` and `urllib2` modules make use of multiple protocols, including HTTP and FTP, which are commonly provided by web servers.

We'll start with the high-level procotols: HTTP and how this serves web pages for people. We'll look at using this to create a web service, also. Then we'll look at lower-level protocols like FTP. Finally, we'll look at how Python deals with the low-level socket abstraction for network communications. Then we'll look at some higher-level modules that depend on sockets implicitly.

Web Servers and the HTTP protocol

The most widely-used protocol is probably HTTP. It is the backbone of the World Wide Web. The HTTP protocol defines two parties: the client (or browser) and the server. The browser is generally some piece of software like FireFox, Opera or Safari. These products handle half the HTTP conversation.

A web server handles the other half of the HTTP conversation. We have a number of choices of how to handle this.

- We can write our own from scratch. Python provides us three seed modules from which we can build a working server. In some applications, where the volume is low, this is entirely appropriate. See the `BaseHTTPServer`, `SimpleHTTPServer` and `CGIHTTPServer` modules.

- We can plug into the Apache server. Apache supports a wide variety of plug-in technologies, including CGI, SCGI, FCGI.

- We can plug into Apache using ModPython. The ModPython project on Source

Forge contains a module which embeds a Python interpreter directly in Apache. This embedded interpreter then runs your Python programs as part of Apache's response to HTTP requests. This is very secure and very fast.

- We can use a web framework. In this case, the web framework plugs into Apache, and the framework handles much of the details of a web request. Using a web framework means that we do much, much less programming. Python has dozens of popular, successful web frameworks. You can look at Zope, Django and TurboGears for just three examples of dozens of ways that the Python community has simplified the construction of web applications.

We can't easily cover ModPython or any of the web frameworks in this book. But we can take a quick look at `SImpleHTTPServer`, just to show what's involved in HTTP. We'll leverage this example to handle some additional requests in subsequent sections.

About HTTP

There are two versions of HTTP, both widely used. Version 1.0 doesn't include cookies or some other features that are essential for modern, interactive web applications. Version 1.1, however, requires some more care in creating the response to the web browser.

An HTTP request includes a number of pieces of information. A few of these pieces of information are of particular interest to a web application.

command

The command is generally GET or POST. There are other commands specified in the protocol (like HEAD or INDEX), but they are rarely provided by browsers.

path

This is the path (after the host name and port). This can include a query string, separated from the main part of the URI by a "?".

headers

There are a number of headers which are included in the query; these describe the browser, and what the browser is capable of. The headers summarize some of the browser's preferences, like the language which is preferred. They also describe any additional data that is attached to the request. The "content-length" header, in particular, tells you that form input or a file upload is attached.

An HTTP reply includes a number of pieces of information. It always begins with a MIME-type string that tells the browser what kind of document will follow. This string us often `TEXT/HTML` or `TEXT/PLAIN`. The reply also includes the status code and a number of headers. Often the headers are version infromation that the browser can reveal via the Page Info menu item in the browser. Finally, the reply includes the actual document, either plain text, HTML or an image.

There are a number of HTTP status codes. Generally, a simple page includes a status code of 200, indicating that request is complete, and the page is being sent. The 30x status codes indicate that the page was moved, the `"Location"` header provides the URL to which the browser will redirect. The 40x status codes indicate problems with the request. The 50x status codes indicate problems with the server, problems that might clear up in the future.

Building an HTTP Server

Your HTTP server has two parts. The control of services in general is handled by `BaseHTTPServer.HTTPServer`. This class has two methods that are commonly used. In

the following examples, `srvr` is an instance of `BaseHTTPServer.HTTPServer`.

`HTTPServer(`*`addresshandlerClass`*`)`

> The `address` is a two-tuple, with server name and port number, usually something like `('',8008)`. The handlerClass is the name of a subclass of `BaseHTTPServer.BaseHTTPRequestHandler`. This server will create an instance of this class, and invoke appropriate methods of that class to serve the requests.

`srvr.handle_request`

> This method of a server will handle just one request. It's handy for debugging.

`srvr.serve_forever`

> This method of a server will handle requests until the server is stopped.

The server requires a subclass of `BaseHTTPServer.BaseHTTPRequestHandler`. The base class does a number of standard operations related to handling web service requests. Generally, you'll need to override just a few methods. Since most browsers will only send `GET` or `POST` requests, you only need to provide `do_GET` and `do_POST` methods. For each request, you'll need to provide a matching `do_X` method function.

`do_GET`

> Handle a GET request from a browser.

`do_POST`

> Handle a POST request from a browser.

This class has several class variables. Generally, you'll want to override `server_version`, with some identification for your server. There are some additional class variables that you might want to use. When using these, the instance qualifier, `self.`, is required.

self.server_version

> A string to identify your server and version. This string can have multiple clauses, each separated by whitespace. Each clause is of the form product/version. The default is `'BaseHTTP/0.3'`.

self.sys_version

> This a version string to identify the overall system. It has the form product/version. The default is `'Python/2.5.1'`.

self.error_message_format

> This is the web page to send back by the send_error method. The send_error method uses the error code to create a dictionary with three keys: `"code"`, `"message"` and `"explain"`. The `"code"` item in the dictionary has the numeric error code. The `"message"` item is the short message from the `self.responses` dictionary. The "explain" method is the long message from the `self.responses` dictionary. Since a dictionary is provided, the formatting string for his error message can include conversion strings: `%(code)d`, `%(message)s` and `%(explain)s`.

self.protocol_version

> This is the HTTP version being used. This defaults to `'HTTP/1.0'`. If you set this to `'HTTP/1.1'`, then you should also use the `"Content-Length"` header to

provide the browser with the precise size of the page being sent.

self.responses

A dictionary, keyed by status code. Each entry is a two-tuple with a short message and a long explanation. The message for status code 200, for example, is `'OK'`. The explanation is somewhat longer.

This class has a number of instance variables which characterize the specific request that is currently being handled. These are proper instance variables, so the instance qualifier, `self.`, is required.

self.client_address

An internet address as used by Python. This is a 2-tuple: (host address, port number).

self.command

The command in the request. This will usually be GET or POST.

self.path

The requested path.

self.request_version

The protocol version string sent by the browser. Generally it will be `'HTTP/1.0'` or `'HTTP/1.1'`.

self.headers

This is a collection of headers, usually an instance of `mimetools.Message`. This is a mapping-like class that gives you access to the individual headers in the request. The header `"cookie"`, for instance, will have the cookies being sent back by the browser. You will need to decode the value of the cookie, usually using the `Cookie` module.

self.rfile

If there is an input stream, this is a file-like object that can read that stream. Do not read this without providing a specific size to read. Generally, you want to get `headers['Content-Length']` and read this number of bytes. If you do not specify the number of bytes to read, and there is no supplemental data, your program will wait for data on the underlying socket. Data which will never appear.

self.wfile

This is the response socket, which the browser is reading. The response protocol requires that it be used as follows:

1. Use `self.send_response( number )` or `self.send_response( number, text )`. Usually you simply send 200.

2. Use `self.send_header( header, value )` to send specific headers, like `"Content-type"` or `"Content-length"`. The `"Set-cookie"` header provides cookie values to the browser. The `"Location"` header is used for a 30x redirect response.

3. Use `self.end_headers()` to finish sending headers and start sending the resulting page.

4. Then (and only then) you can use `self.wfile.write` to send the page content.

5. Use `self.wfile.close()` if this is a HTTP/1.0 connection.

This class has a number of methods which you'll want to use from within your `do_GET` and `do_POST` methods. Since these are used from within your methods, we'll use `self.` as the instance qualifier.

`self.send_error`*number*, ⟨*message*⟩

Send an error response. By default, this is a complete, small page that shows the code, message and explanation. If you do not provide a `message`, the short message from the `self.responses[`*number*`]` mapping will be used.

`self.send_response`*number*, ⟨*message*⟩`()`

Sends a response in pieces. If you do not provide a `message`, the short message from the `self.responses[`*number*`]` mapping will be used. This method is the first step in sending a response. This must be followed by `self.send_header` if any headers are present. It must be followed by `self.end_headers`. Then the page content can be sent.

`self.send_header`*name*, *value*

Send one HTTP header and it's value. Use this to send specific headers, like `"Content-type"` or `"Content-length"`. If you are doing a redirect, you'll need to include the `"Location"` header.

`self.end_headers`

Finish sending the headers; get ready to send the page content. Generally, this is followed by writing to `self.wfile`.

`self.log_request`*status*, ⟨*size*⟩

Uses `self.log_message` to write an entry into the log file fo\r a normal response. This is done automatically by `send_headers`.

`self.log_error`*format*, *args...*

Uses `self.log_message` to write an entry into the log file for an error response. This is done automatically by `send_error`.

`self.log_message`*format*, *args...*

Writes an entry into the log file. You might want to override this if you want a different format for the error log, or you want it to go to a different destination than `sys.stderr`.

Example HTTP Server

The following example shows the skeleton for a simple HTTP server. This sever merely displays the GET or POST request that it receives. A Python-based web server can't ever be fast enough to replace Apache. However, for some applications, you might find it convenient to develop a small, simple application which handles HTTP.

Example 36.1. webserver.py

```
import BaseHTTPServer
```

```
class MyHandler( BaseHTTPServer.BaseHTTPRequestHandler ):
    server_version= "MyHandler/1.1"
    def do_GET( self ):
        self.log_message( "Command: %s Path: %s Headers: %r"
                          % ( self.command, self.path, self.headers.items() ) )
        self.dumpReq( None )
    def do_POST( self ):
        self.log_message( "Command: %s Path: %s Headers: %r"
                          % ( self.command, self.path, self.headers.items() ) )
        if self.headers.has_key('content-length'):
            length= int( self.headers['content-length'] )
            self.dumpReq( self.rfile.read( length ) )
        else:
            self.dumpReq( None )
    def dumpReq( self, formInput=None ):
        response= "<html><head></head><body>"
        response+= "<p>HTTP Request</p>"
        response+= "<p>self.command= <tt>%s</tt></p>" % ( self.command )
        response+= "<p>self.path= <tt>%s</tt></p>" % ( self.path )
        response+= "</body></html>"
        self.sendPage( "text/html", response )
    def sendPage( self, type, body ):
        self.send_response( 200 )
        self.send_header( "Content-type", type )
        self.send_header( "Content-length", str(len(body)) )
        self.end_headers()
        self.wfile.write( body )

def httpd(handler_class=MyHandler, server_address = ('', 8008), ):
    srvr = BaseHTTPServer.HTTPServer(server_address, handler_class)
    srvr.handle_request() # serve_forever

if __name__ == "__main__":
    httpd( )
```

❶ You must create a subclass of `BaseHTTPServer.BaseHTTPRequestHandler`. Since most browsers will only send `GET` or `POST` requests, we only provide `do_GET` and `do_POST` methods. Additionally, we provide a value of `server_version` which will be sent back to the browser.

❷ The HTTP protocol allows our application to put the input to a form either in the URL or in a separate data stream. Generally, a forms will use a POST request; the data is available

❸ This is the start of a debugging routine that dumps the complete request. This is handy for learning how HTTP works.

❹ This shows the proper sequence for sending a simple page back to a browser. Thi s technique will work for files of all types, including images. This method doesn't handle complex headers, particularly cookies, very well.

❺ This creates the server, `srvr`, as an instance of `BaseHTTPServer.HTTPServer` which uses `MyHandler` to process each request.

Web Services: The `xmlrpclib` Module

When we looked at HTTP in the section called “Web Servers and the HTTP protocol”, we were interested in its original use case of serving web pages for people. We can build on HTTP, creating an interface between software components, something called a *web service*. A web service leverages the essential request-reply nature of HTTP, but takes the elaborate human-centric HTML web page out of the response. Instead of sending back something for people to read, web services use XML to send just the facts.

Web services allow us to have a multi-server architecture. A central web server provides interaction with people. When a person's browser makes a request, this central web server can make web service requests of other servers to gather the information. After gathering the information, the central server aggregates it and builds the final HTML-based presentation, which is the reply sent to the human user.

Web services are an adaptation of HTTP; see the section called “About HTTP” for a summary. Web services rely on a number of other technologies. There are several competing alternatives, so we'll look at web services in general before looking at the `xmlrpclib` module in particular.

Web Services Overview

There are a number of ways of approaching the problem of coordinating work between clients and servers. All of these alternatives have their advantages and disadvantages.

XML-RPC

> The XML-RPC protcol uses XML notation to make a remote procedure call (RPC). It works by sending an HTTP request that contains the name of the procedure to call and the arguments to that procedure. This protocol uses HTTP "POST" requests to provide the XML document.

SOAP

> There are two variations on the Simple Object Access Protocol (SOAP): remote procedure call variation and document. The RPC variant is basically the next generation of XML-RPC, where an XML document encodes the name of the procedure and the arguments. The document variant merely sends an XML document; the document provides all the information required by the server. This protocol is heavily supported by additional standards like Web Services Definition Language (WSDL).

REST

> The Representational State Transfer (REST) protcol uses the HTTP operations (POST, GET, PUT, DELETE) and Uniform Resource Identifiers (URI) to manipulate remote objects. This protocol is perhaps the simplest of the web services protocols; for this reason it is very popular.

We'll focus on the XML-RPC since it is relatively simple and well-supported by Python. REST is also very popular because it can be done largely using `urllib2` features (see the section called “Mid-Level Protocols: The `urllib2` Module”).

The essence of RPC is that we are calling a procedure that resides on another, remote computer. In order to do this, the argument values on our local computer must be marshalled and sent through the internet to this remote service. The service must unmarshall the argument values, evaluate the procedure, marshall the results, and send them back to the original requester. Finally, the requester must unmarshall the response.

We have, therefore, three separate issues that we have to address.

1. Packaging the data. This means writing the argument values in XML notation. We'll see how the `xmlrpclib` module handles this transformation between XML and Python. The Simple Object Access Protocol (SOAP) is an alternative to the XMLRPC approach to sending objects from one computer to another; it is not widely used in the Python community.

2. Making the client request. This means marshalling the arguments, making the request, and unmarshalling the response. Since this is based on HTTP, this is a kind of HTTP client, akin to what a browser does when it makes a POST request.

3. Serving requests. This means unmarshalling arguments, doing something useful, and marshalling a response. Since this is based on HTTP, this is handling a POST request with an XML request, and providing an XML reply.

The essential ingredient in making RPC work is to have a local object which acts as a

proxy for the remote service. This `ServerProxy` appears as if it is doing the work. In fact, it is merely marshalling arguments, transmitting the request via HTTP and unmarshalling the response.

Web Services Client

Let's imagine that a colleague has built a web service which provides us with an extremely good simulation of a roulette wheel. (We'll actually build this in the next section.) Our colleague has provided us with the following summary of this web service.

host

> 10.0.1.5. While IP address numbers are the lowest-common denominator in naming, some people will create Domain Name Servers (DNS) which provide interesting names instead of numeric addresses.

port number

> 8008. While the basic HTTP service is defined to run on port 80, you may have other web services which, for security reasons, aren't available on port 80. Port numbers from 1024 and up may be allocated for other purposes, so port numbers are often changed as part of the configuration of a program.

path

> /. Your HTTP handler may have different families or collections of web services, each with a different path.

method name

> spin

return

> Python tuple with the outcome. The tuple includes number, color, even/odd, high/low for the result.

To create a web services client, we can use the `xmlrpclib` module to access an XML-RPC protocol web service. We'll need to define a proxy for this service.

Example 36.2. wheelclient.py

```
#!/usr/bin/env python
""" Quick Demo of the spin service.
"""
import xmlrpclib

server= xmlrpclib.ServerProxy( "http://10.0.1.5:8008/" )
for i in range(10):
    print server.spin()
```

❶ We import the `xmlrpclib` module.

❷ We synthesize the interface information (protocol, host, port and path) into a URI which identifies the web service. This statement creates a local object that appears to have all of the methods that are part of the remote service.

❸ We evalulate the spin method on the remote server, and we get back a result that we simply print. In this case, we expect to get a tuple with number, color, even/odd and high/low attributes of the number.

There are some limitations on what kind of structures can be marshalled by the XML-RPC protocol. For example, Python makes a distinction between tuple and list. The XML-RPC protocol, however, can only create lists.

Web Services Server

A web services is usually built into a more complete web application framework. Often the server will have a human interface as well as a web service interface. The human interface will use HTML and port 80. The web service interface will use XML and some other port number, usually a number above 8000. Since web services are built in HTTP, we can adapt our `SimpleHTTPServer` example toward providing web services.

A common technique is to have a single server process that includes a dispatcher. The dispatch method examines the path of the request to determine which group of web services are being invoked. In this example, we won't include any path dispatching.

The following example shows how to implement the wheel service.

Example 36.3. wheelservice.py

```
#!/usr/bin/env python
"""Wheel Server.
"""
import SimpleXMLRPCServer as xmlrpc
import random

class Wheel( object ):
    redSet= [1,3,5,7,9,12,14,16,18,19,21,23,25,27,30,32,34,36]
    def _color( self, number ):
        if number in ("0", "00"): return "GREEN"
        elif int(number) in self.redSet: return "RED"
        else: return "BLACK"
    def _even( self, number ):
        if number in ("0", "00"): return "ZERO"
        elif int(number) % 2 == 0: return "EVEN"
        else: return "ODD"
    def _high( self, number ):
        if number in ("0", "00"): return "ZERO"
        elif int(number) <= 18: return "LOW"
        else: return "HIGH"
    def __init__( self ):
        self.rng= random.Random()
        self.wheel= map( str, range(0,37) ) + ["00"]
    def spin( self ):
        number= self.rng.choice( self.wheel )
        return ( number, self._color(number), self._even(number), self._high(number) )
    def spinList( self, spins=1 ):
        return [ self.spin() for i in range(spins) ]

def server():
    theWheel= Wheel()
    service = xmlrpc.SimpleXMLRPCServer(("", 8008))
    service.register_instance( theWheel )
    service.serve_forever()

if __name__ == "__main__":
    server()
```

1. We import the `SimpleXMLRPCServer` module. Since the name is so long, we provide an alias, `xmlrpc`, to make it easier to type. We also import the `random` module.
1. This class contains the state and methods that we will expose as a web service. We included some private methods, with names prefaced by _. The `SimpleXMLRPCServer` class uses this "leading _" convention to identify a private method that isn't published on the web.
1. We've defined three private methods, `_color`, `_even`, and `_high`. These methods will be used as part of responding to web service requests.
1. We initialize an instance of Wheel by creating a random number generator. We

also initialize the set of numbers on the wheel. In this case, we are folding in double zero to make an American-style wheel. A subclass could modify this initialization to create a European wheel with only a single zero.

❶ The `spin` and `spinList` methods are the public interface to this class. The `spin` method makes a random choice of the numbers on the wheel. It then creates a tuple with the number, the color, the even/odd and the high/low values. The `spinList` method returns a list of tuples by calling `spin` repeatedly.

❷ The `server` function creates and runs the web server. First, it creates the `Wheel` object, `theWheel`, that we'll use. Second, it creates the server, `service`, using the server's web address and port number. The empty string is a special short-hand meaning "this host's IP address". We register all public methods of `theWheel`. The `server_forever` method handles requests until we stop the Python interpreter by killing the process.

❸ We use the main switch so that we can easily reuse this `Wheel` class definition.

Mid-Level Protocols: The `urllib2` Module

A central piece of the design for the World-Wide Web is the concept of a Uniform Resource Locator (URL) and Uniform Resource Identifier (URI). A URL provides several pieces of information for getting at a piece of data located somewhere on the internet. A URL has several data elements. Here's an example URL: `http://www.python.org/download/`.

- A protocol (http)
- A server (www.python.org)
- A port number (80 is implied if no other port number is given)
- A path (download)
- An operation (browsers use GET or POST, some web services use PUT and DELETE, also)

It turns out that we have a choice of several protocols, making it very pleasant to use URL's. The protocols include

- FTP - the File Transfer Protocol. This will send a single file from an FTP server to our client. For example, `ftp://aeneas.mit.edu/pub/gnu/dictionary/cide.a` is the identifier for a specific file.
- HTTP - the Hypertext Transfer Protocol. Amongst other things that HTTP can do, it can send a single file from a web server to our client. For example, `http://www.crummy.com/software/BeautifulSoup/download/BeautifulSoup.py` retrieves the current release of the Beautiful Soup module.
- FILE - the local file protocol. We can use a URL beginning with `file:///` to access files on our local computer.

HTTP Interaction. A great deal of information on the World Wide Web is available using simple URI's. In any well-design web site, we can simply GET the resource that the URL identifies.

A large number of transactions are available through HTTP requests. Many web pages provide HTML that will be presented to a person using a browser.

In some cases, a web page provides an HTML form to a person. The person may fill in a form and click a button. This executes an HTTP `POST` transaction. The `urllib2` module allows us to write Python programs which, in effect, fill in the blanks on a form and submit that request to a web server.

Example. By using URL's in our programs, we can write software that reads local files as well as it reads remote files. We'll show just a simple situation where a file of content can be read by our application. In this case, we located a file provided by an HTTP server and an FTP server. We can download this file and read it from our own local computer, also.

As an example, we'll look at the *Collaborative International Dictionary of English*, CIDE. Here are three places that these files can be found, each using different protocols. However, using the `urrllb2` module, we can read and process this file using any protocol and any server.

FTP

> `ftp://aeneas.mit.edu/pub/gnu/dictionary/cide.a` This URL describes the `aeneas.mit.edu` server that has the CIDE files, and will respond to the FTP protocol.

HTTP

> `http://ftp.gnu.org/gnu/gcide/gcide-0.46/cide.a` This URL names the `ftp.gnu.org` server that has the CIDE files, and responds to the HTTP protocol.

FILE

> `file:///Users/slott/Documents/dictionary/cide.a` This URL names a file on my local computer.

Example 36.4. urlreader.py

```
#!/usr/bin/env python
"""Get the "A" section of the GNU CIDE Collaborative International Dictionary of English
"""
import urllib2

#baseURL= "ftp://aeneas.mit.edu/pub/gnu/dictionary/cide.a"
baseURL= "http://ftp.gnu.org/gnu/gcide/gcide-0.46/cide.a"
#baseURL= "file:///Users/slott/Documents/dictionary/cide.a"

dictXML= urllib2.urlopen( baseURL, "r" )
print len(dictXML.read())
dictXML.close()
```

❶ We import the `urllib2` module.
❷ We name the URL's we'll be reading. In this case, any of these URL's will provide the file.
❸ When we open the URL, we can read the file.

Client-Server Exercises

1. **HTTP Client.** Write a simple client that can make HTTP requests. An HTTP request consists of three elements: a *method*, URI and a protocol name. The request header may be followed by one or more lines of header fields. RFC 2068 has all of the details of this protocol.

   The OPTIONS request returns information about the server.

   ```
   OPTIONS * HTTP/1.1
   ```

   The GET request returns a particular HTML page from a server.

   ```
   GET http://www.python.org/index.html HTTP/1.1
   ```

The following request is two lines, with a header field that specifies the host.

```
GET /pub/WWW/ HTTP/1.1
Host: www.w3.org
```

Your client will need to open a socket on port 80. It will send the request line, and then read and print all of the reply information.

2. **FTP Client.** Write a simple, special-purpose FTP client that establishes a connection with an FTP server, gets a directory and ends the connection. The FTP directory commands are "DIR" and "LS". The responses may be long and complex, so this program must be prepared to read many lines of a response.

   For more information, RFC 959 has complete information on all of the commands an FTP server should handle. Generally, the DIR or LS command, the GET and PUT commands are sufficient to do simple FTP transfers.

   Your client will need to open a socket on port 21. It will send the command line, and then read and print all of the reply information. In many cases, you will need to provide additional header fields in order to get a satisfactory response from a web server.

3. **Stand-Alone Web Application.** You can easily write a desktop application that uses web technology, but doesn't use the Internet. Here's how it would work.

   - Your application is built as a very small web server, based on `BaseHTTPServer.HTTPServer`. This application prepares HTML pages and forms for the user.
   - The user will interact with the application through a standard browser like Firefox, Opera or Safari. Rather than connect to a remote web server somewhere in the Internet, this browser will connect to a small web server running on your desktop.
   - You can package your application with a simple shell script (or `.BAT` file) which does two things. (This can also be done as a simple Python program using the `subprocess` module.)

     a. It starts a subprocess to run the HTTP server side of your application.

     b. It starts another subprocess to run the browser, providing an initial link of `'http://localhost:8008'` to point the browser at your application server.

   This assures that your desktop application has a web browser look and feel.

   The Input Form. While the full power of HTML is beyond the scope of this book, we'll provide a simple form using the `<form>`, `<label>`, `<button>` and `<input>` tags. Here's an example of a simple form. This form will send a POST request to the path `/response` when the user clicks the Submit button. The input will include two name-value pairs with keys of `field` (from the `<input type="text">`) and `action` (from the `<button type="submit">`).

```
<html><head><title>Test</title><head>
<body><form action="/response" method="POST">
<label>Field</label> <input type="text" name="field"/>
<br/>
<button type="submit" name="action" value="submit">Submit</button>
</form>
</body>
</html>
```

An Overview of the Server. You'll create a subclass of `BaseHTTPServer.BaseHTTPRequestHandler` which provides definitions for `do_GET` and `do_POST` methods.

Your main script will create an instance of `BaseHTTPServer.HTTPServer`, provide it an address like `('', 8008)`, and the name of your subclass of `BaseHTTPRequestHandler`. This object's `serve_forever` method will then handle HTTP requests on port 8008.

Your subclass of `BaseHTTPRequestHandler` will have `do_GET` and `do_POST` methods. Both methods need to examine the path to determine the name of the form they are responding to. For example, if you are going to do a celsius to farenheit conversion, a GET for a path of `/farenheit` would present a form that accepts a celsius temperature. A POST for a path of `/farenheit` would check that the input is a number, and would produce a form with the farenheit value.

Handling Form Input. By making our form's method is `"POST"`, we've assured that the `do_POST` method will be called when someone clicks the Submit button. The user's input is made available to your program in the form of a data stream, available from the socket through a file object, `self.rfile`. The `do_POST` method must read the data waiting in this file.

Because of the HTTP protocol rules, this socket is left open. It cannot be simply read to end of file. The `do_POST` method must first determine how much data is there, then read just that number of bytes and no more.

The `formText` received from the browser will contain the user's input encoded as a single, long string that includes all of the form's field (and button) names and input values. The `cgi` module has functions for parsing this. Specifically `cgi.parse_qs` and `cgi.parse_multipart` functions handle this parsing nicely.

```
length= int( self.headers['content-length'] )
formText= self.rfile.read( length )
formInput= cgi.parse_qs( formText )
```

The resulting `formInput` object is a dictionary with all of the various input fields and button names as keys.

Sending HTML Output. The response from a web server is a kind of MIME message. It has a status line, some headers, and the "payload" or content from the web server. There are some methods defined in `BaseHTTPRequestHandler` that help us create this response correctly. Specifically, `send_response`, `send_header` and `end_headers` are used to create the message header, including the proper MIME type.

Here's an example of the correct way to send an HTML page back to a browser. It shows just two headers, `content-type` and `content-length`. This example also shows where any cookie information would be included in the headers. The `Cookie` module helps to encode and decode cookies.

```
self.send_response( 200 )
self.send_header( "Content-type",  type )
self.send_header( "Content-length", str(len(body)) )
self.wfile.write( theCookie.output() + "\n" )
self.end_headers()
self.wfile.write( body )
```

When the content type is "text/html", the body is expected to be an HTML document; the browser will format the document using the HTML markup. For a content type of "text/plain" the browser will not format the document. Other content types will lead to other kinds of processing. For example, a content type

of "image/jpeg" would describe a JPEG file being sent as the response, the browser will display the image. A content type of "audio/x-aiff" would describe an AIFF file being sent; most browsers will start playing the audio file.

4. **Roulette Server.** We'll create an alternative implementation of the simple Roulette server shown in the section called "Web Services Server". Rather than use web services, we'll create a new protocol and use sockets to interact with the server.

   Our new protocol will involve a stream of commands from the client to the server. This protocol will be limited to a single client at a time -- it isn't appropriate for games like chess or poker where there will be multiple, interacting clients.

   We'll define three commands that a client can send to the server.

   The `bet` command will define a bet placed by a client. It will save the proposition and the amount being bet. The proposition is any of the available Roulette bets: a specific number, a color, high or low, even or odd. If the bet and amount are valid, the server should respond with a "201" message. For example, `"201 Bet $5.00 BLACK accepted"`. If the bet or amount are invalid, the server should respond with a "501" message. For example, `"501 'BET asd EVEN' has invalid amount"`. Additionally, a bet over the table maximum will also get a "501" message.

   The client can send as many bet requests as necessary. For example `"BET BLACK 5"` and `"BET EVEN 5"`. Each bet will receive a confirmation. The table minimum and maximum might be $10 and $500.

   The command to spin the wheel will be `spin`. There are two possible responses. If the total value of the bets is less than the table minimum, a "401" status is sent with a message that includes the table minimum and the amount bet. For example `"401 only $8.00 bet at a $10.00 table"`. If the total of the bets is greater than the table minimum, a "203" status code is sent. This message includes the result, and the list of bets placed and the result for each bet. This can be quite a long message. For example, it might be `"203 Spin (14, RED, LOW, EVEN). Bet $5.00 on BLACK: lose. Bet $5 on EVEN: win."`

   Checking the status of bets will be `show`. This command will get an 202 status code with all of the possible bets and the amounts placed. This can be a single long line of data, or multiple lines using indentation to show that this is a multiple-line response. For example `"202 EVEN 0, BLACK 5, ODD 0, EVEN 5, HIGH 0, LOW 0"`.

   We can leverage the `handle` method of a handler to parse the input command. We would send a "505" response if the message cannot be interpreted.

```
class RouletteHandler( StreamRequestHandler ):
    def handle():
        input= self.rfile.read()
        if input[:3] == "bet":
            reply= self.placeBet(input)
        elif input[:4] == "show":
            reply= self.showBets(input)
        elif input[:4] == "spin":
            reply= self.spinWheel()
        else:
            reply= "505 unknown request: " + input
        self.wfile.write(reply)
```

   You'll need to assign this to some suitable port number, for example, 36000.

5. **Roulette Client.** Write a simple client which places a number of bets on the

Roulette server, using the Martingale betting strategy. The strategy is relatively simple. First, each bet will be placed on just one proposition (Red, Black, Even, Odd, High or Low). A base betting amount (the table minimum) is used. If a spin is a winner, the betting amount is reset to the base. If a spin is a loser, the betting amount is doubled.

Note that bet doubling will lead to situations where the ideal bet is beyond the table maximum. In that case, your simulation can simply stop playing, or adjust the bets to be the table maximum.

6. **Roulette Web Application.** We can write a web application which uses our Roulette Server. This will lead to a fairly complex (but typical) architecture, with two servers and a client.

    - We'll have the Roulette Server from exercise 4, running on port 36000. This server accepts bets and spins the wheel on behalf of a client process. It has no user interaction, it simply maintains state, in the form of bets placed.

    - We'll have a web server, similar to exercise 3, running on port 8008. This application can present a simple form for placing a bet or spinning the wheel. If the user filled in the fields and clicked the Bet button, the web application will make a request to the Roulette Server and present the results in the HTML page that is returned to the user. If the user clicked the Spin button, the web application will make a request to the Roulette Server and present th eresults in the HTML page. This application will include a large amount of fancy HTML presentation.

    - We'll use a browser to contact our web server. This client will browse "http://localhost:8008" to get a web page with a simple form for placing a bet or spinning the wheel.

    A simple HTML form might look like the following.

```
<html><head><title>Roulette</title></head>
<body>
<p>Results from previous request go here</p>
<form action="/roulette" method="POST">
<label>Amount</label> <input type="text" name="amount"/>
<br/>
<label>Proposition</label> <select name="proposition">
<option>Red</option>
<option>Black</option>
<option>Even</option>
<option>Odd</option>
<option>High</option>
<option>Low</option>
</select>
<br/>
<button type="submit" name="action" value="bet">Bet</button>
<button type="submit" name="action" value="spin">Spin</button>
</form>
</body>
</html>
```

    Your web server will see requests with a path of `"/roulette"`, the action on the form. The input will include three name/value pairs, named `amount`, `proposition` and `action`. When the `action` field is "bet", then the `amount` and `proposition` will define the bet request sent to the Roulette server. When the action field is "spin", then the spin request should be sent to the Roulette server.

7. **Chess Server.** In the section called "Chessboard Locations" we described some of the basic mechanics of chess play. A chess server would allow exactly two clients

to establish a connection. It would then a chess moves from each client and respond to both clients with the new board position.

We'll create a web service, using XML-RPC, that has a number of methods for handling chess moves. To do this, we'll need to create a basic ChessBoard class which has a number of methods that establish players, move pieces, and report on the board's status.

It's essential that the ChessBoard be a single object that maintains the state of the game. When two players are connected, each will need to see a common version of the chessboard.

Here are some of the methods that are essential to making this work.

`__ini__`

> This method will initialize the chessboard with all pieces in the starting position. It will also create two variables to store the player names. These two player names will initially be None, since no player has connected.

`connect(`*`name`*`)` → message string

> This method will allow a player to connect to the chess server. If two players are already connected, this will return an Error message. Otherwise, this will return an acceptance message that includes the player's assigned color (White or Black.)

`move(`*`player`*`, `*`from`*`, `*`to`*`)` → message string

> A request by a player to move a piece from one location to another location. Both locations are simply file (a-h) and rank (1-8). If the move is allowed, the response is an acknowledgement. Otherwise, an error response is sent. The chess server will need to track the moves to be sure they players are alternating moves properly.

`board` → position string

> The response is a 128-character string that reports the state of the board. It has each rank in order from 8 to 1; within each rank is each file from a to h. Each position is two characters with a piece code or spaces if the position is empty.

A client process for this web service will attempt a connection. If that succeeds, the client will provide moves from the given player. Periodically, the client will also ask for the status of the board.

If the client program is a desktop application, it may poll the chess server's `board` method fairly often to provide a response to one player as soon as the other player has moved.

If the client program is a web application, there may be a JavaScript prorgam which is polling the board periodically. The HTML may include an automatic refresh via `<meta http-equiv="refresh" content="120">` with the `<head>` tags. Or the user may be expected to refresh the page manually to check for any change to the board.

Socket Programming

Socket-level programming isn't our first choice for solving client-server problems. Sockets are nicely supported by Python, however, giving us a way to create a new protocol when the vast collection of existing internetworking protocols are inadequate.

Client-server applications include a client-side program, a server, a connection and a protocol for communication betweem the two processes. One of the most popular and enduring suite of client-server protocols is based on the Internetworking protocol: TCP/IP. For more information in TCP/IP, see *Internetworking with TCP/IP* [Comer95].

All of the TCP/IP protocols are based on the basic *socket*. A socket is a handy metaphor for the way that the Transport Control Protocol (TCP) reliably moves a stream of bytes between two processes.

The `socket` module includes a number of functions to create and connect sockets. Once connected, a socket behaves essentially like a file: it can be read from and written to. When we are finished with a socket, we can close it, releasing the network resources that were tied up by our processing.

Client Programs

When a client application communicates with a server, the client does three things: it establishes the connection, it sends the request and it reads the reply from the server. For some client-server relationships, like a databsae server, there may be multiple requests and replies. For other client-server requests, for example, the HTTP protocol, a single request may involve a number of replies.

To establish a connection, the client needs two basic facts about the server: the IP address and a port number. The IP address identifies the specific computer (or host) that will handle the request. The port number identifies the application program that will process the request on that host. A typical host will respond to requests on numerous ports. The port numbers prevent requests from being sent to the wrong application program. Port numbers are defined by several standards. Examples include FTP (port 21) and HTTP (port 80).

A client program makes requests to a server by using the following outline of processing.

1. **Develop the server's address.** Fundamentally, an IP address is a 32-bit host number and a 16-bit port number. Since these are difficult to manage, a variety of coding schemes are used. In Python, an address is a 2-tuple with a string and a number. The string representes the IP address in dotted notation (`"194.109.137.226"`) or as a domain name (`"www.python.org"`); the number is the port number from 0 to 65535.

2. **Create a socket and connect it to this address.** This is a series of function calls to the `socket` module. When this is complete, the socket is connected to the remote IP address and port and the server has accepted the connection.

3. **Send the request.** Many of the standard TCP/IP protocols expect the commands to be sent as strings of text, terminated with the `\n` character. Often a Python file object is created from the socket so that the complete set of file method functions for reading and writing are available.

4. **Read the reply.** Many of the standard protocols will respond with a 3-digit numeric code indicating the status of the request. We'll review some common variations on these codes, below.

Developing an Address. An IP address is numeric. However, the Internet provides *domain names*, via Domain Name Services (DNS). This permits useful text names to be associated with numeric IP addresses. We're more used to `"www.python.org"`. DNS resolves this to an IP address. The `socket` module provides functions for DNS name resolution.

The most common operation in developing an address is decoding a host name to create

the numeric IP address. The `socket` module provides several functions for working with host names and IP addresses.

`gethostname` → string

> Returns the current host name.

`gethostbyname` (*`host`*) → address

> Returns the IP address (a string of the form '255.255.255.255') for a host.

`gethostbyaddr` (*`address`*) → (name, aliaslist, addresslist)

> Return the true host name, a list of aliases, and a list of IP addresses, for a host. The host argument is a string giving a host name or IP number.

`getservbyname` (*`servicename`*, *`protocolname`*) → integer

> Return a port number from a service name and protocol name. The protocol name should be 'tcp' or 'udp'.

Typically, the `socket.gethostbyname` function is used to develop the IP address of a specific server name. It does this by makig a DNS inquiry to transform the host name into an IP address.

Port Numbers. The port number is usually defined by your application. For instance, the FTP application uses port number 21. Port numbers from 0 to 1023 are assigned by RFC 1700 standard and are called the *well known ports*. Port numbers from 1024 to 49151 are available to be registered for use by specific applications. The Internet Assigned Numbers Authority (IANA) tracks these assigned port numbers. See http://www.iana.org/assignments/port-numbers. You can use the private port numbers, from 49152 to 65535, without fear of running into any conflicts. Port numbers above 1024 may conflict with installed software on your host, but are generally safe.

Port numbers below 1024 are restricted so that only priviledged programs can use them. This means that you must have root or administrator access to run a program which provides services on one of these ports. Consequently, many application programs which are not run by root, but run by ordinary users, will use port numbers starting with 1024.

It is very common to use ports from 8000 and above for services that don't require root or administrator privileges to run. Technically, port 8000 has a defined use, and that use has nothing to do with HTTP. Port 8008 and 8080 are the official alternatives to port 80, used for developing web applications. However, port 8000 is often used for web applications.

The usual approach is to have a standard port number for your application, but allow users to override this in the event of conflicts. This can be a command-line parameter or it can be in a configuration file.

Generally, a client program must accept an IP address as a command-line parameter. A network is a dynamic thing: computers are brought online and offline constantly. A "hard-wired" IP address is an inexcusable mistake.

Create and Connect a Socket. A socket is one end of a network connection. Data passes bidirectionally through a socket between client and server. The `socket` module defines the `SocketType`, which is the class for all sockets. The `socket` function creates a socket object.

`socket` (*`family`*, *`type`*, [*`protocol`*]) → `SocketType`

> Open a socket of the given type. The *`family`* argument specifies the address

family; it is normally `socket.AF_INET`. The *`type`* argument specifies whether this is a TCP/IP stream (`socket.SOCK_STREAM`) or UDP/IP datagram (`socket.SOCK_DGRAM`) socket. The protocol argument is not used for standard TCP/IP or UDP/IP.

A `SocketType` object has a number of method functions. Some of these are relevant for server-side processing and some for client-side processing. The client side method functions for establishing a connection include the following. In each definition, the variable `s` is a socket object.

`s.connect (`*`address`*`)`

Connect the socket to a remote address; the address is usually a (host address, port #) tuple. In the event of a problem, this will raise an exception.

`s.connect_ex (`*`address`*`)` → integer

Connect the socket to a remote address; the address is usually a (host address, port #) tuple. This will return an error code instead of raising an exception. A value of 0 means success.

`s.fileno` → integer

Return underlying file descriptor, usable by the `select` module or the `os.read` and `os.write` functions.

`s.getpeername` → address

Return the remote address bound to this socket; not supported on all platforms.

`s.getsockname` → address

Return the local address bound to this socket.

`s.getsockopt (`*`level`*`, `*`opt`*`, [ `*`buflen`*` ] )` → string

Get socket options. See the UNIX man pages for more information. The level is usually `SOL_SOCKET`. The option names all begin with `SO_` and are defined in the module. You will have to use the `struct` module to decode results.

`s.setblocking (`*`flag`*`)`

Set or clear the blocking I/O flag.

`s.setsockopt (`*`level`*`, `*`opt`*`, `*`value`*`)`

Set socket options. See the UNIX man pages for more information. The *`level`* is usual `SOL_SOCKET`. The option names all begin with `SO_` and are defined in the module. You will have to use the `struct` module to encode parameters.

`s.shutdown (`*`how`*`)`

Shutdown traffic on this socket. If how is 0, receives are disallowed; if how is 1, sends are disallowed. Usually this is 2 to disallow both reads and writes. Generally, this should be done before the `close`.

`s.close`

Close the socket. It's usually best to use the `shutdown` method before closing the socket.

Sending the Request and Receiving the Reply. Sending requests and processing

replies is done by writing to the socket and reading data from the socket. Often, the response processing is done by reading the `file` object that is created by a socket's `makefile` method. Since the value returned by `makefile` is a conventional file, then `readlines` and `writelines` methods can be used on this file object.

A `SocketType` object has a number of method functions. Some of these are relevant for server-side processing and some for client-side processing. The client side method functions for sending (and receiving) data include the following. In each definition, the variable `s` is a socket object.

`s.recv` (*`bufsize`*, [*`flags`*]) → string

> Receive data, limited by *`bufsize`*. *`flags`* are `MSG_OOB` (read out-of-band data) or `MSG_PEEK` (examine the data without consuming it; a subsequent `recv` will read the data again).

`s.recvfrom` (*`bufsize`*, [*`flags`*]) → (string, address)

> Receive data and sender's address, arguments are the same as `recv`.

`s.send` (*`string`*, [*`flags`*]) → (string, address)

> Send data to a connected socket. The `MSG_OOB` flag is supported for sending out-of-band data.

`s.sendto` (*`string`*, [*`flags`*,] *`address`*) → integer

> Send data to a given address, using an unconnected socket. The flags option is the same as `send`. Return value is the number of bytes actually sent.

`s.makefile` (*`mode`*, [*`bufsize`*]) → `file`

> Return a file object corresponding to this socket. The *`mode`* and *`bufsize`* options are the same as used in the built in `file` function.

Example. The following examples show a simple client application using the `socket` module.

This is the `Client` class definition.

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

class Client( object ):
    rbufsize= -1
    wbufsize= 0
    def __init__( self, address=('localhost',7000) ):
        self.server=socket.socket( socket.AF_INET, socket.SOCK_STREAM )
        self.server.connect( address )
        self.rfile = self.server.makefile('rb', self.rbufsize)
        self.wfile = self.server.makefile('wb', self.wbufsize)
    def makeRequest( self, text ):
        """send a message and get a 1-line reply"""
        self.wfile.write( text + '\n' )
        data= self.rfile.read()
        self.server.close()
        return data

print "Connecting to Echo Server"
c= Client()
response= c.makeRequest( "Greetings" )
print repr(response)
print "Finished"
```

A `Client` object is initialized with a specific server name. The host (`"localhost"`) and port number (`8000`) are default values in the class `__init__` function. The address of `"localhost"` is handy for testing a client and a server on your PC. First the socket is created, then it is bound to an address. If no exceptions are raised, then an input and output file are created to use this socket.

The `makeRequet` function sends a message and then reads the reply.

Server Programs

When a server program starts, it creates a socket on which it listens for requests. The server has a three-step response to a client. First, it accepts the connection, then it reads and processes the client's request. Finally, it sends a reply to the client. For some client-server relationships, like a database server, there may be multiple requests and replies. Since database requests may take a long time to process, the server must be multi-threaded in order to handle concurrent requests. In the case of HTTP, a single request will lead to multiple replies.

A server program handles requests from a client by using the following outline of processing.

1. **Create a Listener Socket.** A listener socket is waiting for client connection requests.

2. **Accept a Client Connection.** When a client attempts a connection, the socket's `accept` method will return a "daughter" socket connected to the client. This daughter socket is used for all subsequent processing.

3. **Read the request.** Many of the standard TCP/IP protocols expect the commands to be sent as strings of text, terminated with the `\n` character. Often a Python file object is created from the socket so that the complete set of file method functions for reading and writing are available.

4. **Send the reply.** Many of the standard protocols will respond with a 3-digit numeric code indicating the status of the request. We'll review some common variations on these codes, below.

Create and Listen on a Socket. The following methods are relevant when creating server-side sockets. These server side method functions are used for establishing the public socket that is waiting for client connections. In each definition, the variable `s` is a socket object.

`s.bind (`*`address`*`)`

> Bind the socket to a local address tuple of (IP Address and port number). This tuple is the address and port that will be used by clients to connect with this server. Generally, the first part of the tuple is simply "" to indicate that this server uses the address of the computer on which it is running.

`s.listen (`*`queueSize`*`)`

> Start listening for incoming connections, queueSize specifies the number of queued connections.

`s.accept` → (socket, address)

> Accept a client connection, returning a socket connected to the client and client address.

Once the socket connection has been accepted, processing is a simple matter of reading and writing on the daughter socket.

We won't show an example of writing a server program using simple sockets. The best way to make use of server-side sockets is to use the `SocketServer` module.

Practical Server Programs with SocketServer

Generally, we use the `SocketServer` module for simple socket processing. Usually, we create a `TCPSocket` using this module. This can simplify the processing of requests and replies. The SocketServer module, for example, is the basis for the `SimpleHTTPServer` (see the section called "Web Servers and the HTTP protocol") and `SimpleXMLRPCServer` (see the section called "Web Services: The `xmlrpclib` Module") modules.

Much of server-side processing is encapsulated in two classes of the `SocketServer` module. You will subclass the `StreamRequestHandler` class to process TCP/IP requests. This subclass will include the methods that do the essential work of the program.

You will then create an instance of the `TCPServer` class and give it your `RequestHandler` subclass. The instance of `TCPServer` will to manage the public socket, and all of the basic processing. For each connection, it will create an instance of your subclass of `StreamRequestHandler` to handle the connection.

Define a `RequestHandler`. Defining a handler is done by creating a subclass of `StreamRequestHandler` or `BaseRequestHandler` and adding a `handle` method function. The `BaseRequestHandler` defines a simple framework that `TCPServer` can use when data is received on a socket.

Generally, we use a subclass of `StreamRequestHandler`. This class has methods that create files from the socket. This alliows the `handle` method function to simply read and write files. Specifically, the superclass will assure that the variables `self.rfile` and `self.wfile` are available.

For example, the echo service runs in port 7. The echo service simply reads the data provided in the socket, and echoes it back to the sender. Many Linux boxes have this service enabled by default. We can build the basic echo handler by creating a subclass of `StreamRequestHandler`.

```
#!/usr/bin/env python
"""My Echo"""
import SocketServer

class EchoHandler( SocketServer.StreamRequestHandler ):
    def handle(self):
        input= self.request.recv(1024)
        print "Input: %r" % ( input, )
        self.request.send("Heard: %r\n" % ( input, ) )

server= SocketServer.TCPServer( ("",7000), EchoHandler )
print "Starting Server"
server.serve_forever()
```

This class can be used by a `TCPServer` instance to handle requests. In this, the `TCPServer` instance named `server` creates an instance of `EchoHandler` each time a connection is made on port 7. The derived socket is given to the handler instance, as the instance variable `self.request`.

A more sophisticated handler might decode input commands and perform unique processing for each command. For example, if we were building an on-line Roulette server, there might be three basic commands: a place bet command, a show status command and a spin the wheel command. There might be additional commands to join a table, chat with other players, perform credit checks, etc.

Methods of TCPServer. In order to process requests, there are two methods of a `TCPServer` that are of interest. In the following examples the `TCPServer` instance is the variable `s`.

`s.handle_request`

Handle a single request: wait for input, create the handler object to process the request.

`s.serve_forever`

Handle requests in an infinite loop. Runs until the loop is broken with an exception.

Protocol Design Notes

Generally, basic web services do almost everything we need; and they do this kind of thing in a simple and standard way. Using sockets is done either to invent something knew or to cope with something very old. Generally, using web services is a better choice than inventing your own protocol.

If you can't, for some reason, make suitable use of web services, here are some lessons gleaned from the reading the Internetworking Requests for Comments (RFCs).

Many protocols involve a request-reply conversational style. The client connects to the server and makes requests. The server replies to each request. Some protocols (for example, FTP) may involve a long conversation. Other protocols (for example, HTTP) involve a single request and (sometimes) a single reply. Many web sites leverate HTTP's ability to send multiple replies, but some web sites send a single, tidy response.

Many of the Internet standard requests are short 1- to 4-character commands. The syntax is kept intentionally very simple, using spaces for delimeters. Complex syntax with optional clauses and sophisticated punctuation is often an aid for people. In most web protocols, a sequence of simple commands are used instead of a single, complex statement.

The responses are often 3-digit numbers plus explanatory comments. The application depends on the 3-digit number. The explanatory comments can be written to a log or displayed for a human user. The status numbers are often coded as follows:

1yz

Preliminary reply, more replies will follow.

2yz

Completed.

3yz

More information required. This is typically the start of a dialog.

4yz

Request not completed; trying again makes sense. This is a transient problem like a deadlock, timeout, or file system problem.

5yz

Request not completed because it's in error; trying again doesn't make sense. This a syntax problem or other error with the request.

The middle digit within the response provides some additional information.

x0z

The response message is syntax-related.

x1z

The response message is informational.

x2z

The response message is about the connection.

x3z

The response message is about accounting or authentication.

x5z

The response message is file-system related.

These codes allow a program to specify multi-part replies using 1*yz* codes. The status of a client-server dialog is managed with 3*yz* codes that request additional information. 4*yz* codes are problems that might get fixed. 5*yz* codes are problems that can never be fixed (the request doesn't make sense, has illegal options, etc.)

Note that protocols like FTP (RFC 959) provide a useful convention for handling multi-line replies: the first line has a - after the status number to indicate that additional lines follow; each subsequent lines are indented. The final line repeats the status number. This rule allows us to detect the first of many lines, and absorb all lines until the matching status number is read.

Projects

Projects to Build Skills

Programming language skills begin with the basic syntax and semantics of the language. They advance through the solution of small exercises and are refined through solving more complete problems.

"Real-world" applications, used every day in business and research, are less than ideal for learning a programming language. The business-oriented problems often have a very narrow in focus; the solutions are dictated by odd budgetary constraints or departmental politics. Reasearch problems are also narrowly focused, often lacking a final "application" to surround the interesting parts of the programming and create a final, finished product.

This part provides several large exercises that provide for more advanced programming than the smaller exercises at the end of each section. These aren't real-world in scope, but they are quite a bit larger than the small exercises at the end of each chapter.

We cover several problems that have interesting algorithms. These are ranked in order of difficulty.

Chapter 37, *Areas of the Flag*. Computing the area of the symbols on the American flag.

Chapter 38, *The Date of Easter*. Finding the date for Easter in a given year.

Chapter 39, *Musical Pitches*. Computing the frequencies of various musical pitches.

Chapter 40, *Bowling Scores*. Computing the score in a game of bowling.

Table of Contents

Chapter 37. Areas of the Flag

From Robert Banks, *Slicing Pizzas, Racing Turtles, and Further Adventures in Applied Mathematics* [Banks02].

This project is simple: it does not use loops, if-statements or any data structures. This exercise focuses on expressions and assignment statements.

Facts about the American flag. The standard dimension is the *width* (or hoist). This is the basic unit that all others are multiplied by. A flag that is 30 inches wide will simply have 30 inches multiplied by all of the other measurements.

- Width (or *hoist*) Wf=1.0
- Length (or *fly*) Lf=1.9 × Wf
- Width of union (or *canton*) Wu=7/13 × Wf
- Length of union Lu=0.76 × Wf
- Radius of a star R=0.0308 × Wf

These are other facts; they are counts, not measurements.

- Number of red stripes Sr=7
- Number of white stripes Sw=6
- Number white stars Ns=50
- Width of a stripe Ws=1/(Sr+Sw) × Wf

Red Area. There are 4 red stripes which abut the blue field and 3 red stripes run the full length of the flag.

We can compute the area of the red, since it is 4 short rectangles and 3 long rectangles. The short rectangle areas are the width of a stripe (Ws) times the whole length (length of the fly, Lf) less the width of the blue union (Wu). The long rectangle areas are simply the width of the stripe (Ws) times the length of the fly (Lf).

Red = 4×Ws×(Lf-Lu) + 3×Ws×(Lf).

White Area. There are 3 white stripes which abut the blue field, 3 whie stripes run the full length of the flag, plus there are the 50 stars.

We can compute the basic area of the white using a similar analysis as area of the red, and adding in the areas of the stars, 50*S*. We'll return to the area of the stars, 50*S*, last.

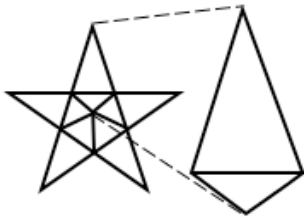
White = 3×Ws×(Lf-Lu) + 3×Ws×(Lf) + 50S.

Blue Area. The blue area is the area of the union, less the area of the stars, 50*S*.

Blue = (Lu-Wu) - 50S.

Area of the Stars. A 5-sided star (pentagram) can be analyzed as 5 kites of 2 triangles. The area of each kite, *K*, is computed as follows.

Figure 37.1. Kite and Star



The inner angles of all five kites fill the inside of the pentagram, and the angles must sum to 360°, therefore each inner angle is 72°. Angles of a triangle sum to 180°, therefore the lower triangle has two side angles of (180°-72°)/2 = 54°.

We see some straight lines that contain an outer triangle and two inner triangles. We know the inner triangles add to 108°, a straight line is 180° so the outer triangle has two 72° corners and a 36° peak. The area of the two triangles can be computed as

a=36°, b=72°

Equation 37.1. Area of a triangle, version 1

$$K = \frac{\sin\frac{a}{2}\sin\frac{b}{2}}{\frac{1}{2}\sin(a+b)}R^2$$

Equation 37.2. Area of a triangle, version 2

$$K = \frac{\sin\frac{b}{2}}{\phi^2}R^2$$

Note that the math library `sin` and `cos` functions operate in radians, not degrees. The conversion rule is π radians = 180 degrees. Therefore, we often see something like `sin ( a×π/180 )` to convert `a` degrees to radians.

The Golden Ratio is $\phi = \frac{1+\sqrt{5}}{2}$ (about 1.61803).

The total area of a star is

S=5×K.

Given a specific width of a flag (in feet), we can compute the actual areas of each color.

Check Your Results. Blue is 18.73% of the total area.

Chapter 38. The Date of Easter

Don Knuth, in the *Art of Computer Programming, Volume I, Fundamental Algorithms*, suggests, "There are many indications that the sole important application of arithmetic in Europe during the Middle Ages was the calculation of Easter date, and so such algorithms are historically significant."

We present several variants on the basic problem of finding the date of Easter. We will not delve into the history and politics of this Christian holiday, but merely observe the huge intellectual effort put forth into locating the correct date.

In 1582, Pope Gregory decreed the change from the old Julian calendar to the Gregorian calendar. The most significant changes were dropping 10 days, fixing the leap year rules and fixing the rule for finding Easter. This was not adopted in England (and its colonies like the U.S. colonies) until 1782.

$\lfloor x \rfloor$ means round down, which is the usual rule for integer division. x mod y is the remainder when dividing x by y.

Algorithm E dates from the sixteenth century, and will find Easter for years after 1582. The author is D. Knuth, and it is published in *Art of Computer Programming, Volume I, Fundamental Algorithms* [Knuth73].

Procedure 38.1. Algorithm E.

(Date of Easter after 1582.) Let Y be the year for which the date of Easter is desired.

1. **Golden Number.** Set $G \leftarrow (Y \bmod 19) + 1$. ($G$ is the "golden number" of the year in the 19-year Metonic cycle.)

2. **Century.** Set $C \leftarrow \lfloor Y \div 100 \rfloor + 1$. (When Y is not a multiple of 100, C is the century number; i.e., 1984 is in the twentieth century.)

3. **Corrections.** Set $X \leftarrow \lfloor 3C \div 4 \rfloor - 12, Z \leftarrow \lfloor (8C+5) \div 25 \rfloor - 5$. ($X$ is the number of years, such as 1900, in which leap year was dropped in order to keep in step with the sun. Z is a special correction designed to synchronize Easter with the moon's orbit.)

4. **Find Sunday.** Set $D \leftarrow \lfloor 5Y \div 4 \rfloor - X - 10$. (March $((-D) \bmod 7)$ actually will be a Sunday.)

5. **Epact.** Set $E \leftarrow (11G + 20 + Z - X) \bmod 30$. If $E = 25$ and the golden number G is greater than 11, or if $E = 24$, then increase E by 1. (E is the "epact," which specifies when the full moon occurs.)

6. **Find full moon.** Set $N \leftarrow 44 - E$. If N < 21 then set $N \leftarrow N + 30$. (Easter is

> supposedly the "first Sunday following the first full moon which occurs on or after March 21." Actually perturbations in the moon's orbit do not make this strictly true, but we are concerned here with the "calendar moon" rather than the actual moon. The *N*th of March is a calendar full moon.

7. **Advance to Sunday.** Set $N \leftarrow N + 7 - ((D + N) \bmod 7)$.

8. **Get month.** If $N > 31$, the date is $(N - 31)$ April; otherwise the date is N March.

Knuth also observes the following:

> A fairly common error in the coding of this algorithm is to fail to realize that the quantity (11G + 20 + Z – X) in step E5 may be negative, and so the positive remainder mod 30 is sometimes not computed. For example, in the year 14250, we would find G=1, X=95, Z=40; so if we had E=–24 instead of E=+6 we would get the ridiculous answer "42 April". An interesting variation on this algorithm is to find the earliest year for which this remainder problem would cause the wrong date to be calculated for Easter.

For dates between 464 and 1582, the Julian calendar was in use, with different leap year rules, algorithm J computes Easter according to that calendar. The author is D. Knuth, and it is published in *The Art of Computer Programming, Volume I, Fundamental Algorithms* [Knuth73].

Procedure 38.2. Algorithm J.

(Date of Easter between 464 and 1582.) Let Y be the year for which the date of Easter is desired.

1. **Golden Number.** Set $G \leftarrow (Y \bmod 19) + 1$. ($G$ is the "golden number" of the year in the 19-year Metonic cycle.)

2. **Find Sunday.** Set $D \leftarrow \lfloor 5Y \div 4 \rfloor$. (March $((-D) \bmod 7)$ actually will be a Sunday.)

3. **Epact.** Set $E \leftarrow (11G - 4) \bmod 30 + 1$. ($E$ is the "epact," which specifies when the full moon occurs.)

4. **Find full moon.** Set $N \leftarrow 44 - E$. If N < 21 then set $N \leftarrow N + 30$. (Easter is supposedly the "first Sunday following the first full moon which occurs on or after March 21." Actually perturbations in the moon's orbit do not make this strictly true, but we are concerned here with the "calendar moon" rather than the actual moon. The *N*th of March is a calendar full moon.

5. **Advance to Sunday.** Set $N \leftarrow N + 7 - ((D + N) \bmod 7)$.

6. **Get month.** If $N > 31$, the date is $(N - 31)$ April; otherwise the date is N March.

Algorithm A was first published in 1876. The author is Jean Meeus and it is published in *Astronomical Algorithms* [Meeus91].

Procedure 38.3. Algorithm A.

(Date of Easter after 1582.) Let Y be the year for which the date of Easter is desired.

1. $A \leftarrow (Y \bmod 19)$.

2. $B \leftarrow \lfloor Y / 100 \rfloor$.

3. $C \leftarrow (Y \bmod 100)$.

4. $D \leftarrow \lfloor B / 4 \rfloor$.

5. $E \leftarrow B \bmod 4$.
6. $F \leftarrow \lfloor (B + 8) / 25 \rfloor$.
7. $G \leftarrow \lfloor (B - F + 1) / 3 \rfloor$.
8. $H \leftarrow ((19A) + B - D - G + 15) \bmod 30$.
9. $I \leftarrow \lfloor C / 4 \rfloor$.
10. $K \leftarrow C \bmod 4$.
11. $X \leftarrow (32 + (2E) + (2I) - H - K) \bmod 7$.
12. $M \leftarrow \lfloor (A + (11H) + (22X)) / 451 \rfloor$.
13. $Q \leftarrow H + X - 7M + 114$.
14. $N \leftarrow \lfloor Q / 31 \rfloor$.
15. $P \leftarrow Q \bmod 31$.
16. Easter is day P+1 of month N.

Algorithm N is a variant on Algorithm A, but gives Easter prior to 1583. The rules for Easter are somewhat unclear prior to 325, but the author states that Easter is April 12 for 179, 711 and 1243. The author is Jean Meeus and it is published in *Astronomical Algorithms* [Meeus91].

Procedure 38.4. Algorithm B.

(Date of Easter prior to 1583.) Let Y be the year for which the date of Easter is desired.

1. $A \leftarrow (Y \bmod 4)$.
2. $B \leftarrow (Y \bmod 7)$.
3. $C \leftarrow (Y \bmod 19)$.
4. $D \leftarrow (19C + 15) \bmod 30$.
5. $E \leftarrow (2A + 4B - D + 34) \bmod 7$.
6. $H \leftarrow D + E + 114$.
7. $F \leftarrow \lfloor H / 31 \rfloor$.
8. $G \leftarrow H \bmod 31$.
9. Easter is day G+1 of month F.

The following algorithm is from T. M. O'Beirne, it is published in *Puzzles and Paradoxes* [OBeirne65].

Procedure 38.5. Algorithm O.

(Date of Easter after 1582.) Let Y be the year for which the date of Easter is desired.

1. $A \leftarrow (Y \bmod 19)$.
2. $B \leftarrow \lfloor Y / 100 \rfloor$.

3. $C \leftarrow (Y \bmod 100)$.
4. $D \leftarrow \lfloor B / 4 \rfloor$.
5. $E \leftarrow B \bmod 4$.
6. $G \leftarrow \lfloor (8B + 13) / 25 \rfloor$.
7. $H \leftarrow ((19A) + B - D - G + 15) \bmod 30$.
8. $M \leftarrow \lfloor (A + 11H) / 319 \rfloor$.
9. $I \leftarrow \lfloor C / 4 \rfloor$.
10. $K \leftarrow C \bmod 4$.
11. $F \leftarrow (2E + 2I - K - H + M + 32) \bmod 7$.
12. $N \leftarrow \lfloor (H - M + F + 90) / 25 \rfloor$.
13. $P \leftarrow (H - M + F + N + 19) \bmod 32$.
14. Easter is day P of month N.

The following algorithm is from T. M. O'Beirne, it is published in *Puzzles and Paradoxes* [OBeirne65].

Procedure 38.6. Algorithm P.

(Date of Easter after 1582.) Let Y be the year for which the date of Easter is desired.

1. $B \leftarrow \lfloor Y / 100 \rfloor$.
2. $C \leftarrow (Y \bmod 100)$.
3. $A \leftarrow (5B + C) \bmod 19$.
4. $T \leftarrow 3B + 75$.
5. $D \leftarrow \lfloor T / 4 \rfloor$.
6. $E \leftarrow T \bmod 4$.
7. $G \leftarrow \lfloor (8B + 88) / 25 \rfloor$.
8. $H \leftarrow ((19A) + D - G) \bmod 30$.
9. $M \leftarrow \lfloor (A + 11H) / 319 \rfloor$.
10. $T \leftarrow 300 - 60E + C$.
11. $J \leftarrow \lfloor T / 4 \rfloor$.
12. $K \leftarrow T \bmod 4$.
13. $F \leftarrow (2J - K - H + M) \bmod 7$.
14. $T \leftarrow H - M + F + 110$.
15. $N \leftarrow \lfloor T / 30 \rfloor$.
16. $Q \leftarrow T \bmod 30$.

17. $P \leftarrow (Q + 5 - N) \bmod 32$.
18. Easter is day P of month N.

The following algorithm is from J.-M. Oudin. It is published in *Standard C Date/Time Library* [Latham98].

Procedure 38.7. Algorithm F.

(Date of Easter after 1582.) Let Y be the year for which the date of Easter is desired.

1. $C \leftarrow \lfloor Y / 100 \rfloor$.
2. $N \leftarrow (Y \bmod 19)$.
3. $K \leftarrow \lfloor (C - 17) / 25 \rfloor$.
4. $I \leftarrow (C - \lfloor C / 4 \rfloor - \lfloor (C - K) / 3 \rfloor + 19N + 15) \bmod 30$.
5. $I \leftarrow I - \lfloor I/28 \rfloor (1 - \lfloor I/28 \rfloor \times \lfloor 29/(I+1) \rfloor \times \lfloor (21-N)/11 \rfloor)$.
6. $J \leftarrow (Y + \lfloor Y / 4 \rfloor + I + 2 - C + \lfloor C / 4 \rfloor) \bmod 7$.
7. $X \leftarrow I - J$.
8. $M \leftarrow 3 + \lfloor (X + 40) / 44 \rfloor$.
9. $D \leftarrow \mathrm{X} + 28 - 31 \lfloor M / 4 \rfloor$.
10. Easter is day D of month M.

The following algorithm is from Gauss. It is published in *Standard C Date/Time Library* [Latham98].

Procedure 38.8. Algorithm G.

(Date of Easter between 1583 and 2199.) Let Y be the year for which the date of Easter is desired.

1. $H \leftarrow \lfloor Y / 100 \rfloor$.
2. When H is then A is and B is

H	A	B
15	22	2
16	22	2
17	23	3
18	23	4
19	24	5
20	24	5
21	24	6

3. $C \leftarrow (19(Y \bmod 19) + A) \bmod 30$.
4. $D \leftarrow (2(Y \bmod 4) + 4(Y \bmod 7) + 6C + B) \bmod 7$.
5. $day \leftarrow 22 + C + D$, $month \neg 3$.
6. If $day > 31$, set $day \leftarrow C + D - 9$, increment $month$.
7. If $month = 4$ and $day = 26$, set $day \leftarrow 19$.

8. If $C = 38$ and (Y mod 19) > 10 and *month* = 4 and *day* = 25, set *day* ← 18.

9. Easter is day *day* of month *month*.

The following variation on algorithm G is from Dershowitz and Reingold, *Calendrical Calculations* [Dershowitz97]. This depends on a very clever idea of doing the calculation strictly as a number of days offset from a Gregorian Epoch date. This day number is called a *Rata Die*, RD. Once computed, the RD for Easter is simply converted to a Gregorian date. This algorithm requires the Gregorian to RD algorithm and the RD to Gregorian algorithm.

Procedure 38.9. Algorithm R.

(Date of Easter after 1582.) Let Y be the year for which the date of Easter is desired.

1. **Century.** Set $C \leftarrow \lfloor Y / 100 \rfloor + 1$. (When Y is not a multiple of 100, C is the century number; i.e., 1984 is in the twentieth century.)

2. **Shifted Epact.** Set $E \leftarrow (14 + 11(Y \bmod 19) - \lfloor 3C / 4 \rfloor + \lfloor (5+8C) / 25 \rfloor) \bmod 30$.

3. **Adjust Epact.** If $E = 0$ or ($E = 1$ and $10 < (Y \bmod 19)$), then add 1 to E.

4. **Paschal Moon.** Set $R \leftarrow$ the RD for April 19 of year Y. Set $P \leftarrow R - E$.

5. **Easter.** Locate the Sunday after the Paschal Moon. Set $Q \leftarrow P + 7 - (P \bmod 7)$.

6. **Get Date.** Compute the gregorian date for RD number Q.

In order to recover the actual date from the RD, the following algorithms must be used to get the year, the month and the day. These, in turn depend on another algorithm, given below, to compute the RD for a given date.

Procedure 38.10. Algorithm L.

(Leap Year Test.) Let Y be the year for which a leap day test should be done.

1. **Year Test.** Set $R \leftarrow Y \bmod 4$.

2. **400-Year Test.** Set $C \leftarrow Y \bmod 400$.

3. **Final Rule.** If $R = 0$ and not (C=100 or C=200 or C=300), then Y is a leap year; otherwise Y is a standard year.

We can build up a RD from a Gregorian date with a relatively simple algorithm. This depends on a number of subtle observations, detailed by Dershowitz and Reingold in their book. This algorithm relies on Algorithm L to determine if a leap day is required.

Procedure 38.11. Algorithm RD.

(RD from a Gregorian Date.) Let Y, M and D be the year, month and day of a Gregorian date for which an RD is required.

1. **Days for all Years.** Set $R_1 \leftarrow 365(Y-1) + \lfloor (Y-1)/4 \rfloor - \lfloor (Y-1)/100 \rfloor + \lfloor (Y-1)/400 \rfloor$.

2. **February Adjustment.** If $M \le 2$, then set $f \leftarrow 0$. If $M > 2$ and this year is a leap year, then set $f \leftarrow -1$. If $M > 2$ and this year is not a leap year, then set $f \leftarrow -2$.

3. **Days for this year.** Set $R_2 \leftarrow \lfloor (367*M-362) / 12 \rfloor + f$.

4. **Final RD.** Set RD← $R_1 + R_2 + D$.

Now that we can convert a Gregorian date to an RD, we can use algorithm RD to convert an RD back to a Gregorian date.

Procedure 38.12. Algorithm Y.

(Extract Gregorian year from RD number.) Let *RD* be the *Rata Die* date for which we want the Gregorian Year.

1. **Offset by Gregorian Start Date.** Set $d0 \leftarrow RD - 1$.
2. **400 year cycles.** Set $n_{400} \leftarrow \lfloor d0 / 146097 \rfloor$, set $d1 \leftarrow d0 \bmod 146097$.
3. **100 year cycles.** Set $n_{100} \leftarrow \lfloor d1 / 36524 \rfloor$, set $d2 \leftarrow d1 \bmod 35624$.
4. **4 year cycles.** Set $n_4 \leftarrow \lfloor d2 / 1461 \rfloor$, set $d3 \leftarrow d2 \bmod 1461$.
5. **1 year cycles.** Set $n_1 \leftarrow \lfloor d3 / 365 \rfloor$, set $d4 \leftarrow d3 \bmod 365 + 1$.
6. **Year.** Set $Y \leftarrow 400n_{400} + 100n_{100} + 4n_4 + n_1$. If $n_{100}=4$ or $n_1=4$, Y is the Gregorian year. Otherwise, $Y+1$ is the Gregorian year.

Given a Gregorian year for an RD number, we can compute the RD number of the first day of the year, and subtract to find the day number. A bit more math will yield the month within the year. This relies on Algorithm Y and Algorithm L as well as Algorithm RD.

Procedure 38.13. Algorithm M.

(Extract month from the RD number.) Let *RD* by the *Rata Die* date for which we want the Gregorian month.

1. **Get key dates.** Use algorithm Y to set *Y* to the year for *RD*. Use algorithm RD to set *jan1* to the RD number for Jan. 1 of year *Y*. Use algorithm RD to set *mar1* to the RD number for Mar. 1 of year *Y*.
2. **Get prior days in this year.** Set $P \leftarrow RD - jan1$.
3. **Leap year correction.** If $RD < mar1$, then set $c \leftarrow 0$. If $RD > mar1$ and year *Y* is a leap year, then set $c \leftarrow 1$. If $RD > mar1$ and year *Y* is not a leap year, set $c \leftarrow 2$.
4. **Compute month.** Set $M \leftarrow \lfloor (12*(P+c)+373) / 367 \rfloor$. *M* is the month.

Finally, we can combine algorithm Y and M (along with L and RD) to compute the day of the month for an RD number. This is done by getting the year and month, and then finding the RD number for the first of the month. We then subtract the RD number from the RD number for the first of the month. The remainder is the number of days after the first.

Procedure 38.14. Algorithm D.

(Extract day from the RD number.) Let *RD* by the *Rata Die* date for which we want the day of the Gregorian month.

1. **Get key date.** Use algorithm Y to set *Y* to the year for *RD*. Use algorithm M to set *M* to the month for *RD*. Use algorithm RD to set *mon1* to the first day of month *M*, year *Y*.
2. **Get day of this month.** Set $D \leftarrow RD - mon1 + 1$. *D* is the day of the month.

Check Your Results. Find a calendar or web site with dates for Easter.

Chapter 39. Musical Pitches

Table of Contents

A musician will call the frequency of a sound its *pitch*. When the frequencies of two pitches differ by a factor of two, we say they harmonize. This perception of harmony happens because the two sounds reinforce each other completely. Indeed, some people have trouble telling two notes apart when they differ by an octave.

Classical European music divided that perfectly harmonious "factor of two" interval into eight asymetric steps; for this historical reason, it is called an octave. Other cultures divide this same interval into different numbers of steps with different intervals.

More modern European music further subdivides the octave, creating a 12-step system. The modern system has 12 equally spaced intervals, a net simplification over the older 8-step system. The pitches are assigned names using *flats* (♭) and *sharps* (♯), leading to each pitch having several names. We'll simplify this system slightly, and use the following 12 names for the pitches within a single octave: A, A#, B, C, C#, D, D#, E, F, F#, G, G#.

The eight undecorated names (A through G and the next A) form our basic octave; the additional notes highlight the interesting asymetries. For example, the interval from A to B is called a whole step or a second, with A ♯ being half-way between. The interval from B to C, however is only a half step to begin with. Also, it is common to number the various octaves as though the octaves begin with the C, not the A. So, some musicians consider the basic scale to be C, D, E, F, G, A, B, and the C is in the next higher octave. The higher C is twice the frequency of the lower C.

The tuning of an instrument to play these pitches is called its *temperament*. A check on the web for reference material on tuning and temperament will reveal some interesting ways to arrive at the tuning of a musical instrument. It is suprising to learn that there are many other ways to arrive at the 12 steps of the scale. This demonstrates that our ear is either remarkably inaccurate or remarkably forgiving of errors in tuning. We'll explore a number of alternate systems for deriving the 12 pitches of a scale. We'll use the simple equal-temperament rules, plus we'll derive the pitches from the overtones we hear, plus a more musical rule called the *circle of fifths*, as well as a system called *Pythagorean Tuning*.

Interesting side topics are the questions of how accurate the human ear really is, and can we really hear the differences? Clearly, professional musicians will spend time on ear training to spot fine gradations of pitch. However, even non-musicians have remarkably accurate hearing and are easily bothered by small discrepencies in tuning. The musicians will divide the octave into 1200 cents. Errors on the order of 50 cents, 1/24 of the octave, are noticable even to people who claim they are "tone deaf". When two tunings produce pitches with a ratio larger than 1.0293, it is easily recognized as out of tune.

These exercises will make extensive use of loops and the list data structure.

Equal Temperament

The equal temperament tuning divides the octave into twelve equally sized steps. Moving up the scale is done by multiplying the base frequency by some amount between 1 and 2. If we multiply a base frequency by 2 or more, we have jumped to another

octave. If we multiply a base frequency by a value between 0 and 0.5, we have jumped into a lower octave. When we multiply a frequency by values between 0.5 and 1, we are computing lower pitches in the same octave. Similarly, multiplying a frequency by values between 1 and 2 computes a higher pitch in the same octave.

We want to divide the octave into twelve steps: when we do a sequence of twelve multiplies by this step, we should arrive at an exact doubling of the base frequency. The steps of the octave, then, would be b, $b \times s$, $b \times s \times s$, ... up to $b \times s^{12} = 2 \times b$. This step value, therefore is the following value.

$$s = e^{\frac{\log 2}{12}}$$

If we multiply this 12 times for each of the 12 steps, we find the following.

$$e^{12\frac{\log 2}{12}} = 2$$

For a given pitch, p, from 0 to 88, the following formula gives us the frequency. We can plug in a base frequency, b of 27.5 Hz for the low A on a piano and get the individual pitches for each of the 88 keys.

Equation 39.1. Musical Pitches

$$f = b\, e^{p\frac{\log 2}{12}}$$

Equal Temperament Pitches. Develop a loop to generate these pitches and their names. If you create a simple tuple of the twelve names shown above (from "A" to "G#"), you can pick out the proper name from the tuple for a given step, `s`, using `int( s % 12 )`.

Check Your Results. You should find that an "A" has a pitch of 440, and the "G" ten steps above it will be 783.99 Hz. This 440 Hz "A" is the most widely used reference pitch for tuning musical instruments.

Overtones

A particular musical sound consists of the fundamental pitch, plus a sequence of *overtones* of higher frequency, but lower power. The distribution of power among these overtones determines the kind of instrument we hear. We can call the overtones the spectrum of frequencies created by an instrument. A violin's frequency spectrum is distinct from the frequency spectrum of a clarinet. The overtones are usually integer multiples of the base frequency. When any instrument plays an A at 440 Hz, it also plays A's at 880 Hz, 1760 Hz, 3520 Hz, and on to higher and higher frequencies. While we are not often consciously aware of these overtones, they are profound, and determine the pitches that we find harmonious and discordant.

If we expand the frequency spectrum through the first 24 overtones, we find almost all of the musical pitches in our equal tempered scale. Some pitches (the octaves, for example) match precisely, while other pitches don't match very well at all. This is a spread of almost five octaves of overtones, about the limit of human hearing.

Even if we use a low base frequency, b, of 27.5 Hz, it isn't easy to compare the pitches for the top overtone, $b \times 24$, with a lower overtone like $b \times 8$: they're in two different octaves. However, we divide each frequency by a power of 2, which will normalize it into the lowest octave. Once we have the lowest octave version of each overtone pitch, we can compare them against the equal temperament pitch for the same octave.

The following equation computes the highest power of 2, p_2, less than or equal to some frequency, f compared against our base frequency, b, of 27.5 Hz.

Equation 39.2. Highest Power of 2, p_2

$$p_2 = int\left[\frac{\log\frac{f}{b}}{\log 2} - 1\right]$$

Given this highest power of highest power of 2, p_2, we can normalize a frequency by this simple division. This will create what we'll call the *first octave pitch*, f_0.

Equation 39.3. First Octave Pitch

$$f_0 = \frac{f}{2^{p_2}}$$

The list of first octave pitches arrives in a peculiar order. You'll need to collect the values into a list and sort that list. You can then produce a table showing the 12 pitches of a scale using the equal temperament and the overtones method. They don't match precisely, which leads us to an interesting musical question of which sounds "better" to most listeners.

Overtone Pitches. Develop a loop to multiply the base frequency of 27.5 Hz by values from 3 to 24, compute the highest power of 2 required to normalize this back into the first octave, p_2, and compute the first octave values, f_0. Save these first octave values in a list, sort it, and produce a report comparing these values with the closest matching equal temperament values.

Note that you will be using 22 overtone multipliers to compute twelve scale values. You will need to discard duplicates from your list of overtone frequencies.

Check Your Results. You should find that the 6th overtone is 192.5 Hz, which noralizes to 48.125 in the fist octave. The nearest comparable equal-tempered pitch is 48.99 Hz. This is an audible difference to some people; the threshold for most people to say something sounds wrong is a ratio of 1.029, these two differ by 1.018.

Circle of Fifths

When we look at the overtone analysis, the second overtone is three times the base frequency. When we normalize this back into the first octave, it produces a note with the frequency ratio of 3/2. This is almost as harmonious as the octave, which had a frequency ratio of exactly 2. In the original 8-step scale, this was the 5th step; the interval is called a *fifth* for this historical reason. It is also called a *dominant*. Looking at the names of our notes, this is "E", the 7th step of the more modern 12-step scale.

This pitch has an interesting mathematical property. When we look at the 12-step tuning, we see that numbers like 1, 2, 3, 4, and 6 divide the 12-step octave evenly. However, numbers like 5 and 7 don't divide the octave evenly. This leads to an interesting cycle of notes that are separated by seven steps: A, E, B, F#, C#, ... We can see this clearly by writing the notes around the outside of a circle, and walking around the circle in groups of seven pitches. This is called the *Circle of Fifths* because we see all 12 pitches by stepping through the names in intervals of a fifth.

This also works for the 5th step of the 12-step scale; the interval is called a *fourth* in the old 8-step scale. Looking at our note names, it is the "D". If we use this interval, we create a *Circle of Fourths*.

Write two loops to step around the names of notes in steps of 7 and steps of 5. You can use something like `range( 0, 12*7, 7 )` or `range( 0, 12*5, 5 )` to get the steps, `s`. You can then use `names[s % 12]` to get the specific names for each pitch.

You'll know these both work when you see that the two sequences are the same things in

opposite orders.

Circle of Fifths Pitches. Develop a loop similar to the one in the overtones exercise; use multipliers based on 3/2: 3/2, 6/2, 9/2, ... to compute the 12 pitches around the circle of fifths. You'll need to compute the highest power of 2, using Equation 39.2, "Highest Power of 2, p_2", and normalize the pitches into the first octave using Equation 39.3, "First Octave Pitch". Save these first octave values in a list, indexed by `s % 12`; you don't need to sort a list, since the pitch can be computed directly from the step.

Check Your Results. Using this method, you'll find that "G" could be defined as 49.55 Hz. The overtones suggested 48.125 Hz. The equal temperament suggested 48.99 Hz.

Pythagorean Tuning

When we do the circle of fifths calculations using rational numbers instead of floating point numbers, we find a number of simple-looking fractions like 3/2, 4/3, 9/8, 16/9 in our results. These fractions lead to a geometrical interpretation of the musical intervals. These fractions correspond with some early writings on music by the mathematician Pythagoras.

We'll provide one set of commonly-used list of fractions for Pythagorean tuning. These can be compared with other results to make the whole question of scale tuning even more complex.

Name	Ratio
A	1:1
A#	256:243
B	9:8
C	32:27
C#	81:64
D	4:3
D#	729:512
E	3:2
F	128:81
F#	27:16
G	16:9
G#	243:128

Pythagorean Pitches. Develop a simple representation for the above ratios. A list of tuples works well, for example. Use the ratio to compute the frequencies for the various pitches, using 27.5 Hz for the base frequency of the low "A". Compare these values with equal temperament, overtones and circle of fifths tuning.

Check Your Results. The value for "G" is 27.5 * 16 / 9 = 48.88Hz.

Five-Tone Tuning

The subject of music is rich with cultural and political overtones. We'll try to avoid delving too deeply into anything outside the basic accoustic properties of pitches. One of the most popular alternative scales divides the octave into five equally-spaced steps. This tuning produces pitches that are distinct from those in the 12 pitches available in European music.

The original musical tradition behind the blues once used a five step scale. You can revise the formula in Equation 39.1, "Musical Pitches" to use five steps instead of twelve. This will provide a new table of frequencies. The intervals should be called something distinctive like "V", "W", "X", "Y", "Z" and "V" in the second octave.

Five-Tone Pitches. Develop a loop similar to the 12-tone Equal Temperament (the section called "Equal Temperament") to create the 5-tone scale pitches. Note that the 12-tone scale leads to 88 distinct pitches on a piano; this 5-tone scale only needs 36.

Compare 12-Tone and 5-Tone Scales. Produce a three column table with the 12-tone pitch names and frequencies aligned with the 5-tone frequencies. You will have to do some clever sorting and matching. The frequencies for "V" will match the frequencies for "A" precisely. The other pitches, however, will fall into gaps.

The resulting table should look like the following

name	12-tone	5-tone
A	440.0	440.0
A#	466.16	
B	493.88	
		505.42
…		

Chapter 40. Bowling Scores

Bowling is played in ten frames, each of which allows one or two deliveries. If all ten pins are bowled over in the first delivery, there is no second delivery.

Each frame has a score based on the delivery in that frame, as well as the next one or two deliveries. This means that the score for a frame may not necessarily be posted at the end of the frame. It also means that the tenth frame may require a total of three deliveries to resolve the scoring.

- **Rule A.** The score for a frame is the total pins bowled over during that frame, if the number is less than ten (an open frame, or error or split depending some other rules beyond the scope of this problem).
- **Rule B.** If all ten pins are bowled over on the first delivery (a strike), the score for that frame is 10 + the next two deliveries.
- **Rule C.** If all ten pins are bowled over between the first two deliveries (a spare), the score for that frame is 10 + the next delivery.

A game can be as few as twelve deliveries: ten frames of strikes require two additional deliveries in the tenth frame to resolve the rule B scoring. A game can be as many as twenty-one deliveries: nine open frames of less than 10 pins bowled over during the frame, and a spare in the tenth frame requiring an extra delivery to resolve the rule C scoring.

There is a relatively straight-forward annotation for play. Each frame has two characters to describe the pins bowled during the delivery. The final frame has three characters for a total of 21 characters.

Rule A: If the frame is open, the two characters are the two deliveries; the total will be less than 10. If a delivery fails to bowl over any pins, a `-` is used instead of a number.

Rule B: If the frame is strike, the two characters are `X␣`. No second delivery was made.

Rule C: If the frame is a spare, the first character is the number of pins on the first delivery. The second character is a `/`.

For example:

```
8/9-X␣X␣6/4/X␣8-X␣XXX
```

This can be analyzed into ten frames as follows:

Frame	First delivery	Second delivery	Scoring rule	Frame Score	Total
1	8	2	C- spare = 10 + next delivery	19	19
2	9	-	A- open = 9	9	28
3	10	(not taken)	B- strike = 10 + next 2 deliveries	26	54
4	10	(not taken)	B- strike = 10 + next 2 deliveries	20	74
5	6	4	C- spare = 10 + next delivery	14	88
6	4	6	C- spare = 10 + next delivery	20	108
7	10	(not taken)	B- strike = 10 + next 2 deliveries	18	126
8	8	-	A- open = 8	8	134
9	10	(not taken)	B- strike = 10 + next 2 deliveries	30	164
10	10	10 and 10	B- strike = 10 + next 2 deliveries, two extra deliveries are taken during this 10th frame.	30	194

Each of the first nine frames has a two-character code for each delivery. There are three forms:

- **X␣**.
- ***n*/** where *n* is - or 1-9.
- ***mm*** where *m* is - or 1-9. The two values cannot total to 10.

The tenth frame has a three-character code for each of the deliveries. There are three forms:

- **XXX**
- ***n*/*r*** where *n* is -, 1-9 and *r* is X, -, 1-9.
- ***mm*␣** where *m* is - or 1-9. The two values cannot total to 10.

Write a `valid(`*game*`)` function that will validate a 21-character string as describing a legal game.

Write a scoring function, `scores(`*game*`)`, that will accept the 21-character scoring string and produce a sequence of frame-by-frame totals.

Write a reporting function, `scoreCard(`*game*`)` will use the validation and scoring functions to produce a scorecard. The scorecard shows three lines of output with 5 character positions for each frame.

The top line has the ten frame numbers: 2 digits and 3 spaces for each frame.

The second line has the character codes for the delivery: 2 or 3 characters and 3 or 2 spaces for each of the ten frames.

The third line has the cumulative score for each frame: 3 digit number and 2 spaces.

The game shown above would have the following output.

```
 1    2    3    4    5    6    7    8    9   10
8/   9-   X    X    6/   4/   X    8-   X   XXX
 19   28   54   74   88  108  126  134  164  194
```

Chapter 41. Mah Jongg Hands

Table of Contents

The game of Mah Jongg is played with a deck of tiles with three suits with ranks from one to nine. There are four sets of these 27 tiles. Additionally there are four copies of the four winds and three dragons. This gives a deck of 136 tiles.

The three suits are bamboo, "characters" (wan, *万* simplified or *萬* traditional), and dots. The ranks are the numbers one to nine.

The winds and dragons are collectively called "honors". There are four winds: East (*東*), South (*南*), West (*西*), and North (*北*). There are three dragons: White (*白*), Red (*中*), and Green (*發*). These tiles don't have ranks, merely names.

In some variations of the game there are also jokers, seasons and flowers. We'll leave these out of our analysis for the moment.

Tile Class Hierarchy

We can define a parent class of `Tile`, and two subclasses: `SuitTile` and `HonorTile`. These have slightly different attributes. The `SuitTile` has suit and rank information. The `HonorTile` merely has a unique name. The comparison function, `__cmp__`, must compare `self.getName` to `other.getName` to see if the other tile has the same name. Additionally, `SuitTile` objects should return a suit and rank information. Honors tiles should return None for these methods.

We'll define `Tile` as an *abstract superclass*. Some of the methods in the superclass will either raise `NotImplementedError` or return `NotImplemented`. The superclass can provide any common functions that are shared by all of the subclasses. Each subclass will provide overriding method definitions that actually do useful work and are specialized for that particular family of tiles.

```
class  Tile:
  def __init__(self, name):
  def __str__(self):
  def __cmp__(self, other):
  def getSuit(self):
  def getRank(self):
  def getName(self):
```

For `HonorsTile`, `getSuit` and `getRank` must return `None`. However, for `getSuit`, it should return the tile's name. The `__cmp__` is inherited from the superclass.

For `SuitTile`, `getSuit` and `getRank` return the proper suit and rank values. However, for `getName`, it should return `None`. The function `__cmp__` function must compare `self.getSuit` to `other.getSuit`; if those are equal, it must compare `self.getRank` to `other.getRank`. In the case of matching suit, it can compare the ranks, which will tend to put the suit tiles into order properly.

Note that we defined the `Tile` superclass with a `name` attribute. We can use this to keep the suit information for `SuitTile`. We must be sure, however, that `getName` returns `None`, not the suit.

Also, if we use the names `"Bamboo"`, `"Character"` and `"Dots"`, this makes the suits occur alphabetically in front of the honors without any further special processing. If, on the other hand, we want to use Unicode characters for the suits, we should add an additional sort key to the `Tile` that can be overridden by `SuitTile` and `HonorsTile` to force a particular sort order.

Note that the ranks of one and nine have special status among the suit tiles. These are called terminals, ranks two through eight are called simples. Currently, we don't have a need for this distinction.

Build The Tile Class Hierarchy. First, build the tile class hierarchy. This includes the `Tile`, `SuitTile`, `HonorTile` classes. Write a short test that will be sure that the equality tests work correctly among tiles.

Wall Class

You hould also define a Mah Jongg `Wall` class which holds the initial set of 136 tiles. We can create additional subclasses to add as many as a dozen more tiles to include jokers, flowers and seasons.

> Mah Jongg tiles are too large to manipulate like playing cards. They are shuffled by stirring them face down in the middle of the table. Then the tiles are stacked to make a wall with four sides. Each side is a row 17 tiles long and two tiles tall. Since this is a gambling game, there are fairly colorful procedures for establishing where people will sit, who will deal first, which of the four sides will be dealt from and where along the side the dealing will begin.

The `Wall` will need an `__init__` that enumerates four copies of each of the following tiles: the twenty-seven combinations of each suit (dot, bamboo, and character) and each rank (one through nine). It also needs to enumerate four copies of each honor tile (east, south, west, north, red, white, green).

In addition to creating the set of tiles, this class will need methods to shuffle and deal. In this way the wall is nearly identical with a deck of playing cards. See the section called “Advanced Class Definition Exercises” for more guidance on this class design.

Build The Wall Class. Second, build the `Wall`. Write a short test that will be sure that it shuffles and deals tiles properly.

Set Class Hierarchy

A winning Mah Jongg hand has five scoring *sets* exactly one of which is a pair. There are four varieties of set: pair, three of a kind, three in a row of the same suit, and four of a kind. The most common winning hands have 14 tiles: 4 sets of three and a pair. Each four of a kind extends the hand by one tile to accomodate the larger set.

A Mah Jongg `Hand` object, then, is a list of `Tiles`. This class needs a method, `mahjongg` that returns `True` is the hand is a winning hand. The evaluation is rather complex, because a tile can participate in a number of sets, and several alternative interpretations may be necessary to determine the appropriate use for a given tile.

Consider a hand with 2, 2, 2, 3, 4 of bamboo. This is either a set of three 2's and a non-scoring 3 and 4, or it is a pair of 2's and a sequence of 2, 3, 4.

The `mahjongg` method, then must create five `Set` objects, assigning individual `Tiles` to the `Sets` until all of the `Tiles` find a home. The hand is a winning hand if all sets are full, there are five sets, and one set is a pair.

We'll cover the design of the `Set` classes in this section, and return to the design of the

Hand class in the next section.

We can create a class hierarchy around the four varieties of Set: pairs, threes, fours and straights. A PairSet holds two of a kind: both of the tiles have the same name or the same suit and rank. A ThreeSet and FourSet are similar, but have a different expectation for being full. A SequenceSet holds three suit tiles of the same suit with adjacent ranks. Since we will sort the tiles into ascending order, this set will be built in ascending order, making the comparison rules slightly simpler.

We'll define a Set superclass to hold a sequence of Tiles. We will be able to add new tiles to a Set, as well as check to see if a tile could possibly belong to a Set. Finally, we can check to see if the Set is full. The superclass, Set, is abstract and returns NotImplemented for the full. The sublasses will override this methods with specific rules appropriate to the kind of set.

```
class  Set:
  def __init__(self):
  def __str__(self):
  def canContain(self, aTile):
  def add(self, aTile):
  def full(self):
  def fallback(self, tileStack):
  def pair(self):
```

- The Set __init__ function should create an internal list to store the tiles.
- The add method appends the new tile to the internal list. A pop function can remove the last tile appended to the list.
- The superclass canContain method returns True if the list is empty; it returns False if the list is full. Otherwise it compares the new tile against the last tile in the list to see if they are equal. Since most of the subclasses must match exactly, this rule is what is used. The straight subclass must override this to compare suit and rank correctly.
- The superclass fallback pushes all of the tiles from the Set back onto the given stack. The superclass version pushes the tiles and then returns None. Each subclass must override this to return a different fallback Set instance.
- The superclass pair returns False. The PairSet subclass must override this to return True.

An important note about the fallback is that the stack that will be given as an argument in tileStack is part of the Hand, and is using is maintainted by doing tileStack.pop(0) to get the first tile, and tileStack.insert(0, aTile) to push a tile back onto the front of the hand of tiles.

We'll need the following four subclasses of Set.

- **FourSet.** Specializes Set for sets of four matching tiles. The full method returns True when there are four elements in the list. The fallback method pushes the set's tiles onto the given tileStack; it returns a new ThreeSet with the first tile from the tileStack.
- **ThreeSet.** Specializes Set for sets of three matching tiles. The full method returns True when there are three elements in the list. The fallback method pushes the set's tiles onto the given tileStack; it returns a new SequenceSet with the first tile from the tileStack.
- **SequenceSet.** Specializes Set for sets of three tiles of the same suit and ascending rank. The belongs returns True for an empty list of tiles, False for a full list of

tiles, otherwise it compares the suit and rank of the last tile in the list with the new tile to see if the suits match and the new tile's rank is one more than the last tile in the list. The `full` method returns `True` when there are three elements in the list. The `fallback` method pushes the set's tiles onto the given `tileStack`; it returns a new `PairSet` with the first tile from the `tileStack`.

- **`PairSet`.** The `full` method returns `True` when there are two elements in the list. The `fallback` method is inherited from the superclass method in `Set`; this method returns None, since there is no fallback from a pair. This subclass also returns `True` for the `pair` method.

The idea is to attempt to use a `FourSet` to collect a group of tiles. If this doesn't work out, we put the tiles back into the hand, and try a `ThreeSet`. If this doesn't work out, we put the tiles back and try a `SequenceSet`. The last resort is to try a `PairSet`. There is no fallback after a pair set, and the hand cannot be a winner.

Hand Class

A Mah Jongg `Hand` object, then, is a list of `Tiles`. The `mahjongg` creates an assignment of individual `Sets`. It checks these sets to see if all of them are full, if there are five of them and if one of the five is a pair. If so, it returns `True` because the hand is a winning hand.

If we sort the tiles by name or suit, we can more effectively assign tiles to sets. The first step in the `mahjongg` function is to sort the tiles into order. Then the tiles can be broken into sets based on what matches between the tiles.

Procedure 41.1. Hand Scoring

The `mahjongg` function examines a hand to determine if the tiles can be assigned to five scoring sets, one of which is a pair.

1. **Sort Tiles.** Sort the tiles by name (or suit) and by rank for suit tiles where the suit matches. We will treat the hand of tiles as a the tile stack, popping and pushing tiles from position 0 using `pop(0)` and `insert(0,tile)`.

2. **Stack of Sets.** The candidate set definition is a stack of `Set` objects. Create an empty list to be used as the candidate stack. Create a new, empty `FourSet` and push this onto the top of the candidate stack.

3. **Examine Tiles.** Use the `examine` function to examine the tiles of the hand, assigning tiles to `Sets` in the candidate stack. When this operation is complete, we may have a candidate assignment that will contain a number of `Sets`, some of which are full, and some are incomplete. We may also have an empty stack because we have run out of fallback `Sets`.

4. **While Not A Winner.** While we have `Sets` in the candidate stack, use the `allFull` to see if all `Sets` are full, there are five sets, and there is exactly one pair. If we do not have five full `Sets` and a single pair, then we must fallback to another subclass of `Set`.

    a. **Retry.** Use the `retry` method to pop the last candidate `Set`, and use that `Set`'s `fallback` to create a different `Set` for examination. Save this `Set` in `n`.

    b. **Any More Assignments?** If the result of `retry` is `None`, there are no more fallbacks; we can return `False`.

    c. **Examine Tiles.** Append the `Set`, `n`, returned by retry to the candidate stack. Use the `examine` function to examine the tiles of the hand, assigning tiles to `Sets`. When this operation is complete, we may have a candidate

assignment that will contain a number of `Sets`, some of which are full, and some are incomplete.

5. **Winner?** If we finish the loop normally, it means we have a candidate set assignment which has five full sets, one of which is a pair. For some hands, there can be multiple winners; however, we won't continue the examination to locate additional winning assignments.

The `allFull` function checks three conditions: all `Sets` are full, there are five `Sets`, and is one `Set` is a pair. The first test, all `Sets` full, is an "and-reduce", using something like the following `reduce( lambda a,b: a and b, [s.full() for s in sets], True )`.

Procedure 41.2. Examine All Tiles

The `examine` function requires a non-empty stack of candidate `Sets`, created by the `mahjongg` method. It assigns all of the remaining tiles beginning with the top-most candidate `Set`. Initially, the entire hand is examined. After each retry, some number of tiles will have been pushed back into the hand for re-examination.

- **While More Tiles.** If the tile stack is empty, we are done.

    a. **Next Tile.** Pop the next unexamined tile from the tile stack, assigning it to the variable `t`.

    b. **Topmost Set Full?** If the topmost set on the set stack is full, push a new, empty `FourSet` onto the top of the set stack. This is also a handy place to use a **print** statement to watch the progress of the evaluation.

    c. **Topmost Set Can Contain?** If the top-most `Set` can contain `t`, add this tile to the set. We're done examining this tile.

    d. **Topmost Set Can't Contain.** Put the tile `t` back into the stack of tiles to be examined. Use the `retry` function to pop the `Set` from the stack, and fallback to another subclass of `Set`.

    e. **Another Retry?** If the result of the `retry` is `None`, we've run of alternatives, return from this function. Otherwise, append the new `Set` created by `retry` to the stack of candidate sets.

Procedure 41.3. Retry a Set Assignment

The `retry` function requires at least one `Set` in the assignments. This will pop that `Set`, pushing the tiles back into the hand. It will then use the popped `Set`'s `fallback` method to get another flavor of `Set` to try.

1. **Pop.** Pop the top-most set from the set stack, assign it to `s`. Call `s fallback` method to get a new top-most `Set`, assign this to `n`.

2. **Out Of Fallbacks?** While the set stack is not empty and `n` is `None`, there was no fallback.

    a. **Pop Another.** Pop the top-most set from the set stack, assign it to `s`. Call `s fallback` method to get a new top-most `Set`, assign this to `n`.

3. **Done?** If `n` is `None` and the set stack is empty, the hand is incomplete and we are out of fallback sets. Otherwise, append `n` to the stack of sets.

Some Test Cases

The following test case is typical. Bamboo: 2, 2, 2, 3, 4, 5, 5, 5; dots: 2, 2, 2; green

dragon, green dragon, green dragon. In this case, we will attempt to put the three 2 bamboo tiles into a set of four, pop that set, and put them into a set of three. The 3 will be put into a set of four, a set of three and then a straight with the 4 and 5. The next two fives will be put into a set of four, a set of three, a straight and a pair. The 2 dots tiles and the green dragon tiles will both be put into four sets and three sets. The final set stack will have a three set, a straight, a pair, and two three sets.

```
def testHand1():
    t1= [ SuitTile( 2, "Bamboo" ), SuitTile( 2, "Bamboo" ),
        SuitTile( 2, "Bamboo" ), SuitTile( 3, "Bamboo" ),
        SuitTile( 4, "Bamboo" ), SuitTile( 5, "Bamboo" ),
        SuitTile( 5, "Bamboo" ), SuitTile( 5, "Bamboo" ),
        SuitTile( 2, "Dot" ), SuitTile( 2, "Dot" ),
        SuitTile( 2, "Dot" ), HonorsTile( "Green" ),
        HonorsTile( "Green" ), HonorsTile( "Green" ), ]
    h1=  Hand( *t1 )
    print h1.mahjongg()
```

The following test case is a little more difficult. Bamboo: 2, 2, 2, 2, 3, 4, 3 × green dragon, 3 × red dragon, 3 × north wind. The initial run of four 2 bamboo tiles will be put into a set of four. The next 3 bamboo and 4 bamboo will be put into a four set, three set and straight. The first green dragon won't fit into the straight, causing us to pop the straight, attempt a pair, and pop this. We then pop the initial set of four two's and replace that with a set of three. The 2 bamboo and 3 bamboo will be checked against a four set and a three set before being put into a straight.

Here's a challenging test case with two groups of tiles that require multiple retries.

```
def testHand2():
    t2= [ SuitTile( 2, "Bamboo" ), SuitTile( 2, "Bamboo" ),
        SuitTile( 2, "Bamboo" ), SuitTile( 3, "Bamboo" ),
        SuitTile( 4, "Bamboo" ), SuitTile( 5, "Bamboo" ),
        SuitTile( 5, "Bamboo" ), SuitTile( 5, "Bamboo" ),
        SuitTile( 2, "Dot" ), SuitTile( 2, "Dot" ),
        SuitTile( 2, "Dot" ), SuitTile( 2, "Dot" ),
        SuitTile( 3, "Dot" ), SuitTile( 4, "Dot" ), ]
    h2=  Hand( *t2 )
    print h2.mahjongg()
```

Ideally, your overall unit test looks something like the following.

```
import unittest

class TestHand(unittest.TestCase):
    def testHand1( self ):
        body of testHand1
        self.assert_( h1.mahjongg() )
        self.assertEqual( str(h1.sets[0]),
            "ThreeSet['2B', '2B', '2B']" )
        self.assertEqual( str(h1.sets[1]),
            "SequenceSet['3B', '4B', '5B']" )
        self.assertEqual( str(h1.sets[2]),
            "PairSet['5B', '5B']" )
        self.assertEqual( str(h1.sets[3]),
            "ThreeSet['2D', '2D', '2D']" )
        self.assertEqual( str(h1.sets[4]),
            "ThreeSet['Green', 'Green', 'Green']" )
    def testHand2( self ):
        body of testHand2
        self.assert_( h2.mahjongg() )
        check individual Sets

if __name__ == "__main__":
    unittest.main()
```

A set of nine interesting test cases can be built around the following set of tiles: 3×1's, 2, 3, 4, 5, 6, 7, 8, and 3×9's all of the same suit. Adding any number tile of the same suit to this set of 13 will create a winning hand. Develop a test function that iterates through the nine possible hands and prints the results.

Hand Scoring - Points

A hand has a point value, based on the mixture of `Sets`. This point value is used to resolve the amount owed to the winner by the losers in the game. There is a subtlety to this evaluation that we have to gloss over, and that is the rules about for *concealed* and *exposed* or *melded* sets. For now, we will assume that all sets are concealed.

> During the play of Mah Jongg, a player will drawn a tile from the `Wall` and evaluate their hand. If the hand is a winner, the game is over. Otherwise, the player will discard a tile. With some additional considerations, a player can draw the last discarded tile to make a complete set; the player must expose the set to do this. If the player completes a set with a tile drawn from the wall, they do not have to expose it, so the set is concealed. Concealed sets are worth more than exposed sets, for obvious reasons.
>
> There are number of variations in the rules for drawing discarded tiles and exposing sets. We'll avoid much of this complexity, and focus on assigning point values to the sets using just the rules for concealed sets.

We need to expand our definition of `SuitTile`. There are two different score values for `SuitTiles`: the terminals (one and nine) have one score, and the simples (two through eight) have a different score. This will lead to two subclasses of `SuitTile`: `TerminalSuitTile` and `SimpleSuitTile`.

A winning hand has a base value of 20 points plus points assigned for each of the four scoring sets and the pair.

Set	Simples	Terminals or Honors
`SequenceSet`	0	0
`ThreeSet`	4	8
`FourSet`	16	32

The `PairSet` is typically worth zero points. However, the following kinds of pairs can add points to a hand.

- A pair of dragons is worth 2 points.
- A pair of winds associated with your seat at the table is worth 2 points.
- A full game consists of four rounds. Each round has a *prevailing* wind. Within each round, each of the players will be the dealer. A pair of the round's prevailing winds is worth 2 points.
- A *double wind* pair occurs when your seat's wind is also the prevailing wind. A pair of this wind is worth 4 points.

There are a few more ways to add points, all related to the mechanics of play, not to the hand itself.

Update the Tile Class Hierarchy. You will need to add two new subclass of `SuitTile`: `TerminalSuitTile` and `SimpleSuitTile`.

You will need to add a `simple` method to the `Tile` class which returns `False`. The

`SimpleSuitTile` (ranks 2 to 8), however, will override this method to return `True`.

You will need to add a `lucky(`*`prevalingWind`*`, `*`myWind`*`)` method to the `Tile` class which returns `False`. The `HonorsTile` will override this method to return `True` if the name is a dragon ("Red", "Green" or "White") or *`prevalingWind`* or *`myWind`*.

You will want to upgrade `Wall` to correctly generate the various `HonorsTile`, `TerminalSuitTile` and `SimpleSuitTile` instances.

You may also want to create a **Generator** for tiles. A function similar to the following can make programs somewhat easier to read.

```
def tile( *args ):
    """tile(name) -> HonorsTile
    tile( rank, suit ) -> SuitTile
    """
    if len(args) == 1:
        return HonorsTile( *args )
    elif args[0] in ( 1, 9 ):
        return TerminalSuitTile( *args )
    else:
        return SimpleSuitTile( *args )
```

Update the Set Class Hierarchy. You can then add a `points(`*`prevailingWind`*`, `*`myWind`*`)` to the `Set` class hierarchy. This function will examine the first `Tile` of the `Set` to see if it is `simple` or not, and return the proper number of points. The wind isn't used for most `Sets`.

In the case of `PairSet`, however, the first `Tile` must be checked against two rules. If *`prevailingWind`* is the same as *`myWind`* and the same as the tile's name, this is worth 4 points. If the tile's `lucky` method is `True` (a dragon, or one of the two winds), then the value is 2 points.

Update the Hand Class. You'll want to add a `points` function which computes the total number of points for a hand. You may also want to write a `pointReport` which prints a small scorecard for the hand, showing each set and the points awarded.

You will want to revise your unit tests, also, to reflect these changes. You'll also need to add additional unit tests to check the number of points in each hand.

For the first test cases in the previous the section called “Some Test Cases”, here are the scores.

Set	Points
Winning	20
ThreeSet['2B', '2B', '2B']	4
StraightSet['3B', '4B', '5B']	0
PairSet['5B', '5B']	0
ThreeSet['2D', '2D', '2D']	4
ThreeSet['Green', 'Green', 'Green']	8
Points	36

For the second test cases in the section called “Some Test Cases”, here are the scores.

Set	Points
Winning	20
ThreeSet['2B', '2B', '2B']	4
StraightSet['3B', '4B', '5B']	0
PairSet['5B', '5B']	0
ThreeSet['2D', '2D', '2D']	4

StraightSet['2D', '3D', '4D']	0
Points	28

Be sure to add a test case with *lucky tiles* (dragons or winds) as the pair.

Hand Scoring - Doubles

The point value for a hand can be doubled a number of times for a variety of rare achievements. Most of these rules of these are additional properties of `Sets` that are summarized by the `Hand`.

Each non-pair `Set` that contains lucky tiles is worth 1 double (2×). In the case of the player's wind being the prevailing wind, a `Set` of this wind is worth 2 doubles (4×)

A hand of four `ThreeSet` or `FourSet` (i.e., no `SequenceSet`) merits a double. Depending on how the hand was played and how many of these triples were concealed or melded, the hand can have a second double, or possibly even pay the house limit, something we'll look into in the section called "Limit Hands". For now, we are ignoring these mechanics of play issues, and will simply double the score if there are no `SequenceSets`.

A hand that has three consecutive `SequenceSets` in the same suit is doubled. There are many rule variations on this theme, including same-rank sequences from all three suits, same-rank `ThreeSets` or `FourSets` from all three suits. We'll focus on the three-consecutive rule for now.

If the hand is worth exactly 20 points (it is all `SequenceSets` and an unlucky `PairSet`), then it merits one double.

There are six different consistency tests. These are exclusive and at most one of these conditions will be true.

1. If the hand is all terminals and honors (no simples), it is doubled.

2. If each set in the hand has one terminal or an honor in it, the hand is doubled. A hand could have four `SequenceSets`, each of which begins with one or ends with nine, and a pair of honors or terminals to qualify for this double.

3. If the hand is all simples (no terminals or honors), it is doubled.

4. If all of the `SuitTiles` are of the same suit, and all other tiles are `HonorsTiles`, this is doubled.

5. If all of the `SuitTiles` are of the same suit, and there are no `HonorsTiles`, this is doubled four times (16×).

6. If the hand contains `Sets` of all three dragons, and one of those sets is a `PairSet`, this is called the *Little Three Dragons*, and the hand's points are doubled.

There are a few more ways to add doubles, all related to the mechanics of play, not to the hand itself.

Update Set Class Hierarchy. You'll need to add the following functions to the `Set` classes.

- A `lucky` function that returns `True` for a `ThreeSet` or `FourSet` with lucky tiles (dragons, the prevailing wind or the player's wind. It returns `False` for other kinds of `Sets` or `Sets` without lucky tiles.

- A `triplet` function that returns `True` for a `ThreeSet` or `FourSet`. It returns `False` for other kinds of `Sets`.

- A `sequenceSuit` function that returns the suit of a `SequenceSet`. It returns `None` for other kinds of `Sets`.
- A `sequenceRank` function that returns the lowest rank of a `SequenceSet`. It returns `None` for other kinds of `Sets`.
- An `allSimple` function that returns `True` if the `Set` contains only simple tiles.
- An `noSimple` function that returns `True` if the `Set` contains no simple tiles. This is an all terminals and honors `Set`.
- An `oneTermHonor` function that returns `True` if the `Set` contains one terminal or honor. Since we only have a `simple` function, this is one non-simple `Tile` in the `Set`.
- An `suit` function that returns the suit for a `Set` of all `SuitTiles` (either `SimpleSuitTiles` or `TerminalSuitTiles`). It returns `None` if the `Set` contains `HonorsTiles`.
- An `bigDragon` function that returns `True` for a `TreeSet` or `FourSet` of dragons.
- An `littleDragon` function that returns `True` for a `PairSet` of dragons.

Update Hand Class. You'll need to add the following functions to the `Set` classes.

- `luckySets(`*`prevailWind`*`,`*`myWind`*`)` returns the number of lucky sets. This function also checks for the double wind conditions where *`prevailWind`* is the same as *`myWind`* and one of the `Sets` has this condition and throws an additional doubling in for this.
- `groups` returns `1` if all `Sets` have the `triple` property `True`.
- `sequences` returns `1` if three of the `Sets` have the same value for `sequenceSuit`, and the values for `sequenceRank` are 1, 4, and 7.
- `noPoints` returns `1` all of the `Sets` are worth zero points.
- `consistency` returns `1` or `4` after checking for the following conditions. If `allSimple` is true for all `Sets`, return `1`. If `noSimple` is true for all `Sets`, return `1`. If `oneTermHonor` is true for all `Sets`, return `1`. If every `Set` has the same value for `suit` and there are no `Set` where `suit` is None, return `4`. If every `Set` has the same value for `suit` or the value for `suit` is None, return `1`. If there are two `Sets` for which `bigDragon` is true, and one `Sets` for which `littleDragon` is true, return `1`.

The sum of the double functions is the total number of doubles for the hand. This is `d = luckySets(`*`prevailWind`*`,`*`myWind`*`) + groups + sequences + noPoints + consistency` . This sum is the power of 2 to use: the score is multiplied by 2^d. An amazing hand of all one suit with three consecutive sequences leads to 5 doubles, 32× the base number of points.

You'll want to add a `doubleReport` which prints a small scorecard for the hand, showing each double that was awarded. You can then write a `scoreCard` which products the `pointReport`, the `doubleReport` and the final score of *points* × $2^{doubles}$.

The total score is often rounded to the nearest 10, as well as limited to 500 or less to produce a final score. This final score is used to settle up the payments at the end of the game.

There are a number of variations on the payments at the end of the game. The

simplest version has all losers paying an equal amount to the winner. There are number of variations which, for example, penalizes the player who's discard allowed another player to win. Generally, if the dealer wins, the payments to the dealer are doubled, and when the dealer loses, the payment to the winner is doubled.

Limit Hands

At the end of a hand of Mah Jongg, the winner is paid based on the final score of the hand. Generally, the final score is limited to 500 points. There are, however, some extraordinary hands which simply score this limit amount. These conditions are checked first; if none of these are true, then the normal hand scoring is performed.

- The *Big Three Dragons* hand has three `Sets` for which the `bigDragon` function is true.
- The *Little Four Winds* hand has three `ThreeSets` or `FourSetss` for which the `wind` function is true and a `PairSet` for which `wind` is true.
- The *Big Four Winds* hand has four `ThreeSets` or `FourSetss` for which the `wind` function is true.
- The *All Honors* hand has all `Sets` composed of `HonorsTiles`; these will all have either `wind` or `dragon` true.
- The *All Terminals* hand has all `Sets` composed of `TerminalSuitTiles`.

An additional hand that pays the limit also breaks many of the rules for a winning hand. This is the *Thirteen Orphans* hand, which is one each of the various terminals and honors: three dragons, four winds, three one's, three nine's and any other of the thirteen terminal and honor tiles. This requires a special-case test in `Hand` that short-cuts all of the evaluation algorithm.

An interesting limit hand is the *Nine Gates* hand, which is 3×1's, 2, 3, 4, 5, 6, 7, 8, and 3×9's all of the same suit. Any other tile of this suit will create a winning hand that pays the limit. Just considering the hand outside the mechanics of play, it would get four doubles because it is all one suit, plus the possibility of an additional double for consecutive sequences. The Nine Gates hand is only a limit hand if the player draws it as a completely concealed hand.

There are a few other limit hands, including all concealed triplets, or being dealt a winning hand. These, however, depend on the mechanics of play, not the hand itself.

Update Set Class Hierarchy. You'll want to add `wind` and `dragon` functions to the `Set` hierarchy. These return `True` if all `Tiles` in the `Set` are a wind or a dragon, respectively.

Update Hand Class. You can add six additional functions to `Hand` to check for each of these limit hands.

The final step is to update the `finalScore` to check for limit hands prior to computing points and doubles.

Chapter 42. Chess Game Notation

Table of Contents

Descriptive Notation
Game State
PGN Processing Specifications

See the section called "Chessboard Locations" for some additional background.

Chess is played on an 8x8 board. One player has white pieces, one has black pieces. Each player's pieces include eight pawns, two rooks, two knights, two bishops, a king and a queen. The various pieces have different rules for movement. Players move alternately until one player's king is in a position from which it cannot escape but must be taken, or there is a draw. There are a number of rules that lead to a draw, all beyond the scope of this problem. White moves first.

A game is recorded as a log of the numbered moves of pieces, first white then black. The Portable Game Notation (PGN) standard includes additional descriptive information about the players and venue.

There are two notations for logging a chess game. The newer, *algebraic* notation and the older *descriptive* notation. We will write a program that will process a log in either notation and play out the game, showing the chess board after each of black's moves. It can also be extended to convert logs to completely standard PGN notation.

Algebraic Notation

We'll present the formal definition of algebraic notation including Algebraic Notation (LAN) and Short Algebraic Notation (SAN). We'll follow this with a summary and some examples. This section will end with some Algorithm R, used to resolve which of the available pieces could perform a legal move.

Definition

Algebraic notations uses letters `a-h` for the files (columns across the board) from white's left to right, and numbers for the ranks (rows of the board) from white (1) to black (8).

Piece symbols in the log are as follows:

Piece	Symbol	Move Summary
Pawn	(omitted)	1 or 2 spaces forward
Rook	R	anywhere in the same rank or same file
Knight	N	2 in one direction and 1 in the other "L-shaped"
Bishop	B	diagonally, any distance
Queen	Q	horizontal, vertical or diagonal, any distance
King	K	1 space in any direction

The game beings in the following starting position. For white the pieces are arranged as follows:

White	Black	Piece
a1	a8	rook
b1	b8	knight
c1	c8	bishop
d1	d8	queen
e1	e8	king
f1	f8	bishop
g1	g8	knight
h1	h8	rook
a2-h2	a7-h7	pawns

There are two forms of algebraic notation: short (or standard) algebraic notation (SAN), where only the destination is shown, and long algebraic notation (LAN) that shows the piece, the starting location and the destination.

The basic syntax for LAN is as follows:

[*P*]*frmfr*[*n*]

P is the piece name (omitted for pawns), *f* is file (a-h), *r* is rank (1-8), *m* is move (`-` or `x`) and *n* is any notes about the move (`+`, `#`, `!`, `!!`, `?`, `??`). The notes may include `=` and a piece letter when a pawn is promoted.

Short notation omits the starting file and rank unless they are essential for disambiguating a move. The basic syntax is as follows:

[*P*] [*m*] [*d*]*fr*[*n*]

The move, *m*, is only specified for a capture (`x`). The optional *d* is either a file (preferred) or a rank or both used to choose which piece moved when there are multiple possibilities.

In both notations, the castle moves are written `O-O` or `O-O-O` (capital letters, not numbers). Similarly, the end of a game is often followed with a `1-0` (white wins), `0-1` (black wins), `1/2-1/2` (a draw), or `*` for a game that is unknown or abandoned.

Each turn is preceeded by a turn number. Typically the number and a `.` preceeds the white move. Sometimes (because of commentary), the number and `...` preceed the black move.

The PGN standard for notes is `$` and a number, common numbers are as follows:

1 good move (traditional "!")

2 poor move (traditional "?")

3 very good move (traditional "!!")

4 very poor move (traditional "??")

Each piece has a legal move. This is a critical part of processing abbreviated notation where the log gives the name of the piece and where it wound up. The legal moves to determine which of two (or eight) pieces could have made the requested move. This requires a simple search of pieces to see which could have made the move.

A pawn moves forward only, in its same file. For white, the rank number must increase by 1. It can increase by 2 when it is in its starting position (rank 7 for white or 2 for black). For black, the rank number must decrease by 1 or 2. A pawn captures on the diagonal: it will move into an adjacent file and forward one rank, replacing the piece that was there. There is one origin for all but the opening pawn moves (one rank back on the file in which the pawn ended its move); one origin for an opening pawn move that lands in rank 4 or 5 (two ranks back on the file where the pawn ended its move); two possible origins for any pawn capture (one position on a file adjacent to the one in which the pawn ended its move).

A rook moves in ranks or files only, with no limit on distance. There are 16 possible origins for any rook move, including the entire rank or the entire file in which the rook ended its move.

A knight makes an "L-shaped" move. It moves two spaces in one direction, turns 90-degrees and moves one more space. From g1, a knight can move to either f3 or h3. The rank changes by 2 and the file by 1; or the file changes by 2 and the rank changes by 1.

There are 8 places a knight could start from relative to its final location.

A bishop moves diagonally. The amount of change in the rank must be the same as the change in the file. There are 16 places a bishop can start from on the two diagonals that intersect the final location.

The queen's move combines bishop and rook: any number of spaces diagonally, horizontally or vertically. There are 16 places on the diagonals, plus 16 more places on the horizontals and verticals where the queen could originate. Pawns that reach the opposite side of the board are often promoted to queens, meaning there can be multiple queens late in the game.

The king is unique, there is only one. The king can only move one space hoizontally, vertically or diagonally.

The king and a rook can also engage in a move called *castling*: both pieces move. When the king and the closest rook (the one in file h) castle, this is *king side* and annoted `o-o`. The king moves from file e to file g and the rook from fiel h to file f. When the king and the queen's side rook (the one in file a) castle, this is annotated `o-o-o`. The king moves from file e to file c and the rook move from file a to file d. Castling can only be accomplished if (a) neither piece has moved and (b) space between them is unoccupied by other pieces. Part a of this rule requires that the game remember when a king or rook moves, and eliminate that side from available castling moves. Moving the rook in file a eliminates queen-side castling; moving the rook in file h eliminates king-side castling. Moving the king eliminates all castling.

Summary and Examples

Here's a suymmary of the algebraic notation symbols used for annotating chess games. This is followed by some examples.

Symbol	Meaning
a-h	file from white's left to right
1-8	rank from white to black
R, N, B, Q, K	rook, knight, bishop, king, queen
-	move (non-SAN)
x	capture; the piece that was at this location is removed
+	check, a note that the king is threatened
#	checkmate, a note that this is the reason for the end of the game
++	checkmate (non-SAN), a note that this is the reason for the end of the game
=	promoted to; a pawn arriving at the opposite side of the board is promoted to another piece, often a queen.
0-0	castle on the king's side; swap the king and the rook in positions e1 and h1 (if neither has moved before this point in the game)
0-0-0	castle on the queen's side; swap the king and the rook in positions e1 and a1 (if neither has moved before this point in the game)
e.p.	en passant capture (non-SAN), a note that a pawn was taken by another pawn passing it. When a pawn's first move is a two space move (from 7 to 5 for black or 2 to 4 for white) it can be captured my moving behind it to the 6th rank (white taking black) or 3rd rank (black taking white).
ep	en passant capture (non-SAN), see `e.p.`
?, ??, !, !!	editorial comments (non-SAN), weak, very weak, strong, very strong

Here's parts of an example game in abbreviated notation:

1. **e4 e5.** White pawn moves to e4 (search e3 and e2 for the pawn that could do this); black pawn moves to e5 (search e6 and e7 for a pawn that could do this)

2. **Nf3 d6.** White knight moves to f3 (search 8 positions: g1, h2, h4, g5, e5, d4, d2, e1 and g1 for the knight that could do this); black pawn moves to d6 (search d7 and d8 for the pawn).

3. **d4 Bg4.** White pawn moves from d4 (search d3 and d2 for the pawn); black bishop moves to g4 (search the four diagonals: f5, e6, d7, c8; h5; h3; f3, e3, and d3 for the bishop that could do this).

4. **dxe5 Bxf3.** A white pawn in d takes a piece at e5, the pawn must have been at d4, the black pawn at e5 is removed; a black bishop takes a piece at f3 (search the four radiating diagonals from f3: e4, d5, c6, b7, a8; g4, h5; g2, h1; e2, d1).

5. **Qxf3 dxe5.** The white queen takes the piece at f3; the black pawn in d4 takes the piece in e5.

Here's a typical game in abbreviated notation:

```
1.c4 e6 2.Nf3 d5 3.d4 Nf6 4.Nc3 Be7 5.Bg5 0-0 6.e3 h6 7.Bh4 b6
8.cxd5 Nxd5 9.Bxe7 Qxe7 10.Nxd5 exd5 11.Rc1 Be6 12.Qa4 c5
13.Qa3 Rc8 14.Bb5 a6 15.dxc5 bxc5 16.0-0 Ra7 17.Be2 Nd7
18.Nd4 Qf8 19.Nxe6 fxe6 20.e4 d4 21.f4 Qe7 22.e5 Rb8
23.Bc4 Kh8 24.Qh3 Nf8 25.b3 a5 26.f5 exf5 27.Rxf5 Nh7
28.Rcf1 Qd8 29.Qg3 Re7 30.h4 Rbb7 31.e6 Rbc7 32.Qe5 Qe8
33.a4 Qd8 34.R1f2 Qe8 35.R2f3 Qd8 36.Bd3 Qe8 37.Qe4 Nf6
38.Rxf6 gxf6 39.Rxf6 Kg8 40.Bc4 Kh8 41.Qf4 1-0
```

Here's a small game in full notation:

```
1.f2-f4 e7-e5 2.f4xe5 d7-d6 3.e5xd6 Bf8xd6 4.g2-g3 Qd8-g5
5.Ng1-f3 Qg5xg3+ 6.h2xg3 Bd6xg3#
```

Algorithms for Resolving Moves

Algebraic notation is terse because it is focused on a human student of chess. It contains just enough information for a person to follow the game. Each individual move cannot be interpreted as a stand-alone (or “context-free” statement). Each move's description only makes sense in the context of the game state established by all the moves that came before it. Therefore, in order to interpret a log of chess moves, we also need to maintain the state of the chess board.

Given that we have a model of the chess board, Algorithm G can locate the pieces and execute the moves an entire game.

Procedure 42.1. Algoritm G

(Resolve chess moves in SAN notation, playing out the entire game.) We are given a block of text with a sequence of chess turns. Assume that line breaks have been removed and the game ending marker has been removed from the block of text.

1. **Parse turn into moves.** Locate move number, white move and black move. Lines that don't have this form are some kind of commentary and can be ignored.

2. **Parse each move.** For each move, parse the move locating the piece (`R`, `B`, `N`, `Q`, `K`, pawn if none of these), optional file (`a-h`) or rank (`1-8`) for the source, the optional `x` for a capture, the destination file (`a-h`) and rank (`1-8`), and other material like `+` or `#` for checks, `=`*x* for promotions, or `!`, `!!`, `?`, `??` for editorial comments.

3. **Castling?** If the move is simply `0-0` or `0-0-0`, move both the king (in file e) and the appropriate rook. For `0-0` it is the rook in file h moves to f, the king moves from e to g. For `0-0-0` it is the rook in file a moves to d, the king moves from e to c.

4. **Fully specified from location?** If a two-character from-position is given, this is the starting location. Execute the move with step 7.

5. **Partially specified from location?** If a one-character from-position is given (a-h or 1-8), restrict the search for the source to this rank for file. Use the piece-specific version of Algorithm S with rank or file restrictions for the search. After the starting location is found, execute the move with step 7.

6. **Omitted from location?** Search all possible origins for the from-position for this piece. Each piece has a unique search pattern based on the piece's movement rules. Use the piece-specific version of Algorithm S with no restrictions for the search. After the starting location is found, execute the move with step 7.

7. **Execute move.** Move the piece, updating the state of the board, removing captured pieces. Periodically during game processing, print the board position. The board, by the way, is always oriented so that a1 is a dark square in the lower-left.

8. **Next move.** Loop to step 2, processing the black move after the white move in this turn. If the black move is omitted or is one of the ending strings, skip the black move.

9. **Next turn.** Loop to step 1, processing the next turn. If the turn number is omitted or is one of the ending strings, this is the end of the game.

We have to design six different kinds of searches for possible starting pieces. These searches include pawns, rooks, knights, bishops queens and the king. We'll provide formal algorithms for pawns and rooks, and informal specifications for the other pieces.

Procedure 42.2. Algorithm P

(Search for possible pawn starting locations.) Given a destination location, piece color, and optional restrictions on starting rank or starting file.

1. **First move.** If the destination is rank 4 (white) or rank 5 (black), search two spaces back for the first move of a pawn (from rank 7 or rank 2). If moving this piece will not put the king into check, this is the stating location.

2. **Previous space.** Search the previous space (rank -1 for white, rank +1 for black) for a move. If moving this piece will not put the same color king into check, this is the stating location.

3. **Capture.** Search the adjacent files one space back for a pawn which performed a capture. If moving this piece will not put the same color king into check, this is the stating location.

Procedure 42.3. Algorithm R

(Search for possible rook starting locations.) Given a destination location, piece color, and optional restrictions on starting rank or starting file.

1. **To Right.** Set r ← +1.

2. **Target.** Target position has file offset by r.

3. **Check.** If this is off the board, or a non-rook was found, skip to 4. If moving this piece will not put the king into check, this is the stating location.

4. **Loop.** Increment r. Repeat from step 2.

5. **To Left.** Set r ← -1.

6. **Target.** Target position has file offset by r.

7. **Check.** If this is off the board, or a non-rook was found, skip to 8. If moving this piece will not put the king into check, this is the stating location.

8. **Loop.** Decrement r. Repeat from step 5.

9. **To Black.** Set r ← +1.

10. **Target.** Target position has rank offset by r.

11. **Check.** If this is off the board, or a non-rook was found, skip to 12. If moving this piece will not put the king into check, this is the stating location.

12. **Loop.** Decrement r. Repeat from step 9.

13. **To White.** Set r ← -1.

14. **Target.** Target position has rank offset by r.

15. **Check.** If this is off the board, or a non-rook was found, we have a problem. If moving this piece will not put the king into check, this is the stating location.

16. **Loop.** Decrement r. Repeat from step 13.

Procedure 42.4. Algorithm N

(Search for possible knight starting locations.) Given a destination location, piece color, and optional restrictions on starting rank or starting file.

1. Search the eight possible starting positions for a knight's move. Four of the positions have a file offset of +1 or -1 and rank offsets of +2 or -2. The other four positions have file offsets of +2 or -2 and rank offsets of +1 or -1.

2. When evaluating a piece, moving this piece cannot put the king into check.

Procedure 42.5. Algorithm B

(Search for possible bishop starting locations.) Given a destination location, piece color, and optional restrictions on starting rank or starting file.

1. Search radially out the diagonals until edge of board or an intervening piece or the correct bishop was found. This algorithm is similar to the rook algorithm, except the offsets apply to both rank and file. Applying +1 to both rank and file moves north-east; applying -1 to both rank and file moves south-west. Applying +1 to rank and -1 to file moves south east; applying -1 to rank and +1 to file moves north west.

2. When evaluating a piece, moving this piece cannot put the king into check.

Procedure 42.6. Algorithm Q

Search for possible queen starting locations.) Given a destination location, piece color, and optional restrictions on starting rank or starting file.

1. Search radially out the ranks, files and diagonals until edge of board or an intervening piece or the correct queen was found. This combines the bishop and rook algorithms.

2. When evaluating a piece, moving this piece cannot put the king into check.

Procedure 42.7. Algorithm K

Search for possible king starting locations.) Given a destination location, piece color, and optional restrictions on starting rank or starting file.

- Search the 8 adjacent locations. These are all combinations of -1, 0, +1 for rank offset and -1, 0, +1 for file offset; skipping the one combination with a 0 offset to rank and a 0 offset to file.

Descriptive Notation

Descriptive notation uses a different scheme for identifying locations on the board. Each file is named for the pieces at it's top and bottom ends as the game begins. The board is divided into King's side and Queen's side. The files are KR, KKt, KB, K, Q, QB, QKt, QR. These are known as a, b, c, d, e, f, g, h in Algebraic notation.

The ranks are counted from the player's point of view, from their back row to the far row. Consequently, white's row 1 is black's row 8. White's Q1 is Black's Q8; Black's KB5 is White's KB4.

The notation has the following format:

***piece*[(*file rank*)]*move*[*file rank*][*note*]**

The symbol for the *piece* to be moved is one of p, B, N, R, Q, K.

If capturing, the *move* is `x`, followed by the symbol of the captured piece. Examples: pxp, NxQ. A search is required to determine which piece can be taken.

If not capturing, the *move* is `-`, followed by *filerank* to name the square moved to, from the perspective of whoever is moving, black or white

If 2 pieces are both be described by a move or capture, write the location of the intended piece in parentheses. Examples: `p(Q4)xR` means pawn at queen's rook four takes Rook, `N(KB3)-K5` means knight at KB3 moves to K5

Special moves include king's side castling, designated O-O, Queen's side castling, designated O-O-O.

Notes. If a pawn captures *en passant* or *in passing* it is designated `ep` in the *note*. A move resulting in a check of the king is followed by `ch` in the *note*. `!` means `good move`; `?` means `bad move` in the *note*.

If the pawn in front of the king is moved forward two spaces, it is described as `P-K4`. If the pawn in front of the queenside knight is moved forward one space, it is `P-QN3`. If a knight at K5 captures a rook on Q7, it would be `NxR` or if clarification is needed, `NxR(Q7)` or `N(K5)xR`.

Game State

In order to process a log, a model of the chess board's current position is essential. In addition to the basic 64 squares containing the pieces, several additional facts are necessary to capture the game state. The current state of a chess game is a 6-tuple of the following items:

Piece Placement. An 8-tuple showing pieces in each rank from 8 down to 1. Pieces are shown as single letters, upper case for white (PRNBQK), lower case for black (prnbqk). Pieces are coded P for pawn, R for rook, N for knight, B for bishop, Q for queen and K for king. Empty spaces are shown as the number of contiguous spaces. The entire rank can be coded as a 1-8 character string. 8 means no pieces in this rank. 4p3 means four empty spaces (a-d), a black pawn in file e, and 3 empty spaces (f-h). The entire 8-tuple

of strings can be joined to make a string delimited by /'s. For example rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R.

Active Color. A value (w or b) showing who's turn to move is next. This color is *active* because this player is contemplating their move. The starting position, for instance, has an active color of w, because white moves first.

Castling Availability. A string with 1 to 4 characters showing which castling moves are still allowed. If none are allowed, a - is shown. White codes are capital letters, black are lower case. When king side castling is possible, a K (white) or k (black) is included. When queen side caslting is possible a Q (white) or q (black) is included. At the start of the game, there are four characters: KQkq. As the game progress and kings castle or are moved for other reason, or rooks are moved, the string reduces in size to -.

***En Passant* target.** Either - or a square in rank 6 or 3. When a pawn's first move advances two spaces (from 7 to 5 for black or 2 to 4 for white), the skipped-over space is named here on the next turn only. If an opposing pawn moves to this space, an *En Passant* capture has occured. If no *En Passant* vulnerability, a - is given.

Half Move Count. How many 1/2 moves since a pawn was advanced or a piece captures. This is zero after a pawn moves or a piece is captured. This is incremented after each 1/2 move (white or black) where no pawn moves and no piece is captured. When this reaches 50, the game is technically a draw.

Turn. This is the turn count, it increments from 1, by 1, after black's move.

PGN Processing Specifications

There are several parts to a PGN processing program. There is the parsing of a PGN input file, the resolution of moves, and maintenance of the game state. Each can be dealt with separately with suitable interfaces. Each of these modules can be built and tested in isolation.

First, some preliminaries. In order to resolve moves, the game state must be kept. This is a dictionary of locations and pieces, plus the five other items of information that characterize the game state: active color (w or b), castling availability, *en passant* target, half-move draw count and turn number. The board has an interface that accepts a move and executes that move, updating the various elements of board state.

Moves can use the Command design pattern to separate king-side castle, queen-side castle, moves, captures and promotions. The Board object will require a fully-specified move with source location and destination location. The source location is produced by the source resolution algorithm.

A well-defined Board object could be used either for a single-player game (against the computer) or as part of a chess game server for two-player games.

Second, the hard part: resolution of short notation moves. Based on input in algebraic notation, a move can be transformed from a string into a 7-tuple of color, piece, fromHint, moveType, toPosition, checkIndicator and promotionIndicator.

- The color is either w or b.
- The piece is omitted for pawns, or one of RNBQK for the other pieces.
- The fromHint is the from position, either a file and rank or a file alone or a rank alone. The various search algorithms are required to resolve the starting piece and location from an incomplete hint.
- The moveType is either omitted for a simple move or x for a capturing move.
- The toPosition is the rank and file at which the piece arrives.
- The checkIndicator is either nothing, + or #.
- The promotionIndicator is either nothing or a new piece name from QBRK.

This information is used by Algorithm G to resolve the full starting position information for the move, and then execute the move, updating the board position.

Finally, input parsing and reporting. A PGN file contains a series of games. Each game begins with identification tags of the form `[Label "value"]`. The labels include names like `Event`, `Site`, `Date`, `Round`, `White`, `Black`, `Result`. Others labels may be present. After the identification tags is a blank line followed by the text of the moves, called the "movetext". The movetext is supposed to be SAN (short notation), but some files are LAN (long notation). The moves should end with the result (`1-0`, `0-1`, `*`, or `1/2-1/2`), followed by 1 or more blank lines.

In order to handle various forms for the movetext, there have to be two move parsing classes with identical interfaces. These polymorphic classes implement long-notation and short-notation parsing. In the event that a short-notation parser object fails, then the long-notation parser object can be used instead. If both fail, the file is invalid.

A PGN processing program should be able to read in a file of games, execute the moves, print logs in various forms (SAN, LAN and Descriptive), print board positions in various forms. The program should also be able to convert files from LAN or Descriptive to SAN. Additionally, the processor should be able to validate logs, and produce error messages when the chess notation is invalid.

Additionally, once the basic PGN capabilities are in place, a program can be adapted to do analysis of games. For instance it should be able to report only games that have specific openings, piece counts at the end, promotions to queen, castling, checks, etc.

Bibliography

Use Cases

[Jacobson92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. Copyright © 1992. 0201544350. Addison-Wesley. *Object-Oriented Software Engineering*. A Use Case Driven Approach.

[Jacobson95] Ivar Jacobson, Maria Ericsson, and Agenta Jacobson. Copyright © 1995. 0201422891. Addison-Wesley. *The Object Advantage*. Business Process Reengineering with Object Technology.

Computer Science

[Boehm81] Barry Boehm. Copyright © 1981. 0138221227. Prentice-Hall PTR. *Software Engineering Economics*.

[Comer95] Douglas Comer. Copyright © 1995. 3rd edition. 0132169878. Prentice-Hall. *Internetworking with TCP/IP*. Vol. I. Principles, Protocols, and Architecture.

[Cormen90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Copyright © 1990. 0262031418. MIT Press. *Introduction To Algorithms*.

[Dijkstra76] Edsger Dijkstra. Copyright © 1976. 0613924118. Prentice-Hall. *A Discipline of Programming*.

[Gries81] David Gries. Copyright © 1981. 0387964800. Springer-Verlag. *The Science of Programming*.

[Holt78] R. C. Holt, G. S. Graham, E. D. Lazowska, and M. A. Scott. Copyright © 1978. 0201029375. Addison-Wesley. *Structured Concurrent Programming with Operating Systems Applications*.

[Knuth73] Donald Knuth. Copyright © 1973. 0201896834. Addison-Wesley. *The Art of Computer Programming*. Volume 1. Fundamental Algorithms.

[Meyer88] Bertrand Meyer. Copyright © 1988. 0136290493. Prentice Hall. *Object-Oriented Software Construction*.

[Parnas72] D. Parnas. Copyright © 1972. Communications of the ACM. *On the Criteria to Be Used in Decomposing Systems into Modules*. 5. 12. December 1972. 1053-1058.

[SEIstr04] Software Engineering Institute. Copyright © 2004. http://www.sei.cmu.edu/str/descriptions/index.html. *Software Technology Roadmap*.

Design Patterns

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Copyright © 1995. 0201633612. Addison-Wesley Professional. *Design Patterns*. Elements of Object-Oriented Software.

[Larman98] Craig Larman. Copyright © 1998. 0137488807. Prentice-Hall. *Applying UML and Patterns*. An Introduction to Object-Oriented Analysis and Design.

[Lott05] Steven Lott. Copyright © 2005. Steven F. Lott. http://homepage.mac.com/s_lott/books/oodesign/oodesign.pdf. *Building Skills in Object-Oriented Design*. Step-by-Step Construction of A Complete Application.

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. Copyright © 1991. 0136298419. Prentice Hall. *Object-Oriented Modeling and Design*.

Languages

[Geurts91] Leo Geurts, Lambert Meertens, and Steven Pemberton. Copyright © 1991. 0-13-000027-2. Prentice-Hall. *The ABC Programmer's Handbook*.

[Gosling96] Gosling and McGilton. Copyright © 1996. Sun Microsystems. http://java.sun.com/docs/white/langenv/. *Java Language Environment White Paper*.

[Harbison92] Samuel P. Harbison. Copyright © 1992. 0-13-596396-6. Prentice-Hall. *Modula-3*.

[vanRossum04] Guido van Rossum and Fred L. Drake, jr.. Copyright © 2004. Python Labs. http://www.python.org/doc/. *Python Documentation*.

[Sun04] Copyright © 2004. Sun Microsystems. http://java.sun.com/reference/docs/index.html. *Java Technology Reference Documentation*.

[Wirth74] Copyright © 1974. North-Holland. *Proceedings of the IFIP Congress 74*. “On the Design of Programming Languages”. Niklaus Wirth. 386-393.

Problem Domains

[Aceten04] . Copyright © 2004. http://www.ace-ten.com. *Ace-Ten*.

[Banks02] Robert B. Banks. Copyright © 2002. 0-691-10284-8. Princeton University Press. *Slicing Pizzas, Racing Turtles, and Further Adventures in Applied Mathematics*.

[Dershowitz97] Nachum Dershowitz and Edward M. Reingold. Copyright © 1997. 0-521-56474-3. Cambridge University Press. *Calendrical Calculations*.

[Latham98] Lance Latham. Copyright © 1998. 0-87930-496-0. Miller-Freeman. *Standard C Date/Time Library*.

[Meeus91] Jean Meeus. Copyright © 1991. 0-943396-35-2. Willmann-Bell Inc..

Astronomical Algorithms.

[Neter73] John Neter, William Wasserman, and G. A. Whitmore. Copyright © 1973. 020503853. 4th edition. Allyn and Bacon, Inc.. *Fundamental Statistics for Business and Economics*.

[OBeirne65] T. M. O'Beirne. Copyright © 1965. Oxford University Press. *Puzzles and Paradoxes*.

[Shackleford04] Michael Shackleford. Copyright © 2004. http://www.wizardofodds.com. *The Wizard Of Odds*.

[Silberstang05] Edwin Silberstang. Copyright © 2005. 0805077650. 4th edition. Owl Books. *The Winner's Guide to Casino Gambling*.

[Skiena01] Steven Skiena. Copyright © 2001. 0521009626. Cambridge University Press. *Calculated Bets*. Computers, Gambling, and Mathematical Modeling to Win.

Colophon

The following toolset was used for production of this book.

- Python 2.5.1.
- Java 1.5.0_07-164.
- Docbook XSL stylesheets 1.68.1. These had to be extended to correctly format class synopsis in Python.

- XMLMind XML Editor 3.5.1.

- Saxon 6.5.5.
- FOP 0.20.5.
- NeoOffice 2.1