Programming for Non-Programmers

Release 2.6.2

Steven F. Lott

January 23, 2012

CONTENTS

Part I

How To Write Your Own Software Using Python

The Walrus and the Carpenter – Lewis Carroll

"The time has come," the Walrus said,
"To talk of many things:
Of shoes – and ships – and sealing-wax –
Of cabbages – and kings –
And why the sea is boiling hot –
and whether pigs have wings."

CHAPTER

ONE

PREFACE

1.1 Why Read This Book?

You'll need to read this book when you have the following three things happening at the same time:

- You have a problem to solve that involves *data* and *processing*.
- You've found that the common desktop tools (word processors, spread sheets, databases, organizers, graphics) won't really help. You've found that they require too much manual pointing and clicking, or they don't do the right kinds of processing on your data.
- You're ready to invest some of your own time to learn how to write customized software that will solve your problem.

You'll want to read this book if you are tinkerer who likes to know how things really work. For many people, a computer is just an appliance. You may not find this satisfactory, and you want to know more. People who tinker with computers are called hackers, and you are about to join their ranks.

Python is what you've been looking for. It is an easy-to-use tool that can do any kind of processing on any kind of data. Seriously: any processing, any data. *Programming* is the term for setting up a computer to do the processing you define on your data. Once you learn the Python language, you can solve your data processing problem.

Our objective is to get you, a non-programming newbie, up and running. When you're done with this book, you'll be ready to move on to a more advanced Python book. For example, a book about the Python libraries. You can use these libraries can help you build high-quality software with a minimum of work.

1.2 What Is This Book About?

This book is about many things. The important topics include Python, programming, languages, data, processing, and some of the skills that make up the craft of programming. We'll talk about the core intellectual tools of *abstraction*, *algorithms* and the formality of computer languages. We'll also touch on math and logic, statistics, and casino games.

Python. Python is a powerful, flexible toolbox and workbench that can help solve your data processing problem. If you need to write customized software that does precisely what you want, and you want that software to be readable, maintainable, adaptable, inexpensive and make best use of your computer, you need Python.

Programming. When we've written a sequence of statements in the Python language, we can then use that sequence over and over again. We can process different sets of data in a standard, automatic fashion. We've created a program that can automate data processing tasks, replacing tedious or error-prone pointing and clicking in other software tools. Also, we can create programs that do things that other desktop tools can't do at all.

The big picture is this: the combination of the Python *program* plus a unique sequence of Python language *statements* that we create can have the effect of creating a new application for our computer. This means that our application uses the existing Python program as its foundation. The Python program, in turn, depends on many other libraries and programs on your computer. The whole structure forms a kind of *technology stack*, with our program on top, controlling the whole assembly.

Languages. We'll look at three facets of a programming language: how you write it, what it means, and the additional practical considerations that make a program useful. We'll use these three concepts to organize our presentation of the language. We need to separate these concepts to assure that there isn't a lot of confusion between the real meaning and the ways we express that meaning.

The sentences "Xander wrote a tone poem for chamber orchestra" and "The chamber orchestra's tone poem was written by Xander" have the same meaning, but express it different ways. They have the same *semantics*, but different *syntax*. For example, in one sentence the verb is "wrote", in the other sentence it is "was written by" : different forms of the verb *to write*. The first form is written in *active voice*, and second form is called the *passive voice*. Pragmatically, the first form is slightly clearer and more easily understood.

The *syntax* of the Python language is covered here, and in the *Python Reference Manual* [PythonRef]. Python syntax is simple, and very much like English. We'll provide many examples of language syntax. We'll also provide additional tips and hints focused on the newbies and non-programmers. Also, when you install Python, you will also install a *Python Tutorial* [PythonTut] that presents some aspects of the language, so you'll have at least three places to learn syntax.

The *semantics* of the language specify what a statement really means. We'll define the semantics of each statement by showing what it makes the Python program do to your data. We'll also be able to show where there are alternative syntax choices that have the same meaning. In addition to semantics being covered in this book, you'll be able to read about the meaning of Python statements in the *Python Reference Manual* [PythonRef], the *Python Tutorial* [PythonTut], and chapter two of the *Python Library Reference* [PythonLib].

In this book, we'll try to provide you with plenty of practical advice. In addition to breaking the topic into bite-sized pieces, we'll also present lots of patterns for using Python that you can apply to real-world problems.

Extensions. Part of the Python technology stack are the extension *libraries*. These libraries are added onto Python, which has the advantage of keeping the language trim and fit. Software components that you might need for specialized processing are kept separate from the core language. Plus, you can safely ignore the components you don't need.

This means that we actually have two things to learn. First, we'll learn the language. After that, we'll look at a few of the essential libraries. Once we've seen that, we can see how to make our own libraries, and our own application programs.

1.3 Audience

Programming and Computer Skills. We're going to focus on programming skills, which means we have to presume that you already have general computer skills. You should fit into one of these populations.

- You have good computer skills, but you want to learn to program. You are our target crew. Welcome aboard.
- You have some programming experience, and you want to learn Python. You'll find that most of *Getting Started* is something you can probably skim through. We've provided some advanced material that you may find interesting.

What skills will you need? How will we build up your new skills?

Skills You'll Need. This book assumes an introductory level of skill with any of the commonly-available computer systems. Python runs on almost any computer; because of this, we call it *platform-independent*. We won't presume a specific computer or operating system. Some basic skills will be required. If these are a problem, you'll need to brush up on these before going too far in this book.

- Can you download and install software from the internet? You may need to do this to get the Python distribution kit from http://www.python.org. If you've never downloaded and installed software before, you may need some help with that skill.
- Do you know how to create text files? We will address doing this using a program called **IDLE**, the Python Integrated Development Environment. If you don't know how to create folders and files, or if you have trouble finding files you've saved on your computer, you'll need to expand those skills before trying to do any programming.
- Do you know some basic algebra? Some of the exercises make use of some basic algebra. A few will compute some statistics. We shouldn't get past high-school math, and you probably don't need to brush up too much on this.

How We Help. Newbie programmers with an interest in Python are our primary audience. We provide specific help for you in a number of ways.

- Programming is an activity that includes the language skills, but also includes design, debugging and testing; we'll help you develop each of these skills.
- We'll address some of the under-the-hood topics in computers and computing, discussing how things work and why they work that way. Some things that you've probably taken for granted as a user become more important as you grow to be a programmer.
- We won't go too far into software engineering and design. We need to provide some hints on how software gets written, but this is not a book for computer professionals; it's for computer amateurs with interesting data or processing needs.
- We cover a few of the most important modules to specifically prevent newbie programmers from struggling or – worse – *reinventing the wheel* with each project. We can't, however, cover too much in a newbie book. When you're ready for more information on the various libraries, you're also ready for a more advanced Python book.

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

- Use the core language constructs: variables, statements, exceptions, functions and classes. There are only twenty statements in the language, so this is an easy undertaking.
- Use the Python *collection* classes to work with more than one piece of data at a time.
- Use a few of the Python extension libraries. We're only going to look at libraries that help us with finishing a polished and complete program.

A Note on Clue Absorption. Learning a programming language involves accumulating many new and 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 below this limit, we've found that it helps to build up the language as ever-expanding layers. We'll start with a very tiny, easy to understand subset of statements; to this we'll add concepts until we've covered the entire Python language and all of the built-in data types.

Our part of the agreement is to do things in small steps. Here's your part: you learn a language by using it. In order for each layer to act as a foundation for the following layers, you have to let it solidify by doing small programming exercises that exemplify the layer's concepts. Learning Python is no different from learning Swedish. You can read about Sweden and Swedish, but you must actually use the language to get it off the page and into your head. We've found that doing a number of exercises is the only way to internalize each language concept. There is no substitute for hands-on use of Python. You'll need to follow the examples and do the exercises. As you can probably tell from this paragraph, we can't emphasize this enough.

The big difference between learning Python and learning Swedish is that you can immediately interact with the **Python** program, doing real work in the Python language. Interacting in Swedish can more difficult. The point of learning Swedish is to interact with people: for example, buying some *kanelbulle* (cinnamon buns) for *fika* (snack). However, unless you live in Sweden, or have friends or neighbors who speak Swedish, this interactive part of learning a human language is difficult. Interacting with Python only requires a working computer, not a trip to Kiruna.

Also, your Swedish phrase-book gives you little useful guidance on how to pronounce words like *sked* (spoon) or *sju* (seven); words which are notoriously tricky for English-speakers to get right. Python, however, is a purely written language so you don't have subtleties of pronunciation, you only have spelling and grammar.

1.4 Conventions Used in This Book

Here is how we'll show Python programs in the rest of the book. The programs will be in separate boxes, in a different font, often with numbered "callouts" to help explain the program. This example is way too advanced to read in detail (it's part of *Mappings : The dict*) it just shows what examples look like.

Python Example

```
1  from __future__ import print_function, division
2  combo = { }
3  for i in range(1,7):
4      for j in range(1,7):
5          roll= i+j
6          combo.setdefault( roll, 0 )
7          combo[roll] += 1
8  for n in range(2,13):
9      print("{0:d} {1:.2f}".format(n, combo[n]/36))
```

Line 1 creates a Python dictionary, a map from key to value. In this case, the key will be a roll, a number from 2 to 12. The value will be a count of the number of times that roll occurred.

Line 5 assures that the rolled number exists in the dictionary. If it doesn't exist, it will default, and will be assigned frequency count of 0.

Line 7 prints each member of the resulting dictionary.

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()`, *attribute*, which includes both class variables or instance variables.

Sidebars

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

Tip: Tips

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 programmers.

CHAPTER

TWO

GETTING STARTED

Tools and Toys

This part provides some necessary background to help non-programming newbies get ready to write their own programs. If you have good computer skills, this section may be all review. If you are new to programming, our objective is to build up your skills by providing as complete an introduction as we can. Computing has a lot of obscure words, and we'll need some consistent definitions.

In *Let There Be Python: Downloading and Installing* we'll describe how to install Python. This is mostly for folks using Windows. Mac OS X and Linux users will find they already have Python installed. This chapter has the essential first step in starting to build programs: getting our tools organized.

We'll describe two typical problems that Python can help us solve in *Two Minimally-Geeky Problems : Examples of Things Best Done by Customized Software*. We'll provide many, many more exercises and problems than just these two. But these are representative of the problems we'll tackle.

2.1 About Python

Our goal is to write Python programs to solve problems that involve data and processing. Doing Python programming involves two things.

1. Designing and writing *programs*: statements in the Python language that could control a computer system.
2. Running the Python program to evaluate (or interpret) these statements and actually do the useful work we imagined. This is the goal.

This leads to the very important distinction:

- Python is a *program* that does data processing. This is an operational view.
- You control the Python program using the Python *programming language*. This is a software design view.

What does this distinction mean? First, there is an opportunity for us to confuse Python (the program) and Python (the language). We'll attempt to be as clear as we can on the things the Python program does when you give it commands in the Python Language. This is the distinction beytween semantics (what Python does) and syntax (how we spell it).

For people very new to programming, this raises further questions like "what is a programming language?" and "why can't it just use English?" and "what if I'm not good with languages?" We'll return to these topics in *Concepts FAQ's*. For now, we'll emphasize the point that the Python language is more precise than English, but still very easy to read and write.

The other thing that the distinction between program and language means is that we will focus our efforts on learning the language. The data processing will be completely defined by a sequence of statements in the Python language. Learning a computer language isn't a lot different from learning a human language, making our job relatively easy. We'll be reading and writing Python in no time.

We'll look at the concepts of software design in *About Programming*.

For now, however, let's look at the Python the program.

2.1.1 What is a Program?

We're going to look a closely at these things called "programs". The concept is multi-faceted, so we'll have to look at programs from several points of view. In *Software Terminology* there's a kind of road map to computers and software that may provide helpful background.

The essence of a program is the following:

> **A program sets up a computer to do a specific task**.

We could say that it is a program which *applies* a general-purpose computer to a specific problem. That's why we call them *application programs* or sometimes just *applications*; A program applies a generalized computing appliance to a specific data processing purpose.

There is a kind of parallel between a computer system running programs and a television playing a particular TV show. Without the program, the computer is just a pile of inert electronics. Similarly, if there is no TV show, the television just sits there showing a blank screen. (When I was a kid, a TV with no program showed a flickering "noise" pattern. Modern TV's don't do this, they just sit there.)

We will take two differing views of a program: the *data* side and the *processing* side. We're separateing information from action. We'll be learning to write programs which read and write files of data, much like our ordinary desktop tools open and save files. These programs will consume a file of data, the processing will update an internal state (like a count or a total) and possibly create a resulting file of data.

We aren't excluding game programs or programs that control physical processes. A game's input data includes the control actions from the player (or sensors attached to a device) plus the description of the game's levels and environments. The processing that a game does consumes the inputs; based on the current state, it determines what happens next; and it creates a new game state.

2.1.2 Program Varieties

To understand how a program we write depends on the underlying Python program, we need to make a distinction between some varieties of programs: specifically, between *binary executable* and *script* programs.

A binary executable is a program that is given direct control computer's processor. We call it binary because it uses the binary codes specific to the processor chip inside the computer. If you haven't encountered "binary" before, see *Binary Codes*. Most programs that you buy or download fit this description. Most of the office applications you use are binary executables.

The `python` program (named `python.exe` in Windows) is a binary executable. It takes direct control over the processor.

Note: Other Pythons

There are other varieties of Python, including CPython, Jython and PyPy. We're focused on CPython. Jython and PyPy aren't actually binary exectables, making them somewhat more indirect. This has the undesirable effect of making the explanation a hair more complex, so we'll ignore them.

A script, on the other hand, is used to control a program. A script doesn't take direct control over the processor. It doesn't rely (directly) on binary codes. The Python language is a scripting language; it controls the computer system indirectly, via the Python binary program.

Our programs will be scripts that control the underlying Python program.

Your operating system is a complex collection of binary executables and scripts. These operating system programs don't solve any particular problem, but they enable the computer to be used by folks who do have a particular problem to solve.

A binary executable's direct control over the processor is beneficial because it gives the best speed and uses the fewest resources. However, the cost of this control is the relative opacity of the coded instructions that control the processor chip. The processor instruction codes are focused on the electronic switching arcana of gates, flip-flops and registers. They are not focused on data processing at a human level. If you want to see how complex and confusing the processor chip can be, go to Intel or AMD's web site and download the technical specifications for one of their processors.

One subtlety that we have to acknowledge is that even the binary applications don't have *complete* control over the entire computer system. A computer system loads a kernel of software when it starts. The parts we interact with are actually outside this kernel. The binary applications we use do parts of their work by using the kernel. This important design feature of the operating system assures that all of the application programs behave consistently and share resources politely.

Binary Codes

Binary codes were invented by the inhabitants of the planet Binome, the Binome Individual uniTs, or BITs. These creates had two hands of four fingers each, giving them eight usable digits instead of the ten that most Earthlings have. Unlike Earthlings, who use their ten fingers to count to ten, the BITs use only their right hands and can only count to one.

If their hand is down, that's zero. If they raise their hand, that's one. They don't use their left hands or their fingers. It seems like such a waste, but the BITs have a clever work-around

If a BIT want to count to a larger number, say ten, they recruit three friends. Four BITs can then chose positions and count to ten with ease. The right-most position is worth 1. The next position to the left is worth 2. The next position is worth 4, and the last position is worth 8.

The final answer is the sum of the positions with hands in the air.

Say we have BITs named Alpha, Bravo, Charlie and Delta standing around. Alpha is in the first position, worth only 1, and Delta is in the fourth position, worth 8. If Alpha and Charlie raise their hands, this is positions worth 1 and 4. The total is 5. If all four BITs raise their hands, it's 8+4+2+1, which is 15. Four BITs have 16 different values, from zero (all hands down) to 15 (all hands up).

Delta (8)	Charlie (4)	Bravo (2)	Alpha (1)	total
down	down	down	down	0
down	down	down	up	1
down	down	up	down	2
down	down	up	up	$2 + 1 = 3$
down	up	down	down	4
down	up	down	up	$4 + 1 = 5$
down	up	up	down	$4 + 2 = 6$
down	up	up	up	$4 + 2 + 1 = 7$
up	down	down	down	8
up	down	down	up	$8 + 1 = 9$
up	down	up	down	$8 + 2 = 10$

A party of eight BITs can show 256 different values from zero to 255. A group of thirty-two BITs can count to over 4 billion.

The reason this scheme works is that we only have two values: on and off. This two valued (binary) system is easy to build into electronic circuits: a component is either on or off. Internally, our processor chip works in this binary arithmetic scheme because its fast and efficient.

2.1.3 The Python Program and What It Does

In *What is a Program?* we noted the difference between a binary executable and a script. A binary program uses codes that are specific to our processor chip and operating system. A script, written in an easy-to-read language like Python, controls that binary program.

The **Python** program (`python` or `python.exe`) is described as an *interpreter* or a *virtual machine*. The **Python** program's job is to read statements in the Python language and execute those statements. For historical reasons, this process is called "interpreting" the program. You might think that an interpreter should be *translating* the program into another language; after all, that's what human interpreters do. Software people have bent the meaning of this term, and the process of executing a script is called interpreting.

The term virtual machine summarizes the way that the principle of abstraction is applied. Rather than write Python programs that are specific to each unique machine (with it's different devices and chipsets and memory), Python offers a uniform, easier-to-work-with virtual machine that sits squarely on the back of the real machine.

Ultimately, everything that happens in a computer is the result of the processor executing it's internal binary instructions. In the case of **Python**, the `python` program contains a set of binary instructions that will read the Python-language statements we provided and execute those statements.

2.1.4 Layers of Abstraction

There is a sort of parallel between a computer running a program and a TV playing a particular TV show. The TV device is the computer and the show is the application software.

At a more detailed view, however, our computer is really a composite concept of computer hardware plus Operating System. It is really this hardware-plus-software device which runs our application programs.

Our TV metaphor starts to break down. We don't have to get a kernel TV show that allows our TV to watch a specific channel. A standard television is complete by itself.

A computer, however, can't do anything without *some* software. We start with a kernel of OS software to help us run our desired application software.

This is a slightly more helpful picture.

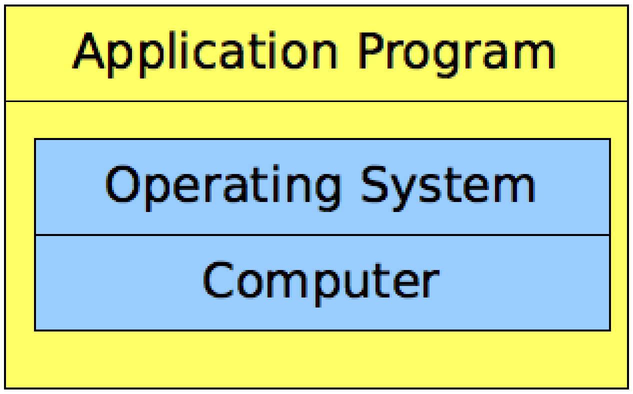

Figure 2.1: Layers of Abstraction

Looking More Deeply. When we start to write software, however, we need to be aware of multiple layers of meaning.

- The computer system running programs – in a broad and general sense – is like a TV set playing a TV show.
- The computer system running the Operating System program can be viewed as a single device. The computer hardware combined with the operating system software makes a new, abstract device. The composite (hardware-plus-software) device runs our application software.
- The computer system plus the Operating System plus the **Python** program can *also* be viewed as a single device. This complex, multi-layered device is what runs the application scripts that we write.

  This isn't very much like TV at all: we never tune to one channel (the operating system) that enables us to watch another channel (**Python**) that finally lets us watch the video we made with our own video camera.

Software builds up in stacks and layers, based on the principle of abstraction. TV simply switches channels. It looks like computers are so strange that metaphors will only cause more confusion. There aren't many things that are like computer systems where the behavior we see is built up from independent pieces. We can't really talk about them metaphorically, which makes computers a unique intellectual challenge.

Here's a picture that shows the abstract Computer-Plus-Operating system. This hardware plus software combination creates a "virtual" machine. The real machine is controlled by a binary executable. The virtual machine (hardware plus operating system plus Python) is controlled by our Python-language program.

Bottom Line. When we write Python-language statements, those statements will control the **Python** program; the **Python** program controls the OS kernel; the OS kernel controls our computer. These are the

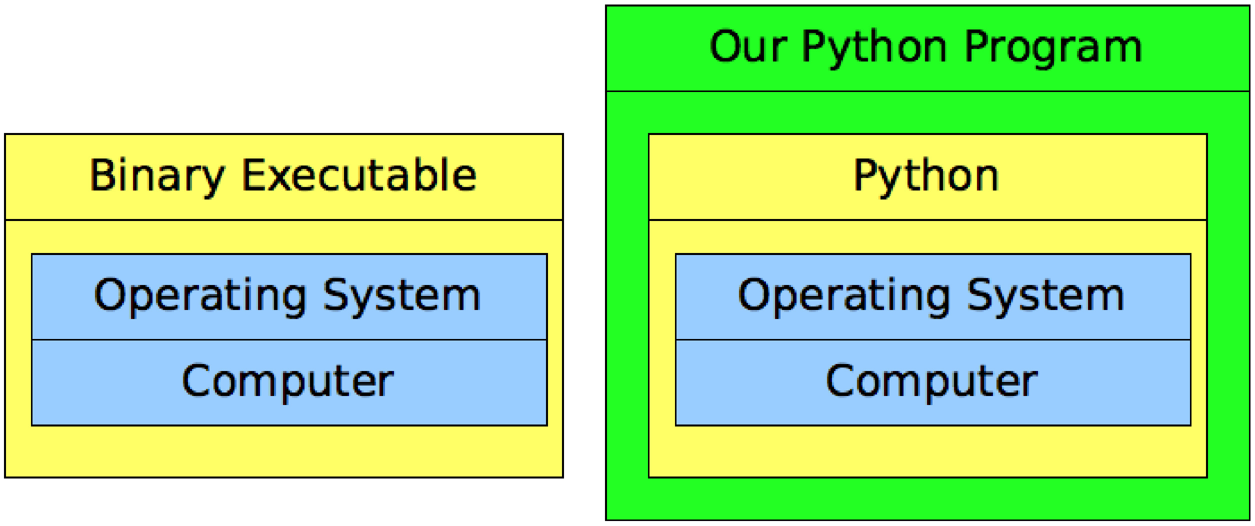

Figure 2.2: **The Python Virtual Machine**

most obvious and influential layers of abstraction. It turns out that there are other parts of this technology stack, but we can safely ignore them. The OS makes hardware differences largely invisible; and **Python** makes many OS differences invisible.

The cost of these layers of indirection is programs that are somewhat slower than those which use the computer's internal codes. The benefit is a huge simplification in how we write and use software: we're freed from having to understand the electro-techno-mumbo-jumbo of our processor chip and can describe our data and processing clearly and succinctly.

Note: Additional Factoids

The Python program was written in the C language and then compiled into a binary executable that is specific for your hardware chip and operating system. That's why the various distribution files for Python have names that include "i386" for Intel 80386-compatible chips and "fc9" for Fedora Core 9 GNU/Linux. While a bit beyond our scope, we'll talk about this a little in *So How Do They Create Binary Executables?*.

2.1.5 It's Turtles All The Way Down

The Iroquois creation myth involved the world carried on the back of a turtle. The turtle swam through the infinite void of space, surrounded by stars and planets and other spirits.

Think of our Python program as a world. Our program rests on the back of the Python executable, which is its turtle. The Python program seems to swim through space, directed by our script.

We now know that the Python turtle is actually a world of it's own. That world is resting on other pieces of software. This other software is the turtle that supports the Python turtle. In the case of CPython, this turtle is the standard C-language libraries. In the case of Jython, this turtle is the Java Virtual Machine. In the case of PyPy, it's the RPython kernel.

That software is really a world sitting on the back of another turtle. That turtle supporting Python is the operating system. The OS turtle is actually seveal turtles standing on the back of a kernel. And the kernel is standing squarely on the back of the hardware turtle. This is the **Abstraction** principle in action.

We often say this:

It's turtles all the way down.

2.1.6 Where Are The Programs?

Our programs (and our data) reside in two places. When we're using a program, it must be stored in memory. However, memory is volatile, so when we've turned our computer off, the program must also reside in persistent storage (e.g., a disk) somewhere. Since our disks are organized into file systems, we find programs residing in files.

When we look at the various files on our computer, we'll see a number of broad categories.

1. Applications or Programs. These are executable files, they will control the computer-plus-operating system abstract machine. There are two kinds of programs:
   (a) Binary Executable programs use the processor chip's binary codes. We use these, but won't be building them.
   (b) Script programs use a script language like Python. We'll build these.
2. Documents. Our OS associates each document with a program. This is a convenient short-cut for us, and allows us to double-click the document and have the proper program start running.

When we use the Finder's **Get Info** to look at the detailed information for an application icon in MacOS or GNU/Linux, we can see that our application program icons are marked "executable" and the file type will be "application" . In Windows, a binary executable program must have a file name that ends with `.exe` (or `.com`, but this is rare).

Starting A Program. Our various operating systems give us several user interface actions that will load a program into memory so that we can start to use it. Since starting a program is the primary purpose of an operating system, there are many ways to accomplish this.

- Double click an application icon
- Double click a document icon
- Single click something in the dock or task bar
- Click on a **Run...** menu item in the **Start...** menu
- Use the Windows **Command Prompt**; in GNU/Linux or the MacOS it is called a **Terminal**. Through the terminal window we interact with a shell program that allows us to type the name of another program to have that started.

All of these actions are just different ways to get the operating system to locate the binary executable, load it into memory and give it the resources to do its unique task.

2.1.7 Running a Python Program

The Python program's job is to read statements in the Python language and execute those statements.

From the operating system's point of view, all of our various Python programs and tools are really just the **python** program. Let's pretend you're running a program you wrote named `roulette.py`.

When you double click the Python program file you created (usually a file with a name that ends in `.py`), something like this happens under the hood. This is the conceptual sequence of events.

1. The OS looks up the binary executable associated with the `roulette.py` file. This is the Python program, `python` or `python.exe`.
2. The OS loads the binary executable **Python** program into memory, allocates resources and starts it running. It uses the kernel to share these resources politely with all other programs.

3. The OS provides the file name you double-clicked (`roulette.py`) to the Python program.
4. The Python program reads the `roulette.py` file and executes the Python language statements it finds.
5. When the statements are finished the Python program has nothing more to do, so it terminates.
6. The OS releases the resources allocated to the Python program.

It turns out that step four can have some sub-steps to it. The Python program doesn't always do a simple read of our file of statements. There's room for a small optimization in this step. Under some circumstances (see *Modules : The unit of software packaging and assembly*), Python will create a "compiled" version of our file to save a little bit of time.

What you'll observe are files with an extension of `.pyc`. These are compiled versions of a file. They're smaller, and encoded in a way that makes them very easy to read and work with.

2.1.8 Concepts Exercises

1. **Inventory Your System**.

   It helps to inventory the various devices and interfaces on your computer system. Start with the central processor (which may hang behind the display on some iMac's), and work your way around your desktop to identify each part.

   Use your word processor to write down all of the computer system parts. You'll need a folder for your Python programming projects. Create that folder; this inventory will be the first file in that folder.

2. **Get Info/Properties**.

   Find your web browser application. You may have a desktop shortcut, or a MacOS dock icon, a Windows start menu icon or a Windows toolbar icon for your browser.

   In Windows, you can ask for the properties of an application icon. If it is a short cut, you can use the **Find Target...** button to locate the real application file. With MacOS, you can use control-click to get information on a particular icon.

   In the MacOS, you can ask for the information about an application icon. In the MacOS Finder, you can click on an application icon and then use the **File Get Info...** to get information on an icon.

3. **Get Info/Properties**.

   Locate a file you made with your favorite word processor. The first exercise in this section was an opportunity to make a new document file.

   In Windows, you can ask for the properties of a document icon. If it is a short cut, you can use the **Find Target...** button to locate the real application file. With MacOS, you can use control-click to get information on a particular icon. The properties name the application that is associated with this document. In Windows, you can see the *Type of File* and *Opens With* information about the file.

   Using MacOS, you can ask for information about a document icon. In the MacOS Finder, you can click on an application icon and then use the **File Get Info...** to get information on an icon. The information has a *Kind* description. The *Open With* label shows the application that will open this document.

2.2 About Programming

Our job as a programmer is to create *programs*, which are statements in the Python language. When we run those programs, they will control our computer system to do the data processing we specified. This chapter takes a closer look at what a program really is and what is means to "run" a program. This will lead us to the program named `python` (or `python.exe`) and how we control it with statements in the Python language.

2.2.1 What is Programming?

While the coffee-shop description of programming is "how we create programs", that doesn't help very much. We can identify a number of skills that are part of the broadly-defined craft of programming. We'll stick to two that are foundational: designing Python statements and debugging problems with those statements.

We take much of our guidance on this from *Software Project Management* [Royce98]. Royce identifies four stages of software development, with distinct kinds of activities and skills.

- **Inception**. We have to start by defining the problem. The central skills you use here are observing and writing. You need to observe the problem and write a clear, simple description of what is wrong and how software can be used to fix it. If we clearly state our problem, then all of the rest of the programming activities are directed at a single goal.

  If we aren't clear on what we're trying to accomplish, we're very likely to go astray at this point. It's more important to clearly define the problem than it is to try and design software. We'll get to the software design in stages. We need the problem defined or we'll never get anywhere.

  We'll return to this in *Inception – Getting the Characters Right*.

- **Elaboration**. We elaborate our solution into solid description of how software will sove the problem. This is the first part of the design effort.

  We move from a description of how software will be used to identifying what we will need to buy, what we will build and what we will download from the open source community. We use design and architecture skills to create a solution to our problem. We need to be sure that our elaborated solution really will solve our problem. We also need to be sure that the cost is appropriate for the value we will create.

  This is rarely shows up as a single good idea for a Python program. Instead, this is often a series of experiments where we imagine something that would solve the problem, and then try to design a more complete solution. It takes a lot of practice to imagine something that can be written as a Python program.

  We'll return to this in *Elaboration – Overcoming Obstacles*.

- **Construction**. This is where we create our Python language statements and put them into module files and script files. Here we are building the solution that we designed during the elaboration stage. This is also involves design skills; but the design is at a much finer grain than the design work done during Elaboration.

  Construction goes beyond the Python language skills. It includes testing our programs to make sure they work, and debugging our programs to find out why they don't work.

- **Transition**. Our programs have to make the transitions from engineering effort to useful tool. That means they have to be installed on a computer where they can be used. Here is where the problem we started with in inception is actually solved by using the software.

  We know that we've done this phase well when we have a nice file that we can double-click, or run from the Terminal window that does the job we imagined and solves the original problem.

  We'll return to this in *Transition – Installing the Final Product*.

We're going to focus on two skills in this book: creating Python language statements, and debugging problems when we make mistakes. The language is central. However, techniques for debugging are almost as important as the language itself.

Other skills in include testing and problem analysis. Testing is a rich subject; it would double the size of this book to talk about appropriate testing techniques. The analytical skills for inception and elaboration don't require knowledge of Python, just common sense and clear thinking.

2.2.2 Goal-Directed Activities

Computer use is a goal-directed activity. When we're done using our software, we've finished some task and (hopefully) are happy and successful. We can call this a *state change*. Before we ran our program, we were in one mental state: we were curious about something or needed to get some data processing done. After we ran our program, we were in a different mental state of happy and successful. For example, we got the total value of our stock portfolio from a file of stock purchases.

This "state of being" view of how programs work is very deep. We can think of many things as state changes. When we clean our office, it goes from a state of messy to a state of clean (or, in my case, less messy). When we order breakfast at the coffee shop we go through a number of state changes: from hungry to waiting for our toast, to eating our toast, to full and ready to start the day.

Let's work backwards to see what had to happen to get us to a happy and successful state of being.

Success! The program has done the desired job, cleaned up, and we are back looking at the operating system interface (the Finder, Explorer or a Terminal prompt). We say that the program has reached a *terminating state* (also known as "all finished"). Therefore, one goal of programming is to create a program that finishes it's work normally so that the operating system can deallocate the resources and regain control of our computer.

In order to finish, what had to be true? Clearly, the program had to run and do what we wanted.

Running. The program is running, doing the job we designed it to do. Perhaps it is controlling a device or computing something. The program is undergoing a number of internal state changes as it moves from its initial or start-up state to its terminating state. Often it reaches terminating state because we clicked the **Quit** menu item. Another goal of programming, then, is to have the program behave correctly when it is running.

In order to run properly and do what we wanted, what had to be true? The program had to start running.

Starting. The operating system loads the program into memory from files on a disk. The operating system may load additional standard modules that the program requires. The operating system also creates a schedule for our program. Most operating systems interleave several activities, and our program is only one of many programs sharing the time and memory of the computer. Once everything is in place, we see it start running.

Another goal of programming is to have a program that cooperates with the operating system to start in a simple way and follow all the rules for allocating resources.

When we reverse this sequence, it is a path from start to finish. This goal-focused, state-transition principle is very important. It will pervade everything we do.

We'll revisit this in *Where Exactly Did We Expect To Be?* and show some techniques for reviewing the state changes of our programs.

2.2.3 The Tools Of Programming

Designing a program is mostly an intellectual exercise. It can be done with pencils and paper or sophisticated software modeling tools. For the most part, pencil-and-paper design is still the best and most common way to design software. Often, the fancy modeling tools are used to fill in details and create a document ready for publication.

As we noted above in *Goal-Directed Activities*, a program is focused on the goal of successful completion. Programming must, therefore, focus on the final outcome, also. The difficult part is to determine two things.

- What's the last step in creating the desired successful state.
- What's the precondition for that last step.

Once we've figured out what the last thing to do is, we now have a new problem that focuses on the next-to-last thing to do.

Generally, it helps to think backwards from desired outcome to necessary pre-conditions. At some point, we get to a necessary condition which is so trivial that we write the first statement and we're done with the programming effort.

2.2.4 Programming Exercises

1. **Operating System and Platform**.

   To successfully download and install software on your computer, you'll need to know your operating system information. It can also help to know your processor chip information.

   - **Windows**. You get to the control panels with **Start Settings Control Panel**. One of your control panels is the **System** control panel. When you double-click this, it shows the operating system and computer information.
   - **MacOS**. In your **Applications** folder, you have a **Utilities** folder. One of these utilities is the **System Profiler**. Double-click this icon to see a complete description of your Macintosh, including the operating system and processor chip information.
   - **GNU/Linux**. There are two relevant commands for examining your GNU/Linux architecture: **arch** and **uname**. The **arch** command prints a simple architecture string, like `i686` to tell you about the processor chip. The **uname -i** command shows a similar string for the "platform", which is the general family for your processor. In my case, it is a "i386"; one of the Intel processors.

2. **Other Applications**.

   What other applications are installed on your computer?

   - **Windows**. Your **Start** menu lists a number of programs. This isn't the complete list, since the Windows operating system has a number of additional binary programs tucked away in places where they don't appear on the start menu.

     You get to the control panels with **Start Settings Control Panel**. One of your control panels is the :application:‘ Add/Remove Programs‘ control panel. When you double-click this, it shows many of the application programs that you've installed on your computer.
   - **MacOS**. In your **Applications** folder, you have a **Utilities** folder. One of these utilities is the **System Profiler**. Double-click this icon to see a complete description of your Macintosh, including the list of application programs.
   - **GNU/Linux**. There are a number of standard places where GNU/Linux application programs are kept. You can use the **ls** command to look at directories like `/bin`, `/usr/bin`, `/usr/local/bin`. Notice the common theme to the directory names: bin is short for binary, as in binary executable.

2.2.5 So How Do They Create Binary Executables?

This is a digression for the curious. It may help you understand how the team that wrote the Python program did it. It can help you demystify programming. It may not help you learn the Python language, so feel free to skip it.

Binary executable files are created by a program called a *compiler*. A compiler translates statements from some starting language into the processor's native instruction codes. This leads to blazing speed. This approach is typified by the C language. One consequence of this is that we must recompile our C language programs for each different chip set and operating system.

The C language isn't terribly easy to read. The language was designed to be relatively easy for the compiler to read and translate. It reflects an older generation of smaller, slower computers.

The GNU Tools. For the most part, the GNU C Compiler and C language libraries are used to write binary executables like **Python**. The C language has been around for decades, and has evolved a widely-used style that makes it appropriate for a variety of operating systems and processors. The GNU C compiler has been designed so that it can be tailored for all processors currently used to build computers. Many companies make processors, include Intel, National Semiconductor, IBM, Sun Microsystems, Hewlett-Packard, and AMD. The GNU C Compiler can produce appropriate binary codes for all of these various processor chips.

In addition to the processor (or "chip architecture"), binary executables must also be specific to an operating system. Different operating systems provide different kernel services and use different formats for their binary executable files. Again, the GNU C Compiler can be made to work with a wide variety of operating systems, producing binary executable files with all the unique features for that operating system.

The ubiquity of the GNU C compiler leads to the ubiquity of Python. By depending on the GNU C compiler, the authors of Python assured that the **python** program can be compiled for any processor chip and any operating system.

2.2.6 How Are Compilers Written?

Think about this conundrum for a moment. A compiler is a binary executable that creates binary executables. How do you create that first binary executable compiler? You don't have the compiler yet, you're in the process of writing the compiler.

Think about using a lathe to build yourself a lathe. Or, think about laying brick to make a furnace in which you can make bricks. Or think about creating a Makerbot that creates more Makerbots.

Wait, it gets worse. The operating system is really just a complex binary executable. How do you write an operating system before you have an operating system that runs your editor and your compiler?

In the early days of computers, this was a difficult and complex problem. Before the operating system exists, the computer is just an inert collection of electronic good ideas. The earliest computer programmers had to fool with the hardware just to load the binary instructions into memory. For example, many early computers had switches on a front panel that allowed the programmer to manually set the contents of memory. This meant tedious setup time to run a program.

Nowadays, the people who write compilers and operating systems have a variety of sophisticated programs sometimes called "cross-compilers". They use one computer to create the binary executables for a different kind of computer. With some care, the inventors of a new computer system can cross compile and build up a complete operating system in a series of steps. The eventual goal is to be able to use the operating system to rebuild itself.

2.2.7 Concepts FAQ's

What is a programming language? This is actually a complex question that exposes the very heart of computing. The essence of a computer is the processor chip. This chip is a very complex electronic circuit built up from a number of simpler circuit elements that we'll call "flip-flops" . A flip-flop is either on or off and can be flipped on or off electrically. There are a number of kinds of flip-flops with different electronic connections to flip (or flop) and detecting if the circuit is presently flipped or flopped. In addition to flip-flops are logic gates to do things like determine if two flip-flops are on at the same time ("and") or if one of two flip-flops is on ("or"), or if a flip-flop is off ("not").

The designers of computers will often group the flip-flops into bunches and call them registers. These register specific values or conditions within the processor. For example, one register may contain the memory address of the next instruction to fetch. Another register might have a numeric value on which we are calculating. Another register might be a clock that counts up automatically from zero when the processor is turned on.

A computer's memory, it turns out, is just a collection of billions of flip-flops.

The processor chip does two things: it fetches instructions from memory, and executes those instructions. The fetching part is a relatively simple process of reading data from the memory chips and changing registers to reflect that instruction. The execution part is more complex, and involves changing the state of other flip-flops based on the instruction itself, data in memory and the state of the various processor registers.

The instructions in memory form a kind of "language" for controlling the processor. At this level, the language is very primitive. Since it is narrowly focused on the ways the processor works, it is almost incomprehensible. The language can only express a few simple imperative commands in a very precise – essentially numeric – form.

The idea that computers are controlled with a kind of language is an example of an abstraction that has immense and far-reaching consequences.

- It lets us translate from more expressive languages into the machine's native language. We call this kind of translator a compiler.
- It lets us design more expressive languages that better describe the problems we are trying to solve.
- It changes our view of computing. We are no longer controlling an electronic chip thingy; we are capturing knowledge about data and processing.

Why can't programming be done in English? There are a number of reasons why we don't try to do programming in English.

- English is vague. More precisely, English has many subtle shades of meaning. Try to explain the difference between "huge" and "immense" . Further, English has words borrowed from a number of languages, making it more difficult to assign precise meanings to words.
- English is wordy. Data processing can be very simple; however, English is a general-purpose language. Because we're only talking about data processing, it helps to have a number of simplifying assumptions and definitions.

Over the years there have been a number of attempts at "natural language" processing, with varying degrees of success. It takes quite a bit of computing horsepower to parse and understand general English-language writing. All of this horsepower would then make the Python program large and slow; a net loss in value.

In order to keep to short, focused statements, we would do well to use only a limited number of words. We would also find it handy to allow only a few of the available English sentence forms. We should also limit ourselves to just one verb tense. By the time we've focused ourselves to a small subset of English, we've created an artificial language with only a small resemblance to English. We might as well do another round of simplification and wind up with a language that looks like Python.

What if I'm no good with languages? First, we aren't learning a complete natural language like Swedish. We're learning a small, artificial language with only about twenty kinds of statements. Second, we aren't trying to do complex interpersonal exchanges like asking someone which bus will get us to Slottberg in Gamla Stan. Interpersonal interactions are a real struggle because we don't have all day to look up the right words in our phrase book. Python is all done as written exchanges: we have hours to look things up in our various reference books, think about the response from the Python program, and do further research on the Internet.

Also, the Python language lacks subtle shades of meaning. It is a mathematical exercise; the meanings are cut and dried. The meanings may be novel, but the real power of software is that it captures knowledge in a rigorous formal structure.

Why is the terminology so confusing? One of the biggest sources of confusion is the overuse of the word "system" . Almost everything related to computers seems to be a system. We have computer systems, software systems, operating systems, systems programmers, system architects and network systems. Most of this is just casual misuse of the words. We'll limit "system" to describing the computer hardware system.

Another big source of confusion is overuse of "architecture" and the wandering meaning of "platform" . We'll try to avoid these words because they aren't really going to help us too much in learning Python. However, we have software architectures and hardware architectures. The hardware architecture and the platform are both, in essence, the processor chip and supporting electronics.

Generally, however, the biggest issue is that computers and computing involve a number of very new concepts. These new concepts are often described by using existing words in a new sense. For example, when we talk about computer systems being "clients" or "servers" , we aren't talking about a lawyer's customers or a restaurant's wait staff.

2.3 Let There Be Python: Downloading and Installing

Before we can use Python, it must be installed on our computer. This chapter will cover a number of installation scenarios.

You'll need to have access to a reasonably modern computer. This can be either a Macintosh with MacOS X, a Windows machine with Windows XP or higher, or any of the wide variety of GNU/Linux or UNIX machines. The computer doesn't need to be spectacular or huge, just a machine that works reliably.

You'll also need a few basic computer skills; if you're new to computing, you might need a couple of "For Dummy's" books to fill in your background. Since you're going to download and install software, you'll need access to the Internet, plus authority to install software on your computer. In an office or academic environment, you might not have permission to install new software; in this case, you'll need to work through the organization that provides your computer to do the installation for you.

This chapter has a number of sections, but you'll only really need to read a little bit of this chapter, depending on your operating system.

- **Windows**. You need to work through *Windows Installation*, where we describe downloading Python 2.6 (or newer) and installing it.
- **Mac OS**. Python is included. You may want to upgrade your Python to the latest and greatest. We'll look at a Mac OS upgrade in *Macintosh Installation*.
- **GNU/Linux**. Generally, Python is included. Often, the upgrades are automatically done. We'll look at the common variations on the GNU/Linux installation in *GNU/Linux and UNIX Overview*. If you have YUM, for example, see *YUM Installation*.

Once we have Python installed, we can move on to interact with the Python program in the next chapter.

2.3.1 Windows Installation

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.

Download. After making a backup, go to the http://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 2.6. In that case, the kit is `python-2.6.6.msi`. When you click on the filename, your browser should start downloading the file. Save it in your `downloads` folder.

A newer Python 2 (e.g. 2.7) is also a candidate for a download. Python 3, however, has enough differences that it will be confusing; do not download Python 3.

Prepare. 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

Double-click the Python installer (`python-2.6.6.msi`).

You should get a "Security Warning" asking if you want to run this file. The answer is to click **Run**.

First, you'll be asked if you want to install for all users or just yourself. You require administrator privileges to install for all users. If you're using a corporate PC, for example, you might not have administrator privileges. If you have the privileges, then install for all users. Otherwise, install for yourself. Click **Next** to continue.

The next step is to select a destination directory. The default destination should be `C:\Python26`. Note that Python does not expect to live in the `C:\My Programs` folder. There's a subtle problem with the `My Programs` folder: it has a space in the middle of the name, something that is atypical for operating systems other than Windows. This space is sometimes unexpected by Python programs, and can cause no end of obscure problems. Consequently, Python folks prefer to put Python into `C:\Python26` on Windows machines. Click **Next** to continue.

The next step is to customize list of components to install. You have a list of five components. You have no reason to change these.

- Register Extensions. You want this.
- Tcl/Tk (Tkinter, IDLE, pydoc). You want this, so that you can use IDLE to build programs.
- Documentation (Python HTML Help file). This is some reference material that you'll probably want to have.
- Utility scripts (Tools/). We won't be making any use of this; it's simplest 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.

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.

Tip: 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 2.6** 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 *IDLE Time : Using Tools To Be More Productive*.

Important: 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 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.1500 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.

2.3.2 Macintosh Installation

Python is part of the MacOS environment.

This from-the-factory installation includes a copy of IDLE, but it isn't always obvious where it is located on your Macintosh. You can skip down to *Adding IDLE Without An Install on Mac OS X* for information on making use of IDLE without doing an install.

It's easier to upgrade your copy of Python to 2.6.6. This will make IDLE available as a first-class icon in your Applications folder.

In order to install 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

Backup. Before installing software, back up your computer.

Download. After making a backup, go to the http://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 2.6. In that case, the kit is `python-2.6.6-macosx.dmg`. When you click on the filename, your browser should start downloading the file. Save it in your `downloads` folder.

A newer Python 2 (e.g. 2.7) is also a candidate for a download. Python 3, however, has enough differences that it will be confusing; do not download Python 3.

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

- A backup
- The Python installer

Macintosh Installation

When you double-click the `python-2.6.6-macosx.dmg` file, it will create a disk image named `Python 2.6`. This disk image has your license, a ReadMe file, a Build file and the `MacPython.mpkg`.

When you double-click the `Python.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 `Python 2.6` folder, which contains a number of applications.

- BuildApplet

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

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

Important: 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 2.6.6 (r266:84374, Aug 31 2010, 11:00:51)
[GCC 4.0.1 (Apple Inc. build 5493)] 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 2.6.6
>>>
```

At the top of the window, you'll see a menu named **IDLE** with the menu item **Quit IDLE**. 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 that you also downloaded and installed.

Adding IDLE Without An Install on Mac OS X

If you did an install of Python 2.6, you should have **IDLE** available in your `Python 2.6` folder in the `Applications` folder. Nothing else needs to be done. Go directly to the next chapter.

If you did not install an upgrade, you will want to add IDLE to your environment. There are two relatively simple approaches.

- One choice is to move the icon that starts IDLE into your Applications folder.
- A second choice is put the Python binaries on your `PATH`. This allows you to easily run **IDLE** from the Terminal tool.

The following directory has the IDLE program:

```
/System/Library/Frameworks/Python.framework/Versions/Current/bin/idle2.6
```

You can do any one of the following alternatives to make IDLE available without a complete installation. Don't do all of them.

1. **Move the idle icons**.

   This is probably the simplest aproach.

First, create a `Python 2.6` folder in your Applications folder.

To move the existing `idle` and `idle2.6` icons, you'll have to start from your `Macintosh HD`, you can locate the bin directory which contains the files named `idle` and `idle2.6`.

Second, drag these two folders into your new `Python 2.6` folder.

Now the **IDLE** icon is easy to find. You're ready to move to the next chapter.

2. **Add idle to the PATH.**

   This happens automatically as part of installing Python 2.6. You can do this step manually instead of doing a complete installation. However, there is a huge technical hurdle: it's difficult to edit the hidden files in your home directory. Essentially, the job is to edit your `~/.bash_profile` to add the following lines. Because the name begins with a `.`, it's considered "hidden", and most Mac OS tools won't touch it.

   ```
   PATH="/System/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"
   PATH="${PATH}:/usr/local/bin"
   export PATH
   ```

   It's beyond the scope of this book to address the various tools that can edit files like your `~/.bash_profile`.

   Now you can type '`idle &`' at the Terminal prompt and run **IDLE**. You're ready to move to the next chapter.

2.3.3 GNU/Linux and UNIX Overview

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.

Note that we typed `Ctrl-D` to finish using Python.

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.

2.3.4 YUM Installation

Some Linux distributions use tools like **Yum**. For example, if you are a Fedora user, you will have **Yum**. Other Linux distributions have similar tools.

If you have an out-of-date Python, you can enter the following commands in the Terminal window to do an upgrade.

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

The first command will upgrade Python to the latest and greatest version.

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.

2.4 Two Minimally-Geeky Problems : Examples of Things Best Done by Customized Software

There are a couple of problems that we'll use throughout this book to show how (and why) you use Python. Both problems are related to casino games. We don't embrace gambling; indeed, as you work through these sample problems, you'll see precisely how the casino games are designed to take your money.

We like using casino games because they are (a) moderately complex and (b) not very geeky. Really complex problems require whole books just to discuss the problem and its solution. Simple problems can be solved with a spreadsheet. In the middle are moderately complex problems that require Python.

There are numerous geeky problems. Most computer-science textbooks are packed with geeky problems that are relevant to professional programmers, but hard to explain to newbies. Rather than dig into geeky problems like stacks, queues, state machines, or parsers, we'll stick with games.

While it's pretty safe to assume that you know a little about casino gambling, we'll provide a few definitions in *About Gambling* just to be sure. From there, we'll define the Roulette problem in *The Roulette Problem*. We'll look at the Craps problem in *The Craps Problem*. We'll stake out our overall strategy in *Directions*. We'll answer some questions in *Problem FAQ's*.

2.4.1 About Gambling

The casino table games of Craps and Roulette (and a number of similar games) allow the bettor to place a bet (or wager) on an outcome or set of outcomes. Some random device (cards, dice, a wheel, a spinner) is used to make a selection. This selection usually resolves the bets as winners, losers, or a "push" where your money is returned.

In Roulette, each random event defines a complete set of outcomes, and all bets are resolved. You see this in the play at the Roulette table: people place bets, the wheel is spun, all of the bets are resolved. Once the bets are resolved as winners or losers, players are permitted to bet again.

In Craps, each random event does not define a complete set of outcomes. Some bets are not resolved when the dice are thrown; instead, the bets remain. Craps is played as a series dice throws that are part of a round or turn. The turn can be as short as a single throw of the dice, or it can be indefinitely long. It is unlikely (about a 1% chance) for a turn to take more than seven throws of the dice, but not impossible.

Generally, the person throwing the dice, the "shooter", holds the dice as long as they win their round. When they lose, the dice move to a new shooter. These nuances of casino play has no impact on the actual game, so we'll ignore details like these.

Odds. If the outcome you bet on is likely, your payout is rather small. If the outcome you bet on is rare, your payout may be huge. They call this the odds of winning. When the odds are small, the event is pretty likely. For example, almost half the Roulette wheel has numbers colored red. Betting on red, then, is pretty safe. Since it's about half the numbers, the payout is 1:1. If you bet \$10, you could win an additional \$10.

Contrast red (or black) with the number zero, which is just one of the thirty eight bins on the wheel. Since zero is so rare, it pays off at 35:1. If you bet \$10 on zero, and it comes up, you could win \$350. They call these long odds or a long shot.

2.4.2 The Roulette Problem

Here's the short form of the question: "How well does the Martingale betting system work for Roulette?". After we define any unknown jargon in this question, we'll see that it is not terribly complex and it will lead us to some related questions. All of these questions can be answered with simple Python programs.

Roulette. In Roulette players make bets and wheel is spun to determine which bets win and which bets lose. The Roulette table has a number of positions on which you can place bets by stacking up chips. The Roulette wheel is a collection of numbered bins. When the wheel is spun, a small ball is dropped into it, and the ball will eventually come to rest in one of the bins. The bin selected by the ball determines which of the betting positions are winners and which are losers. Each position has a payout ratio that determines how much you win based on how much you bet.

There are over a hundred possible bets on the Roulette table, and a wide variety of payout ratios. We'll define a few of them, and focus on just six of the available bets.

- **The 38 individual numbers**. The numbers go from one to thirty-six, colored red and black. Additionally, there are zero and double-zero, colored green. The numbers all pay off at 35:1.
- **Groups of numbers**. You can place bets between pairs of numbers, groups of three, four or six numbers. You can also place a bet on zero, double zero, one, two and three as a combination of five numbers. If any of the numbers wins, your bet is a winner. The more numbers in your combination, the lower the payoff odds.
- **The Columns**. The numbers form three columns of 12 values. If any of the numbers in the columns wins, the column as a whole pays off at 2:1. Zero and double zero are not part of any column, if they are spun, all column bets lose.
- **The Ranges**. Like the columns, the table is also blocked off into three ranges: one to twelve, thirteen to twenty four and twenty five to thirty six. If any number in the range wins, the range pays off at 2:1. Zero and double zero are not part of any range, if they are spun, all range bets lose.
- **Red, Black, Even, Odd, High and Low**. All of the numbers except zero and double zero are colored red or black, are even or odd, or are low (between one and eighteen) or high (between nineteen and thirty six). These bets all include a large range of values and pay off at 1:1. We'll focus on these bets because they are so simple and so commonly used.

Martingale Betting. The Martingale betting system suggests that you organize your casino play as follows:

1. Establish a budget with a minimum bet. Since tables vary in the size of bets required, we'll just call this amount *b*, the basic betting unit. At a \$10 table, it would often be \$10.
2. Bet the minimum amount, *b*, on one of the 1:1 bets (red, block, even, odd, high or low).
3. If the bet wins, you're way ahead. Reset your bet back to the minimum amount, *b*. If the bet loses, you double your bet. In the even of several losses, you'll be betting $2 \times b$, $4 \times b$ or even $8 \times b$.

Now, let's look at our question again. How well does this Martingale system work? We can see that the green zero and double zero complicate the analysis. There are ways to work out the details, but rather than learn a lot of math, we'll learn a little Python and simulate the whole thing. We can collect some statistics showing the results of our simulated Roulette game.

We can ask a whole family of related questions by replacing the Martingale betting system with more complex systems. We can ask questions based on extending the Martingale system to include additional bets. This is the beauty of writing our own simulation: we can modify our program to try out different variations on our betting procedure.

2.4.3 The Craps Problem

Here's the short form of the question: "How well does the Field bet pay in Craps?". We'll define the gambling jargon and then look at this question again, in a little more detail.

In Craps, players make bets and a pair of dice are thrown to determine the state of the game. Some dice throws are significant events and will resolve some or all of the bests as winners or losers. Some dice throws are less significant and resolve some bets. Some dice throws don't change the state of the game at all.

The Craps table has a number of positions on which you can place bets by stacking up chips, as well as a token that shows the state of the game. A shooter will throw two dice; the number on the dice will do several things. First, the number will pay off any *proposition bets* based on just this throw of the dice. Second, the number will pay off any of the various *number bets* that can be placed. Third, the number may change the state of the game, which can also resolve certain kinds of bets.

The Craps game has two states: point "off" and point "on". The casino will place a large black and white disk on the table to show the state of the game.

Point Off or the Come Out Roll. The first time the shooter throws the dice, the point is off. If the shooter throws 7 or 11, this turn is an immediate winner, and bets are resolved. If the shooter throws 2, 3 or 12, this turn is an immediate loser and bets are resolved. In this case, the point is still off, the game didn't change state, it's still just beginning.

When the point is off, and the shooter throws 4, 5, 6, 8, 9 or 10, the game changes state, and now the point is on. The casino will flip over the large disk to show the "on" side, and put it on the betting space that shows the point number.

Point On. When the point is on, a number of additional bets are allowed. Additionally, the disk sits in a number's space to prevent certain other bets. We'll avoid the complexity of these conditionally permitted bets. In a casino, however, you would see a flurry of activity when a point is established.

When the point is on, and the shooter throws this point number, the round is a winner, and most of the bets are resolved. There are some bets that will persist, however. When the shooters throws a seven, the round is a loser, and all bets are resolved.

Throwing a seven means the shooter lost, and most bets are losers. There are, however, some "don't" bets that will be winners when the shooter doesn't win. These are sometimes called "wrong bets", and involve a more sophisticated odds calculation. In general, you can put up a lot to win a little when you make wrong bets.

When the shooter throws 2, 3, 11 or 12, nothing much can happen. Certain one-roll proposition bets are resolved, but these four numbers are neither points nor are they 7, which ends the game.

Other Bets. There are a number of bets which don't depend on the state of the game. These are one roll "proposition" bets. The field is one of these bets. You place your bet in the box marked "Field" before the dice are thrown. The number on the dice determines the field bet result immediately.

The field bet wins on any of 2,3,4,9,10,11, or 12. The 3,4,9,10, and 11 pay 1:1 ("even money") and the 2 and 12 pay off at 2:1.

Analysis. There are a number of questions about the field bet. We can create a simple simulator to see the basic outcome. We can use a more sophisticated simulation of doing Martingale betting (see *The Roulette Problem*) to see how that changes the performance of this bet. Some people use an even more complex betting system for the field by increasing their bet with each win and decreasing it with each loss. We'll stick with a simple simulation as a way to learn Python

2.4.4 Directions

We aren't going to describe the solutions to any of these casino game problems here – that would rob you of the intellectual fun of working out your own solutions to these problems. Instead, we want to provide some hints and nudges that will parallel the course this book will take.

This may already be obvious, but we're going to address these problems by writing new software in the Python language. The reason why it is important to restate the (potentially) obvious is that in *Using Python* we're going to spend time on learning to control the **python** program in a simple, manual way. Then, when we write programs, we'll control **python** with our programs to do more sophisticated work.

Any solution to these kinds of problems will involve some simple math. Almost all computing involves some kind of math. Business programming tends to involve the simplest math. Engineering and science can involve some really complex math. Statistics is often in the middle ground, which is why we will look at it closely in *Arithmetic and Expressions*.

By the way, in addition to math-oriented computing, there is also computing that could be termed "symbolic" in nature. It might involves words or XML documents or things that aren't obviously mathematical; we'll set this aside as atypical for newbies.

Sequential Thinking. A program in Python is often a sequence of operations. In the casino game definitions, we saw that each game was a sequence of individual steps. We can often summarize programs by looking at their inputs, their processing steps and their outputs. This input-process-output model reflects the sequential order of processing: first, read the inputs; second, do the processing; third, print the outputs. More sophisticated programs (like games or web servers) will interleave these operations. We'll look at this in *Programming Essentials*.

The sequence of operations is rarely fixed and immutable. With casino games, we have some bets which are winners and some bets which are losers. We have conditional operations of collecting losing bets and paying winning bets. Additionally, we'll have some operations which have to be repeated for a number of simulations, or until some condition is satisfied. We'll look at this in *Some Self-Control*.

Our exploration of Python starts with arithmetic expressions and moves on to statements, then to sequences of statements. We'll add conditional and iterative statements. The next step will be a simple organizing principle called a function definition. We'll introduce this in *Organizing Programs with Function Definitions* and use it to package parts of our program until a useful, discrete components that can help us control the overall complexity of our program.

Other Side Of the Coin. Beginning with *Getting Our Bearings* we'll turn to a different tack. The first parts of our exploration were focused on the processing, and the procedural nature of our problems. The second part of our exploration will look at the data and collections of data.

If we are going to simulate a number of sessions at the Roulette wheel, following our Martingale strategy, we'll need to collect the results and do statistical analysis on the collection. We'll look at collections of data items in *Basic Sequential Collections of Data*.

We'll address some programming techniques in *Additional Processing Control Patterns* that make our Python programs more reliable and also a bit simpler. Simplification is a touchy subject: simplifications aren't always appreciated until you see the more complex alternative. Further, since we're approaching Python by moving from the elementary to the advanced, some things we'll look at will be complex but elementary. As we learn more, we can replace them with something simple but advanced.

In *More Data Collections* we'll look at some additional data structures that can help us develop truly useful solutions to our problems. These additional data structures will give us foundational knowledge of the Python language and the built-in data types that we can use.

Successful Collaboration. When we look at our problems, we see that there is considerable interaction among a number of objects. For example, in Roulette, we have the following kinds of things:

- the wheel, which returns a random bin,
- the table, which holds bets,
- the player, which uses the Martingale strategy to place bets

This interaction between player, table and wheel forms a larger thing, called the game, which lasts until the player wins big, loses big, or has spent too much time at the table. Each game produces a final result of zero dollars, big bucks or some number of dollars that was available when time ran out. These, in turn are collected for statistical analysis. An even bigger assembly of objects does the simulation and analysis. We'll learn how to define these collaborating objects in *Data + Processing = Objects*.

A lot of the basic components that make a program robust and reliable are already packaged as Python modules, and we'll cover these in *Modules : The unit of software packaging and assembly*. We'll also use the built-in modules as templates for designing our own modules; this allows us to organize our program neatly into discrete, easy-to-manage pieces.

Our final section, *Fit and Finish: Complete Programs*, will cover some final issues. These are the things that separate a fragile mess that almost works most of the time from a useful program that can be trusted.

2.4.5 Problem FAQ's

Why Casino Gambling? We think we've got two compelling reasons for using casino gambling for programming problems in this book.

- Casino games have an almost ideal level of complexity. If they were too simple, the *house edge* would be too obvious and people would not play them. If they were too complex, people would not enjoy them as simple recreation. Years (centuries?) of experience in the gaming industry has fine-tuned the table games to fit nicely with the limits of our human intellect.
- The results are sophisticated but easy to interpret. *Probability theory* has been applied by others to develop precise expectations for each game. These simulations should produce results consistent with the known probabilities. This book will skim over the probability theory in order to focus on the programming. For a few exercises, the theoretical results will be provided to serve as checks on the correctness of the student's work.

This book does not endorse casino gaming. Indeed, one of the messages of this book is that all casino games are biased against the player. Even the most casual study of the results of the exercises will allow the student to see the magnitude of the :firstterm:' *house edge* ' in each of the games presented.

Why not Something Simpler? While many problems are simpler than casino gambling, they don't require customized software written with a powerful language like Python to solve them. It's hard to locate things that are both simpler than casino gambling and still interesting enough to provide more than one trivial exercise.

Why not *Your Subject Here* ? As an author, I'm not as knowledgeable in *Your Subject Here* as you are, and can't do it justice. Also, I had to pick something, and I chose something that I knew a little bit about.

More importantly, however, the point of this book is to equip you to go out and tackle *Your Subject Here* using your new-found programming skills.

2.5 Why Python is So Cool

We'll ramble on a bit about Python and the reasons why it is so cool. This won't really help you learn the language. It's mostly op-ed material to provide some justification for why someone would invest time in learning Python. In *Core Coolness* we'll cover some fundamental reasons why Python is cool. The FAQ in *Coolness FAQ's* touches on a few more questions that sometimes get asked.

2.5.1 Core Coolness

Python reflects a number of growing trends in how people develop new computer programs. It is a very simple language, supported by an *interpreter* and surrounded by a vast library of add-on modules. It is an *open source* project, supported by dozens of individuals; this encourages you to build complete solutions from smaller components and partial solutions. We'll look at each of these facets separately.

Planet Python. Python is really four separate elements in a single, tidy package. I like to think of it as an wonderfully efficient planet that we can visit. To get things done on that planet you have to learn the language. Once you've learned the language, however, you find that the whole planet is organized to do everything you ask precisely and very quickly. Like any well-run organization, it has a number of services that make life convenient and safe, and assure the common good of all the inhabitants. Finally, it offers a kind of public forum for making your requests and seeing the results of those requests.

This mythical Planet Python is the **Python** program itself, we'll call it **python** in this book. Windows users may see it as `python.exe`. The Python program, **python**, runs on your computer, and carries out statements written in the Python language. The program has just one purpose – execute Python language statements – so it is small and efficient. Because it is so tightly focused, it is wonderfully reliable.

The planet's services are the Python *libraries*. These libraries include programs you can extend, and pieces of programs that you might use to create a more complete program. Some parts of the libraries are both: things you extend to add new features, and then use in your final program. I think of these as essential services like police departments, public libraries, laundromats, and telephone sanitizers. You build your complete organization or enterprise using these pre-built organization units.

The public forum is the *integrated development environment* (IDE). This is the environment where you develop your Python program. It's integrated because all the tools you might want are right there in a single program. In this case, the program's name is **IDLE**. You use **IDLE** to write Python statements, execute sequences of Python statements and read any resulting messages.

Simplicity. Python is a relatively simple programming language that allows you to express data processing in clear, precise terms. The Python language has an easy-to-use syntax, focused on the programmer who must type, read and understand a program. The language is designed to look a bit like a natural language, with simple punctuation and indentation. Computer languages are more rigid than human languages: when you misspell something in English, people can often determine what you meant, and make sense of what you wrote. The Python program, **python** , is only a simple piece of software: spelling and punctuation really matter. While Python is easier than most other programming languages, you must still be precise.

On Simplicity

The simplicity of Python is so important that we're going to emphasize it heavily. In other languages, desirable features were often added as new statements in the language. The language then evolved into a complicated mixture of optional extensions and operating-system features muddled up with the original core statements of the language. A poorly designed language rarely works the same on different computers or operating systems, or it requires many compromises to achieve portability. This kind of badly designed language is always hard to learn.

One hint that a language has too many features is that a language subset is available. The most outstanding example of this is COBOL. There are a number of subsets with different kinds of compatibilities with different tools and operating systems. While originally easy-to-read, COBOL has evolved into a monstrously complex problem for many businesses.

The Python language has only twenty statements, the language is easy to learn, and there is no need to create a simplified language subset.

Interpreted. The computer science folks characterize the Python program, **python** , as an *interpreter* : it interprets and executes the Python language statements, doing your data processing. Because it is interpreting our statements, it can provide useful diagnostic information when something goes wrong. It can also handle all of the tedious housekeeping that is part of how programs make use of the computer's resources. As users, we don't see this housekeeping going on, and as newbie programmers we shouldn't have to cope with it, either.

The computer-science types make a distinction between interpreters (like Python) and *compilers* (used for the C language). The C compiler (controlled by a program named **cc**) translates C language statements into a program that uses the hardware-specific internal codes used by your computer. The operating system can then directly execute that resulting program. After you see the results of execution you might make changes, recompile and re-execute. This compilation step makes everything you do somewhat indirect. The compiler translates your C statements into another language which is then executed. This indirection makes compiled languages harder to learn; it also makes diagnosing a problem very hard.

Here's a diagram that may help clarify how Python differs from a language like C. For a C programmer, they will use a complex IDE which includes the C Compiler to translate their C statements into a binary executable program from their statements. For a Python programmer, a simpler IDE uses the **python** program to execute the Python statements.

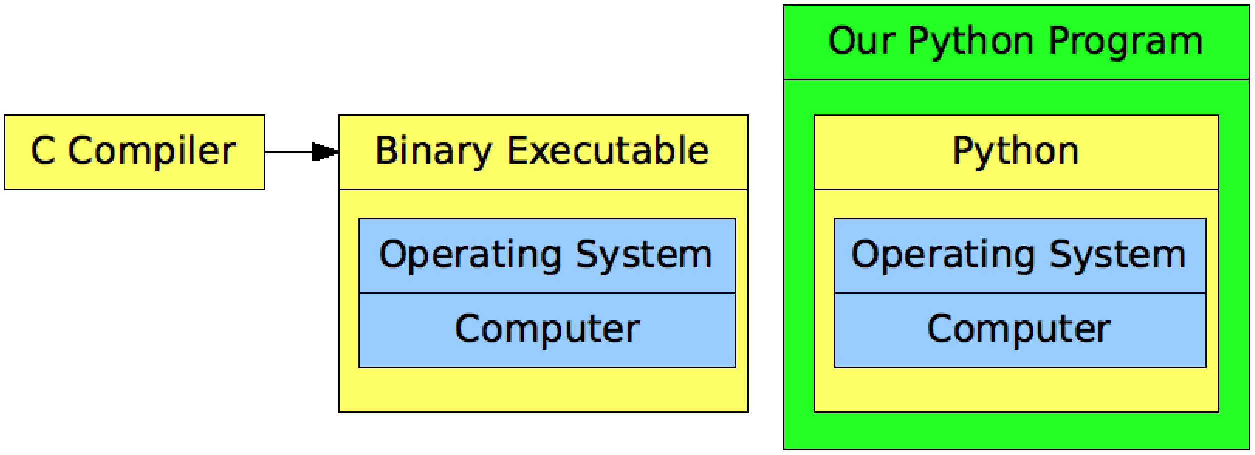

Figure 2.3: **C Compiler vs. Python**

The binary executables have relatively direct control over the operating system and computer. A Python-language program controls **Python**.

Technical Digression

The Python interpreter, which runs Python-language programs, is implemented in the C programming language and relies on the extensive, well understood, portable C libraries. Using the C-language under the hood means that it fits seamlessly with Unix, GNU/Linux and POSIX environments. Since these standard C libraries are widely available for the various MS-Windows variants, Python runs similarly in just about all computers and operating systems. Because of the abstraction created by the C libraries, you'll find it impossible to find meaningful differences between Windows-2000, Windows-XP, Red Hat GNU/Linux and MacOS.

Why does anyone use a compiled language like C? C is more complex than Python and writing C requires the programmer to keep careful track of a number of housekeeping details. The program that results from the C compiler is hardware-specific and consequently very fast. This is the key to why Python helps us out so much. The **Python** program, having been written in C, and compiled to be specific to our computer's hardware, is very efficient. However, since we can express our data processing needs in the (easy to learn) Python language we can use all this speed without having to learn C or how to compile C-language statements into a program.

When we need blazing speed, we have to write in C. When we need simplicity, we find it easier to write in Python. We can have the best of both worlds. Most programs only need amazing performance in small sections of the program. We can, with some care, write just those small sections in C, and then make that component available to Python. This gives us the speed of C where we need it and the simplicity of Python everywhere else.

It turns out that Python often does a secret compilation pass on your Python statements in order to speed things up a hair. It doesn't change the fundamental benefit that accrues because Python is a kind of interpreter. It only blurs the distinction between compiled and interpreted languages.

Libraries. Python, the project, includes a rich set of supporting libraries. These libraries contain the basic gears, sprockets, flywheels and drive-shafts that you can use to make a program. By separating the library tool-boxes from the core language, the designers of Python could keep the language simple, which means the interpreter can be very efficient and reliable. Yet, they can provide an extensive feature set as separate extensions. Every new idea can be added as another extension.

There are other consequences to having extensive and separate libraries. Principally, good ideas can be preserved and extended, and bad ideas can be ignored. This basic evolution saves programmers from having to design everything perfectly the first time. As you get more experience with the Python programming community, you will see ideas come and go. Some extensions will blossom and become widely used, where others will be quietly ignored because something better has come along.

Another consequence of having separate libraries is that any programming project should begin with a survey of available libraries. This can replace unproductive programming with more productive research and reuse.

Development Environment. Finally, we see that Python also comes with a development environment, or workbench, that you can use to write and execute your Python statements. The *integrated development environment* (IDE) includes an editor for writing Python files, and the Python interpreter, plus some other tools for searching the Python libraries.

Interestingly, the Python development environment is just another Python program. When you double-click on the **IDLE** icon, you are starting a Python program that helps you write Python programs. At first, this seems like a real mind-wrenching problem. You might think of it as similar to asking "which came first, the chicken or the egg?" . It isn't all that bad a problem however. In this case, someone else wrote **IDLE** to help you write your program. Your program, and **IDLE** (and a large number of other programs) all share the **Python** program as the driving engine.

Timeline. 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 http://www.cwi.nl/~steven/abc/. For information on Modula-3, see *Modula-3* [Harbison92], as well as http://www.modula3.org/.

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

2.5.2 Coolness FAQ's

If Python uses C, why not cut out the middleman and just learn C? We have a number of reasons for avoiding C. First, programming in C is a more difficult proposition because of the number of tools involved: C uses a compiler to build programs: you don't interact directly with C; you build a program, then interact with the operating system to run that program. Second, the C language is designed to make the C compiler work efficiently, it wasn't designed to be easy to write or easy to read. Third, C exposes a number of house-keeping chores that professionals can exploit for efficiency; they won't help newbies get their first program written.

If Python is so cool, why doesn't everyone use it? That's like asking why everyone doesn't like the Boston Red Sox, Philly cheese steaks, and the Red Hot Chili Peppers. Some people prefer Mom, Apple Pie and the Beatles. There's really no accounting for taste.

Some languages like Visual Basic and C# have the powerful and sophisticated marketing arm of Microsoft backing them. Other languages, like Java, have Sun backing them, and a large, well-established open-source community.

Some languages, like COBOL, are entrenched in the way data is processed at large organizations. While Python may be superior, it appears cheaper (in the short run) to leave the COBOL programs in place rather than convert them to something less complex and less expensive to operate and maintain.

The most important reason, however, is that languages are often specialized around particular tasks or data structures. Some languages, like SQL, express some operations more precisely and with a useful level of abstraction.

CHAPTER

THREE

USING PYTHON

Taking Your First Steps

Now that you have Python installed, we can start using it. We'll look at a number of ways that we can interact with the **Python** application. We'll use these interactions to learn the language.

In later sections, after we've got a more complete grip on the language and start to write programs, we'll move on to more advanced ways to use the **Python** program. Our goal is to use **Python** in an automated fashion to do data processing. Here in phase one, we'll be using **Python** manually to learn the language.

We'll describe the direct use of the **python** to process Python-language statements in *Instant Gratification : The Simplest Possible Conversation*. This will help us get started; it provides immediate gratification, but isn't the easiest way to work.

We'll dig into **IDLE** in *IDLE Time : Using Tools To Be More Productive*. We'll emphasize this as a good way to learn the language as well as build programs.

3.1 Instant Gratification : The Simplest Possible Conversation

There are two ways to exercise the **Python** program: interactively and with a script. In *interactive* mode, **Python** responds to each statement that we type in. In *script* mode, we give the **Python** program a file with a script of statements and turn it loose to interpret all of the statements in that script. Both modes produce identical results. Our goal is to write finished programs that will be run as a script. It's a long journey to scripting, which begins with some first small steps. We have to start with experimenting and exploring, so we'll use Python interactively. This gives us the instant gratification of a dialog with the **Python** program.

To be sure that we've got the basics installed and working, we'll use Python directly for our interactions. In the next chapter, we'll add the **IDLE** tool to the mix.

We'll look at starting Python in several sections: *The Windows Command Prompt*, *The Mac OS Terminal Tool*, and *The GNU/Linux Terminal Tool*. We'll look at ending out conversation in *How Do We Stop?*.

The real work starts in *Your First Conversation in Python: miles per gallon*. We'll look at numbers in *Decimal-Points and Accuracy* and look at more arithmetic in *More Conversations on Arithmetic*.

We'll examine some core features of the language in *Parenthesis and Precedence*, *Long-Winded Statements*, and *More About Punctuation*. We'll answer a few questions in *Direct Python Interaction FAQ*.

The Command or Terminal tool use of **Python** is the simplest and most ubiquitous way to use Python. This doesn't have flashy, interactive, colorful screens; it's just plain text. When we get to *Fit and Finish: Complete Programs*, we'll see that this way of using Python has an elegant simplicity that the experts use heavily.

3.1.1 The Windows Command Prompt

The **command prompt** is sometimes hard to find in Windows. In Windows 2000, you have to look in the **Start** menu under **Programs**, and then under **Accessories** to find the Command Prompt.

You can also use the **Run...** menu item in the **Start** menu. This will give you a small dialog box where you can type the name of a program. The name of the command prompt is just **cmd**. You can type `cmd`, and click **Okay**.

When you run the command tool, it will present a black window with a prompt from the operating system that looks something like `C:\Documents and Settings\SLott>`. Here, you can type the word `python`, hit return, and you're off and running.

Tip: Debugging Windows Command Prompt

In the unlikely event that you can't use Python from the **Command Prompt**, you have an issue with your Windows "path". Your path tells the **Command Prompt** where to find the various commands. The word **python** becomes a command when the `python.exe` file is on the system's path.

Generally, you should reinstall Python to give the Python installer a chance to set the path correctly for you. If, for some reason, that doesn't work, here's how you can set the system path in Windows.

Setting the Windows Path

1. Open the Control Panel.

   Use the **Start** menu, **Settings** sub menu to locate your **Control Panel**.

2. Open the System Control Panel

   Double-click the **System** Control Panel. This opens the *System Properties* panel.

3. Open the Advanced Tab of the System Control Panel

   Click the **Advanced** tab on the **System** Control Panel.

   There are three areas: Performance, Environment Variables and Startup and Recovery. We'll be setting the environment variables.

4. Open the Environment Variables of the Advanced Tab of the System Control Panel

   Click the **Environment Variables...** button.

   This dialog box has a title of *Environment Variables*. It shows two areas: user variables and System variables. We'll be updating one of the system variables.

5. Edit the Path variable

   This dialog box has a title of *Environment Variables*. Scroll through the list of System variables, looking for `Path`. Click on the `Path` to highlight it.

   Click the **Edit...** button.

   This dialog box has a title of *Edit System Variable*. It has two sections to show the variable name of `Path` and the variable value.

6. Add Python's location to the Path value

   This dialog box has a title of *Edit System Variable*. It has two sections to show the variable name of `Path` and the variable value.

Click on the value and use the right arrow key to scroll through the value you find. At the end, add the following `;C:\python26`. Don't forget the `;` to separate this search location from other search locations on the path.

Click **OK** to save this change. It is now a permanent part of your Windows setup on this computer. You'll never have to change this again.

7. Finish Changing Your System Properties

   The current dialog box has a title of *Environment Variables*. Click **OK** to save your changes.

   The current dialog box has a title of *System Properties*. Click **OK** to save your changes.

3.1.2 The Mac OS Terminal Tool

In the `Applications` folder, you'll find a `Utilities` folder. In the `Utilities` folder, you'll find a program named **Terminal**. Double click **Terminal** and you'll get a window with a prompt from the operating system that looks something like `[DVDi-Mac-1:~] slott%`. Here, you can type **env python**, and you're off and running.

You might want to drag the **Terminal** icon onto your dock to make it easier to find.

3.1.3 The GNU/Linux Terminal Tool

The Fedora Linux desktop, for example, 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** and you'll get a window which prompts you by showing something like `[slott@linux01 slott]$`. In response to this prompt, you can type **env python**, and you're off and running.

For non-Gnome Linux and Unix variants, you must find your Terminal tool, into which can type `python`.

3.1.4 How Do We Stop?

Once the **Python** program has started, it looks something like the following. It doesn't matter whether Python starts up from the command prompt or terminal window, the basic operation is the same. The window title and background color may be different, but our interaction with Python will be the same.

```
MacBook-5:PythonBook-2.6 slott$ python
Python 2.6.6 (r254:67917, Dec 23 2008, 14:57:27)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When we get the `>>>` prompt, the Python interpreter is listening to us. We can type any Python statements we want. Each complete statement is executed when it is entered.

We can ask Python to stop by using the `exit()` function. We enter `exit()` at the `>>>` prompt.

We can also enter the end-of-file character. This varies slightly from operating system to operating system.

MacOS and GNU/Linux. The polite way to tell **Python** that we're done is to enter `Ctrl-D`.

Windows. The polite way to finish a conversation with **Python** is `Ctrl-Z`, followed by `Enter`.

3.1.5 Your First Conversation in Python: miles per gallon

The **Python** program's job is very simple: it prompts you for a statement, executes the statement you entered, and then responds back to you with the result of executing the statement. This little three-step loop is all that **Python** does. Of course, as we will see, the real power comes from the wide variety of statements we provide in the Python language.

Important: Don't Forget to Run Python

While this may seem like a silly reminder, it's important to start the Python program. We emphasize it because it isn't always obvious what piece of software processes our Python language statements.

In *The Python Program and What It Does* we described our multi-tiered device built on top of the computer hardware. We're looking at these tiers: Computer System, Operating System, **Terminal** (or **Command Prompt**) and Python.

First, we start the computer system with the power button. The Operating System starts more-or-less automatically. Second, we use to OS to locate the **Terminal** (or **Command Prompt**), called a "shell". Third, from the Terminal, we run the **Python** program.

Each tier of software has it's own unique prompt. The basic operating system presents a slick GUI desktop metaphor with colorful icons and menus. The shell provides a technical-looking prompt like `C:\Documents and Settings\SLott>`, or `[DVD-iMac-2:~] slott%`. Python provides the `>>>` prompt that tells us we can enter a Python statement.

Bottom Line. We're always interacting with some program on our computer. We can't "simply type things"; we have to run a program which will respond appropriately when we give that program statements in a language it can process. If you don't see the `>>>` prompt, you're not interacting with Python.

Since we are all newbies to programming, we'll start with some very simple Python interactions, just to see what kinds of things Python can do. We'll start the **Python** program and then type Python statements that evaluate a simple formula. For these first few examples, we'll include the reminder to start running Python.

This first example will show the mathematical operation of ÷. If you look at your computer keyboard, you won't find the ÷ key. Python uses `/` (and `//`) for division.

```
1  MacBook-5:~ slott$ python
2  Python 2.6.6 (r254:67917, Dec 23 2008, 14:57:27)
3  [GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>> 351/18
6  19
```

1. The shell prompted me with `MacBook-5:~ slott$`. I typed `python` to start the python program running.
2. Python provided some information on itself.

5. Python prompted me with `>>>`. I typed `351 / 18` to compute miles per gallon I got driving to Newark and back home. This is a complete Python statement, and Python will evaluate that statement.

6. Python responded with `19`: a rotten 19 miles per gallon. I've got to get a new car that uses less gasoline.

This shows **Python** doing simple integer arithmetic. There were no fractions or decimal places involved. When I entered `351 / 18` and then hit `Return`, the Python interpreter evaluated this statement. Since the value not `None`, Python printed the results.

The usual assumption for numbers is that they are integers, sometimes called whole numbers.

Note that Python does not like `,` in numbers. Outside Python, we write large numbers with `,` to break the numbers up for easy reading. (The exception is the calendar year, where we omit the `,`: we write 2007, not 2,007.) Python can't cope with `,` in the middle of numbers. The mileage on my odometer reads 19,241. But, in Python we write this as `19241`.

Bottom Line. For now, be comfortable that Python is perfectly happy with whole numbers. Remember to avoid commas. We sometimes call these numbers *ints*, short for integers. Later, we'll see that Python has a pretty expansive set of numbers available to work with.

3.1.6 Decimal-Points and Accuracy

That calculation was nice, but you'll notice that whole numbers aren't really very accurate. If you pull out a calculator, you'll see that Python got a different answer than your calculator shows.

If you include a period in your numbers, you get "floating decimal point" numbers. We call these *floating-point* or *floats*. The number of digits on either side of the decimal point can "float". Floating point numbers are handy for many kinds of calculations.

Our previous conversation used whole numbers. Let's try again, using floating-point numbers.

```
>>> 351. / 18.
19.5
```

1. I typed `351. / 18.` to compute miles per gallon I got driving to Newark and back home.
2. Python responded with `19.5`: the more accurate 19.5 miles per gallon.

Floating-point isn't adequate for everything, so there's another kind of number that we'll get to later. When we do financial calculations on US dollars, the decimal point is fixed; we have two digits to the right of the decimal point and no more. These fixed-decimal point numbers aren't a built-in feature of Python, but there are ways to extend Python with a library that gives us this capability.

Bottom Line. For now, be comfortable that Python is perfectly happy with floating-point numbers that have about 17 total digits of accuracy, but a range that is huge. Remember to include a decimal point to tell Python that you want to see decimal places in the calculation. Also, remember to avoid commas, they're just confusing.

3.1.7 More Conversations on Arithmetic

So far, we've given arithmetic expressions to Python and Python's response is to evaluate those expressions. When we look at our keyboard, we can see that we have `/` (for division), `+` for addition and `-` for subtraction. What about multiplication ($\times$)? Square roots ($\sqrt{}$)? Raising to a power? These aren't keys on a standard keyboard.

Here are the arithmetic operations that Python recognizes in forms very similar to the way mathematics is written:

- Addition ($+$) is the `+` character. You say `123+456` to add two numbers.
- Subtraction (-) is the `-` character. You say `9116-8765` to find the difference between two numbers.
- Multiplication ($\times$) is the `*` character. You say `19*18` to find the product of two numbers.
- Division ($\div$) is the `/` character and the `//` sequence of characters. You say `351/18` to find the quotient of two numbers.
- Raising to a power (a^p) is the `**` sequence of characters. You can say `5**2` to raise 5 to the 2nd power, 5^2.

Additionally, Python (and many other programming languages) provide two handy operators that mathematicians don't normally write down in this form. Mathematicians may talk about "modular" arithmetic, with something like $a \bmod m$. This is written in Python using the `%` character. For non-mathematicians, this is the remainder after division.

```
>>> 355 % 113
16
>>> 355 / 113
3
```

Here's what this shows us: 113 goes into 355 with 16 left over. Mathematically, $355 = 3 \times 113 + 16$.

We'll look at all of these operators closely in *Simple Arithmetic : Numbers and Operators*.

3.1.8 Parenthesis and Precedence

The usual mathematical rules of operator precedence apply to Python expressions: multiplies and divides will be done before adds and subtracts. Plus, we get to use `()` are used to group terms against precedence rules. Unlike mathematics, we can't use `[]` and `{}` in arithmetic expressions. Mathematicians can use these, but in Python, we have to limit ourselves to just `()`.

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

```
>>> (65 - 32) * 5 / 9
18
>>> ((65 - 32) * (5 / 9))
0
>>> ((65 - 32) * (5 / 9.))
18.333333333333336
```

We have to put the `65-32` in parenthesis so that it is done before the multiply and divide. Also, you'll note that when one number is floating point (`9.`) it forces the calculation to be done as floating-point.

What would happen if we said `65-32*5/9`? Try it first, to see what happens.

If we don't include the `()` for grouping, then Python would do what every mathematician would do: compute `32*5/9` first and then the difference between that and 65. Python did what we said, but not what we meant. We know the answer is wrong because 65 ° Fahrenheit can't be the impossibly hot 48 ° Celsius.

In the second example, we put in extra `()` that don't change the resulting answer.

3.1.9 Long-Winded Statements

Python prompts us with the basic "I'm listening" prompt of `>>>` When we type an expression statement, Python prints the result for us, and then another prompt.

Python has a second prompt that you will see from time to time. It indicates that your statement isn't complete, and more is required. It's the "I'm still listening" prompt of `...`. Here's how it works.

For this section only, we'll emphasize the usually invisible `Return` key by showing it as . When we start using compound statements in *Processing Only When Necessary : The if Statement*, we'll add some additional syntax rules. For now, however, we have to emphasize that statements can't be indented; they must begin without any leading spaces or tabs. Here's a simple case: converting 65 ° Fahrenheit to Celsius.

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

What happens when the expression is obviously incomplete?

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

This leads us to the first of many syntax rules. We'll present them in order of relevance to what we're doing. That means that we're going to skip over some syntax rules that don't apply to our situation.

Important: Syntax Rule One

Statements must be complete on a single line. If the statement is incomplete, you'll get a `SyntaxError` response.

Just to be complete, we'll present syntax rule two, but it doesn't really have much impact on what we're going to be doing.

Important: Syntax Rule Two

The invisible end-of-line character is slightly different on different platforms. On Windows it is actually two non-printing characters, where on GNU/Linux and MacOS it is a single non-printing character. You may notice this when moving files back and forth between operating systems.

There is an escape clause that applies to rule one ("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
>>>
```

Yes, we skipped rules three, four and five.

Important: Syntax Rule Six

Python needs to see matching `()` pairs before it will consider the statement complete.

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

```
>>> 5 + 6 * \
... 7
47
>>>
```

This is called an *escape* and it allows you to break up an extremely long statement. It creates an escape from the usual meaning of the standard 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".

Important: Syntax Rule Five

You can use `\` to escape the usual meaning of a character.

Using \ at the end of a line escapes the meaning of the enter key, and allows a statement go continue onto multiple lines. While legal, this isn't the best policy, and we're going to avoid doing this.

3.1.10 More About Punctuation

We've been ignoring spaces in our expressions. There are some spaces in the examples, but we haven't been dwelling on precisely how many spaces and where the spaces are allowed. It turns out that spaces *other than indentation* are very flexible. Indentation is not flexible.

Important: Syntax Rule Nine

You can use spaces and tabs freely to separate tokens, or language elements.

We have found some kinds of mistakes that we can make with unclosed `()`s and an extra \ at the end of the line. This leads us to an important debugging tip.

Tip: Debugging Typing a Python Statement

When you see the `...` prompt from Python, it means that your statement is incomplete. Are you missing a `)` to make the `()` pairings complete? Did you accidentally use the \? Hit **Return** twice and you'll get a nice syntax error and you'll be back at the `>>>` where you can try again.

Also, you'll get unexpected errors if you try to use `[]`, and `{}` the way mathematicians do. Python only uses `()` to group expressions. If you try to use `[]`, you'll get a `TypeError: unsupported operand type(s) for [] : 'list' and 'int'`. If you try to use `{}`, you get a `SyntaxError: invalid syntax`.

3.1.11 Getting Help

Python has a help mode and a `help()` function.

When you enter `help()`, you'll wind up in help mode. This has a different prompt, to make it perfectly clear what kinds of things Python is doing.

```
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

You can see that the prompt is now `help>`. To go back to ordinary Python programming mode, enter `quit`.

```
help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

You can see that the prompt is now `>>>`.

If you ask for help on a specific topic, you'll enter something like this:

```
>>> help("EXPRESSIONS")
```

You'll get a page of output, ending with a special prompt from the program that's helping to display the help messages. The prompt varies: Mac OS and GNU/Linux will show one prompt, Windows will show another.

Mac OS and GNU/Linux. In standard OS's, you're interacting with a program named **less**; it will prompt you with `:` for all but the last page of your document. For the last page it will prompt you with `(END)`.

This program is very sophisticated. The four most important commands you need to know are the following.

q Quit the **less** help viewer.

h Get help on all the commands which are available.

Enter a space to see the next page.

b Go back one page.

Windows. In Windows, you're interacting with a program named **more**; it will prompt you with `-- More --`. The four important commands you'll need to know are the following.

q Quit the **more** help viewer.

h Get help on all the commands which are available.

Enter a space to see the next page.

3.1.12 Conversational Python 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

3.1.13 Direct Python Interaction FAQ

What Are The Missing Syntax Rules? Yes, we did skip over rules three, four, seven and eight. These are more advanced topics.

We'll look at rules three and four in *Comments and Scripts*. These rules have to do with lines Python ignores, called "comments" and character encoding for Python files.

We'll look at rules seven and eight in *Processing Only When Necessary : The if Statement*. These rules have to do with indenting and completing compound statements.

3.2 IDLE Time : Using Tools To Be More Productive

We'll look at how we can use the **IDLE** program to make our lives easier. **IDLE** puts an *Integrated Development Environment* (IDE) around the **Python** program. Rather than work with Python directly, using the text-only interface, we'll add some nice features that help us spot errors, save files, and generally see that's going on.

We'll look at this what this environment helps us do in *The Development Environment*. We'll look at how to start and stop **IDLE** in *Starting and Stopping IDLE*. We'll touch on the basic features of **IDLE** in *Using IDLE's "Python Shell" Window*. We'll do the real work of entering simple Python statements in *A Conversation Using IDLE*.

In *IDLE Interaction FAQ* we'll answer some common questions.

3.2.1 The Development Environment

Python programming is often done using an Integrated Development Environment (IDE) named **IDLE**. The **IDLE** program does a number of things that make programming easier.

- IDLE runs the Python interpreter in interactive mode for us. You'll see this in a window named "Python Shell". You can type Python statements and have them executed immediately. We'll make heavy use of this mode to continue to get instant gratification.
- IDLE has an easy-to-use text file editor. When you create a new window, **IDLE** opens a very nice editor. This editor knows the Python language and can highlight syntax in color, making the program easier to read.
- When we start writing scripts, we'll find that **IDLE** can run our scripts for us, saving the output from the script for us.

IDLE isn't your only choice for an IDE. It is, however, free and so easy to use that we'll focus on it in this book. There are some alternatives that might want to explore.

- ActiveState's **Komodo** is a very sophisticated editor that knows Python.
- MacOS programmers sometimes use **BBEdit** or **TextMate**.
- Windows programmers sometimes use **Textpad** or **notepad++**.
- Linux programmers can work directly from the command line, using **vi** or **emacs** or any of the other text file editors available in GNU/Linux.

- The **LEO** editor can be used to create complex programs. It is a *literate programming* editor with outlines. **LEO** isn't as easy for newbies to use because it is focused on experts. **LEO** is written in Python, also.

3.2.2 Starting and Stopping IDLE

After we get past the operating-specific details, we'll see that the *Python Shell* window of **IDLE** is the same on all of out various operating systems.

Windows

You can use the **Start** menu, **Programs** submenu, **Python 2.6** submenu to locate the **IDLE (Python GUI)** menu item. This will open the *Python Shell* window.

MacOS

You can go to your `Applications` folder, find the `Python 2.6` 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.

It can help to drag the `IDLE` icon onto your dock to make it easier to find.

GNU/Linux

You have two choices for starting IDLE under GNU/Linux: from the **Terminal** or using the GUI. Configuring GNOME or KDE to include an icon for starting **IDLE** is beyond the scope of this book. It isn't hard, but it makes the book too big. So we'll skip straight to using the **Terminal** to start **IDLE**.

Ideally. If your Linux is setup correctly, you may find that the system's `PATH` includes `/usr/lib/python2.6/idlelib/`. If this is the case, then entering the following command will start IDLE.

```
idle.py
```

If this doesn't work, you'll see a response like this

```
MacBook-5:~ slott$ idle.py
-bash: idle.py: command not found
```

Less Ideal. From the **Terminal** prompt, you can type the following command to start **IDLE** .

```
env python /usr/lib/python2.6/idlelib/idle.py &
```

Yes, this is long. There are some ways to shorten this up. We'll cover some of them because they tell us a lot about how GNU/Linux really works. You only have to do one of these. Pick the one method that seems simplest to you and ignore the others.

- Write a script. This is a short file that becomes a new Linux command.
- Update your `PATH` setting. This is a change to your environment that makes the `idle.py` file usable by your shell.
- Create an alias. This is a change to your environment that creates a new Linux command.
- Create a link. This adjusts the file system so that idle appears to be in your home directory. This is a bit risky because your file system may not be organized the same as mine, meaning my example may not work for you.

Write a Script. To create a script, you'll put a command in a file, and mark that file as executable. Once you've done these two steps, you've effectively added a new command to your GNU/Linux environment.

1. Use an editor (I like **gedit**) to create a file named `idle`. Put this above into that file as the only line. Save the file into your home directory.

   ```
   env python /usr/lib/python2.6/idlelib/idle.py &
   ```

2. Execute the command **chmod +x idle** to mark your new file as executable.

Now you can type **./idle** to start **IDLE**. When you do it this way, you've written your first program! Okay, it's only one line, but it's a program.

Tip: Debugging A Script

If your `idle` script file doesn't work, there are some common things to confirm:

- Your file is in the same directory that the **Terminal** starts in. If you are unsure, you can use the **pwd** to print the working directory. In my case it is `/home/slott`. That's where I put my `idle` startup file.
- Your file is plain text. A word processor won't save files as plain text automatically, so you should use something like **gedit** to assure that you're creating a plain text file.

Update your PATH. To update your path, you must make sure that the shell sets the environment you want.

Most shells, it turns out, read a hidden file named `.profile` every time you log in to GNU/Linux. The **bash** shell reads `.bash_profile` . There's a two step process to creating an alias. Once you've done these two steps, you've configured your shell environment.

1. Use an editor (I like **gedit**) to update your `.profile` or `.bash_profile` file.

   You won't see this file in ordinary directory listings; the '`.`' in the name means that it's hidden; use **ls -a** to see *all* files. Insert the following line at the very end. Note that the apostrophes are essential to making this work.

   ```
   export PATH=$PATH:/usr/lib/python2.6/idlelib
   ```

2. Log out. That way, when you log in again, your `.profile` is executed.

Now you can type **idle.py** to run the **IDLE** program.

Create an Alias. To create an alias, you have to make sure that the **alias** command is executed every time you log in.

Most shells, it turns out, read a hidden file named `.profile` every time you log in to GNU/Linux. The **bash** shell reads `.bash_profile` . There's a two step process to creating an alias. Once you've done these two steps, you've configured your shell environment.

1. Use an editor (I like **gedit**) to update your `.profile` or `.bash_profile` file.

   You won't see this file in ordinary directory listings; the '`.`' in the name means that it's hidden; use **ls -a** to see *all* files. Insert the following line at the very end. Note that the apostrophes are essential to making this work.

   ```
   alias idle='env python /usr/lib/python2.6/idlelib/idle.py &'
   ```

2. Log out. That way, when you log in again, your `.profile` is executed.

Now you can type **idle** to run the **IDLE** program. This is a handy technique, but we don't want to go overboard creating too many aliases.

Tip: Debugging An Alias

If your alias doesn't work, there are some common things to confirm:

- Your `.profile` works correctly. You can type **sh -v .profile** or **bash -v .bash_profile** to test it. If you see error messages, likely you missed an apostrophe or messed up the spaces.

Create a Link. To create a link, you'll execute one command. This will make the `idle.py` file exist in your home directory as well as the Python library directory.

1. Execute the command **ln /usr/lib/python2.6/idlelib/idle.py idle.py** to create the link.

You may get an error if your Python environment isn't the same as mine. You may have to search around for the correct location for your Python implementation.

Now you can type **env python2.6 idle.py** to start **IDLE**. This is at least short and reasonably easy to remember.

3.2.3 Using IDLE's "Python Shell" Window

After we get past the operating-specific details, we see that the *Python Shell* window of **IDLE** is the same on all of the various operating systems. The *Python Shell* window will have the following greeting.

```
Python 2.6.6 (r266:84374, Aug 31 2010, 11:00:51)
[GCC 4.0.1 (Apple Inc. build 5493)] 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 2.6.6
>>>
```

If you have personal firewall software and it does warn you about IDLE, you can ignore your personal firewall's messages. Your firewall is detecting ordinary activity called "interprocess communication" among the various components of IDLE. Rather than a personal firewall, I buy routers that do this for all the computers in my home.

You can use the **File** menu, item **Exit** to exit from IDLE. You can also close the window by clicking on the close icon.

3.2.4 A Conversation Using IDLE

When **IDLE** is running, the *Python Shell* shows the Python >>> prompt. This is the same "I'm Listening" prompt we saw in *Your First Conversation in Python: miles per gallon*. Interacting with Python through **IDLE** is the same as interacting with Python directly.

```
>>> -20 * 9/5 + 32
-4
```

Interesting. When the Stockholm weather says -20 Celsius, that is -4 Fahrenheit. That's cold.

Drat! We used numbers without any decimal points. That means we used integer division, which won't be very accurate. We'd like to try that statement again without having to retype the entire thing from scratch.

IDLE has a couple of features to make it possible to work more efficiently.

First, we have ordinary copy and paste capabilities. If you look on the **Edit** menu, you'll see the usual culprits: **Cut** , **Copy** and **Paste** . If you are new to this sort of thing, here's the play-by-play.

1. Click and drag to highlight the `-20 * 9/5 + 32`.
2. Use **Edit** menu, **Copy** menu item to copy this. Or, you can use `Ctrl-C` to copy this.
3. Click after the last `>>>` prompt.
4. Use **Edit** menu, **Paste** menu item to paste the saved command. Or, you can use `Ctrl-V` to paste it.

Once you have the almost-correct statement, you can use the left-arrow key to move across the line and add decimal points to make the line look like this: `-20 * 9./5. + 32`.

The cooler technique is to use the up-arrow key to go back to our original command. Here's the play-by-play.

1. Hit the `Up Arrow` to go back to the original command. Twice should do it.
2. Hit `Enter` and this line will be entered again, and the blinking cursor will be at the right end of the new line.

Once you have the almost-correct statement, you can now use the left-arrow key to move across the line and add decimal points. This up-arrow and left-arrow editing is perhaps the easiest way to re-enter a command with minor changes.

3.2.5 IDLE Exercises

1. **Check the IDLE Version**.

   Note the version number of **IDLE**. You have a **Help** menu, look here for information.

2. **Enter simple commands**.

   In the initial window of **IDLE**, you can enter the following one-line commands to Python:

   - copyright
   - license
   - credits
   - help

3. **Save a file**.

   Use **New Window** under the **File** menu to create a simple file. Write a few words in this file. Save this file. It is very important to note the directory that appears initially when you save. Be sure to pay careful attention to where the file gets saved. If it doesn't get saved to your own home directory, you'll need to figure out two things:

   (a) What directory IDLE starts working in. Your operating system should provide some suitable hints on the directory you're using.

   (b) How to navigate to your home directory. Again, your operating system should provide ways to save your file to your preferred working directory.

3.2.6 IDLE Interaction FAQ

Why are the multiple ways to use Python? Python can be used a variety of ways, depending what problem you are solving.

We can interact directly with Python at the "command-line". This was what we saw in *Instant Gratification : The Simplest Possible Conversation*. This is available because Python is must usable when it is a shell program.

A tool like **IDLE** makes it easier to enter Python statements and execute them. **IDLE** shows us a *Python Shell* window, which is effectively the command-line interaction with Python, plus offers a handy text editor as a bonus. **IDLE** is both written in Python and uses Python as a shell program.

A tool like **BBEdit** or **TextPad** is a handy text editor that can execute the Python command-line tool for us. This interaction is made possible because "under the hood", **Python** is a command-line program with the ultra simple character-oriented command-line interface.

Why all the colors? Can I turn that off? Some newbies find syntax coloring distracting. Most experienced programmers find it very handy because the colors provide immediate feedback that the syntax of the statement is sensible.

If you want to change or disable the syntax coloring, use the **Options Configure IDLE...** to provide different syntax coloring.

CHAPTER

FOUR

ARITHMETIC AND EXPRESSIONS

Before Reading and 'Riting comes 'Rithmetic

The heart of Python is a rich variety of numeric types and arithmetic operators. We can use these various numeric types to do basic mathematical operations on whole numbers, real numbers and complex numbers. We'll look at the basics in *Simple Arithmetic : Numbers and Operators*.

In addition to the basic arithmetic capabilities, many kinds of problems need additional mathematical and financial functions. We'll look at some of the built-in functions and some functions in add-on modules in *Better Arithmetic Through Functions* and *Extra Functions: math and random*.

For more specialized problems, Python has a variety of additional operators. We'll look more deeply at these additional operators in *Special Ops : Binary Data and Operators*.

We'll cover some optional topics in *More Advanced Expression Topics*, including different approaches to *execution* of Python statements, some notes on Python writing style.

4.1 Simple Arithmetic : Numbers and Operators

Python provides four slightly different flavors of numbers: plain integers, long integers, floating-point numbers and complex numbers. Each of these have their various strengths and weaknesses. The mathematical abstraction of *number* doesn't really exist inside the computer. Instead, we have different representations of numbers, each reflecting a distinct tradeoff in the amount of computer memory required and the speed of performing operations.

In *Plain Integers, Also Known As Whole Numbers* we'll look at basic numbers. In *Floating-Point Numbers, Also Known As Scientific Notation* we'll look at numbers with a wider range of values. We'll look at Python's ability to handle very large numbers in *Long Integers – Whole Numbers on Steroids*. We'll review the rules for mixing different species of numbers in *Mixing Numbers, Some More Rules*.

For the mathematicians and engineers, we'll look at complex numbers in *Complex Numbers – For The Mathematically Inclined*; this is optional material unless you're really curious.

We'll look at strings of characters in *Strings – Anything Not A Number*.

For the most part, Python uses conventional decimal numbers, in base 10. However, for specialized computer-related tasks, Python can also work in base 8 or base 16. There is a hidden shoal here, so we'll look at alternate bases in *Octal and Hexadecimal – Counting by 8's or 16's*.

Warning: Python 3 Changes
In Python 3, the distinction between integers and long integers will be almost invisible to the naked eye.
Some of the distinctions we mention here will go away.
Also, the `/` division operator will change it's meaning slightly.

4.1.1 Plain Integers, Also Known As Whole Numbers

Plain integers in Python are written as strings of digits without commas, periods, or dollar signs. A negative number begins with a single `-`. Plain integers have range ± 2 billion, or about 9 decimal digits.

Internally, an integer is compact, using just four bytes of memory. It's also blazingly fast for most mathematical operations. However, this small size and high speed also mean that it has a limited range of values.

Here are some examples of integers. Note the absence of `,`, `.`, `$` or other punctuation. We can only use `-` to mean a negative number.

- `0`
- `2005`
- `8675309`
- `-42`

Here are the basic arithmetic operations that Python recognizes:

- Addition (+) is the `+` character.
- Subtraction (-) is the `-` character.
- Multiplication (×) is the `*` character.
- Division (÷) is the `/` character for standard division, and `//` for integer-like division.
- Raising to a power (a^p) is the `**` sequence of characters.
- Modulus (remainder in division) is the `%` character.
- Grouping is done with the `(` and `)` characters.

Here are some examples.

```
>>> 32 - 42
-10
>>> 42 * 19 + 21 / 6
801
>>> 2**10
1024
>>> 241 % 16
1
>>> (18-32)*5/9
-8
```

Pay close attention to `42 * 19 + 21 / 6`. In particular, remember that your desktop calculator may say that $21 \div 6 = 3.5$. However, since these are all integer values, Python uses integer division, discarding fractions and remainders. `21/6` is precisely `3`.

Does Python Round? Try this to see if Python rounds. If Python does not round, the answers will all be 2. If Python does round, the answers will be 2, 2, 3 and 3.

```
8 / 4
9 / 4
10 / 4
11 / 4
```

What happened? It shouldn't be any surprise that integer arithmetic is done very simply. For more sophistication, we'll have to use floating-point numbers and complex numbers, which we'll look at in later sections.

New Syntax: Functions. More sophisticated math is separated into the `math` module, which we will look at in *The math Module – Trig and Logs*. Before we get to those advanced functions, we'll look at a few less-sophisticated functions.

The absolute value (sometimes called the magnitude or absolute magnitude) operation is done using a slightly different syntax than the conventional mathematical operators like `+` and `-` that we saw above. A mathematician would write $|n|$, but this can be cumbersome for computers. Instead of copying the mathematical notation, Python uses a kind of syntax that we call *prefix* notation. In this case, the operation is a prefix to the operands.

Here are some examples using the `abs()` function.

```
>>> abs(-18)
18
>>> abs(6*7)
42
>>> abs(10-28/2)
4
```

The expression inside the parenthesis is evaluated first. In the last example, the evaluation of `10-28/2` is -4. Then the `abs()` function is applied to -4. The evaluation of `abs(-4)` is 4.

Here's the formal Python definition for the absolute value function.

`abs`(*number*) → number

Returns the absolute value of *number*.

For non-numeric arguments, raises a `TypeError`.

This tells us that `abs()` 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 *Basic Sequential Collections of Data*.

4.1.2 Floating-Point Numbers, Also Known As Scientific Notation

Floating decimal point numbers are written as strings of digits with one period to show the decimal point. They can't have commas or dollar signs. A negative number begins with a single `-`. We call them floats or floating point numbers for short.

These numbers are different from the "fixed-point" decimal numbers that we use for financial calculations. With fixed-point numbers, the number of positions to the right of the decimal place is fixed. Doing fixed-point processing in Python is done with an add-on library; we'll cover this in *Currency Calculations: Fixed Point Math*.

A floating-point number takes at least eight bytes, making it twice the size of a plain integer. The extra complexity of scientific notation makes them much slower than plain integers. They have about 17 digits of useful precision, but they can represent values with an astronomical range.

Here are some examples:

- `0.`

- `3.1415926`
- `867.5309`
- `-42.0`

We can, if we want, write our numbers in scientific notation. A scientist might write 6.022×10^{23}. In Python, they use the letter `E` or `e` instead of $\times 10$. Here are some examples.

- `6.022e23`
- `1.6726e-27`
- `8.675309e3`
- `2.998e8`

All of the arithmetic operators we saw in *Plain Integers, Also Known As Whole Numbers* also apply to floating-point numbers. Here are a couple of examples.

```
>>> 6.62e-34 * 2.99e8
1.97938e-25
>>> 3.1415926 * 3.5**2
38.484509350000003
```

You Call That Accurate? What is going on with that last example? What is that "0000003" hanging off the end of the answer?

That tiny, tiny error amount is the difference between the decimal (base 10) display and the binary (base 2) internal representation of the numbers. That tiny, annoying error can be made invisible when we look at formatting our output in *Sequences of Characters : str and Unicode*. For now, however, we'll leave this alone until we have a few more Python skills under our belt.

One consequence is that some fractions are spot-on, while others involve an approximation. Anything that involves halves, quarters, eighths, etc., will be represented precisely. 3.1 has to be approximated, where 3.25 is something that Python handles exactly.

Important: Mixing Numbers

When you mix numbers, as in `2 + 3.14159`, Python *coerces* the integer value to a floating-point value. This assures that you never lose any information. It also means that you don't have to meticulously check every number in a statement to be sure that they are all floating-point. As long as some numbers are floating-point, the others will likely get promoted properly.

The coercion rules are done for each individual operation. `2+3/4.0` and `2.0+3/4` will do different things. We'll return to this below.

Scientific Notation. Floating point numbers are stored internally using a fraction and an exponent, in a style some textbooks call "scientific notation". Usual scientific notation uses a power of 10. In the Python language, we write the numbers as if we were using a power of 10. We think of a number like `123000` as `1.23e5`. Mathematically, it means the following,

$$n = 1.23 \times 10^5$$

While the Python language allows us to enter our numbers in good-old decimal, our computer doesn't use base 10, it uses base 2. Really, our floating point numbers are converted to the following form.

$$n = b \times 2^b$$

Specifically, it becomes this inside the computer's hardware:

$$n = 0.93841552734375 \times 2^{17}$$

This emphasizes how two conversions – between the value of `1.23` (as entered in base 10) to `0.938... blah blah blah` (in base 2) and then back to base 10 to display it for human consumption – reveals tiny differences in how a decimal fraction is approximated by a binary fraction.

One important consequence of this is the need to do some algebra before simply translating a formula into Python. Specifically, if we subtract two nearly-equal floating point numbers, we're going to magnify the importance of the stray error bits that are artifacts of conversion.

4.1.3 Some Words on Division

Now that we've seen integers and floating-point numbers, we can look more closely at division.

Here's the example of `/` division: integer values give an integer answer, floating-point values give a floating point answer, mixed values lead to coercion, and a float-point answer.

```
>>> 355/113
3
>>> 355./113.
3.1415929203539825
>>> 355./113
3.1415929203539825
```

Division introduces a subtle question of "what do we expect"?

- An exact answer, as a floating-point value. `355/113` *should* be 3.1415929203539825.
- An answer based on the types of values. `355/113` *should* be 3.

Both are legitimate points of view.

Historically, Python took the second position: answers are based on the types of the values. It turns out that this approach isn't adequate. We really need two division operators that clarify our expectations.

We'll look at this in depth in *The Two Specialized Division Operators: / and //*.

4.1.4 Long Integers – Whole Numbers on Steroids

Python allows us to use very 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.

There's a trade-off with long integers. An ordinary integer uses relatively little memory and the operations are blazing fast. A long integer will use a lot of memory and the operations are quite slow.

We write long integers as a string of digits (no periods) that end in `L` or `l`. Upper case `L` is preferred, since the lower-case `l` looks too much like the digit `1`. Additionally, Python is graceful about converting to long integers when it is necessary.

Here are some examples of long integers. Note the absence of `,`, `.`, `$` or other punctuation. We can only use `-` to mean a negative number.

- `0L`
- `2005L`
- `4294967296L`
- `-42L`

How many different combinations of 32 bits are there? The answer is there are 2^{32} (we write this `2**32` in Python). The answer is too large for ordinary integers, and we get an answer in long integers.

```
>>> 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?

```
2**(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.

Important: Mixing Numbers

When you mix numbers, as in `2 + 3L`, Python coerces the integer value to a long value. This assures that you never loose any information. If you mix long and floating-point numbers, as in `3.14 + 123L`, the long number is converted to floating-point.

4.1.5 Mixing Numbers, Some More Rules

We've noted in a couple of places that when you have mixed kinds of numbers Python will coerce the numbers to be all one kind. The rules aren't that complex, but they're important for understanding the semantics of a mathematical formula.

When you mix integers and longs, the integers are coerced to be longs. The idea here is that a long will preserve all the information of an integer, even though the long works more slowly. It's a fair tradeoff. `2+3L` is `5L` because the `2` was coerced to `2L`.

When you mix integers or long integers with floating-point, the integers are coerced to floating-point. Again, the idea is to preserve as much information as possible. However, the floating-point version of a number might not preserve every digit.

A floating-point number can represent a vast range of values, but it only has about 17 digits of precision. A long integer can have any number of digits. If your long integer is over 17 digits, some of the precision has to be sacrificed, and it will be the right-most digits of the long integer.

Remember that a floating-point number's right-most digits aren't all perfectly accurate; we're reminded of that every time we see a dangling "0000003". Consequently, making a floating-point value into a long integer doesn't work out well. Some of the digits on the right-hand end of such a number are more error than precision.

How Coercion Happens. Coercion is something Python does as it evaluates each operator. Here's something you can try to see the effect of these rules.

```
>>> 2+3/4.0
2.75
>>> 2.0+3/4
2.0
```

In the first example, the first expression to be evaluated (`3/4.0`) involves coercing `3` to `3.0`, with a result of `0.75`. Then the `2` is coerced to `2.0` and the two values added to get `2.75`.

In the second example, the first expression to be evaluated (`3/4`) is done as integer values, with a result of `0`. Then this is coerced to `0.0` and added to `2.0` to get `2.0`.

As we'll see in *Functions are Factories (really!)* we can force specific conversions if Python's automatic conversions aren't appropriate for our problem.

4.1.6 Complex Numbers – For The Mathematically Inclined

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 + 14j`. Note that Python always prints these in `()`s; for example `(2+14j)`.

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

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

4.1.7 Strings – Anything Not A Number

A string is a sequence of characters without a specific meaning. We surround strings with quotes to separate them from surrounding numbers and operators. Unlike a number, which supports arithmetic operations, a string supports different kinds of operations including *concatenation* and *repetition*.

You can use either apostrophes (`'`) or quotes (`"`) to surround string values. This gives you plenty of flexibility in what characters are in your strings. You can put an apostrophe into a quoted string, and you can put quotes into an apostrophe'd string. The full set of quoting rules and alternatives, however, will have to wait for *Sequences of Characters : str and Unicode*.

Here are some examples of strings. We use apostrophes for the strings that have quotes. We use quotes for the strings that have apostrophes.

- `"Hello world"`
- `'"The time has come," the walrus said'`
- `"Alice's Adventures in Wonderland"`

What if we need both quotes and apostrophes in a single string value? We have to use a technique called an *escape*. In a quoted string, we may need to escape from the usual meaning of the quote as the end of the string. We use the character `\` in front of the quote as an escape. In a quoted string, we use `\"` to include a quote inside the string. In an apostrophe string, we use `\'` to embed an apostrophe.

- `"Larry said, \"Don't do that.\""`
- `'Natalie said, "I won\'t."'`

The first example shows a quoted string with a quotation inside it. If we tried `"Larry said, "Don't do that.""`, we would have a syntax error. We'd have a quoted string (`"Larry said, "`), some random letters (`Don't do that.`), and another zero-length quoted string (`""`). We have to escape the meaning of the two internal quotes, so we use `\"` for them.

The second example shows an apostrophe'd string with an apostrophe inside it. To escape the meaning of the apostrophe, we use `\'`.

String Operators. Strings have two basic operators:

- Concatenation is the `+` operator; it puts two strings together to make a new string.
- Repetition is the `*` operator; it repeats a strings several times to make a new string.

Here are some examples. Note that we had to include spaces in our strings so that the concatenation would look good.

```
>>> "Walrus" + " and the " + "Carpenter"
'Walrus and the Carpenter'
>>> "Pigs " * 3
'Pigs Pigs Pigs '
>>> "Whether " + ("pigs have " * 2 ) + "wings"
'Whether pigs have pigs have wings'
```

We'll use strings more heavily in *Seeing Results : The print Statement*. It turns out that strings are actually very sophisticated objects, so we'll defer exploring them in depth until *Sequences of Characters : str and Unicode*.

Note: Adjacent String Literals

As a special case, Python will automatically concatenate adjacent string literals. This only works for quoted strings, but sometimes you'll see programs that look like this.

```
big_string = "First part of the message, " \
    "second part of the message." \
    "The end of the message."
```

Remember from *Syntax Rule 5* that the \ extends the statement to the next line. This statement is three adjacent string literals. Python will concatenate these three strings to make one long message.

4.1.8 Octal and Hexadecimal – Counting by 8's or 16's

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

You might say to yourself, "Why am I reading this section? I'm not a computer heavyweight!" It turns out that there is a hidden shoal lurking just under the surface of the numbers we've seen so far. The debugging tip, below, is the reason we have to mention this topic.

For much, much more information on bits, bytes, octal and hexadecimal, see *Special Ops : Binary Data and Operators*.

Base 8 – "Octal". A number with a leading 0 (zero) is octal and uses the digits 0 to 7. Here are some examples:

- `0`
- `0123`
- `-077`
- `012`

When you enter one of these numbers, Python evaluates it as an expression, and responds in base 10.

```
>>> 0123
83
>>> 0777
511
```

An attempt to use digits 8 and 9 in an octal number is illegal; In base 8, we only have the digits 0 to 7. It doesn't make sense to try and use 8 and 9 in an octal value.

Consequently, there's a strange looking error message if you do try.

```
>>> 09
  File "<stdin>", line 1
    09
     ^
SyntaxError: invalid token
```

In the obscure parlance of language parsing, any symbol, including a number is a *token*. In this case, the token could not be parsed because it began with a zero, but it did not continue with digits between 0 and 7. It isn't a proper numeric token.

Tip: Debugging Octal Numbers (Leading Zero Alert)

A number that begins with a zero is supposed to be in base 8. If you are copying numbers from another source, and that other uses leading zeros, you may be surprised by what Python does. If the number has digits of 8 or 9, it's illegal. Otherwise, the number isn't decimal.

I spent a few hours debugging a program where I had done exactly this. I was converting a very ancient piece of software, and some of the numbers had zeroed slapped on the front to make them all line up nicely. I typed them into Python without thinking that the leading zero meant it was really base 8 not base 10.

Base 16 – "Hexadecimal". A number with a leading `0x` or `0X` is hexadecimal, base 16. In order to count in base 16, we'll need 16 distinct digits. Sadly, our alphabet only provides us with ten digits: 0 through 9. The computer folks have solved this by using the letters a-f (or A-F) as the missing 6 digits. This gives us the following way to count in base 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, 12, 13, 14, etc.

In math textbooks, they sometimes write this: 53_{16} to indicate that it's a value using base 16. We figure out the value by applying base 16 place values of 1, 16, 256, etc. So we have $53_{16} = 5 \times 16 + 3 = 83_{10}$.

In Python, we use `0x` as a prefix.

Here are some examples of hexadecimal numbers:

- `0x0`
- `0x123`
- `-0xcbb2`
- `0xbead`

When you enter one of these numbers, Python evaluates it as an expression, and responds in base 10.

```
>>> 0x53
83
>>> 0x1ff
511
>>> 0xffcc33
16763955
```

Hex or octal notation can be used for long numbers. `0x234C678D098BAL`, for example is `620976988526778L`.

4.1.9 Expression Exercises

1. **Some Simple Expressions**.

   Evaluate each of the following expressions. In some places, changing integers to floating-point produces a notably different result. For example `(296/167)**2` and `(296.0/167.0)**2`.

   `pow( 2143/22, 0.25 )`

```
355/113 * ( 1 - 0.0003/3522 )
```

```
22/17 + 37/47 + 88/83
```

```
(553/312)**2
```

2. **Stock Value**.

   Compute value from number of *shares* × *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 expression 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.

3. **Convert Between ° C and ° F**.

   Convert temperature 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).

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

   Create an expression to convert 18 °C to °F.

   Create an expression to convert -4 °F to °C.

4. **Periodic Payment**.

   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.

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

   Mortgage with payments due at the end of each period:

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

   Mortgage woth payments due at the beginning of each period:

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

   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.

5. **Surface Air Consumption Rate**.

   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.

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

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 expressions 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.

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

For example, tanks you own might have a starting pressure of 2500 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 expressions that show how long one can dive at 60 feet and 70 feet.

1. **Wind Chill**. 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.

   Wind Chill, new model

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

   Wind Chill, old model

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

   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.

   You can do square root of a given wind speed, *V*, using an expression like `V ** 0.5`. For example, a 20 mph wind would use `20 ** 0.5` in the formula.

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

2. **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).

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

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

3. **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 an expression 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.

4. **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 zeros 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 an expression 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 an expression that shows the difference between the pay out and the real odds.

4.1.10 Expression FAQ

Why are numbers 32 bits? The coffee-shop answer is "that's the way computers are built".

The real answer is that the use of 32 bits has a long engineering history. One very important consideration is parallelism. The processor chip designers want to have many things happen at the same time. In the case of retrieving data from memory, getting data in 4-byte chunks will take 1/4 the time of getting data in 1-byte chunks. Modern processors often fetch a very large number of bits from memory and keep it in a special cache buffer on the processor chip.

The number of bits used to represent data has varied somewhat. A comfortable group of bits is called a *byte*. Some older computers used 9-bits in each byte, and put four of these together to make 36-bit numbers. Early modems used a signal protocol optimized to send 7-bits in each byte.

The 7-bit byte allows for 128 values in a single byte. If we take the US Latin alphabet (26 lower case letters, 26 upper case letters, 10 digits, 40-odd punctuation marks) have about 96 characters. Adding some additional codes for housekeeping, we have 128 character codes, which only needs a 7-bit number to encode each character.

We can then use an eighth bit to carry a primitive error-detection code. We can insist that each valid character code have an even number of bits switched on. If we receive a character with an odd number of bits, we know that a bit got garbled. This is one of the many historical precedents that made 8-bit bytes appealing.

Also, of course, there is an elegant symmetry to using 8-bit bytes when we are using binary number coding. The powers of two that we use for binary number positions are 1, 2, 4, 8, 16, 32, 64 and 128. This sequence of numbers has almost mystic significance. Of course we would prefer 8-bit bytes over 9-bit bytes. 32-bit numbers fit this sequence of numbers better than 36-bit numbers.

From Bytes to Words. Once we've settled on 8-bit bytes, the next question is how many bytes make up a respectable "word". Early computers had 64 kilobytes of memory, a number that requires only 16 bits (2 bytes) to represent. We can use a two-byte register to identify any of the bytes in memory. Many early microprocessors made use of this. The legendary Apple][PC had a 6502 processor chip that worked this way. Growing this to 640K only adds 4 more bits to the address information, a kind of half-byte compromise that Microsoft made use of to create DOS for the Intel 8088 processor chip.

In the metric measurement system, a kilometer is 1,000 meters. In the world of computers, there is an elegant power-of-two number that we use instead: 1024. A kilobyte, then is 1024 bytes; a megabyte is 1024*1024 = 1,048,576 bytes; a gigabyte is 1,073,741,824 bytes.

As the amount of memory grew, the size of numbers had to grow so that each location in memory could have a unique numeric address. Currently, 32-bit numbers are oriented around computers with 2 gigabytes of memory. Newer, larger computers use 64-bit numbers so that they can comfortably handle more than 2 Gb of memory.

Is the 8-bit byte still relevant? When we look at the world's alphabets, we discover that our 26-letter US Latin alphabet isn't really very useful. For most European languages that use the Latin alphabet we'll need to add a number of accented characters. For mathematics, we'll need to add a huge number of special characters. Once we open the door, we might as well provide for non-Latin alphabets like Greek, Arabic, Cyrillic, Hebrew and others. We're going to need a lot more than 128 character codes. And then there's the Chinese problem: there are thousands of individual characters. This is solved by having *Multi-byte Character Sets* (MBCS). Currently the Unicode standard uses as many as four bytes to represent the world's alphabets.

Since a byte is no longer an individual character, it is not relevant for that purpose. However, it is the unit in which memory and data are measured, and will be for the foreseeable future.

4.2 Better Arithmetic Through Functions

We've seen one Python function, `abs()`, that is also a standard mathematical function. The usual mathematical notation is $|x|$. Some mathematical functions are difficult to represent with simple lines of text, so the folks who invented Python elected to use "prefix" notation, putting the name of the function first.

This function syntax is pervasive in Python, and we'll see many operations that are packaged in the form of functions. We'll look at many additional function definitions throughout this book. In this chapter, we'll focus on built-in functions.

We'll look at a few basic functions in *Say It With Functions*; we'll show how formal definitions look in *pow() and round() Definitions*. We'll show how you can evaluate complex expressions in *Multiple Steps*. We'll touch in the accuracy issue in *Accuracy?*. We'll look at how Python gives you flexibility through optional features in *Another Round, Please*.

There are a number of conversion or *factory functions* that we'll describe in *Functions are Factories (really!)*. In *Going the Other Way* we'll see how we can use conversion functions to make strings from numbers. Finally, in *Most and Least*, we'll look at functions to find the maximum or minimum of a number of values.

4.2.1 Say It With Functions

Many of the Python processing operations that we might need are provided in the form of functions. Functions are one of the ways that Python lets us specify how to process some data. A *function*, in a mathematical sense, is a transformation from some input to an output. The mathematicians sometimes call this a *mapping*, because the function is a kind of map from the input value to the output value.

We looked at the `abs()` function in the previous section. It maps negative and positive numbers to their absolute magnitude, measured as a positive number. The `abs()` function maps -4 to 4, and 3.25 to 3.25.

We'll start out looking at two new mathematical functions, `pow()` and `round()`. Here are some examples of `abs()`, `pow()` and `round()`.

```
>>> abs(-18)
18
>>> pow(16, 3)
4096
>>> round(9.424)
9.0
>>> round(12.57)
13.0
```

A function is an expression, with the same syntactic role as any other expression, for example `2+3`. You can freely combine functions with other expressions to make more complex expressions. Additionally, the arguments to a function can also be expressions. Therefore, we can combine functions into more complex expressions pretty freely. This takes some getting used to, so we'll look at some examples.

```
1 >>> 3*abs(-18)
2 54
3 >>> pow(8*2, 3)*1.5
4 6144.0
5 >>> round(66.2/7)
6 9.0
7 >>> 8*round(abs(50.25)/4.0, 2)
8 100.48
```

1. In the first example, Python has to compute a product. To do this, it must first compute the absolute value of -18. Then it can multiply the absolute value by 3.

3. In the second example, Python has to compute a product of a `pow()` function and 1.5. To do this, it must first compute the product of 8 times 2 so that it can raise it to the 3rd power. This is then multiplied by 1.5. You can see that first Python evaluates any expressions that are arguments to the function, then it evaluates the function. Finally, it evaluates the overall expression in which the function occurs.

5. In the third example, Python computes the quotient of 66.2 and 7, and then rounds this to the nearest whole number.

7. Finally, the fourth example does a whopping calculation that involves several steps. Python has to find the absolute value of 50.25, divide this by 4, round that answer off to two positions and then multiply the result by 8. Whew!

4.2.2 `pow()` and `round()` Definitions

The function names provide a hint as to what they do. Here are the formal definitions, the kind of thing you'll see in the Python reference manuals.

`pow`(*x*, *y*[, *z*]) → number

Raises x to the y power.

If z is present, this is done modulo z: $x^y \bmod z$.

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

Rounds *number* to *ndigits* to the right of the decimal point.

The [and]'s are how we show that some parts of the syntax are optional. We'll summarize this in *Function Syntax Rules*.

Important: Function Syntax Rules

We'll show optional parameters to functions by surrounding them with [and]. We don't actually enter the [and]'s; they're just hints as to what the alternative forms of the function are.

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

Rounds *number* to *ndigits* to the right of the decimal point.

In the case of the `round()` function, the syntax summary shows us there are two different ways to use this function:

- We can use `round()` with the one required parameter, *number*. Example: `round( 3.14159 )`
- We can use `round()` with two parameters, *number* and *ndigits*. Example: `round( 3.14159, 2 )`

```
>>> round(2.459, 2)
2.46
>>> round(2.459)
2.0
```

Note that there is some visual ambiguity between using `[` and `]` in our Python programming and using [and] as markup for the grammar rules. Usually the context makes it clear.

4.2.3 Multiple Steps

We can use the `pow()` function for the same purpose as the `**` operator. Here's an example of using `pow()` instead of '`x**y`'.

```
>>> 2L**32
4294967296L
>>> pow(2L, 32)
4294967296L
```

Note that `pow(x,0.5)` is the square root of *x*. Also, the function `math.sqrt()` is the square root of x. The `pow()` function is one of the built-in functions, while the square root function is only available in the math library. We'll look at the `math` library in *The math Module – Trig and Logs*.

In the next example we'll get the square root of a number, and then square that value. It'll be a two-step calculation, so we can see each intermediate step.

```
>>> pow(2, 0.5)
1.4142135623730951
>>> _ ** 2
2.0000000000000004
```

The first question you should have is "what does that `_` mean?"

The _ is a Python short-cut. During interactive use, Python uses the name _ to mean the result it just printed. This saves us retyping things over and over. In the case above, the "previous result" was the value of `pow( 2, 0.5 )`. By definition, we can replace a _ with the entire previous expression to see what is really happening.

```
>>> pow(2, 0.5) ** 2
2.0000000000000004
```

Until we start writing scripts, this is a handy thing. When we start writing scripts, we won't be able to use the _, instead we'll use something that's a much more clear and precise.

4.2.4 Accuracy?

Let's go back to the previous example: we'll get the square root of a number, and then square that value.

```
>>> pow( 3, 0.5 )
1.7320508075688772
>>> _ ** 2
2.9999999999999996
>>> pow( 3, 0.5 ) ** 2
2.9999999999999996
```

Here's a big question: what is that ".9999999999999996" foolishness?

That's the left-overs from the conversion from our decimal notation to the computer's internal binary and back to human-friendly decimal notation. We talked about it briefly in *Floating-Point Numbers, Also Known As Scientific Notation*. If we know the order of magnitude of the result, we can use the round function to clean up this kind of small error. In this case, we know the answer is supposed to be a whole number, so we can round it off.

```
>>> pow( 3, 0.5 ) ** 2
2.9999999999999996
>>> round(_)
3.0
```

Tip: Debugging Function Expressions

If you look back at *Syntax Rule 6*, you'll note that the `()`s need to be complete. If you accidentally type something like `round(2.45` with no closing `)`, you'll see the following kind of exchange.

```
>>> round(2.45
...
...
... )
2.0
```

The `...` is Python's hint that the statement is incomplete. You'll need to finish the `()`s so that the statement is complete.

4.2.5 Another Round, Please

Above, we noted that the `round()` function had an optional argument. When something's optional, we can look at it as if there are two forms of the `round()` function: a one-argument version and a two-argument version.

- The one-argument `round()` function rounds a number to the nearest whole number.

- If you provide the optional second parameter, this is the number of decimal places to round to. If the number of decimal places is a positive number, this is decimal places to the right of the decimal point. If the number of decimal places is a negative number, this is the number of places to the *left* of the decimal point.

```
>>> round(678.456)
678.0
>>> round(678.456, 2)
678.46000000000004
>>> round(678.456, -1)
680.0
```

So, rounding off to -1 decimal places means the nearest 10. Rounding off to -2 decimal places is the nearest 100. Pretty handy for doing business reports where we have to round off to the nearest million.

4.2.6 Functions are Factories (really!)

How do we get Python to do specific conversions among our various numeric data types? When we mix whole numbers and floating-point scientific notation, Python automatically converts everything to floating-point. What if we want the floating-point number *truncated* down to a whole number instead?

Here's another example: what if we want the floating-point number transformed into a long integer instead of the built-in assumption that we want long integers turned into floating-point numbers? How do we control this coercion among numbers?

We'll look at a number of *factory functions* that do number conversion. Each function is a factory that creates a new number from an existing number. Eventually, we'll identify numerous varieties of factory functions.

These factory functions will also create numbers from string values. When we write programs that read their input from files, we'll see that files mostly have strings. Factory functions will be an important part of reading strings from files and creating numbers from those strings so that we can process them.

`float`(*x*) → number
: 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. You can use this function when doing division to prevent getting the simple integer quotient.

For example:

```
>>> float(22)/7
3.1428571428571428
>>> 22/7
3
>>> float("6.02E24")
6.0200000000000004e+24
```

`int`(*x*) → number
: Creates an integer equal to the string or number *x*. This will chop off all of the digits to the right of the decimal point in a floating-point number. If a string is given, it must be a valid decimal integer string.

```
>>> int('1234')
1234
>>> int(3.14159)
3
```

`long`(*x*) → number
: 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)`.

```
>>> long(2)**64
18446744073709551616L
>>> long(22.0/7.0)
3L
```

The first example shows the range of values possible with 64-bit integers, available on larger computers. This is a lot more than the paltry two billion available on a 32-bit computer.

Complex Numbers - Math wizards only. 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`(*real*[, *imag*]) → number
Creates a complex number with the real part of *real*; if the second parameter, *imag*, is given, this is the imaginary part of the complex number, otherwise the imaginary part is zero.

If this syntax synopsis with the [and] is confusing, you'll need to see *Function Syntax Rules*.

Examples:

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

Note that the second parameter, with the imaginary part of the number, is optional. This leads to two different ways to evaluate this function. In the example above, we used both variations.

Conversion from a complex number (effectively two-dimensional) to a one-dimensional integer or float is not directly possible. Typically, you'll use `abs()` to get the absolute value of the complex number. This is the geometric distance from the origin to the point in the complex number plane. The math is straight-forward, but beyond the scope of this introduction to Python.

```
>>> abs(3+4j)
5.0
```

4.2.7 Going the Other Way

If the `int()` function turns a string of digits into a proper number, can we do the opposite thing and turn an ordinary number into a string of digits?

The `str()` and `repr()` functions convert any Python object to a string. The `str()` string is typically more readable, where the `repr()` result can help us see what Python is doing under the hood. For most garden-variety numeric values, there is no difference. For the more complex data types, however, the results of `repr()` and `str()` can be different.

Here are some examples of converting floating-point expressions into strings of digits.

```
>>> str( 22/7.0 )
'3.14285714286'
>>> repr( 355/113. )
'3.1415929203539825'
```

Note that the results are surrounded by `'` marks. These apostrophes tell us that these aren't actually numbers; they're strings of digits.

What's the difference? Try this and see.

```
11+12
11+'12'
```

A string of digits may look numeric to you, but Python won't look inside a string to see if it "looks" like a number. If it is a string (with `"` or `'`), it is not a number, and Python won't attempt to do any math.

Here are the formal definitions of these two functions. These aren't very useful now, but we'll return to them time and again as we learn more about how Python works.

`str`(*object*) → string
: Creates a string representation of *object*.

`repr`(*object*) → string
: Creates a string representation of *object*, usually in Python syntax.

4.2.8 Most and Least

The `max()` and `min()` functions accept any number of values and return the largest or smallest of the values. These functions work with any type of data. Be careful when working with strings, because these functions use alphabetical order, which has some surprising consequences.

```
>>> max( 10, 11, 2 )
11
>>> min( 'asp', 'adder', 'python' )
'adder'
>>> max( '10', '11', '2' )
'2'
```

The last example (`max( '10', '11', '2' )`) shows the "alphabetical order of digits" problem. Superficially, this looks like three numbers (10, 11 and 2). But, they are quoted strings, and might as well be words. What would be result of '`max( 'ba', 'bb', 'c' )`' be? Anything surprising about that? The alphabetic order rules apply when we compare string values. If we want the numeric order rules, we have to supply numbers instead of strings.

Here are the formal definitions for these functions.

`max`(*sequence*) → object
: Returns the object with the largest value in *sequence*.

`min`(*sequence*) → object
: Returns the object with the smallest value in *sequence* .

4.2.9 Basic Function Exercises

1. **Numeric Expressions**.

   Write an expression to convert the mixed fraction 3 5/8 to a floating-point number.

2. **Truncation**.

   Evaluate `(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?

3. **Illegal Conversions**.

   Try illegal conversions like `int('A')`, `int(3+4j )`, `int( 2L**64 )`. Why are exceptions raised? Why can't a simple default value like zero be used instead?

4.3 Extra Functions: `math` and `random`

In *Meaningful Chunks and Modules*, we'll digress to look at the extension libraries. This is because the bulk of the math functions are in a separate *module* or library, called `math`. We'll look at parts of it in *The math Module – Trig and Logs*. We'll also look at the random number generators in *The random Module – Rolling the Dice*.

For those who will be using Python for financial and other fixed-point calculations, we'll look at fixed-point math, also. However, we'll defer this until *Fixed-Point Numbers : Doing High Finance with decimal* because it is a bit more advanced than using the built-in types of numbers.

4.3.1 Meaningful Chunks and Modules

Python's use of modules is a way to break the solution to a problem down into meaningful chunks. We hinted around about this in *Core Coolness*. There are dozens of standard Python modules that solve dozens of problems for us. We're not ready to look at modules in any depth, that comes later in *Modules : The unit of software packaging and assembly*. This section has just a couple of steps to start using modules so that you can make use of two very simple modules: `math` and `random`.

A Python module extends the Python language by adding new classes of objects, new functions and helpful constants. The **import** statement tells Python to fetch a module, adding that module to our working environment. For now, we'll use the simplest form: **import**.

```
import m
```

This statement will tell Python to locate the module named `m` and provide us with the definitions in that module. Only the name of the module, `m`, is added to the local names that we can use. Every name inside module `m` must be *qualified* by the module name. We do this by connecting the module name and the function name with a `.`. When we import module `math`, we get a cosine function that we refer to with "module name dot function name" notation: `math.cos()`.

This module qualification has a cost and a benefit. The cost is that you have to type the module name over and over again. The benefit is that your Python statements are explicit and harbor no assumptions. There are some alternatives to this. We'll cover it when we explore modules in depth.

Another important thing to remember is that you only *need* to import a module once to tell Python you will be using it. By once, we mean once each time you run the Python program. Each time you exit from the Python program (or turn your computer off, which exits all your programs), everything is forgotten. Next time you run the Python program, you'll need to provide the **import** statements to add the modules to Python for your current session.

An Interesting Example. For fun, try this:

```
import this
```

The `this` module is atypical: it doesn't introduce new object classes or function definitions. Instead, well, you see that it does something instead of extending Python by adding new definitions.

Even though the `this` module is atypical, you can still see what happens when you use an extra import. What happens when you try to import `this` a second time?

4.3.2 The `math` Module – Trig and Logs

The `math` module defines the common trigonometry and logarithmic functions. It has a few other functions that are handy, like square root. The `math` module is made available to your programs with:

```
import math
```

Since this statement only adds `math` to the names Python can recognize, you'll need to use the `math` prefix to identify the functions which are inside the `math` module.

Here are a couple of examples of some trigonometry. We're calculating the cosine of 45, 60 and 90 degrees. You can check these on your calculator. Or, if you're my age, you can use a slide rule to confirm that these are correct answers.

```
>>> import math
>>> math.cos( 45 * math.pi/180 )
0.70710678118654757
>>> math.cos( 60 * math.pi/180 )
0.50000000000000011
>>> math.cos( 90 * math.pi/180 )
6.123233995736766e-17
>>> round( math.cos( 90*math.pi/180 ), 3 )
0.0
```

Tip: Debugging Math Functions

The most common problem when using math functions is leaving off the `math` to qualify the various functions imported from this module. If you get a "`NameError: name 'cos' is not defined`" error message, it means you haven't included the `math` qualifier.

The next most common problem is forgetting to '`import math`' each time you run **IDLE** or **Python**. You'll get a "`NameError: name 'math' is not defined`" error if you forgot to import math.

The other common problem is failing to convert angles from degrees to radians.

The `math` module contains the following trigonometric functions. The trig functions all use radians to measure angles. If your problem uses degrees, you'll have to convert from degrees to radians. If your trigonometry is rusty, remember that 2π radians are 360 degrees.

Here are some trig function definitions. We don't provide all of them, because they're on the Python web site. See http://docs.python.org/library/math.html for the complete list of available functions.

Trigonometry Function Definitions

`math.asin(`*x*`)` → number
    Returns the arc sine of *x*.

`math.atan(`*x*`)` → number
    Returns the arc tangent of *x*.

`math.atan2(`*y*, *x*`)` → number
    Returns the arc tangent of *y* / *x*.

`math.cos(`*x*`)` → number
    Returns the cosine of *x* radians.

`math.sin(`*x*`)` → number
    Returns the sine of *x* radians.

`math.tan(`*x*`)` → number
    Returns the tangent of *x* radians.

Additionally, the following constants are also provided.

pi The value of π, 3.1415926535897931

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

Here's an example of using some of these more advanced math functions. Here is a trig identity for the cosine of 39 degrees. We use `39*math.pi/180` to convert from degrees to radians. We also use the square root function (`sqrt()`).

```
>>> import math
>>> math.sqrt( 1-math.sin(39*math.pi/180)**2 )
0.77714596145697101
>>> math.cos( 39*math.pi/180 )
0.7771459614569709
```

Here are some more of these common trigonometric functions, including logarithms, anti-logarithms and square root.

```
>>> math.exp( math.log(10.0) / 2 )
3.1622776601683795
>>> math.exp( math.log(10.0) / 2 )
3.1622776601683795
>>> math.sqrt( 10.0 )
3.1622776601683795
```

Logarithm Function Definitions.

`math.exp(`x`)` → number
: Returns `math.e**x` (e^x), inverse of `log()`.

`math.hypo(`x`)` → number
: Returns the Euclidean distance, `sqrt( x*x + y*y )` ($\sqrt{x^2 + y^2}$), length of the hypotenuse of a right triangle with height of y and length of x.

`math.log(`x`)` → number
: Returns the natural logarithm (base e) of x ($\ln x$), inverse of `exp()`.

`math.log10(`x`)` → number
: Returns the logarithm (base 10) of x ($\log_{10} x$), inverse of 10^x.

`math.pow(`x, y`)` → number
: Returns '`x**y`' (x^y).

`math.sqrt(`x`)` → number
: Returns the square root of x ($\sqrt{x}$). 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()` which creates imaginary numbers as needed.

4.3.3 More `math` Functions

The following batch of functions supplement the basic `round()` function with more sophisticated computations on floating-point numbers. You can probably guess from the names what ceiling and floor mean.

```
>>> math.ceil(2.1)
3.0
>>> math.floor(2.999)
2.0
```

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

Other Floating-Point Function Definitions.

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

`math.fabs`(x) → number
Returns the absolute value of the x as a floating-point number.

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

4.3.4 Misuse and Abuse

Some of the math functions only work for a limited domain of values. Specifically, square root is only defined for non-negative numbers and logarithms are only defined for positive numbers. What does Python do when we violate these rules?

Try the following expressions:

```
math.sqrt(-1)
math.log(-1)
math.log(0)
```

You'll see one of two kinds of results. The details vary among the operating systems.

- You'll see a result of `nan`. This is a special code that means Not a Number.
- You'll see an exception, like `ValueError` or `OverflowError`. An exception will display a bunch of debugging information that ends with the exception name and a short explanation.

Both results amount to the same thing: the result cannot be computed.

4.3.5 The `random` Module – Rolling the Dice

The `random` module defines functions that simulate random events. This includes coin tosses, die rolls and the spins of a Roulette wheel. The `random` module is made available to your program with:

```
import random
```

Since this statement only adds `random` to the names Python can recognize, you'll need to use the `random` prefix on each of the functions in this section.

The `randrange()` is a particularly flexible way to generate a random number in a given range. Here's an example of some of the alternatives. Since the answers are random, your answers may be different from these example answers. This shows a few of many techniques available to generate random data samples in particular ranges.

```
>>> import random
>>> random.randrange(6)
5
>>> random.randrange(1,7)
6
>>> random.randrange(2,37,2)
6
>>> random.randrange(1,36,2)
13
```

1. We're asking for a random number, n, such that $0 \leq n < 6$. The number will be between 0 and 5, inclusive.

2. We're asking for a random number, n, such that $1 \leq n < 7$. The number will be between 1 and 6, inclusive.
3. We're asking for a random even number, n, such that $2 \leq n < 37$. The range function is defined by start, stop and step values. When the step is 2, then the values used are $2, 4, 6, \ldots, 36$.
4. We're asking for a random odd number, n, such that $1 \leq n < 36$. The number will be between 1 and 35, inclusive. Here, we start from 1 with a step of 2; the values used are $1, 3, 5, \ldots, 35$.

The `random` module contains the following functions for working with simple distributions of random numbers. There are several more sophisticated distributions available for more complex kinds of simulations. Casino games only require these functions.

4.3.6 Random Function Definitions

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

`random.random()` → number
: Creates a random floating-point number, r , such that $0 \leq r < 1.0$. Note that `random()` doesn't require any arguments, but does require the empty `()`s to alert Python that it is really the name of a function. We use it like this: `random.random()`.

`random.randrange(`[*start*], *stop*[, *step*]`)`
: Chooses a random element from `range( start, stop , step )`. We'll revisit this in *Built-in Functions for Lists*. For now, we'll stick with the following examples:

  `randrange(6)` returns a number, n, such that $0 \leq n < 6$.

  `randrange(1,7)` returns a number, n, such that $1 \leq n < 7$.

  `randrange(10,100,5)` returns a number, n, between 10 and 95 incremented by 5's, $10 \leq 5k < 100$.

`random.uniform(`*a*, *b*`)` → number
: Creates a random floating-point number, r , such that $a \leq r < b$.

4.3.7 Extra Function 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. **Tossing a Coin**.

   There are several ways the `random` module can be used to simulate tossing a coin. Write expressions that use `choice()` and `randrange()`.

   Additionally, using something like '`int(random()*X)`', for some value of X, you can simulate a coin toss. If we use zero for heads and 1 for tails, what is an appropriate value for X? What if we use `round()` instead of `int()`? What about `ceil()` and `floor()`?

   Can you also use `uniform()` to simulate a coin toss? Do you need to also use `int()`?

4.3.8 Function FAQ's

Why do functions have an usual syntax? Python functions all consistently look a little like the mathematical `sin(x)`. Mathematicians have evolved a number of other forms for functions. Python's syntax is, at least, consistent. Rather than ask why Python looks the way it does, we prefer to ask why the mathematicians have so many different forms for functions.

Why is `math` (or `random` or `decimal`) a separate module? There are two reasons for keeping `math` (or `random` or `decimal`) in separate modules.

- Not everyone needs `math` so why include it needlessly?
- There will always be new implementations of basic numeric algorithms with different trade-offs for range, precision, speed and amount of storage. Rather than pick one, Python leaves it to you to select among the various alternatives and pick that one that best meets your needs.

Everything I do involves `math` (or `random` or `decimal`), why do I have to import it in every single script? There is one very important reason for importing the module in every single script. It keeps you (and Python) honest: nothing is assumed. You said you needed `math`, making it clear to everyone else who reads your script. While you know that all your programs are mathematical, almost no one else knows this. The import statement sets the assumptions on a script-by-script basis, making each a successful, stand-alone program, without requiring any insider information or background to understand it.

4.4 Special Ops : Binary Data and Operators

This chapter is optional. If you expect to be working with individual bits, these operators are very helpful. Otherwise, if you don't expect to be working with anything other than plain-old decimal numbers, you can skip this chapter.

While we write numbers using decimal digits, in base 10, computers don't really work that way internally. We touched on the computer's view in *Octal and Hexadecimal – Counting by 8's or 16's*. Internally, the computer works in binary, base 2, which makes the circuitry very simple and very fast. One of the benefits of using Python is that we don't need to spend much time on the internals, so this chapter is optional.

We'll take a close look at data in *Bits and Bytes*, this will provide some justification for having base 8 and base 16 numbers. We'll add some functions to see base 8 and base 16 in *Different Bases and Representations*. Then we'll look at the operators for working with individual bits in *Operators for Bit Manipulation*.

4.4.1 Bits and Bytes

The special operators that we're going to cover in this chapter work on individual bits. First, we'll have to look at what this really means. Then we can look at what the operators do to those things called bits.

A bit is a "binary digit" . The concept of bit closely parallels the concept of decimal digit with one important difference. There are only two binary digits (0 and 1), but there are 10 decimal digits (0 through 9).

Decimal Numbers. Our decimal numbers are a sequence of digits using base 10. Each decimal digit's place value is a power of 10. We have the 1,000's place, the 100's place, the 10's place and the 1's place. A number like 2185 is $2 \times 1000 + 1 \times 100 + 8 \times 10 + 5$.

Binary Numbers. Binary numbers are a sequence of binary digits using base 2. Each bit's place value in the number is a power of 2. We have the 256's place, the 128's place, the 64's place, the 32's place, the 16's place, the 8's, 4's, 2's and the 1's place. We can't directly write binary numbers in Python. We'll show them as a series of bits, like this `1-0-0-0-1-0-0-0-1-0-0-1`. This starts with a 1 in the 2048's place, a 1 in the 128's place, plus a 1 in the 8's place, plus a 1, which is 2185.

Octal Numbers. Octal numbers use base 8. In Python, we begin octal numbers with a leading zero. Each octal digit's place value is a power of 8. We have the 512's place, the 64's place, the 8's place and the 1's place. A number like `04211` is $4 \times 512 + 2 \times 64 + 1 \times 8 + 1$. This has a value of 2185.

Each group of three bits forms an octal digit. This saves us from writing out all those bits in detail. Instead, we can summarize them.

```
Binary: 1-0-0  0-1-0  0-0-1  0-0-1
Octal:    4      2      1      1
```

Hexadecimal Numbers. Hexadecimal numbers use base 16. In Python, we begin hexadecimal numbers with a leading `0x`. Since we only have 10 digits, and we need 16 digits, we'll borrow the letters `a`, `b`, `c`, `d`, `e` and `f` to be the extra digits. Each hexadecimal digit's place value is a power of 16. We have the 4096's place, the 256's place, the 16's place and the 1's place. A number like `0x8a9` is $8 \times 256 + 10 \times 16 + 9$, which has a value of 2217.

Each group of four bits forms a hexadecimal digit. This saves us from writing out all those bits in detail. Instead, we can summarize them.

```
Binary:      1-0-0-0  1-0-1-0  1-0-0-1
Hexadecimal:    8        a        9
```

Bytes. A byte is 8 bits. That means that a byte contains bits with place values of 128, 64, 32, 16, 8, 4, 2, 1. If we set all of these bits to 1, we get a value of 255. A byte has 256 distinct values. Computer memory is addressed at the individual byte level, that's why you buy memory in units measured in megabytes or gigabytes.

In addition to small numbers, a single byte can store a single character encoded in ASCII. It takes as many as four bytes to store characters encoded with Unicode.

An integer has 4 bytes, which is 32 bits. In looking at the special operators, we'll look at them using integer values. Python can work with individual bytes, but it does this by unpacking a byte's value and saving it in a full-sized integer.

4.4.2 Different Bases and Representations

In *Octal and Hexadecimal – Counting by 8's or 16's* we saw that Python will accept base 8 or base 16 (octal or hexadecimal) numbers. We begin octal numbers with `0`, and use digits `0` though `7`. We begin a hexadecimal number with `0x` and use digits `0` through `9` and `a` through `f`.

Python normally answers us in decimal. How can we ask Python to answer in octal or hexadecimal instead?

The `hex()` function converts its argument to a hexadecimal (base 16) string. A string is used because additional digits are needed beyond 0 through 9; a-f are pressed into service. A leading `0x` is placed on the string as a reminder that this is hexadecimal. Here are some examples:

```
>>> hex(684)
'0x2ac'
>>> hex(1023)
'0x3ff'
>>> 0xffcc33
16763955
>>> hex(_)
'0xffcc33'
```

Note that the result of the `hex()` function is technically a string, An ordinary number would be presented as a decimal value, and couldn't contain the extra hexadecimal digits. That's why there are apostrophes in our output.

The `oct()` function converts its argument to an octal (base 8) string. A leading `0` is placed on the string as a reminder that this is octal not decimal. Here are some examples:

```
>>> oct(512)
'01000'
>>> oct(509)
'0775'
```

Here are the formal definitions.

hex(*number*) → string
    Creates a hexadecimal string representation of *number*.

oct(*number*) → string
    Creates an octal string representation of *number*.

More Hexadecimal and Octal tools. The `hex()` and `oct()` functions make a number into a specially-formatted string. The `hex()` function creates a string using the hexadecimal digit characters. The `oct()` uses the octal digits. There is a function which goes the other way: it can convert strings of digit characters into proper numbers so we can do arithmetic.

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.

In case you don't recall how this works, remember that in the number 1985, we're implicitly computing math:*1times 10^3 + 9times 10^2 + 8times 10^1 + 5times 10^0* (`1*10**3 + 9*10**2 + 8*10 + 5`). Each digit has a place value that is a power of some number. That number is the "base" for the numbers we're writing. Python assumes that a string of digits is decimal. A string of digits which begins with 0 is in base 8. A string of digits which begins with 0x is in base 16.

Here are some examples of converting strings that are in other bases to good old base 10 numbers.

```
>>> int('010101',2)
21
>>> int('321',4)
57
>>> int('2ac',16)
684
```

In base 2, the place values are 32, 16, 8, 4, 2, 1. The string `'010101'` is evaluated as $1 \times 16 + 1 \times 4 + 1 = 21$.

In base 4, the place values are 16, 4 and 1. The string `'321'` is evaluated as $3 \times 16 + 2 \times 4 + 1 = 57$.

Recall from *Octal and Hexadecimal – Counting by 8's or 16's* that we have to press additional symbols into service to represent base 16 numbers. We use the letters a-f for the digits after 9. The place values are 256, 16, 1; the string `2ac` is evaluated as $2 \times 256 + 10 \times 16 + 12 = 684$.

While it seems so small, it's really important that numbers in another base are written using strings. To Python, `123` is a decimal number. `'123'` is a string, and could mean anything. When you say `int('123',4)`, you're telling Python that the string `'123'` should be interpreted as base 4 number (which maps to 27 in base 10 notation.) On the other hand, when you say `int('123')`, you're telling Python that the string `'123'` should be interpreted as a base 10 number, which is 123.

int(*object*[, *base*]) → number
    Generates an integer from the value *object*. If *object* is a string, and *base* is supplied, *object* must be proper number in the given base. If *base* is omitted, and *object* is a string, it must be decimal.

4.4.3 Operators for Bit Manipulation

We've already seen the usual math operators: `+`, `-`, `*`, `/`, `%`, `**`; as well as a large collection of mathematical functions. While these do a lot, there are still more operators available to us. In this section, we'll look at operators that directly manipulate the binary representation of numbers. The inhabitants of Binome (see *Binary Codes* are more comfortable with these operators than we are.

We won't wait for the FAQ's to explain why we even have these operators. These operators exist to provide us with a view of the real underlying processor. Consequently, they are used for some rather specialized purposes. We present them because they can help newbies get a better grip on what a computer really is.

In this section, we'll see a lot of hexadecimal and octal numbers. This is because base 16 and base 8 are also nifty short-hand notation for lengthy base 2 numbers. We'll look at hexadecimal and octal numbers first. Then we'll look at the bit-manipulation operators.

There are some other operators available, but, strictly speaking, they're not arithmetic operators, they're logic operations. We'll return to them in *Processing Only When Necessary : The if Statement*.

Precedence. We know one basic precedence rule that applies to multiplication and addition: Python does multiplication first, and addition later. The second rule is that `()`s group things, which can change the precedence rules. `2*3+4` is 10, but `2*(3+4)` is 14.

Where do these special operators fit? Are they more important than multiplication? Less important than addition? There isn't a simple rule. Consequently, you'll often need to use `()`s to make sure things work out the way you want.

The ~ operator

The unary `~` operator flops all the bits in a plain or long integer. 1's become 0's and 0's become 1's. Note that this will have unexpected consequences when the bits are interpreted as a decimal number.

```
>>> ~0x12345678
-305419897
>>> hex(~0x12345678)
'-0x12345679'
```

What makes this murky is the way Python interprets the number has having a sign. The computer hardware uses a very clever trick to handle signed numbers. First, let's visualize the unsigned, binary number line, it has 4 billion positions. At the left we have all bits set to zero. In the middle we have a value where the 2-billionth place is 1 and all other values are zero. At the right we have all bits set to one.

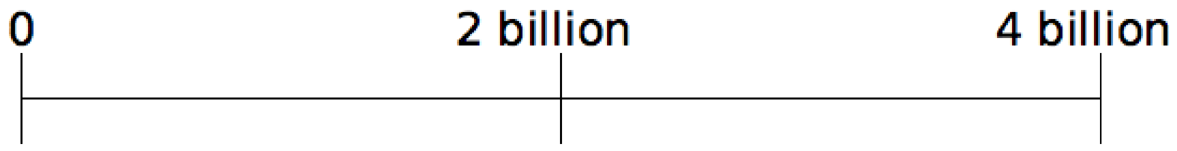

Figure 4.1: The Basic Number Line

Now, let's redraw the number line with positive and negative signs. Above the line, we put the signed values that Python will show us. Below the line, we put the internal codes used. The positive numbers are what we expected: `0x00000000` is the full 32-bit value for zero, 1 is `0x00000001`; no surprise there. Below the 2 billion, we put `0x7fffffff`. That's the full 32-bit value for positive 2 billion (try it in Python and see.) Below the -2 billion, we put `0x80000000`, the full 32-bit value for -2 billion. Below the -1, we put `0xffffffff`.

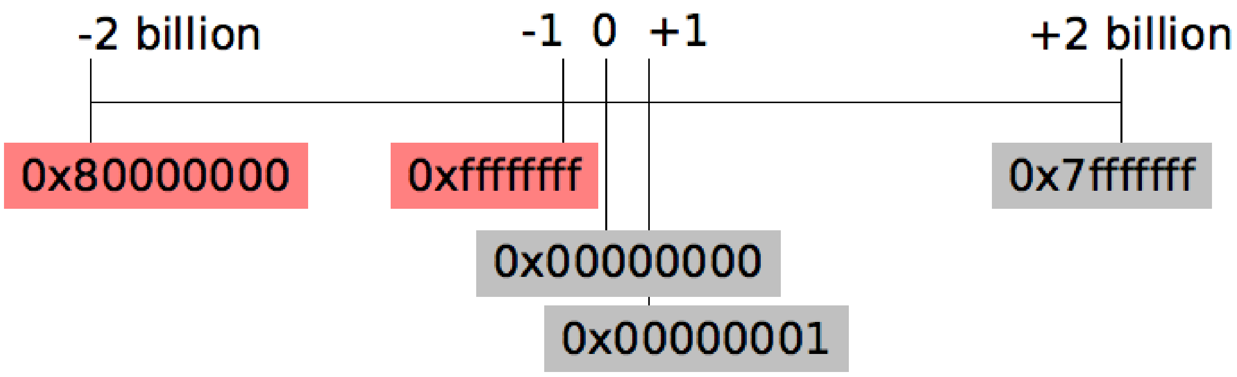

Figure 4.2: Encoding Signs On The Number Line

This works very nicely. Let's start with -2 (`0xfffffffe`). We add 1 to this and get -1 (`0xffffffff`), just what we want. We add 1 to that and get `0x00000000`, and we have to carry the 1 into the next place value. However, there is no next place value, the 1 is discarded, and we have a good-old zero.

This technique is called *2's complement* . Consequently, the `~` operation is mathematically equivalent to adding 1 and switching the number's sign between positive and negative.

This operator has the same very high precedence as the ordinary negation operation, `-` . Try the following to see what happens. First, what's the value of `-5+4`? Now, add the two possible `()`s and see which result is the same: `(-5)+4` and `-(5+4)`. The one the produces the same result as `-5+4` reveals which way Python performs the operations.

Here are some examples of special ops mixed with ordinary operations.

```
>>> -5+4
-1
>>> -(5+4)
-9
>>> (-5)+4
-1
```

The `&` operator

The binary `&` operator returns 1-bits everywhere that the two input bits are both 1. Each result bit depends on one input bit *and* the other input bit both being 1. The following example shows all four combinations of bits that work with the `&` operator.

```
>>> 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.

```
>>> 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
```

This is a very low-priority operator, and almost always needs parentheses when used in an expression with other operators. Here are some examples that show you how `&` and `+` combine.

```
>>> 3&2+3
1
>>> 3&(2+3)
1
>>> (3&2)+3
5
```

The ^ operator

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* operation to distinguish it from the *inclusive or* . Some people write "and/or" to emphasize the inclusive sense of or. They write "either-or" to emphasize the exclusive sense of or.

```
>>> 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.

This is a very low-priority operator, be sure to parenthesize your expression correctly.

The | operator

The binary | operator returns a 1-bit if either of the two inputs is 1. This is sometimes called the *inclusive or* to distinguish it from the exclusive or' operator.

```
>>> 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.

This is a very low-priority operator, and almost always needs parentheses when used in an expression with other operators. When we combine `&`s and `|`s we have to be sure we've grouped them properly. Here's the kind of thing that you'll sometimes see in programs that build up specific patterns of bits.

```
>>> 3&0x1f | 0x80 | 0x100
387
>>> hex(_)
'0x183'
```

Let's look at this in a little bit of detail. Our first expression has two or operations, they're the lowest priority operators. The first or operation has `3&0x1f` or `0x80`. So, Python does the following steps to evaluate this expression.

1. Calculate the *and* of `3` and `0x1f` . This is 3 (try it and see.) You can work it out by hand if you know that 3 is `0-0-0-1-1` in binary and `0x1f` is `1-1-1-1-1`.
2. Calculate the *or* of the previous result (3) and `0x80`.
3. Calculate the *or* of the previous result (`0x83`) and `:0x100`. This has the decimal value of 387.
4. Calculate the hex string for the previous result, using the `_` short-hand for the previously printed result. This shows that the hex value is `0x183`, what we expected.

The << Operator

The '`<<`' is the left-shift operator. The left argument is the bit pattern to be shifted, the right argument is the number of bits. This is mathematically equivalent to multiplying by a power of two, but much, much faster. Shifting left 3 positions, for example, multiplies the number by 8.

This operator is higher priority than `&` , `^` and `|`. Be sure to use parenthesis appropriately.

```
>>> 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 >> Operator

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. This is mathematically equivalent to dividing by a power of two, but much, much faster. Shifting right 4 positions, for example, divides the number by 16.

This operator is higher priority than `&`, `^` and `|` . Be sure to use parenthesis appropriately.

```
>>> 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.

Tip: Debugging Special Operators

The most common problems with the bit-fiddling operators is confusion about the relative priority of the operations. For conventional arithmetic operators, `**` is the highest priority, `*` and `/` are lower priority and `+` and `-` are the lowest priority. However, among `&`, `^` and `|`, `<<` and `>>` it isn't obvious what the priorities are or should be.

When in doubt, add parenthesis to force the order you want.

4.4.4 Special Ops Exercises

1. **Bit Masking**.

   One common color-coding scheme uses three distinct values for the level of red, green and blue that make up each picture element (pixel) in an image. If we allow 256 different levels of red, green and blue, we can mash a single pixel in 24 bits. We can then cram 4 pixels into 3 plain-integer values. How do we unwind this packed data?

We'll have to use our bit-fiddling operators to unwind this compressed data into a form we can process. First, we'll look at getting the red, green and blue values out of a single plain integer.

We can code 256 levels in 8 bits, which is two hexadecimal digits. This gives us a red, green and blue levels from `0x00` to `0xFF` (0 to 255 decimal). We can string the red, green and blue together to make a larger composite number like `0x0c00a2` for a very bluish purple.

What is `0x0c00a2 & 0xff`? Is this the blue value of `0xa2`? Does it help to do `hex( 0x0c00a2 & 0xff)`?

What is `(0x0c00a2 & 0xff00) >> 8`? `hex( (0x0c00a2 & 0xff00) >> 8 )`?

What is `(0x0c00a2 & 0xff0000) >> 16`? `hex( (0x0c00a2 & 0xff0000) >> 16 )`?

2. **Division**.

   How can we break a number down into different digits?

   What is `1956 / 1000`? `1956 % 1000`?

   What is `956 / 100`? `956 % 100`?

   What is `56 / 10`? `56 % 10`?

   What happens if we do this procedure with `1956.`, `956.` and `56.` instead of `1956`, `956` and `56`? Can we use the `//` operator to make this work out correctly?

4.4.5 Special Ops FAQ's

Why is there bit-fiddling? Some processing requires manipulating individual bits. In particular, sound and image data is often coded up in a way that is best processed using these bit-fiddling operations. Additionally, the various compression schemes like MP3 and JPEG use considerable manipulation of individual bits of data.

4.5 More Advanced Expression Topics

Now that we have the basics, we can add details. The first thing we need to look at are the two division operators in *The Two Specialized Division Operators: / and //*.

Currency amounts fall into the cracks between integers (no decimal places) and floating-point numbers (variable number of decimal places.) Currency requires a fixed digits after the decimal point. We'll talk about one way to handle fixed point math in *Currency Calculations: Fixed Point Math*.

This is some additional background in Python and programming. We'll talk a little about what it means to execute the statements in a program and evaluate expressions in *Execution – Two Points of View*. We'll provide some style notes in *Expression Style Notes*.

4.5.1 The Two Specialized Division Operators: / and //

Python 2 harbors an assumption that – it turns out – is a bad idea. Generally, assumptions are a bad idea, and one thing that Python emphasizes is the idea that "explicit is better then implicit". Python 3 remove the assumption permanently.

For users of Python 2, however, we have a balancing act between the legacy assumptions and the new explicitness.

While most features of Python correspond with common expectations from mathematics and other programming languages, the division operator, `/`, has a complexity. This is due to the lack of a common expectation for what division should mean.

- Sometimes we expect division to create precise answers, usually the floating-point equivalents of fractions. `355/113` *should* be 3.1415929203539825.
- Other times, we want a rounded-down integer result. `355/113` *should* be 3.

A basic tenet of Python is that the data determine the result of an operation. Since the two values in samp:*355/113* are integers, the result is an integer.

The are numerous circumstances under which we'd prefer an exact answer, however.

There's no *best* answer. Sometimes we mean one and other times we mean the other. We need to explicitly name the division operation we intent.

To see the effect of this assumption, try the following to see what Python does.

```
355/113
355.0/113
355/113.0
55.0/113.0
```

The Unexpected Integer. Here are two examples of the classical definition of division. We've used the formula for converting 18 ° Celsius to Fahrenheit. The first version uses integers, and gets an integer result. The second uses floating-point numbers, which means the result is floating-point.

```
>>> 18*9/5+32
64
>>> 18.0*9.0/5.0 + 32.0
64.400000000000006
```

In the first example, we got an inaccurate answer from a formula that we are sure is correct. We expected a correct answer of 64.4, but got 64.

In Python 2, when a formula has a `/` operator, the inaccuracy will stem from the use of integers where floating-point numbers were more appropriate. (This can also occur using integers where complex numbers were implicitly expected.)

If we use floating-point numbers, we get a value of 64.4, which was correct. Try this and see.

```
18.0*9.0/5.0 + 32.0
```

Noisy Solutions. The problem we have is reconciling the basic rule of Python (data determines the result) and the two conflicting meanings for division. We have a couple of choices for the solution.

We can solve this by using explicit conversions like `float()` or `int()`. However, we'd like Python be a simple and sparse language, without a dense clutter of conversions to cover the rare case of an unexpected type of data. So this isn't ideal.

This is an over-the-top, worst-case example of using explicit conercion.

```
>>> float(18)*float(9)/float(5) + float(32)
64.400000000000006
```

Explicit is Better. As part of being explicit, Python offers us two division operators.

- For precise fractional results, we'll use `/`.
- When we want division to simply compute the quotient, Python has a second division operator, `//`. This produces rounded-down integer answers, even if both numbers happen to be floating-point.

Old vs. New Division. While usiung Python 2, we need to specify which meaning of `/` should apply. Do we mean the original Python 2 definition (data type determines results)? Or do we mean the newer meaning of `/` (exact results)?

Python 2 gives us two tools to specify the meaning of the `/` operator: a statement that can be placed in a program, as well as a command-line option that can be used when starting the Python program.

Program Statements to Control `/`. To ease the transition from older to newer language features, the statement `from __future__ import division` will changes the definition of the `/` operator from Python 2 (depends on the arguments) to Python 3 (always produces floating-point).

Note that `__future__` module name has two underscores before and after `future`. Also, note that this must be the first executable statement in a script.

Here's the classic division:

```
>>> 18*9/5+32
64
```

Here's the new division

```
1  >>> from __future__ import division
2  >>> 18*9/5+32
3  64.400000000000006
4  >>> 18*9//5+32
5  64
```

1. We set the future definition of the `/` operator.
2. This line shows the new use of the `/` operator to produce precise floating-point results, even if both arguments are integers.
4. This line shows the `//` operator, which always produces rounded-down results.

The **from __future__** statement states that your script uses the new-style floating-point division operator. This allows you to start writing programs with Python 2 that will work correctly with all future versions.

By Python 3, this **import** statement will no longer be necessary, and will have to be removed from the few modules that used them.

Tip: Debugging the **from __future__** statement

There are two common spelling mistakes: omitting the double underscore from before and after `__future__`, and misspelling `division`.

- If you get `ImportError: No module named _future_`, you misspelled `__future__`.
- If you get `SyntaxError: future feature :replaceable:`divinizing` is not defined‘, you misspelled `division`.

Command Line Options to Control `/`. Another tool to ease the transition is an option that we can use as part of the **python** command that starts the Python interpreter. This option can force a particular interpretation of the `/` operator or warn about incorrect use of the `/` operator.

The Python command-line option of *`-Q`* controls the meaning of the `/` operator.

- If you run Python with *`-Qold`*, you get the classical Python 2 definition, where the `/` operator's result depends on the arguments.
- If you run Python with *`-Qnew`*, you get the new Python 3 definition, where the `/` operator's result will be a precise floating-point fraction.

Here's how it looks when we start Python with the *-Qold* option.

```
1  MacBook-5:~ slott$ python -Qold
2  Python 2.5.4 (r254:67917, Dec 23 2008, 14:57:27)
3  [GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>> 355/113
6  3
7  >>> 355./113.
8  3.1415929203539825
9  >>> 355.//113.
10 3.0
```

1. Here is the **python** command with the *-Qold* option. This will set Python to do classical interpretation of the `/` operator.
5. When we do old-style `/` division with integers, we get an integer result.
7. When we do old-style `/` division with floating-point numbers, we get the precise floating-point result.
9. When we do `//` division with floating-point numbers, we get the rounded-down result.

Here's how it looks when we start Python with the *-Qnew* option.

```
1  MacBook-5:~ slott$ python -Qnew
2  Python 2.5.4 (r254:67917, Dec 23 2008, 14:57:27)
3  [GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>> 355/113
6  3.1415929203539825
7  >>> 355./113.
8  3.1415929203539825
9  >>> 355.//113.
10 3.0
```

1. Here is the **python** command with the *-Qnew* option. This will set Python to do the new interpretation of the `/` operator.
5. When we do new-style `/` division with integers, we get the precise floating-point result.
7. When we do new-style `/` division with floating-point numbers, we get the precise floating-point result.
9. When we do `//` division with floating-point numbers, we get the rounded-down result.

Why All The Options?. There are two cases to consider here.

If you have an old program, you may need use *-Qold* to force an old module or program to work the way it used to.

If you want to be sure you're ready for Python 3, you can use the *-Qnew* to be sure that you always have the "exact quotient" version of `/` instead of the classical version.

Important: Debugging The -Q Option

If you misspell the *-Q* option you'll see errors like the following. If so, check your spelling carefully.

```
MacBook-5:~ slott$ python -Qwhat
-Q option should be `-Qold', `-Qwarn', `-Qwarnall', or `-Qnew' only
usage: Python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Try `python -h' for more information.
```

If you get a message that includes `Unknown option: -q`, you used a lower-case `q` instead of an upper-case `Q`.

4.5.2 Currency Calculations: Fixed Point Math

In *Floating-Point Numbers, Also Known As Scientific Notation*, we saw that floating-point numbers are for scientific and engineering use and don't work well for financial purposes. US dollar calculations, for example, are often done in dollars and cents, with two digits after the decimal point.

If we try to use floating-point numbers for currency values, we have problems. Specifically, the slight discrepancy between binary-coded floating-point numbers and decimal-oriented dollars and cents become a serious problem. Try this simple experiment.

```
>>> 2.35
2.3500000000000001
```

There's a classic trick that can be used to solve this problem: use *scaled numbers*. When doing dollars and cents math, you can scale everything by 100, and do the math in pennies. When you print the final results, you can scale the final result into dollars with pennies to the right of the decimal point. This section will provide you some pointers on doing this kind of numeric programming.

Later, in *Fixed-Point Numbers : Doing High Finance with decimal* we'll look at the `decimal` module, which does this in a more sophisticated and flexible way.

Scaled Numbers. When we use scaled numbers, it means that the proper value is represented as the scaled value and a precision factor. For example, if we are doing our work in pennies, the value of $12.99 is represented as a scaled value of 1299 with a precision of 2 digits. The precision factor can be thought of as a power of 10. In our case of 12.99, our precision is 2. We can multiply by $10^{-\text{precision}}$ to convert our scaled number into a floating-point approximation.

We have three cases to think about when doing fixed-point math using scaled integers: addition (and subtraction), multiplication and division. Addition and subtraction don't change the precision. Multiplication increases the precision of the result and division reduces the precision. So, we'll need to look at each case carefully.

Addition and Subtraction. If our two numbers have the same precision, we can simply add or subtract normally. This is why we suggest doing everything in pennies: the precisions are always 2, which always match. If our two numbers have different precisions, we need to shift the smaller precision number. We do this by multiplying by an appropriate power of 10.

What is $12.00 + $5.99? Assume we have 12 (the precision is dollars) and 599 (the precision is pennies). We add them like this: `12*100 + 599`. We applied the penny precision factor of 100 to transform dollars into pennies.

Multiplication. When we multiply two numbers, the result has the sum of the two precisions. If we multiply two amounts in pennies (2 digits to the right of the decimal point), the result has 4 digits of precision. We have to be careful when doing this kind of math to determine the rounding rules, and correctly scale the result.

What is 7.5% of $135.99? Assume we have 13599 (the precision is pennies, 2 digits after the decimal point) and 75 (the precision is 10th of a percent, three digits to the right of the decimal point). When we multiply, our result will have precision of 5 digits to the right of the decimal point. The result (1019925) represents $10.19925. We need to both round and shift this back to have a precision of 2 digits to the right of the decimal point.

We can both round and scale with an expression like the following. The `*.001` resets the scale from 5 digits of precision to 2 digits of precision.

```
>>> int(round(13599L*75,-3)*.001)
1020
```

This means 7.5% of $135.99 is $10.20.

Division. When we divide two numbers, the result's precision is the difference between the numerator's and denominator's precision. If we divide two amounts in pennies (2 digits of precision), the result has zero digits of precision. Indeed, the result is actually a ratio between penny amounts, and isn't actually in pennies. We have to be careful when doing this kind of math to determine the rounding rules, and correctly scale the answer.

Generally, if we want 2 digits of precision in our result, we need to be sure the numerator's precision is at least 2 digits *more* than the denominator's precision. This means scaling the numerator first, then doing the division. If the numerator has too much precision to begin with, we'll have to round and then scale the result after division.

Say we have a bill of $45,276 for 416.15 hours of labor. What is the exact dollars per hour to the penny? Our hours have a precision of two digits, 41615, with a precision factor of 100. We need our dollars to start with five digits of precision because we start with two digits, we'll lose two when dividing by hours, and we want one more digit so we can round properly. We'll represent the dollars as 4527600000 with a precision factor of 100000. The division gives us 108797, with a precision factor of 1000. This can be rounded correctly and divided by 10 to get the value to the penny, properly rounded, of 10880, which means $108.80.

```
>>> int(round(45276L*100000/41615,-1)*.1)
10880
```

This meas that the labor rate was $108.80 per hour.

The Bigger Picture. Whew! It looks like the special cases of adding (and subtracting), multiplying and dividing are really complex.

There's a trick to this, and the trick is to begin with the goal in mind and work backwards to what data we need to satisfy our goal. For adding and subtracting, our goal precision can't be different from our input precision. When multiplying and dividing, we work backwards: we write down our goal precision, we write down the precision from our calculation, and we work out rounding and scaling operations to get from our calculation to our goal.

It turns out that this trick is essential to programming. We'll return to it time and again.

4.5.3 Execution – Two Points of View

What does it mean when a computer "does" a specific task? This is the essential, inner mystery of programming. There are two overall approaches to specifying what should happen inside the computer. Most modern languages are a mixture of both approaches. These two approaches are sometimes called *functional* and *procedural*, or *applicative* and *imperative*. Since the programming language business is very competitive, any term we chose is loaded with meaning and many hairs get split in these conversations. We'll look at both the applicative and imperative views of Python, because Python uses each approach where it is appropriate.

Applicative Approach. Functional or applicative programming is characterized by a style that looks a lot like conventional mathematics. Functions are *applied* to argument values using an *evaluate-apply cycle*.

We can see the applicative approach when we look, for example, at $f = 32 + \frac{9c}{5}$. We start wth "evaluate c" to get its current value (for example, 18); apply a multiply operation using 9 and the current value of c; apply a divide operation with the previous result ($9c$) and 5; apply an addition operation with 32 and the previous result ($\frac{9c}{5}$). The result is 64.4.

We call this process "expression evaluation". We expect our programming language to apply math-like operations and functions using math-like rules: apply the parenthesized operations first, apply the high priority operations (like multiply and divide) in preference to low priority operations (like add and subtract).

Python has some sophisticated expression operators. Some of them transcend the simple add-subtract-multiple-divide category, and include operators that apply a function to a list to create a new list, apply a function to filter a list and apply a function to reduce a list to a single value.

When we evaluate a function like `abs(-4)`, we name the -4 an *argument* to the function `abs()`. When looking at `3+4`, we could consider 3 and 4 to be argument values to the `+()` function. We could – hypothetically – imagine rewriting `3+4` to be `+(3,4)` just to show what it really means.

Imperative Approach. On the other hand, the imperative style is characterized by using a sequential list of individual statements. Donald Knuth, in his *Art of Computer Programming* [Knuth73], shows a language he calls Mix. It is a purely imperative language, and is similar to the hardware languages used by many computer processor chips.

The imperative style lists a series of commands that the machine will execute. Each command changes the value of a register in the central processor, or changes the value of a memory location. In the following example, each line contains the abbreviation for a command and a reference to a memory location or a literal value. Memory locations are given names to make them easy to read. Literal values are surrounded by `=`. The following fragment uses a memory locations named `C` and `F`, as well as a processor register.

```
LDA C
MUL =9=
DIV =5=
ADD =32=
STA F
```

This first command loads the processor's `a` register with the value at memory location `C`. The second command multiplies the register by 9. The third command divides the register by 5. The next command adds 32 to the register. The final command stores the contents of the `a` register into the memory location of the variable `F`.

Python. Python, like many popular languages, has elements drawn from both applicative and imperative realms. We'll focus initially on expressions and expression evaluation, minimizing the imperative statements. We'll then add various procedural statements to build up to the complete language.

The basic rule is that each statement is executed by first evaluating all of the expressions in that statement, then performing the statement's task. The evaluation of each expression is done by evaluating the parameters and applying the functions to the parameters.

This evaluate-apply rule is so important, we'll repeat here so that you can photocopy this page and make a counted cross-stitch sampler to hang over your computer. Yes, it's that important.

Important: The Evalute-Apply Rule

Each statement is executed by (1) evaluating all of the expressions in that statement, then (2) performing the statement's task.

The evaluation of an expression is done by (1a) evaluating all parameters and (1b) applying the function to the parameters.

Example: `(2+3)*4`, evaluates two parameters: `2+3` and `4`, and applies the function `*`. In order to evaluate `2+3`, there are two more parameters: `2` and `3`, and a function of `+`.

While it may seem excessive to belabor this point, many programming questions arise from a failure to fully grasp this concept. We'll return to it several times, calling it the *evaluate-apply cycle*. For each feature of the language, we need to know what happens when Python does its evaluation. This is what we mean by the semantics of a function, statement or object.

Another Imperative Example. Here's another example of the imperative style of programming. This style is characterized by using a sequential list of individual statements. This imperative language is used internally by Python.

In the following example, each line contains an offset, the abbreviation for a command and a reference to a variable name or a literal value. Variable names are resolved by Python's namespace rules. The following fragment uses a variable named *c*.

```
2           0 LOAD_FAST                0 (c)
            3 LOAD_CONST               1 (9)
            6 BINARY_MULTIPLY
            7 LOAD_CONST               2 (5)
           10 BINARY_DIVIDE
           11 LOAD_CONST               3 (32)
           14 BINARY_ADD
```

This first command (at offset 0) pushes the object associated with variable named *c* on the top of the arithmetic processing stack. The second command (at offset 3) loads the constant 9 on the top of the stack. The third command (at offset 6) multiplies the top two values on the stack. This leaves a new value on the top of the stack.

The fourth command (at offset 7) pushes a constant 5 onto the stack. The fifth command (at offset 10) performs a divsion operation between the top two values on the stack.

The sixth command (at offset 11) pushes a constant 32 onto the stack. Finally, the sixth command performances an add operation between the top two values on the stack.

4.5.4 Expression Style Notes

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 http://www.python.org/dev/peps/pep-0008/.

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 indentation become merely conventions; bad-looking, hard-to-read programs are possible. Python makes the line breaks and indentations part of the language, forcing you to create programs that are easier on the eyes.

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)
```

Instead, we prefer:

```
int(22.0/7.0)
```

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

FIVE

PROGRAMMING ESSENTIALS

The Input-Process-Output Pattern

It's often helpful to look at programs using a typical pattern called "input-process-output" . We'll work through this pattern backwards. In order to see output from a script, we'll need to use the **print** statement. We'll look at this in *Seeing Results : The print Statement*.

Once we are comfortable with the **print** statement, we can introduce processing in *Turning Python Loose With a Script*. When we start making more finished and polished programs, we're going to want to make them easy to use. There are a lot of options and shortcuts available to us, and we'll touch on a few of them here. Later, we'll add even more ease-of-use features.

We In order to do processing, we'll introduce variables and the assignment statement in *Expressions, Constants and Variables*. This will allow us to do the basic steps of processing. We'll describe some additional features in *Assignment Bonus Features*

When we add input in *Can We Get Your Input?*, we'll have all three parts to the input-process-output pattern.

5.1 Seeing Results : The print Statement

We write programs so they can produce useful results. We'll start with statements that immediately satisfy our goal: seeing the results. We'll cover the basic **print** statement in *The print() Function*. We'll add some useful features in *Dressing Up Our Output*.

Yes, this chapter is really short; the **print** statement is delightfully simple.

Important: Python 3

In Python 3, the **print** statement will be replaced with a slightly simpler `print()` function. To align the the future, we'll focus on the function version of print, and avoid the statement.

5.1.1 The `print()` Function

The `print()` function takes a list of values and, well, prints them. It converts numbers and other objects to strings and puts the characters on a file called *standard output*. Generally, this standard output file is directed to the *Python Shell* window in **IDLE**. If you run Python directly, it is directed to the **Terminal** (or **Command Prompt**) window where Python was started.

Redirection

It is important to note that each shell has ways to redirect the standard output file. Python has considerable flexibility, and so does the shell that runs Python. Too many choices is either confusing or empowering. We'll limit ourselves to looking at the choices Python gives. You can, however, look at your GNU/Linux shell documentation or Windows Command Prompt documentation to see what additional choices you have.

When we are interacting with Python at the `>>>` prompt and we give Python an expression, the result is printed automatically. This is the way Python responds when interacting with a person. When we run script, however, we won't be typing each individual statement, and Python won't automatically print the result of each expression. Instead, we have to tell **Python** to show us results by writing an statement using the `print()` function that shows the response we want.

The `print()` function isn't automatically available in Python 2. It will be automatically available in Python 3.

To use the `print()` function, we need to include the following statement.

```
from __future__ import print_function
```

This *must* be the first statement in a script. It alerts Python that we won't being using the **print** statement, and we will be using the `print()` function.

The basic `print()` function looks like this:

`print`(*value*, *...*, *sep=' '*, *end='n'*, *file=sys.stdout*)

The `print()` function converts each value to a string and writes them to the given file (by default it's standard output).

Important: Statement Syntax Rules

We used an ellipsis (...) to indicate something that can be repeated. There's no real upper limit on the number of times something can be repeated.

We used *sep=' '* to show two things. First, if this parameter is used, it must be given by name. Second, the parameter has a default value. That meas it can be safely ignored for now.

Here are some examples of a basic `print()` function.

```
from __future__ import print_function
print(22/7, 22./7.)
print(335/113, 335./113.)
print( ((65 - 32) * (5 / 9)) )
```

We'll look at the special purpose *sep*, *end* and *file* arguments separately. For now, it's important to note that they have default values, making them optional.

5.1.2 The Old print Statement

The **print** statement takes a list of values and, well, prints them. It converts numbers and other objects to strings and puts the characters on a file called *standard output*.

The basic **print** statement looks like this: **,**

```
print expression [  expression ] ...
```

The **print** statement converts the expressions to strings and writes them to standard output.

Important: Statement Syntax Rules

We'll show optional clauses in statements by surrounding them with [and]'s. We don't actually enter the [ ]'s, they surround optional clauses to show us what alternative forms of the statement are.

We use a trailing ellipsis (...) to indicate something that can be repeated. There's no real upper limit on the number of times something can be repeated.

Also notice that we put a `,` before the *expression*. This is your hint that expressions are *separated with* `,` characters when you have more than one.

In the case of **print**, the syntax summary shows us there are many different ways to use this statement:

- We can say '`print expression`' with one expression.
- We can say '`print expression, expression`' with two expressions, separated by `,`.
- And so on, for any number of expressions, separated by `,`s.

While our summary doesn't show this, there are several other forms for the **print** statement. All of the extra syntax options and quirks of the **print** statement are really just fodder for confusion.

Here are some examples of a basic print statement.

```
print 22/7, 22./7.
print 335/113, 335./113.
print ((65 - 32) * (5 / 9))
```

We're mostly going to ignore the **print** statement because the `print()` function does the same thing and has no quirks or odd special cases.

5.1.3 Dressing Up Our Output

We can make our printed output easier to read by including quoted strings. See *Strings – Anything Not A Number* to review how we write strings.

For example, the following trivial program prints a string and a number. Since our string had an apostrophe in it, we elected to surround the string with quotes (").

```
from __future__ import print_function
import math
print( "The answer:", 6*7 )
print( 'Value of "pi":', 6.0/5.0*( (math.sqrt(5)+1) / 2 )**2 )
```

It's very important to note that the `from __future__ import print_function` must be provided first.

Tip: Debugging the **print** Statement

One obvious mistake you will make is misspelling **print**. You'll see `NameError: name 'primpt' is not defined` as the error message. I've spelled it "primpt" so often, I've been tempted to rewrite the Python language to add this as an alternative.

The other common mistake that is less obvious is omitting a comma between the values you are printing. When you do this, you'll see a `SyntaxError: invalid syntax` message.

If the result of a print statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**s Python shell to examine the processing one step at a time.

5.1.4 Print Exercises

1. **Print Expression Results**.

   In the *Conversational Python Exercises* exercises in *Instant Gratification : The Simplest Possible Conversation*, we entered some simple expressions into the Python interpreter. Change these expressions into nice-looking `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 )
```

5.1.5 Print FAQ

How can I get more control over the output? The `print()` function's default is to put a space between items, which may not always be desirable.

The string formatting method provides complete control over the formatting of data. We'll cover this in depth in *Sequences of Characters : str and Unicode*. First, we want to introduce a number of programming statements. Once we've got more of the language under our belt, we'll tackle the "fit and finish" issues of nicely formatted output.

However, if you can't wait until then, we'll provide some hints as to what will come in the future.

```
from __future__ import print_function
print( "from this", "via that", "to this", sep="->" )
```

How can I direct output to stderr? We'll talk about this in detail in *External Data and Files*. However, if you can't wait until then, we'll provide some hints as to what will come in the future.

```
from __future__ import print_function
import sys
print( "an error", file=sys.stderr )
```

How can I use multiple statements to create one line of output? The `print()` function's default is to put a newline character at the end, which may not always be desirable.

If we change the *end* parameter, we can piece together a long line of output from multiple uses of the `print()` function. In the following example, the first statement uses `': '` instead of the newline character; the print statement will create a partial line.

```
from __future__ import print_function
print( "335/113", end=': ' )
print( 335.0/113.0 )
```

If we have very complex expressions, this can make our program easier to read by breaking a complex message into understandable chunks.

This is not obvious when working with Python at the `>>>` prompt. When we turn to scripts (in the next chapter), we'll see more use for this.

How do I produce binary output? MP3's, MOV's, JPEG's, etc. We'll talk about this in detail in *External Data and Files*. There's no quick-and-dirty shortcut for that kind of operation; it requires interacting with the file system. Also, these more sophisticated data formats require more sophisticated programming.

5.2 Turning Python Loose With a Script

One of our goals is to have the Python interpreter execute our scripts. Program scripts can vary from a few simple expressions to complex sequences of Python commands. There are two parts to this: creating the script and then running the script. We'll cover the basics of creating the script in *Making A Script File*. We show a simple technique for running a script in **IDLE** in *Running Scripts in IDLE*.

In the long run, however, we'll want to use more direct methods to run our scripts. After all, we don't want to have to start **IDLE** every time we want to run our program. After we get some more experience with programming, we'll look at making our programs a perfectly seamless part of our work environment.

We'll provide some answers to common questions in *Scripting FAQ*.

5.2.1 Making A Script File

The first step in Python programming is to create the program script. For the following examples, we'll create a simple, three-line script, called `example1.py`.

example1.py

```
from __future__ import print_function, division
print(65, "F")
print( (65 - 32) * 5 / 9, "C" )
```

Within **IDLE**, you create a file by using the **File** menu, the **New Window** item. This will create a new window into which you can enter your two-line Python program. Check your spelling and spacing carefully.

When you use the **File** menu, **Save** item, be sure to read where the file is going to be saved. You'll notice that **IDLE** may be starting in `C:\Python26`, your `Macintosh HD`, or `/home/slott` or some other unexpected directory.

For now, it helps to save this file in your home directory. This could be `C:\Documents and Settings\SLott`, or `/home/slott`.

You can't easily use a word processor for this, since word processors include a lot of formatting markup that Python can't read. If you want to try and use an office product to create this kind of file, you have to be absolutely sure that you save the file as pure text.

There are several ways we can run the Python interpreter and have it evaluate our script file. Since **IDLE** is virtually identical on all platforms, we'll cover this next.

Important: Writing and Saving A Script

One of the biggest benefits of using **IDLE** is that your Python script has various syntax elements highlighted. In mine, the keywords show up in orange, strings in green, and my expressions are black. If I misspell **print**, it doesn't show up in orange, but shows up in black.

The most common problem we see is people saving their file to unexpected locations on their disk. It's important to save the file to a directory where you can find it again.

One interesting confusion we've seen arises when people forgetting to save the file in the first place. **IDLE** will ask you if you want to save the file when you attempt to run it. Sometimes this message is unexpected and that can be confusing. Our advice is to save early and save often.

Alternatives to IDLE. If you don't want to use **IDLE** to create text files, you do have several choices for nice program editors. These will require you down download and install additional software.

- **Windows**. You have the built-in **notepad**, or you can purchase any of a large number of programmer's text editors, including **TextPad**. There are free editors like **jEdit**, also.
- **MacOS**. You have the built-in **textedit** application. Be sure to use the **Format** menu, **Make Plain Text** menu item to strip the file down to just text. Or you can purchase any of a large number of programmer's text editors, including **BBEdit**. There are free editors like **jEdit**, also.
- **GNU/Linux**. If you are using GNOME, you have **gedit**. If you want, you can also use **vim** or **emacs**, two very fine sophisticated editors that have been used for decades to write software.

After you create your file outside **IDLE**, you can open the file with **IDLE** in order to run it. You use the **File** menu, **Open...** item to open a file you created outside **IDLE** It's important to take note of where you save files so that you can find them and open them again.

5.2.2 Running Scripts in IDLE

The window for your script file will have a **Run** menu. (The ordinary *Python Shell* window doesn't have this menu.) On the **Run** menu, you'll find the **Run Module** menu item, which will execute your script file. This will show the results in the *Python Shell* window.

Why is it called **Run Module**? Most Python files are called "modules", which are files of definitions. Even though the official name is "module", we'll insist on calling them "scripts", because that is a much more descriptive name.

If you have trouble finding the **Run** menu, be sure you are looking at the correct window. The initial window in **IDLE** is the *Python Shell*. It has the **Shell** menu and shows the `>>>` prompt. When you create a new window to edit a script (or you open a script file), this widow will have the **Run** menu. When you open a script or save a script, the window name reflects the name of the script file.

When we select **Run Module** menu item from the **Run** menu, we see the following in the *Python Shell* window.

```
>>> ================================ RESTART ================================
>>>
65 F
18 C
>>>
```

This shows us that the Python shell was restarted using our script as input. It also shows us the output from our two **print** statements. We ran our first program.

Important: Debugging Aids in IDLE

If you have syntax errors, you'll see a pop-up dialog box named `Syntax error` with a message like `There's an error in your program: invalid syntax`. You'll also notice that some part of your script will be highlighted in red. This is near the error.

Since **IDLE** highlights various syntax elements, you can use the color as a hint. In mine, the keywords show up in orange, strings in green, and my expressions are black. If I misspell **print**, it doesn't show up in orange, but shows up in black.

If you have semantic errors, you'll see these in the shell window in red. For example, I got the following by messing up my program.

```
Traceback (most recent call last):
  File "E:/Personal/NonProgrammerBook/notes/sample1.py", line 1, in -toplevel-
    print(65, "F"/2)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

You can see my erroneous **print** statement: it has `"F"/2`. And you can also see Python's complaint. While the syntax is acceptable, it doesn't mean anything to divide the letter `"F"` by 2.

We can fix our script file, save it, and re-run it. You'll notice that the **Run Module** menu item has a short-cut key, usually `F5`. This edit-save-run cycle is how software gets built.

5.2.3 Script Exercises

1. **Simple Script**.

   Create a Python file with the following three commands, each one on a separate line: `copyright`, `license`, `credits`.

2. **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)`.

   Be sure to include the `from __future__ import print_function` line first.

3. **Another Simple Print Script**.

   Create and run a Python file with commands like the following examples: `print("one red", 18.0/38)`; `print("two reds in a row",(18/38.0)**2)`.

   Be sure to include the `from __future__ import print_function` line first.

4. **Numeric Types**.

   Compare the results of `22/7` and `22.0/7`. Explain the differences in the output.

5.2.4 Scripting FAQ

What are the various ways to use Python? Python can be used a variety of ways, depending what problem you are solving.

- Interactively. We can interact directly with Python at the "command-line". This was what we saw in *Instant Gratification : The Simplest Possible Conversation*. A tool like **IDLE** makes it easier to enter Python statements and execute them. We looked at this in *IDLE Time : Using Tools To Be More Productive*.
- Scripted. Tis makes the processing completely automatic. We'll look at this in *Comments and Scripts* In the long run, this automation is our goal. However, to learn the language, we find the direct interaction is very helpful. Manual interaction via **IDLE** is our "training wheels" for learning the language.
- Through The Web. While beyond the scope of this book, Python can be part of the Internet, with your application running on a web server.

Why are there so many ways to use Python? Or, why can't we just use **IDLE**? The huge number of choices is a natural consequence of creating a simple, flexible program. Many people use Python through **IDLE** and are happy and successful in what they do.

More sophisticated problems, however, often require more complex use of Python. Since the Python program (`python`, or `python.exe`) can be used a variety of ways, we can use Python to build a number of different kinds of solutions to our data processing problems.

In a book like this, we hate to present Python from a single point of view. We prefer to present a number of choices so that different readers can locate one that looks like it will solve their problem.

Do I have to write a script? You don't have to write scripts, you can do everything through interaction with **IDLE**. Scripting is not required, but it is generally the goal of programming. An automated solution should be something that can be double-clicked, or something that is invoked by a web server.

5.3 Expressions, Constants and Variables

Our programs up until this point have been somewhat limited. We've been able to write programs that do exactly one calculation and nothing more than that. How many times do we want to convert 65 degrees F to degrees C?

We can change the original program each morning when we check the weather, but this is second-rate. If we make a mistake, we might break the program completely. We want something more general, something that can convert any temperature from degrees F to C.

To generalize our calculations, we need to replace a *literal* value with a *variable* that can take on any value. Our programs can then work by assigning a specific value to a variable and using that variable in a generic calculation. This simple generalization technique gives us a bunch of new capabilities to examine. We'll look at creating variables in *Putting Name Tags on Results*.

When we assign a value to a variable, it acts as a kind of placeholder; it's a way of saying "we've gotten this far in our work." We'll introduce the **assignment** statement that creates and updates variables in *The Assignment Statement*. Since setting a variable is so important, there are a number of variations on the assignment statement. We'll look at these in *Assignment Combo Package*.

Since our programs are getting more sophisticated, we need ways to test and debug them. We'll look at one of these techniques in *Where Exactly Did We Expect To Be?*.

5.3.1 Putting Name Tags on Results

A Python *variable* is a name that refers to an object. It's easy to think of a variable as a name tag, pinned to the object. An object can be any of the numeric types that we looked at in *Arithmetic and Expressions*. It turns out that many more things than just numbers are objects that can have names pinned to them.

A Python variable name must be at least one letter, and can have a string of numbers, letters and _ to any length. Spaces and punctuation marks other than _ are not allowed in a variable name.

Here are some examples of variable names:

- `x`
- `pi`
- `data`
- `aLongName`
- `multiple_word_name`

- `__str__`
- `_hidden`

Important: Python Name Rules

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 *Defining New Objects* and *Module Definitions – Adding New Concepts*.
- Names that begin with `__` are part of the way the Python interpreter is built. We should never attempt to create variables with names that begin with `__` because it can be lead to confusion between our programs and Python's internals.

The Semantics of Naming. How does a variable name get stuck to an object? Primarily, the **assignment** statement does this labeling. We'll look at the syntax of assignment in the next section. First, we need to talk about what assignment means, then we'll look at how to write it.

An assignment statement has two parts: a variable name and an expression.

1. An **assignment** statement evaluates the expression.
2. An **assignment** statement then assigns the result to a variable. There are two variations on this.
    - If the variable name already existed, the name tag is removed from the old value. The old value is no longer accessible. The name is pinned on the new value.
    - If the variable name did not already exist, the name is created. Then the name is pinned on the new value.

We generally don't worry about creating new variables; there's no cost, they're just names. Create as many as you need to make your program's purpose crystal-clear.

A Python variable has a *scope* of visibility. The scope is the set of statements that can reference this variable. For our first programs, all of our variables will have a *global* scope: we can use any variable anywhere we want. When we look at organizing our programs into separate sections – beginning in *Organizing Programs with Function Definitions* – we'll see how Python's separation between local scopes and a global scope can simplify our programming by localizing a name.

Important: A Script Is A Journey

If we think of a script as a journey from the first statement to the last, we'll often mark our progress along that journey by setting variables. The values that are assigned to our variables amount to a big "You Are Here" arrow showing us where we are.

When a variable is created or changed, the overall position along our journey has changed. The variables start with initial values; the values change as inputs are accepted and our program executes the various statements. Eventually the state of the variables indicates that we have reached our goal, and our program can exit.

We emphasize this because some of the more common program design errors have to do with failure to use variables correctly. We may see a program that doesn't set a variable to indicate when progress has been made. The mistake is either a missing assignment statement or assignment to the wrong variable. We'll look at ways to debug these kinds of problems in *Where Exactly Did We Expect To Be?*.

When we look at it this way, our variables are more than just labels slapped on objects. They have a profound significance; they reflect the "meaning" of our program.

5.3.2 The Assignment Statement

We create and change variables with the *assignment* statement. Here's the syntax summary:

```
variable =  expression
```

First, evaluate *expression*, creating some result object. Then, assign the given variable name to that result object.

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

example3.py

```
1  # Compute the value of a block of stock
2  from __future__ import print_function, division
3  shares= 150
4  price= 3 + 5/8
5  value= shares * price
6  print( value )
```

2. We want to use the `print()` function, and exact division.
3. We create the variable shares, set to the plain integer object 150.
4. We create the variable price, set to the object created by the expression `3 + 5/8`. Prior to 2001, stocks were traded in multiples of 1/8 of a dollar, 12.5 cents.
5. We create the variable value, set to some object created by the expression `shares * price`. Setting this variable advances our program nearly to completion. Once we print the value of this variable, we are finished processing.

Bonus Question. If we omit the `division` from the `from __future__ import`, what happens to the calculation of *price*?

Tip: Debugging the Assignment Statement

There are two common mistakes in the assignment statement. The first is to choose an illegal variable name. If you get a `SyntaxError: can't assign to literal` or `SyntaxError: invalid syntax`, the most likely cause is an illegal variable name.

The other mistake is to have an invalid expression on the right side of the =. If the result of an assignment statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**'s *Python Shell* window to examine the processing one step at a time.

5.3.3 Assignment Combo Package

One very common pattern is to update a variable by performing an operation on that variable. Look at the following example where we update the *sum* variable using the `+` operation.

```
from __future__ import print_function, division
sum= 0
sum= sum + 25
sum= sum + 42
sum= sum + 37
print( "average", sum/3 )
```

This assignment statement pattern is so common that the pattern had to be added to the language as the *augmented assignment* statement. These augment the basic assignment with an additional operation. There are several variations on this combo-pack assignment statement.

The most common application of this pattern is to accumulate a sum. This augmented assignment makes it obvious what we are doing. For example, look at this common augmented assignment statement.

```
sum += v
```

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

```
sum = sum + v
```

We can use this to replace our first example above with something slightly simpler.

```
sum= 0
sum+= 25
sum+= 42
sum+= 37
print( "average", sum/3 )
```

Here's a larger example that does more substantial calculation at each step. This shows the strength of this statement.

Create the following file name `portfolio.py`. In IDLE, you can run this using the **Run Module** item on the **Run** menu.

portfolio.py

```
# Total value of a portfolio made up of two blocks of stock
from __future__ import print_function, division
portfolio = 0
portfolio += 150 * 2 + 1/4
portfolio += 75 * 1 + 7/8
print( portfolio )
```

When we run this script, we see the following.

```
>>> ================================ RESTART ================================
>>>
376.125
>>>
```

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

Here's an interesting use of `/=`. This computes the various digits in a base-10 number from right to left.

```
>>> y=1956
>>> y%10
6
>>> y/=10
>>> y%10
5
>>> y/=10
>>> y%10
9
>>> y/=10
>>> y%10
1
```

Try the same basic sequence of operations using 16 instead of 10. You'll get a sequence of three numbers before y is equal to zero. Compare that sequence of numbers to `hex(1956)`.

Tip: Debugging the Augmented Assignment Statement

There are two common mistakes in the augmented assignment statement. The first is to choose an illegal variable name. If you get a `SyntaxError: can't assign to literal` or `SyntaxError: invalid syntax` the most likely cause is an illegal variable name.

The other mistake is to have an invalid expression on the right side of the assignment operator. If the result of an assignment statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**'s Python shell to examine the processing one step at a time.

5.3.4 Where Exactly Did We Expect To Be?

As our program moves along its journey toward completion, it will create variables and change the value of variables. There are two questions we can ask at the end of each statement:

- Where are we?
- Is this where we should be?

These questions are two variations on a common theme. We know what our sequence of statements *should* do, and we need to be sure it really will do the right thing.

In *Goal-Directed Activities* we provided a hint that a program's progress was really a series of state changes. It turns out that the values assigned to our variables completely defines the state of a program.

We can develop some confidence in what a program does by examining a *trace* of the variable creations and changes. We mentioned this briefly in *Putting Name Tags on Results*. Here we'll look at this notion of execution trace a little more carefully.

The Example. Here's an example program that we'll use for creating a trace of the planned sequence of assignment statements. This is an extension of the *Craps Odds* exercise in *Expression Exercises*.

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.

This little program will compute the odds of winning on the first roll and the odds of losing on the first roll. The remaining probability is the odds of establishing a point.

Here's an example that you can save, named `craps.py`. In IDLE, you can run this using the **Run Module** item on the **Run** menu.

craps.py

```python
from __future__ import print_function, division

# Compute the odds of winning on the first roll
win = 0
win += 6/36 # ways to roll a 7
win += 2/36 # ways to roll an 11

# Compute the odds of losing on the first roll
lose = 0
lose += 1/36 # ways to roll 2
lose += 2/36 # ways to roll 3
lose += 1/36 # ways to roll 12

# 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

# Results
print("first roll win", win)
print("first roll lose", lose)
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.

More Probability

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

```
point = 0
point += 2*3/36 # ways to roll 4 or 10
point += 2*4/36 # ways to roll 5 or 9
point += 2*5/36 # 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.

How To Make A Trace. When we make an execution trace, we start with a clean piece of paper. As we look at our Python source statements, we write down the variables and their values on the paper. From this, we can see the state of our calculation evolve.

When we encounter an assignment statement, we look on our paper for the variable. If we find the variable, we put a line through the old value and write down the new value. If we don't find the variable, we add it to our page with the initial value.

Here's our example from *craps.py script* through the first part of the script. The *win* variable was created and set to `0`, then the value was replaced with `0.16`, and then replaced with `0.22`. The *lose* variable was then created and set to `0`. This is what our trace looks like so far.

win:	~~0.0~~	~~0.16~~	0.22
lose:	0		

Here's our example when *craps.py script* is finished. We changed the variable *lose* several times. We also added and changed the variable *point*.

win:	~~0.0~~	~~0.16~~	0.22	
lose:	~~0.0~~	~~0.027~~	~~0.083~~	0.111
point:	~~1.0~~	~~0.77~~	0.66	

We can use this trace technique to understand what a program means and how it proceeds from its initial state to its final state.

5.3.5 Assignment Exercises

1. **Extend Previous Exercises**.

   Rework the exercises in *Expression Exercises*.

   Each of the exercises in *Expression Exercises* can be rewritten to use variables instead of lengthy expressions. For example, if you want to tackle the Fahrenheit to Celsius problem, you might write something like this:

   ```
   # Convert 8 C to F
   from __future__ import print_function, division
   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 "brownness" sensor and a single heating element. What are the inputs? What are the outputs? Are there internal states while the toaster is making toast?

5.4 Assignment Bonus Features

We'll cover some additional features of the **assignment** statement. In *Combining Assignment Statements* we'll cover "multiple assignment", a handy short-hand. We'll look at the interplay between assignment statements and our interactive exploration of simple expressions in *More About Python Conversations*.

5.4.1 Combining Assignment Statements

The assignment statement can be expanded to assign multiple variables at one time. Python will evaluate all of the expressions and then assign the set of variables all at once. All we have to do is assure that the number of variables on left side of the `=` is the same as the number of expressions on the right side.

The first important part is the obvious match-up between the number of expressions and the number of variables. We generally don't assign more than two or three things at once, so this is an easy rule to observe.

Another important part of this is the "all at once". We use the multiple assignment statement when we're doing two things at the same time and want to lock those two things together. Separate assignment statements imply that there's a sequence to things; that the steps are in this order because one step depends on another. In a some cases, however, a pair of steps may depend on previous steps, but the pair of steps don't depend on each other.

Multiple-Assignment Syntax. A multiple-assignment statement looks like a standard assignment statement. We separate the left-side variables with `,` and the right-side expressions with `,`, also. Here's the syntax for multiple assignment:

```
variable , ... = expression , ...
```

The **...** means that the variable or expression can be repeated any number of times. The **,** means that we separate multiple variables and multiple expressions with **,**s.

We must have the same number of variables on the left as expressions on the right.

Examples. In all of the examples, we'll try to show a pattern where two variables are tightly coupled. We use this when we don't want the assignments to get separated in our program. We want the two assignments in the same statement to emphasize how tightly coupled the two variables are.

```
price, shares = 5 + 3./8., 150
amount = price * shares

hours, minutes, seconds = 8, 18, 24
timestamp = (hours*60 + minutes)*60 + seconds
```

The following script has some more examples of multiple assignment. In this case, we're doing some algebra to compute two values at the same time. The slope, *m*, and the intercept, *b*, both depend on the two points, and can be computed at the same time.

Here's a short example that you can save, named `line.py`. In IDLE, you can run this using the **Run Module** item on the **Run** menu.

line.py

```
# Compute line between two points.
from __future__ import print_function, division
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.

```
y= 1.25 *x+ 0.5
```

This program sets variables *x1*, *y1*, *x2* and *y2*. Then we computed *m* and *b* from those four variables. Then we printed the *m* and *b*.

The All-At-Once Rule. The basic rule is that Python evaluates the entire right-hand side of the **assignment** 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 the values of two variables. This is often quite a bit more complicated in other languages.

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

In *Doubles, Triples, Quadruples : The tuple* we'll see even more uses for this feature.

Tip: Debugging Multiple Assignment Statements

There are three common mistakes in the augmented assignment statement. The first is to choose an illegal variable name. If you get a `SyntaxError: can't assign to literal` or `SyntaxError: invalid syntax` the most likely cause is an illegal variable name.

One other mistake is to have an invalid expression on the right side of the assignment operator. If the result of an assignment statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**'s Python shell to examine the processing one step at a time.

The third mistake is to have a mismatch between the number of variables on the left side of the = and the number of expressions on the right side.

5.4.2 More About Python Conversations

When we first looked at interactive Python in *Instant Gratification : The Simplest Possible Conversation* we noted that Python automatically prints the results of an expression. However, if we enter a complete **assignment** statement, Python executes the statement silently. Sometimes this silence can be inconvenient.

Consider the following example.

```
>>> pi=335/113.0
>>> area=pi*2.2**2
>>> area
14.348672566371683
```

The first two inputs are complete statements, so Python provides no response. Our final calculation of *area* didn't produce a response, so we had to provide a simple expression, *area*, to see our answer.

It turns out that there's a subtle bug in this. It was hidden from us because of the silent execution of statements.

The solution is based on a built-in feature of Python. When you simply enter an expression, Python always assigns the result to the *implicit result* variable, named _. It's as though you typed the following around each *expression*.

```
_ =  expression
print(_)
```

A Longer Conversation. Here's how we use the implicit results variable. We type expressions, and – if the result is helpful – we save the result (_) in a new variable.

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

Our first expression had an error. We fixed that error and saved the correct implicit result into *pi* by saying `pi=_`. When we finished, we saved the last implicit result into *area* with `area=_`.

Th comes in handy when you exploring something rather complex.

Debugging Only. It's important to note that the _ trick only works when we're using Python interactively at the >>> prompt. We can't (and shouldn't) write this in our script files. Our scripts will simply assign expressions to variables directly.

5.4.3 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. The following is considered as a poor way to write Python.

```
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 typically `lower_case_with_underscores()` or `mixedCase()`. Variable names typically begin with lower-case letters.

In addition, the following special forms using leading or trailing underscores are important to recognize:

- *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.

5.5 Can We Get Your Input?

In *Getting Raw Input* we'll introduce a primitive interactive input function. This will finish up the statements we need to fill out the "input-process-output" pattern for simple programs. We'll provide some additional notes in *Additional Notes and Caveats*.

To make complete use of this, we'll need to digress on the standard input, output and error files in *The Standard Files*.

The material on the **del** statement in *The del Statement* is here for completeness. Logically, there's a nice symmetry between creating variables with the **assignment** statement, setting values with input functions and removing variables with the **del** statement. As a practical matter, however, we rarely need to remove a variable.

We'll answer some additional questions in *Input FAQ*.

5.5.1 Getting Raw Input

Python provides simplistic built-in functions to accept input and set the value of variables. These are not really suitable for a all applications, but they will help us learn the language. The more robust and reliable input functions are generally embedded in a web-based application, or a sophisticated GUI, both of which are way beyond what we can cover in this book.

These input functions will gather data from a file called *standard input*. Generally, this standard input file is your keyboard, and the results will show on the *Python Shell* window in **IDLE**. If you run Python directly, the **Terminal** (or **Command Prompt**) where Python was started will manage reading characters from your keyboard.

Raw Input. By "raw", we mean one line of input. The input is what the person typed after handling backspaces. It isn't every keystroke, it's the finished product of typing and backspacing.

The details of how backspacing is handled is actually part of the operating system. Python depends on the OS to provide the line of input via the standard input file, making sure that the `backspace` operates as we expect.

We have to talk about the `raw_input()` from two distinct points of view.

- **What you see**. You will see a prompt on *standard output* – usually the console – and you can respond to that prompt by typing on standard input. This console will usually be the Python Shell window of **IDLE**, or the terminal window, depending on how you are running Python.
- **What you say in Python**. The Python script evaluates a function; the value of that function will be a string that contains the characters someone typed. We'll take a very close look at strings in *Sequences of Characters : str and Unicode*. For now, we do a few simple things with strings.

An Example Script. Here's a very small script that uses `raw_input()` that produces a prompt, reads and prints the result. This will give you a sense of the two worlds in which this function lives: the world of user interaction as well as the world of Python function evaluation.

Create the following file, named `rawdemo.py`. Save it and then run it in **IDLE** using the **Run Module** item in the **Run** menu.

rawdemo.py

```
# get the user's answer
from __future__ import print_function
answer= raw_input( "continue?" )
print("You said:", answer)
```

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

```
continue? why not?
You said: why not?
```

This program begins by evaluating the `raw_input()` function. When `raw_input()` is applied to the parameter of `"continue?"`, it writes the prompt on standard output, and waits for a line of input.

We entered `why not?`. When we hit `enter`, we told the operating system that our input line was complete. The OS hands the completed input line to Python. Python then returns this string as the value of the `raw_input()` function.

Our program saved the value from `raw_input()` int the variable *answer*. The second statement printed that variable.

Definition. Here's a formal definition for the `raw_input()` function.

`raw_input(`[*prompt*]`)` → string

Read a string from standard input. If a prompt is given, it is printed before reading. If the user hits end-of-file (`Ctrl-D` in GNU/Linux or MacOS; `Ctrl-Z` in Windows), an exception is raised.

There's a second input function, named `input()`. It will be removed from Python 3 because it's worse than useless. It's confusing and a potential security nightmare.

Making Raw Input Useful. If we want a numeric value, we must convert the resulting string to a number. In the following example, we'll use the `int()` and `float()` functions to convert the strings we got from the `raw_input()` function into numbers that we can use for calculation.

We'll use the `raw_input()` and `int()` functions to get a number of shares. The resulting number is assigned the name *shares*. Then the program uses the `raw_input()` and `float()` functions to get the price.

Create the following file, named `stock.py`. Save it and then run it in **IDLE** using the **Run Module** item in the **Run** menu.

stock.py

```
# Compute the value of a block of stock
from __future__ import print_function
shares = int( raw_input("shares: ") )
price = float( raw_input("dollars: ") )
print("value", shares * price)
```

Here's what we saw when we ran this program in **IDLE**.

```
shares: 150
dollars: 24.18
value 3627.0
```

Exceptional Input. Exceptions, in Python, often mean exceptionally bad. The `raw_input()` mechanism has some limitations. If the string returned by `raw_input()` is not suitable for use by `int()` function, an exception is raised and the program stops running.

Here's what it looks like when we ran `stock.py` and provided inappropriate input values.

We'll look at ways to handle this in *The Unexpected : The try and except statements*.

5.5.2 Additional Notes and Caveats

If you try to run these examples from TextPad, you'll see that TextPad doesn't have any place for you to type your input. You'll get an immediate end-of-file error. Why does this happen? It happens because TextPad doesn't have a proper file for standard input. Since the file doesn't exist, we get an immediate error when we try to use it.

For MacOS users using tools like BBEdit, you'll can use the **Run In Terminal** item in the **#!** menu to create a **Terminal** window for your interaction. This new window appears when your run your script, showing your standard input and giving you a place for standard input.

Tip: Debugging the `raw_input()` Function

There are two kinds of mistakes that occur. The first kind of mistake are basic syntax errors in the `raw_input()` function call itself.

The other mistake, which is more difficult to prevent, is to provide invalid input to the script when it runs. Currently, we don't quite have all of the language facilities necessary to recover from improper input values. When we cover the **try** statement and exception processing in *The Unexpected : The try and except statements* we'll see how we can handle invalid input.

In the long run, we'll see that the `raw_input()` function is not the most reliable tool. For simple programs, problems with `raw_input()` are easily solved. If you give your software to someone else, the vagaries of what is legal and how Python responds to things that are not legal can be frustrating for them to learn.

Professional quality software doesn't make much use of these functions. Typically, interactive programs use a complete *graphic user interface* (GUI), often written with the `Tkinter` module or the `pyGTK` module. Both of these are beyond the scope of this book: they aren't newbie-friendly modules.

5.5.3 The Standard Files

We need to take a quick digression to look at how your keyboard really works. Generally, we take our keyboard for granted: we start an application, we type and the letters appear. After a while, you get used to letters only showing up in the front-most window. Our operating system tells us which window is active by providing a number of visual cues like bringing the window to the font, showing a blinking cursor and a fancier frame around the window.

Under the hood, your operating system is watching your keyboard device for activity. Most of the buttons on your keyboard generate an "event'. Some of the buttons are "modifiers": the shift, alt, command, and control keys modify the basic key event. The operating system routes these key events to the front-most window. The application program that displays the front-most window is responsible for making sense of the events.

Why "events"? Why not just "characters"? The reason is that some programs don't have characters. A game, for example, doesn't want characters, it wants events that will make the pinball flippers flip, or the person walk.

For many programs, most of these events will be interpreted as characters. There are two really common events that are not characters: the backspace event and the enter event are not letters; instead, they change the sequence of characters being accumulated. The front-most window has to do something with events like backspace. What most of us expect is that backspace removes the previously accumulated character. The enter key alerts the program that you are finished typing and are ready for the program to see the input letters.

Plus, when you look at your keyboard, you've got a dozen (or more) F-keys, plus keys with names like Insert, Delete, Home, End, etc. These are not characters in the same sense as the other keys. These extra keys are used by **IDLE** to control your interactive Python session. When you run Python from the Command Prompt or Terminal tool, you'll see that these extra F-keys do little or nothing. In the Windows Command Prompt, you can use a program named **doskey** to define actions for these non-letter keys. In the MacOS, there are a number of programs that use these additional F-keys for a variety of purposes; for instance, `F12` brings up my dashboard.

Redirection. Each shell also has ways to change the origin of the characters available on standard input. When you *redirect* standard input, it means that your program will wind up reading from a disk file instead of the keyboard. In *The print() Function*, we mentioned the file called *standard output*. Standard output is like standard input: it was opened for you, and it can be redirected outside your program.

This shell redirection technology allows a single program to read from a disk file or read from the keyboard without any changes to the program; there is just a small change to the shell command that starts the program. Similarly, it also allows a program to write the terminal window, or write to a disk file depending on settings provided to the shell. While the details of controlling the shell are outside the scope of this book, the idea is that one program can do any of the preceding, with no change to the program. This gives us a lot of flexibility for no real cost or complexity in our programming.

5.5.4 Simple Input Exercises

Refer back to the exercises in *Expression Exercises* 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 $\frac{1}{2} \times h \times l$. 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.

5.5.5 The del Statement

An assignment statement creates a variable, or assigns a new object to an existing 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. This version of the statement is used rarely; we describe it here to close the circle of the life for a variable.

The most common use for the **del** statement is to remove individual elements from a list object. We'll revist this statement when we look at lists in *Flexible Sequences : The list*. There, we'll find a more practical use for this statement.

The **del** statement looks like this: **,**

```
del  target [  ... ]
```

A *target* is a name of a Python object: a variable, function, module or other object. The variable is removed. Generally, this also means the target object is removed from memory.

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 only reference to an object, the object will be removed from memory also. If, on the other hand, other variables still refer to this object, the object won't be deleted.

Taking Out the Trash

Memory management is silent and automatic, which makes Python programs very reliable with little effort. The computer-science types call the automatic removal of objects *garbage collection*. When done manually (for instance, in the language C++), small, hard-to-find mistakes can lead to *dangling references*: a variable that refers to an object that was deleted prematurely. It can also lead to *memory leaks*, where unreferenced objects are not properly removed from memory. In both cases, the program is unreliable; it works for a while and then behaves badly or stops working.

Python's automated garbage collection means that Python programs suffer from none of the common memory management problems that plague C++ programs. It also means that we rarely need to actually use the **del** statement.

Tip: Debugging the **del** statement

If we misspell a variable name, or attempt to delete a variable that doesn't exist, we'll get an error like `NameError: name 'hack' is not defined`.

5.5.6 Input FAQ

If there is no use for the del statement, why cover it? The **del** statement isn't completely useless for newbies. We cover it for a number of reasons. First, we'll return to it in the chapter on lists and use it to remove an item from a list. Second, when you see it in someone else's program, you'll be able to interpret it. And third, we can say that we covered *all* the language, identifying the parts that meet more advanced needs.

If `raw_input()` is so poor, what's better? We'll take a quick survey of four overall architectures where you will be interacting with your computer. What we want to emphasize is the tremendous differences between these architectures. Since there is so little in common, Python doesn't have a newbie-friendly, reliable, flexible, and richly interactive input mechanism.

- A command-line utility. These are programs like the GNU/Linux **grep** program, where all of the interaction is centered around running the program from the command line or as part of a processing pipeline. These programs read from standard input and write to standard output. They typically use file-oriented `read()` and `write()` methods, not the `raw_input()` function.
- A GUI program. These are programs like those in MS-Office or Open Office; this group also includes all games. These programs generally rely on a "framework" that provides basic graphics capabilities. Python programmers use `Tkinter` or `pyGTK` for this kind of program. The framework handles keyboard and mouse events and provides specialized functions for interacting with the GUI objects.
- A Web program. These are programs like eBay or Yahoo!. In this case, the program exists on a web server and does its processing in response to web requests. These programs rely on the HTTP protocol and HTML web pages for their interaction. There are a number of Python-oriented frameworks for running Python programs from a web server.
- An embedded program. These are programs like those in your microwave oven, thermostat or coffee-maker. You interact with these programs through specially-engineered devices like membrane keyboards, buttons and LED's. These programs often rely on specialized mini-operating system to handle the devices.

With these styles of programming having so little in common, there's very little built-in to the Python language to support a user interface. For all but the first case (command-line utilities), you'll have to master some kind of add-on package appropriate to the kind of programs you want to write.

CHAPTER

SIX

SOME SELF-CONTROL

Making Choices, Doing It All

This section represents a significant milestone. Up until this part, we have presented **Python** as a souped-up desk calculator. This part shows the essential elements of building automated data processing.

We'll start with truth and logic in *Truth and Logic : Boolean Data and Operators*. This is an extension to the expressions and numeric types we started out with. We'll add another data type, `boolean` , and a number of operators, including comparisons and basic logic of **and**, **or** and **not**. With this foundation in logic, we can introduce comparisons in *Making Decisions : The Comparison Operators*.

The basic tools of logic and comparison are the essential ingredient to looking at conditional processing in *Processing Only When Necessary : The if Statement*. Conditional processing is controlled by the **if** statement, and reflects processing that only makes sense when a condition is true.

The other side of this is iterative processing, which we'll cover in *While We Have More To Do : The for Statement*. Iterative processing can depends on a condition – similar to an **if** statement. We'll look at this in *While We Have More To Do : The while Statement*.

We'll cover a number of additional topics *Becoming More Controlling*. This includes the **break** , **continue** and **assert** statements to provide a finer level of control over the processing. Additionally, we'll look at many of the traps and pitfalls associated with iterative processing.

In *Comments and Scripts* we'll return to scripting to make our scripts fit more smoothly with our operating system. We'll look at a complete family tree of processing alternatives using the command line as well as the GUI. There are a number of operating-system specific variations on this theme, and we can't easily cover every alternative.

6.1 Truth and Logic : Boolean Data and Operators

In *Arithmetic and Expressions* we looked at various types of numeric data, including whole numbers and floating-point numbers. We showed all of the arithmetic operations that we could apply to those various kinds of numbers. In *Truth*, we'll introduce another type of data to represent truth. In *Logic* we'll manipulate those truth values with logic operators.

Mathematically, this is the essential foundation of all computing. Pragmatically, we've pushed it back to here where it made more sense.

6.1.1 Truth

The domain of arithmetic involves a large number of values: there are billions of integer values and an almost unlimited range of long integer values. There are also a wide variety of operations, including addition, subtraction, multiplication, division, remainder and others, too numerous to mention. The domain of logic involves two values: `False` and `True`, and a few operations like `and`, `or` and `not`.

This world of logic bridges language, philosophy and mathematics. Because we deal with logic informally all the time, it can seem needless to define a formal algebra of logic. Computers, however, are formally defined by the laws of logic, so we can't really escape these definitions. If you don't have any experience with formal logic, there's no call for panic: with only two values (true and false), how complex can the subject be? Mostly, we have to be careful to follow these formal definitions, and set aside the murky English-language idioms that masquerade as logic.

We also have to be careful to avoid confusing logic and rhetoric. A good public speaker often uses rhetorical techniques to make their point. In some cases, the rhetoric will involve logic, but other times, it will specifically avoid logic. One example is to attack the speaker personally, rather than attack the logic behind the point they're trying to make. Political debates include many examples of rhetorical techniques that have a certain kind of logic, but aren't grounded in the kind of formal mathematical logic that we're going to present here.

Truth. Python has a number of representations for truth and falsity. While we're mostly interested in the basic Python literal of `False` and `True`, there are several alternatives.

- `False`.

  Also `0`, the complex number `0+0j`, the special value `None`, zero-length strings `""`, zero-length lists `[]`, zero-length tuples `()`, and empty mappings `{}` are all treated as `False`. We'll return to these list, tuple and map data structures in later chapters. For now, we only need to know that a structure that is empty of meaningful content is effectively `False`.

- `True`.

  Anything else that is not equivalent to `False`. This means that any non-zero number, or any string with a length of one or more characters are equivalent to `True`.

What about "maybe's" and "unknown's"? You'll need a good book on more advanced logic systems if you want to write programs that cope with shades of meaning other than simple true and false. This kind of fuzzy logic isn't built in to Python. You could write your own extension module to do this.

The bool Function. Python provides a factory function to provide the truth value of any of these objects. In effect, this collapses any of the various forms of truth down to one of the two explicit objects: `True` or `False`.

`bool`(*object*) → boolean

  Returns `True` when the argument *object* is equivalent to true, `False` otherwise.

We can see how this works with the following examples.

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool( "a string" )
True
```

Historical Note

Historically, Python didn't have the boolean literals `True` and `False`. You may find older open-source programs that define variables to have values that mean `True` and `False`. You might see a cryptic dance that looks something like the following:

```
try:
    True, False
except NameError:
    True, False = (1==1), (1==0)
```

This little trick is no longer necessary. We present it here so that you won't be surprised by seeing it in an open source package you're reading.

6.1.2 Logic

Python provides three basic logic operators that work on the domain of `True` and `False` values: `not`, `and` and `or`. This domain and the applicable operators forms a complete algebraic system, sometimes called a Boolean algebra, after the mathematician George Boole.

In Python parlance, the data values of `True` and `False`, plus the operators `not`, `and` and `or` define a *data type*. In *Simple Arithmetic : Numbers and Operators* we saw a number of numeric data types, and we'll look at yet more data types as we learn more about Python.

Truth Tables. The boolean data type has only two values, which means that we can define the boolean operators by enumerating all of the possible results in a table. Each row of the table has a unique combination of `True` and `False` values, plus the result of applying the logic operator to those values. There are only four combinations, so this is a pretty tidy way to define the operators.

We wouldn't want to try this for integer multiplication, since we have almost four billion integer values (including both negative and positive values), which would lead to a table that enumerates all 18 quintillion combinations.

Here's an example of a *truth table* for some hypothetical operator we'll call `cake`. Rather than show `and`, `or` or `not` specifically, we'll use a made-up operator so we can show how a truth table is built.

This table shows all possible results for x cake y. It shows all four combinations of inputs and the result of applying our logic operation to those values.

x	y	x cake y
`True`	`True`	`True` cake `True` = `False`
`True`	`False`	`True` cake `False` = `True`
`False`	`True`	`False` cake `True` = `True`
`False`	`False`	`False` cake `False` = `False`

The not Operator. The following little program creates a truth table that shows the value of `not` x for both vales of x. It may seem silly to take such care over the obvious definition that `not True` is `False`. However, we can use this technique to help us visualize more complex logical operations.

```
from __future__ import print_function
print("x", "not x")
print(True, not True)
print(False, not False)
```

x	`not` x
`True`	`False`
`False`	`True`

The and Operator. This next little program creates a truth table that shows the value of x `and` y for all four combination of `True` and `False`. You can see from this table that x `and` y is only `True` if both of the terms are `True`. This corresponds precisely to the English meaning of "and".

```
from __future__ import print_function
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`

The or Operator. The following table shows the evaluation of x `or` y for all four combination of `True` and `False`. You can see from this table that x `or` y is `True` if one or both of the terms are '`True`'. In English, we often emphasize the inclusiveness of this by writing "and/or" . We do this to distinguish it from the English-language "exclusive or", (sometimes written "either/or") which means "one or the other but not both". Python's x `or` y is the inclusive sense of "or".

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

An important note is that `and` is a higher priority operator than `or`, analogous to the way multiplication is higher priority than addition. This means that when Python evaluates expressions like `a or b and c`, the `and` operation is evaluated first, followed by the `or` operation. This is equivalent to `a or (b and c)`.

Tip: Debugging Logic Operators

The most common problem people have with the logic operators is to mistake the priority rules. The lowest priority operator is **or**. **and** is higher priority and **not** is the highest priority. If there is any confusion, extra parentheses will help.

Other Operators. There are – theoretically – more logic operators. However, we can define all of other the other logic operations using just `not`, `and` and `or`. Other logic operations include things like "if-then", "if-and-only-if". For example, "if a then b" can be understand as `(a and b or not a)`.

One of the more important additional logic operations is "or in an exclusive sense", sometimes called one-or-the-other-but-not-both or exclusive or, abbreviated **xor**. We can understand "a xor b" as `((a or b) and not (a and b))`. The parenthesis are required to create the correct answer.

How can we prove this? Write a short program like the following:

```
from __future__ import print_function
a, b = True, True
print(a, b, ((a or b) and not (a and b)))
a, b = True, False
print(a, b, ((a or b) and not (a and b)))
```

You'll have to repeat this for `False, True` and `False, False` combinations, also.

The claim that we can define *all* logic operations using only `not`, `and` and `or` is a fairly subtle piece of mathematics. We'll just lift up a single observation as a hint to how this is possibly true. We note that given two values and an operation, there are only four combinations of values in the truth table. There are only 16 possible distinct tables built from four boolean values. The logic puzzle of creating each of the 16 results using only `not`, `and` and `or` isn't terribly hard. For real fun, you can try constructing all 16 results using only the not-and operator, sometimes called "nand".

6.1.3 Exercises in Truth

[I love that title.]

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 *Basic Sequential Collections of Data*. For each of the following, work out the value according to the truth tables and the evaluation rules. Since each of the boolean values 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'`

2. **Exclusive Or**.

   Python's **or** operator is the inclusive or, sometimes written "and/or" in English. The exclusive or has a more formal meaning of *x* or *y but not both*. This phrase "but not both" can be implemented as a logic test.

   Remember, first, that "but" means "and". This gives us a hint on how we can proceed. We'll need to write an expression that starts `(x or y) and ...`. What does "both" mean in this context? Would `x and y` implement what we mean by both?

   Does `(x or y) and not (x and y)` create the correct truth table for "exclusive or"?

3. **Not And**.

   Engineers, in an effort to save money creating digital logic, have determined that the not-and operation can be used for a variety of purposes. The engineers call it "nand". We don't really have a proper English phrase for nand, but we can write the nand in Python as `not (a and b)`.

   Create a truth table for the basic nand operation (`not (a and b)`), showing all four results.

   Create a truth table for `not (a and a)`. What truth table does this match?

   What happens when we combine these operations? If we wanted to do the equivalent of something really complex like "(a nand b) nand (a nand b)", we have to do some algebra, resulting in something like `not ((not (a and b)) and (not (a and b)))`. What truth table does this match?

4. **If and Only If**.

   In English, we'll sometimes use the phrase "if and only if", which we might want to abbreviate "iff". When we look at the formal meaning of a hypothetical '`x iff y`', we need it to be true when *x* and *y* have the same truth value. This means that *x* and *y* are both true or *x* and *y* are both false.

   Does `(x and y) or (not x and not y)` create the correct truth table for "if and only if"?

   We haven't covered the `==` operator, but you should also try `x == y` to see if this also works.

5. **Implies**.

   The word implies has a formal logic definition. We say "a implies b" as a short form of "if a, then b". We might say "rain implies a wet lawn", or "if it rains, then the lawn gets wet". In Python, we might want to write '`a implies b`' if Python had a logic operator named "implies". When we look at the formal meaning of our hypothetical '`x implies y`', we want it to be true when x and y are true. When x is false, the truth or falsity of y doesn't really matter. We can say that implication is true when both x and y are true or x is false.

   Does `(x and y) or (not x)` create the correct truth table for "implies"?

6.1.4 Truth and Logic FAQ

Why are there alternatives to the values True and False? In other words, why are all non-zero values True? Is it confusing to have so many alternatives to True?

There are two schools of thought on this subject:

- Boolean data is a *first class* data type, unique and distinct. There should be one – and only one – value for True.
- Boolean operations work just fine on `1` and `0`, don't clutter the language with a specialized type that only has two values.

Boolean data is a unique type of data: it has a unique domain of values and unique operators. The domain is really tiny (`True` and `False`), but it is a proper mathematical domain with as many interesting properties as whole numbers, rational numbers or irrational numbers. For this reason, it deserves its own data type.

Claiming that the values of `True` and `False` are really just aliases for `1` and `0` misses two important points. First, from a historical perspective, the computer engineers borrowed Boole's algebra of logic and used it to build computer circuits. The proper historical context shows us that the engineers figured out how to use the ideas of `True` and `False` to build electronic circuits that could be interpreted as meaning `1` and `0`.

Second, and more important, one common approach to avoiding a boolean type is to use the special operators (*Operators for Bit Manipulation*). This doesn't eliminate the boolean type, it just eliminates plain boolean literals. We have a domain of values (`1` and `0`) and a suite of operators (`&`, `|`, `^`, and `~`) on that domain of values. This creates an ambiguity over the meaning of `1`: does it mean the number one or `True`?

When talking to another intelligent human being, this many appear as needless fussiness over something like truth that is common sense. However, we're not talking with people, we're writing a program in the formal language of Python which will control a mindless collection of transistors. We need to be as precise and inflexible as the circuitry in our computer.

There are two boolean values, 256 unique byte values, 4 billion unique integers. How many unique floating-
The "almost unlimited" at the beginning of the chapter was unsatisfyingly vague.

The domain of values for floating-point numbers is technically finite. The domain depends, to a small extent, on your computer. We'll assume 64-bit floating point numbers. These have 2^{64} distinct values, which is 18.4 quintillion. These values, however, are spread over a range that includes 2^{-1023} (approximately 10^{-308}) as the number closest to zero, and 2^{1024} (approximately 10^{308}) as the number furthest from zero.

Some computers have 80-bit floating-point numbers. The ranges in this case would obviously be somewhat larger.

6.2 Making Decisions : The Comparison Operators

Comparisons are the heart of decision-making. Decision-making is the heart of controlling what our programs do. We'll start off looking at the basic comparison operators in *Greater Than? Less Than?*. We'll look at some more advanced comparison techniques in *More Sophisticated Comparisons*.

We'll take second look at how the logic operators of `and` and `or` work in *Taking Other Short-Cuts*.

We'll answer some additional questions in *Comparison FAQ*.

6.2.1 Greater Than? Less Than?

Ordinary arithmetic operators are functions that map some numbers to another number. For example, addition maps two numbers to the number which is their sum: 3+5 maps to 8. Similarly, multiplication maps two numbers to their product, :we could say math:*3 times 5 mapsto 15*.

A comparison maps two numbers to a boolean value that reflects the relationship between the numbers. For example, $3 < 5 \mapsto$ True because 3 is less than 5; $3 \geq 5 \mapsto$ False for the same reason.

We compare values with the comparison operators. Here are the comparisons that Python recognizes:

- Less than ($<$) is `<`.
- Greater than ($>$) is `>`.
- Less than or equal to ($\leq$) is `<=`.
- Greater than or equal to ($\geq$) is `>=`.
- Equal to ($=$) is `==`.
- Not equal to ($\neq$) is `!=`.

Why is `==` used to test for equality? The problem is that mathematicians use the symbol $=$ pretty freely, but we have to provide more formal definitions. Python uses `=` for the **assignment** statement (*The Assignment Statement*). To distinguish between assignment and comparison, Python uses `==` to mean comparison.

Here are some examples. You can see that a comparison produces a boolean result of either `True` or `False`.

```
>>> 10 > 2
True
>>> 10 >= 9+1
True
>>> 10 >= 9+2
False
>>> 1 == 2
False
>>> 1 != 2
True
```

The = and == Problem. Here's a common mistake. We've used a single `=` (assignment), when we meant to use `==` (comparison). We get a syntax error because we have a literal `99` on the left side of the `=` statement.

```
>>> 99 = 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

An Unexpected Coercion. Here's a strange thing that can happen because of the way Python converts between numeric type data and boolean type data. First, look at the example, then try to figure out what happened.

```
>>> (10 >= 9) + 2
3
```

Because of the `()`, the `10 >= 9` is evaluated first. What is the result of that comparison?

How can it make sense to compute a sum of a boolean value (`True` or `False`) and a number? It doesn't, really, but Python tries anyway. It must have converted the boolean result of `10 >= 9` to a number. Try the following to see what has happened.

```
>>> (10 >= 9) + 2
3
>>> 10 >= 9
True
>>> int( 10 >= 9 )
1
>>> int( 10 >= 9 ) + 2
3
```

Tip: Debugging Comparison Operators

The most common problem people have with the comparison operators is to attempt to compare things which cannot meaningfully be compared. They ask, in essence, "which is larger, the Empire State Building or the color green?" An expression like `123 < 'a'` doesn't really make a lot of sense, even though it is legal Python.

Python copes with senseless comparisons using it's coercion rules. In this case, the number `123` is coerced to a string `'123'`, and the two strings are compared using the ordinary alphabetical order rules. In this case, digits come before letters and any number will be less than any word.

Sorting out the rules for coercion and comparison can be very confusing. Consequently, it should be avoided by exercising reasonable care in writing sensible programs. You should inspect your program to be sure you are comparing things sensibly. You can also put in explicit conversions using the various factory functions in *Functions are Factories (really!)*.

6.2.2 More Sophisticated Comparisons

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

Here's an example of checking for the "middle twelve" in Roulette. Since the number is random, your results may vary.

```
>>> import random
>>> spin= random.randrange(38)
>>> 13 <= spin <= 24
False
>>> spin
12
```

Writing `13 <= somethingComplex <= 24` instead of `13 <= somethingComplex and somethingComplex <= 24` is particularly useful when *somethingComplex* is actually some complex expression that we'd rather not repeat.

Proper Floating-Point Comparison. Exact equality between floating-point numbers is a dangerous concept. During a lengthy computation, round-off errors and conversion errors in floating-point numbers will accrue a tiny error term. In *Better Arithmetic Through Functions*, we saw answers that were off in the

15th decimal place. These answers are close enough to be equal for all practical purposes, but one or more of the 64 bits may not be identical.

The following technique is the appropriate way to do the comparison between floating-point numbers *a* and *b*.

```
abs(a-b)/a<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.

floatequal.py

```
# Are two floating-point values really completely equal?
from __future__ import print_function
a,b = 1/3.0, .1/.3
print(a, b, a==b)
diff= abs(a-b)/a
print(diff, diff < 0.0001)
```

When we run this program, we get the following output

```
$ python floatequal.py
0.333333333333 0.333333333333 False
1.66533453694e-16 True
```

The two values appear the same when printed. Yet, on most platforms, the == test returns `False`. They are not *identical* values. They differ at the 16th digit past the decimal point.

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 most important part of avoiding the problem is never to compare floating-point numbers for exact equality.

6.2.3 The Same Thing or The Same Value? The is Comparison

Python makes an important (but subtle) distinction between the following two questions:

- Do two objects have the same value?
- Are two objects references to the same thing?

Consider the following

```
a = 123456789
b = a
```

The variable *a* refers to `123456789`. The variable *b* also refers to the same object.

When we evaluate the following, the results aren't surprising.

```
>>> a=123456789
>>> b=a
>>> a is b
True
```

```
>>> a == b
True
```

The `is` operator tells us that variable *a* and variable *b* are two different labels attached to the same underlying object.

The `==` operator tells us that variable *a* and variable *b* refer to objects which have the same numeric value. In this case, since `a is b` is `True`, it's not surprising that `a == b`.

Not the Same Thing. In most cases, however, we'll have situations like the following. We'll create two distinct objects that have the same numeric value.

```
>>> a=123456789
>>> c=a*1
>>> c is a
False
>>> c == a
True
>>> c is not a
True
```

In this example. we've evaluated a simple operator (`*`), which created a new object. We know it's a new object because `c is a` is `False` (also, `c is not a` is `True`). However, this new object has the same numeric value as *a*.

Common Use. The most common use for **is** and **is not** is when comparing specific objects, not generic numeric values. We do this mostly with the object `None`.

```
>>> variable = None
>>> variable
>>> variable is None
True
>>> anotherVariable = 355/113.0
>>> anotherVariable is None
False
>>> anotherVariable
3.1415929203539825
```

6.2.4 Comparison Exercises

1. **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).

2. **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 conditions under which the field pays.

6.2.5 Comparison FAQ

Why do the logical operators short-circuit? Isn't that an egregious violation of the eval-apply cycle?

In *Execution – Two Points of View* we emphasized the evaluate-apply cycle like it was a natural law or divine writ, direct from the almighty. Yet, here we're saying that the `and` and `or` operators violate this law.

First, we're only bending the law. The essential principle is still followed, we're just extending the rule a little: all of the *logically necessary* parts of an expression are evaluated first.

The alternatives to the short-circuit sense of `and` and `or` are either much more complex logic operators (`and`, `or`, `cand` and `cor`) or decomposing relatively simple logic into a complex sequence of statements, using a **if** statement instead of a short-circuit logic operator.

The objective of software is to capture knowledge of processing in a clear and formal language. Fussy consistency in this case doesn't help achieve clarity.

6.3 Advanced Logic Operators

There are two advanced logic operators we need to look at. In *The Conditional Expesssion* we'll look at the conditional expression. In *Taking Other Short-Cuts* we'll look at the "short-cut" nature of **and** and **or**.

These operators are rule-benders: they behave in a slightly different way than all other Python operators.

6.3.1 The Conditional Expesssion

Python has a really fancy logic operator, called the "conditional expression" that looks like this.

```
trueValue if condition else falseValue
```

This is a three-part operator that has a condition (in the middle) and two values. If the *condition* is `True`, then the value of the entire expression is the *trueValue*. If the *condition* is `False`, then the value of the entire expression is the *falseValue*.

Note: Terminology

Sometimes you'll hear this called "the ternary operator". This is a confusing name, and shouldn't be used. It's *a* ternary operator: there are three arguments. All other operators are unary (`-a`) or binary (`a+b`).

This happens to be the only ternary operator. There's no reason for calling it *the* ternary operator because others could be fashioned.

Here's an example. Let's say we're monitoring the temperature of a walk-in cooler.

```
status = "in range" if -5 <= freezer <= 0 else "problem"
```

If the temperator is between -5 and 0, the status is "in range". If the temperature is outside the range, the label will be "problem".

This is strictly a two-way true-false comparison. If you need more than two simple choices, you'll need something more sophisticated like the complete **if** statement, *Processing Only When Necessary : The if Statement*.

Rule-Bender. This violates the basic rule we defined in *The Evalute-Apply Rule*. The general rule applies almost everywhere else: every expression is fully evaluated. This conditional expression bends this rule, however, to limit evaluation to the logically necessary sub-expressions rather than every single sub-expression.

Python always evaluates the condition. It then evaluates one of the two other expressions. The remaining expression is not evaluated.

Here's another example.

```
average = float(sum)/count if count != 0 else 0.0
```

In this case, the expression works like this.

1. Python evaluates `count != 0`.
2. If `count != 0` is `True`, then the value of the expression is the value of `float(sum)/count`.

   If `count != 0` is `False`, then the value of the expression is the literal `0.00`.

This short-cut prevents us from getting an error when trying to compute the average of a set of zero values.

6.3.2 Taking Other Short-Cuts

An important feature of the `and` and `or` operators is that they do not evaluate all of its parameters before they are applied.

In the case of `and`, if the left-hand side is equivalent to `False`, the right-hand side is not evaluated, and the left-hand value is returned.

For now, you can try things like the following.

```
>>> False and 0
>>> 0 and False
```

This will show you that when the left-side value is equivalent to `False`, that is what Python returns for **and**. The other value isn't even evaluated.

Try this.

```
>>> import math
>>> False and math.sqrt(-1)
>>> False and 22/0
```

What happens?

The `and` operator doesn't evaluate the right-hand parameter if the value on the left-hand side is `False`.

The `or` operator, similarly, does not evaluate the right-hand parameter if the left-hand side is equivalent to `True`.

Rule-Bender. This violates the basic rule we defined in *The Evalute-Apply Rule*. The general rule applies almost everywhere else: every expression is fully evaluated. The `and` and `or` operators bend this rule, however, to limit evaluation to the logically necessary sub-expressions rather than every single sub-expression.

This short-circuit can be useful. This is an example of the "practicality beats purity" principle that makes Python so cool.

Simplifications. Let's look at the informal logic of English for a moment. When sailing, we might say "if the wind is over 15 knots, we reef the main sail." Reefing, for non-sailors, is a technique for reducing the area of the sail; we do this for a variety of reasons, for example, so that the boat doesn't lean over ("heel") too far in a high wind.

One important consequence of the short-cut rule is that the Python logic operators are very handy for creating a simple, clear statement of some sophisticated processing. One of the most notable examples of this are expressions like the following which summarize our sailing rule very nicely.

```
full = 72 # square feet
reefed = 65 # square feet
sailArea= reefed if windSpeed > 15 else full
```

One More Condition. What if we are motoring, not sailing? In English, we can say something like "when sailing *and* the wind is over 15 knots, we reef the main sail." The implication is that when we are motoring, the wind-speed comparison is irrelevant. After all, it is a bit silly to also check the wind speed when we don't even have the sails up.

If we don't think carefully, we would wind up with the following and incorrect set of conditions.

Engine	Conditions	Configuration
Sailing	Wind Speed <= 15 kn	full
	Wind Speed > 15 kn	reefed
Motoring	Wind Speed <= 15 kn	full
	Wind Speed > 15 kn	reefed

The table above is silly because motoring makes the wind speed and sail positions irrelevant. Why are we checking them needlessly?

Clarification. This table has what we really meant. This clearly states that when we're motoring, we don't need to check the wind-speed.

Engine	Conditions	Configuration
Sailing	Wind Speed <= 15 kn	full
	Wind Speed > 15 kn	reefed
Motoring	Doesn't matter	None

This English-language sense of "and-only-when-it's-relevant" is a handy simplification. This is common in English, and the reason why the Python operators include this same short-cut sense.

We might express this with something like the following in a Python program:

```
sailArea= (reefed if windSpeed > 15 else full) if engine == "Sailing" else None
```

This reflects the "decision tree" in our table. The overall condition is the `if engine == "Sailing"` check. If we're sailing, then there's a second check based on the wind speed.

We can test this with little scripts like the following:

```
>>> engine, wind = "Sailing", 10
>>> (reefed if windSpeed > 15 else full) if engine == "Sailing" else None
72
>>> engine, wind = "Motoring", 25
>>> (reefed if windSpeed > 15 else full) if engine == "Sailing" else None
>>>
```

When sailing in light air (10 knots), we should have the full sail, all 72 square feet deployed.

When motoring in a stiff breeze (25 knots), we should have no sail.

6.3.3 Logic and Comparison Exercises

1. **Develop an "or-guard"**.

   This is the "and-guard" pattern. It guards the division operation, by comparing the divisor with zero. If *count* really is zero, this returns `False` rather than attempting an illegal division.

   ```
   average = count != 0 and float(sum)/count
   ```

   Develop a similar technique using **or** instead of **and**. This will require some reversals of the logic in the above example. We can interpret it as doing a comparison **or** a calculation. If the first clause (the comparison) is false, we want to continue on to the next clause (the calculation).

   This is an application of De Morgan's Laws.

2. **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.

6.4 Processing Only When Necessary : The if Statement

We'll address the meaning of conditional processing in *Conditional Processing*. We'll present the basic Python **if** statement in *The if Statement*. We'll look at some examples in *Example if Statements*.

We'll add a number of conditional processing features in *The elif Condition for Alternatives*, *The else Condition as a Catch-All* and *The pass Statement: a Do-Nothing*.

6.4.1 Conditional Processing

The programs we've seen so far have performed a sequence of steps, unconditionally. We're going to introduce some real sophistication by making some of those steps conditional. We'll start by looking at what it takes to design a sequence of steps that does what we want.

Back in *Where Exactly Did We Expect To Be?* we talked about the significance of putting a variable name on an object. The values we assign to our variables define the program's state of being. Changing a variable's value changes the program's state. When we've planned our program well, each state change moves our program from a nebulous starting state toward the well-defined finished state.

Let's look at converting Celsius temperatures to Fahrenheit temperatures. We'll work backwards from the ending.

- Goal: we need to have both C and F variables defined, and the values must satisfy an equation that maps between the two temperatures. How do we get to this well-defined final state?
- Final Step: Compute the value for F. To do this, we'll need the value for C. Then we can use a formula from the exercises in *Expression Exercises* to set the desired value for F.
- Precondition: Get the value for C. When can get the value for C by using the `input()` function. What's the precondition for getting C?
- Initial Condition. Any initial values of F and C are valid for the start of this program.

When we reverse this list of goals, we have the *algorithm* for computing the Fahrenheit temperature from the Celsius temperature.

This example shows *sequential* execution, where each step is unconditional. Sometimes the processing in a given step depends on a condition being met. When we are planning our programs, we may have to choose one of several statements depending on the data values or the processing goal. We call this *conditional* processing, which is the subject of this chapter.

The next state may depend on a condition being true for *all of* a set of data or finding a condition true for *some of* a set of data. We call this *iterative* processing, and that's the subject of the *While We Have More To Do : The for Statement* chapter.

6.4.2 The if Statement

Many times a program's exact processing or state change depends on a condition. Conditional processing is done by setting statements apart in groups called *suites* that have conditions attached to the suites. The Python syntax for this is an **if** statement. The **if** embodies this semantics of "if a condition is true, execute the suite of statements" .

The basic syntax of an **if** statement looks like this:

```
if expression :
    suite
```

The word **if** and the **:** are essential syntax. The *suite* is an indented block of one or more 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, and we often set BBEdit or TextPad to treat the `tab` key on our keyboard as four spaces.

This is the first *compound statement* we've seen. A compound statement statement makes use of the essential syntax rules we looked at in *Long-Winded Statements*. It also uses two additional syntax rules, that we'll look at next.

Semantics. The **if** statement evaluates the condition *expression* first. When the result is `True`, the *suite* of statements is executed. Otherwise the suite is skipped. Let's look at some examples.

6.4.3 Example if Statements

We'll look at some examples of **if** statements to see some additional examples of how they work. We'll take a couple of examples from the rules for Craps.

Is This Craps?. During the game of Craps, once a point is established, a roll of 7 is "craps", a loser. The following example combines arithmetic expressions, comparison and the **if** statement.

The variables d1 and d2 are set randomly, so each time you run this it may behave differently.

```
d1, d2= random.randint(1,6), random.randint(1,6)

if d1+d2 == 7:
    print("craps")
```

Here's how this **if** statement works.

1. The expression `d1+d2 == 7` is evaluated.
2. The expression `d1+d2` is evaluated to get an integer sum.
3. The `7` doesn't need to be evaluated.
4. The comparison is performed to see if the `d1+d2` sum really is equal to `7`. The result is either `True` or `False`.
5. If the expression's value is `True`, the suite is executed. This will print the message.

   If the value is `False`, the suite is silently skipped.

Come Out Roll. Here's a second example from the game of Craps. On the first roll, the come out roll, if two dice show a total of 7 or 11, the throw is a winner.

```
d1, d2= random.randint(1,6), random.randint(1,6)

if d1+d2 == 7 or d1+d2 == 11:
    print("winner", d1+d2)
```

Here we have a typically complex expression. Here's how the **if** statement works.

1. The expression `d1+d2 == 7 or d1+d2 == 11` is evaluated.

   The `or` operator evaluates the left side of the `or` operation first; if this is `False`, it will then evaluate the right side. If the left side is `True`, the result is `True`.

2. The expression `d1+d2 == 7` is evaluated.
3. If this value is `True`, (*d1* + *d2* really is 7), the entire `or` expression is `True` and evaluation of the expression is complete.
4. If the left side is `False`, then the right side is evaluated. The value of the right side (`d1+d2 == 11`) is the value for the entire `or` operation. This could be `True` or `False`.
5. If the value of the expression is `True`, the suite is executed, which means that a message is printed.

   If the expression is `True`, the suite is skipped.

Syntax Help from IDLE. The suite of statements inside the **if** statement is set apart from other statements by its indentation. This means you have to indent the statements in the suite consistently. Any change to the indentation is, in effect, another suite or the end of this suite.

The good news is the **IDLE** knows this rule and helps us by automatically indenting when we end a line with a `:`. It will automatically indent until we enter a blank line. Here's how it looks in **IDLE**. In order to show you precisely what's going on, we're going to replace normally invisible spaces with _ characters.

```
>>>_if_1+2_==_3:
..._____print("good")
...
good
```

1. You start to enter the **if** statement. When you type the letter `f`, the color of `if` changes to orange, as a hint that **IDLE** recognizes a Python statement. You hit `enter` at the end of the first line of the **if** statement.
2. **IDLE** indents for you. You type the first statement of the suite of statements.
3. **IDLE** indents for you again. You don't have any more statements, so you hit `enter`. The statement is complete, so **IDLE** executes the statement.
4. This is the output. Since 1+2 does exactly equal 3, the suite of statements is executed.

Important: Syntax Rule Eight

Compound 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 _.

```
a=0
if_a==0:
____print("a_is_zero")
else:
____print("a_is_not_zero")
```

Here's an example with spaces shown via _ and tabs shown with :

```
if_a%2==0:
 print("a_is_even")
else:
 print("a_is_odd")
```

While the tab character is allowed, spaces are preferred. Many experience Python programmers set their text editors to replace tab characters with four spaces.

Important: Syntax Rule Seven

When using Python interactively, an entirely blank line ends a multi-line compound statement.

Note that if you're using another editor (BBEdit, TextPad, etc.) you won't get the same level of automatic help that you get from **IDLE**. Other tools can provide syntax coloring and remember your indentation, but they don't all automatically indent when you end a line with a `:` the way **IDLE** does.

6.4.4 The elif Condition for Alternatives

Often there are several alternatives conditions that need to be handled. We encounter this when we have a series of rules that apply to a situation.

A good example is from Roulette, when we have a spin that could be even, odd or zero. We have a situation like the following.

- If the number 0 or 00, bets on even or odd are losers.
- Else, if the number is even (the remainder when divided by 2 is equal to 0), bets on even are winners.
- Else, if the number is odd (the remainder when divided by 2 is equal to 1), bets on odd are winners.

This is done by adding **elif** clauses. This is short for "else-if". We can add an unlimited number of **elif** clauses. The syntax for the **elif** clause is almost identical to the initial **if** clause:

```
elif expression :
    suite
```

Semantics. Python treats the **if** and **elif** sequence of statements as a single, big statement. Python evaluates the **if** expression first; if it is True, Python executes the **if** suite and the statement is done; the **elif** suites are all ignored. If the initial **if** expression is False, Python looks at each **elif** statement in order. If an **elif** expression is True, Python executes the associated suite, and the statement is done; the remaining **elif** suites are ignored. If none of the **elif** suites are true, then nothing else happens

Complete Come Out Roll. Here is a somewhat more complete rule for the come out roll in a game of Craps:

```
d1, d2= random.randint(1,6), random.randint(1,6)

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)
```

Our **if** statement has two clauses.

1. We wrote the condition for winning on 7 or 11. If the first condition is true, Python executes the first suite (set *result* to `"winner"`), ignores the remaining **elif** clauses, and the entire **if** statement is complete.
2. If the first condition is false, then Python moves on to the **elif** condition. If that condition is true, Python executes the second suite (set *result* to `"loser"`), and the entire **if** statement is complete.

If neither condition is true, the **if** statement has no effect. The script prints the result, which will be `None`.

The Roulette Example. Here's the even-odd rule from Roulette. We have one subtlety in Roulette that we have to look at: the problem of zero and double zero. What we'll do is generate random numbers between -1 and 36. We'll treat the -1 as if it was 00, which is like 0, neither even nor odd.

```
from __future__ import print_function
import random
spin= random.randint(-1,36)

result= None
if spin == 0 or spin == -1:
    result= "neither"
elif spin % 2 == 0:
    result= "even"
elif spin % 2 == 1:
    result= "odd"
print(spin, result)
```

Our **if** statement has three clauses.

1. We wrote the condition for zero and double zero. If the first condition is true, Python executes the first suite (set *result* to `"neither"`), ignores the remaining **elif** clauses, and the entire **if** statement is complete.
2. If the first condition is false, then Python moves on to the first **elif** condition. If that condition is true, Python executes the second suite (set *result* to `"even"`), and the entire **if** statement is complete.
3. If the first **elif** condition is false, then Python moves on to the second **elif** condition. If that condition is true, Python executes the second suite (set *result* to `"odd"`), and the entire **if** statement is complete.

If none of these conditions is true, the **if** statement has no effect. The script prints the result, which will be `None`.

6.4.5 The else Condition as a Catch-All

Finally, there is the capability to put a "catch-all" suite at the end of an **if** statement, which handles all other conditions. This is done by adding an **else** clause. The syntax for the **else** clause it somewhat simpler.

```
else:
    suite
```

This clause is always last and, effectively, always `True`. When the **if** expression and all of the **elif** expressions are false, Python will execute any **else** suite that we provide.

Come Out Roll Script. Here's the complete come-out roll rule. In this final example, we've added the necessary **import** and **assignment** statements to make a complete little script.

comeoutroll.py

```
from __future__ import print_function
import random
d1,d2= random.randrange(1,7), random.randrange(1,7)
point= None
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"
else:
    point= d1+d2
    result=  "point is", point
print(result)
```

Here, we used 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 you write it our using **or**. We summarize this complex condition with the **else** clause.

Tip: Debugging the **if** statement.

If you are typing an **if** statement, and you get a `SyntaxError: invalid syntax`, you omitted the `:`.

A common problem with **if** statements is an improper condition. You can put any expression in the if or elif statement. If the expression doesn't have a boolean value, Python will use the `bool()` function to determine if the expression amounts to `True` or `False`. It's far better to have a clear boolean expression rather than trust the rules used by the `bool()` function.

One of the more subtle problems with the **if** statement is being absolutely sure of the implicit condition that controls the **else** clause. By relying on an implicit condition, it is easy to overlook gaps in your logic.

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's the subtle bug in this example. 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 "field pays even money" condition. 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 found that did not match by any of the **if** or **elif** clauses.

6.4.6 The pass Statement: a Do-Nothing

Once in a while you may have a situation where the logic is kind of tangled, and you want to say something like the following:

```
if a > 12:
    do nothing
elif a == 0:
    print("zero")
else:
    print("a between 1 and 12")
```

Unfortunately, the Python languages doesn't allow a suite of statements to be empty. We don't want to have to rearrange our program's logic to suit a limitation of the language. We want to express our processing clearly and precisely. Enter the **pass** statement.

The syntax is trivial.

```
pass
```

The **pass** statement does nothing. It is essentially a syntax place-holder that allows us to have a "do nothing" suite embedded in an **if** statement.

Here's how it looks.

```
if a > 12:
    pass
elif a == 0:
    print("zero")
else:
    print(a, "between 1 and 12")
```

If the value of *a* is greater than 12, the **if** statement's expression is true, and Python executes the first suite of statements. That suite is simply **pass**, so nothing happens.

If the value of *a* is zero, the first **elif** statement's expression is true, and Python executes the second suite of statements. That suite is a `print()` function, and we see a "zero" printed.

If none of the previous expressions are true, Python falls back to the **else** statement, in which case, we would see a message about *a* being between 1 and 12.

6.4.7 Condition Exercises

1. **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 $x \leq y \leq y \leq z$ 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 = X`, `x = Y` or `x = 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.

2. **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.

3. **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"

4. **Hardways 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.

5. **Partial Evaluation**.

   This partial evaluation of the **and** and **or** operators appears to violate the evaluate-apply principle espoused in *Execution – Two Points of View*. 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?

6.5 While We Have More To Do : The for Statement

We'll address the general need for iterative processing in *Iterative Processing*. There are a few specific kinds of iterations that we can identify, and we'll describe these in *Patterns of Iteration*. The most commonly-used Python iterative statement is presented in *The for Statement*. We'll look at some examples in *Multi-Dimensional Loop-the-Loop* and *Simulating All 100 Rolls of the Dice*.

6.5.1 Iterative Processing

A program may have a goal that is best described using the words "for all", where we have to do some calculation *for all* values in a set of values.

Let's look at creating a table of Celsius temperatures and their matching Fahrenheit temperatures. We only need useful temperatures between -20 ° C and 40 ° C. We'll work backwards from the ending.

- Goal: we need to print *all* C and F values; the values must satisfy an equation that maps between the two temperatures; the values for C are between -20 and 40.

  How do we get to this well-defined final state?

- Loop Condition: The value for C is between -20 and 44, and we have not computed and printed the value for F yet. When this condition is true, we have more work to do. When this is false, we have satisfied our *for all* condition.

  What work do we have to do that satisfies the rest of our goal? What precondition is required to make this true initially?

  – Iterative Step 3: Add 2 to the value of C. This satisfies part of our *for all* goal by creating a value of C for which we haven't computed an F.

    What else do we have to do?

  – Iterative Step 2: Print the values for C and F. This satisfies part of our *for all* goal by printing values of C and F.

    What's the precondition for printing C and F?

  – Iterative Step 1: Compute the value for F. To do this, we'll need the value for C. Then we can use a formula from the exercises in *Expression Exercises* to set the desired value for F.

- Precondition: Set the value for C to -20. This sets our Loop Condition to be true.

  What's the precondition for setting C?

- Initial Condition. Any initial values of F and C are valid for the start of this program.

When we reverse this list of goals, we have the *algorithm* for computing a mapping between the Fahrenheit temperature and the Celsius temperature.

We often call iterative or repetitive processing a "loop" because the program statements are executed in a kind of loop. Both the **for** and **while** statements provide a condition that controls how many times the loop is executed. The condition in the **for** statement is trivial, but the pattern is so common that it has a large number of uses. The condition in the **while** statement is completely open-ended, therefore it requires a little more care when designing the statement.

Iterative processing relies on all the elements of sequential and conditional processing that we've already seen. Iterative programming is the backbone of computing. First we'll look at three common kinds of iteration. Then we'll see how to write those kinds of iteration in Python.

6.5.2 Patterns of Iteration

There are three basic species of iterations: mapping, reducing and filtering. As with much of computer science, other words have been borrowed for these rather abstract ideas. Don't think of road maps, weight loss or clothes dryers. Think of mapping in the sense of mapping one value to another, reducing a set of values to a single value, and filtering unwanted values out of a set.

Mapping All Values In A Set. Perhaps the simplest kind of iterations is called a *mapping*. Our iteration maps all values in a set from some domain to some range. We see this kinds of mapping when we look at a Fahrenheit to Celsius conversion table, or a deciliters to cups table. Even an chart that maps the combination of air temperature and wind speed to a wind-chill temperature is a kind of mapping. We'll look at these kinds of iterations extensively in this chapter.

The **for** statement is ideal for performing a mapping that has to be done for all values in a set. The typical pattern for a mapping uses a Python **for** statement and one or more "compute mapped value" statements. It's a Python statement with a suite of statements, and has a general outline like the following:

```
For i is a value in some set:
    Compute mapped result based on i
```

Here's the result of a small program that produces a mapping from Swedish Krona (SEK) to US Dollars (USD); it's a currency exchange table. A Krona is worth about $0.125 right now. We used a Python "for all" loop to iterate through all values from 5 to 50 in steps of 5.

```
5 0.625
10 1.25
15 1.875
20 2.5
25 3.125
30 3.75
35 4.375
40 5.0
45 5.625
50 6.25
```

The result of a mapping is an output set of values; the size of the output set matches the size of the input set. In our example above, we have has many Krona values as Dollar values.

Reducing All Values To One Value. Another common kind of iteration is a *reduction* where all the values in a set are reduced to a single resulting value. When we add up or average a column of numbers, we're doing a reduction. As we look at our two representative problems (see *Two Minimally-Geeky Problems : Examples of Things Best Done by Customized Software*), we see that we will be simulating casino games and computing averages of the results of a number of simulation runs.

The **for** statement is ideal for performing reductions. The typical pattern for a reduction uses a "initializations", a "for all", and one or more statements to "update the reduction".

```
Initialize the Reductions
total= 0
count= 0
For i is a values in some set:
    Update the Reduction
    total += calculation based on i
    count += 1
```

The result of a reduction is a single number created from the input set of values. Common examples are the sum, average, minimum or maximum. It could also be a more sophisticated reduction like the statistical median or mode.

Filtering All Values To Find a Subset. The third common kind of iteration is a *filter* where the iteration picks a subset of the values from all values in a set. For instance, we may want a filter that keeps only even numbers, or only red numbers in Roulette.

In this case, we're introducing a condition, which makes a filter more complex than the map or reduce. A filter combines iteration and conditional processing.

There are two senses of filtering:

- Find all values that match the condition.
- Find some value that matches the condition. This is a slightly more complex case, and we'll return to it several times.

These are sometimes lumped under the category of "search". Search is so important, that several computer science books are on focused on just this subject.

When we look closely at the rules for Craps we see that a game is a kind of filter. Once the game has established a point, the rest of the game is a kind of filter applied to a sequence of dice roles that ignores dice roles except for 7 and the point number. We can imagine adding filter conditions; for example, we could add a filter to keep the dice rolls that win a hardways bet.

The **while** statement can be used for filtering. Additionally, the **break** and **continue** statements can simplify very complex filters. The typical pattern for a filter uses an "initialize", a "while not finished", "filter condition" and an "update the results".

```
Initialize the Results
result = None
While Not Finished Filtering:
    Filter Condition
    If condition based on i :
        Update the results
        result = ...
```

The result of a filter is a subset of the input values. It may be the original set of values, in the rare case that every value passes the filtering test. It may be a single value if we are searching for just one occurrence of a value that passes the filter.

6.5.3 The for Statement

The most common way to do a "for-all" mapping, reduction or filtering is with the **for** statement.

The **for** statement looks like this:

```
for variable in sequence :
    suite
```

The words **for** and **in** and the **:** are essential syntax. The *suite* is an indented block of statements. Any statement is allowed in the block, including indented **for** statements.

There are a number of ways of creating the necessary *sequence* of values. The most common way to create a sequence is to use the `range()` function. First we'll look at the **for** statement, then we'll provide a definition for the `range()` function.

Printing All The Values. This first example uses the `range()` function to create a sequence of six values from 0 to just before 6. The **for** statement iterates for all values of the sequence, assigning each value to the local variable *i*. For each of six values of *i*, the suite of statements inside the for statement is executed.

The suite of statements is just a `print()` function, which has an expression that adds one to *i* and prints the resulting value.

```
for i in range(6):
    print(i+1)
```

We can summarize this as "for all *i* in the range of '0 to one before 6', print *i* +1".

Using A Literal Sequence Display. We can also create the sequence manually, using a literal *sequence display*. A sequence display looks like this: [*expression* **, ...**]. It's a list of expressions; for now they should be numbers separated by commas. The square brackets are essential syntax for marking a sequence. We'll return to sequences in *Basic Sequential Collections of Data*.

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()` function 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")
```

Summing All The Values. The second example sums a sequence of five odd values from 1 to just before 10. The **for** statement iterates through the sequence, assigning each value to the local variable *j*. The `print()` function prints the value.

```
sum= 0
for j in range(1,5*2,2):
    sum += j
print(sum)
```

The `range()` Function. The `range()` function has two optional parameters, meaning it has three forms.

range(x) → sequence
    Generates values from 0 to x-1, incrementing by 1.

range(x, y) → sequence
    Generates values from x to y -1, incrementing by 1. Each value, v will have the property $x \leq v < y$.

range(x, y, z) → sequence
    Generates values from x to y - z, incrementing by z. The values will be $x, x + z, x + 2z, \ldots, x + kz$, where $x + kz < y$.

From this we can see the following features of the `range()` function. If we provide one value, we get a sequence from 0 to just before the value we provided. If we provide two values we get a sequence from the starting value to one before the ending value. If we provide three values, the third value is the increment between each value in the sequence.

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> range(1,7)
[1, 2, 3, 4, 5, 6]
>>> range(1,11,2)
[1, 3, 5, 7, 9]
```

Summary. The **for** statement encapsulates three pieces of information.

- The name of a target variable. This variable will be set to a new value for each iteration of the loop.
- A sequence of values that will be assigned to the target variable. We can provide the values using a list display (`[1, 2, 3]`), or we can provide the values using the `range()` function.
- A suite of one or more statements. The phrase "one or more" means that statements are not optional.

Tip: Debugging the **for** Statement

If you are typing a **for** statement, and you get a `SyntaxError: invalid syntax`, you omitted the `:`.

The most common problem is setting up the sequence properly. Very often, this is because of the complex rules for the `range()` function, and we have one too many or one too few values.

A less common problem is to misspell the variable in the **for** statement or the *suite*. If the variable names don't match, the **for** statement will set a variable not used properly by the suite. An error like `NameError: name 'j' is not defined` means that your suite *suite* expected *j*, but that was not the variable on your **for** statement.

Another problem that we can't really address completely is writing a **for** statement where the *suite* doesn't do the right thing in the first place. In this case, it helps to be sure that the suite works in the first place. An execution trace (see *Where Exactly Did We Expect To Be?*) can help. Also, you can enter the statements from the suite separately to the Python shell to see what they do.

6.5.4 Multi-Dimensional Loop-the-Loop

Our previous examples have had one value which varies. Sometimes we'll have two (or more) values which vary. Here are some examples that have multiple variables.

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()` function will be executed 36 times to print the values of *d1* and *d2*.

```
for d1 in range(6):
    for d2 in range(6):
        print(d1+1, d2+1, '=', d1+d2+2 )
```

We can interpret this as a mapping from two dice to the sum of those two dice. This is a kind of two-dimensional table with one die going down the rows and one die going across the columns. Each cell of the table has the sum written in.

The output from this example, though, doesn't look like a table because it's written down the page, not across the page. To write across the page, we can make use of a feature of the `print()` function. We'll manually set the end-of-line to `','` or `'\n'`.

table.py

```
1 from __future__ import print_function
2 print("", "1", "2", "3", "4", "5", "6")
3 for d1 in range(1,7):
4     print(d1,end=' ')
5     for d2 in range(1,7):
6         print(d1+d2,end=' ')
7     print()
```

2. This is the first line of our table, showing the column titles.
3. Here we print the header for each row. Since this print sets *end* to `' '`, this does not print a complete line.
6. We print a single cell. Since this `print()` function sets *end* to `' '`, this does not finish the output line.
7. This `print()` function does not set *end*. The default value is `'\n'`. Therefore, this is the end of the line. The preceding row label and 6 values will be a complete line.

6.5.5 Simulating All 100 Rolls of the Dice

Here's a program 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*. It turns out that the suite of statements never actually uses the value of *i*, it is just bookkeeping for the state changes until the loop is complete.

We can summarize this as "for all 100 samples, set *d1* to be a random number between 1 and 6, set *d2* to be a random number between 1 and 6, print *d1* + *d2*".

roll100.py

```
from __future__ import print_function
import random
for i in range(100):
    d1= random.randrange(6)+1
    d2= random.randrange(6)+1
    print(d1+d2)
```

This previous example is a mapping from the sample number, (*i*) to two random dice (*d1*, *d2*), and then the two dice are mapped to a single sum.

We'll expand this simple loop to do some additional processing in *While We Have More To Do : The while Statement*.

6.5.6 For Statement Exercises

1. **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; that is total source lines divided by 1000.

   Development Effort, where K is the number of 1000's of lines of source. E is effort in staff-months.

$$E = 2.4 \times K^{1.05}$$

Development Cost, where E is effort in staff-months, R is the billing rate. C is the cost in dollars (assuming 152 working hours per staff-month)

$$C = E \times R \times 152$$

Project Duration, where E is effort in staff-months. D is duration in calendar months.

$$D = 2.5 \times E^{0.38}$$

Staffing, where E is effort in staff-months, D is duration in calendar months. S is the average staff size.

$$S = \frac{E}{D}$$

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.

2. **Wind Chill Table**.

   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 *Expression Exercises* for more information.

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

   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.

3. **Celsius to Fahrenheit 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).

   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).

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

4. **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 *Expression Exercises*.

   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.

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

6.6 While We Have More To Do : The while Statement

In *The while Statement* we'll look at the other iterative statement. We'll put together a big example in *Counting Sevens*.

We'll answer some questions in *Iteration FAQ*.

Later, in *Becoming More Controlling* we'll add some additional control to these statements. We have to set this more advanced material aside until we've learned more Python.

6.6.1 The while Statement

The **for** statement handles a number of patterns of iteration very nicely. However, it has a limitation. We must provide some iterable source of data (we showed the `range()` function, there are others.)

Sometimes, we don't have a tidy, simple iterable source for a **for** statement. Under those circumstances, we can use 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*. Should your program execute the suite of statements without changing any of the variables in the **while** *expression*, nothing will change, and the loop will not terminate.

There's an intentional parallelism between the **while** statement and the **if** statement. Both have a suite which is only executed when the condition is `True`. The **while** statement repeatedly executes the suite, where the **if** statement only executes the suite once.

100 Random Dice. Let's look at some examples. This first example is a revision of the last example in *The for Statement*; it shows that there is considerable overlap between the **while** statement and the **for** statement. Both can do similar jobs.

```
from __future__ imprort print_function
import random
sample= 0
while sample != 100:
    d1= random.randrange(6)+1
    d2= random.randrange(6)+1
    sample= sample + 1
    print(d1+d2)
```

This previous example is a mapping from the sample number, (*sample*) to two random dice, and then the two dice are mapped to a single sum.

Sum of Odd Numbers. Here's a more sophisticated example that computes the sum of odd numbers 1 through 9.

The loop is initialized with *num* and *total* each set to 1. We specify that the loop continues while $num \neq 9$. In the body of the loop, we increment *num* by 2, so that it will be an odd value; we increment *total* by *num*, summing this sequence of odd values.

When this loop is done, *num* is 9, and *total* is the sum of odd numbers less than 9: 1+3+5+7. Also note that the **while** condition depends on *num*, so changing *num* is absolutely critical in the body of the loop.

```
num, total = 1, 1
while num != 9:
    total= total + num
    num= num + 2
```

This example is a kind of reduction. We are reducing a sequence of odd numbers to a sum, which happens to be the square of the number of values we summed. **Roll Dice Until Craps**. Here's a more complex example. This iteration counts dice rolls until we get a 7. Note that our loop depends on *d1* and *d2* changing. Each time the suite inside the **while** statement finishes, we restore the initial condition of having an unknown values for *d1* and *d2*.

```
from __future__ import print_function
import random
rolls= 0
d1,d2=random.randrange(6)+1,random.randrange(6)+1
while d1 + d2 != 7:
    rolls += 1
    d1,d2=random.randrange(6)+1,random.randrange(6)+1
print(rolls)
```

This example is a search. We are search through a sequence of random dice rolls looking for the first seven. We are reducing the list of dice rolls to a count of the number of rolls.

Tip: Debugging the **while** Statement

If you are typing a **while** statement, and you get a `SyntaxError: invalid syntax`, you omitted the `:`.

There are several problems that can be caused by an incorrectly designed **while** statement.

The while loop never stops! The first time you see this happen, you'll probably shut off your computer. There's no need to panic however, there are some better things to do when your computer appears "hung" and doesn't do anything useful.

When your loop doesn't terminate, you can use `Ctrl-C` to break out of the loop and regain control of your computer. Once you're back at the `>>>` you can determine what was wrong with your loop. In the case of a loop that doesn't terminate, the **while** expression is always `True`. There are two culprits.

- You didn't initialize the variables properly. The **while** expression must eventually become `False` for the loop to work. If your initialization isn't correct, you may have created a situation where it will never become `False`.
- You didn't change the variables properly during the loop. If the variables in the **while** expression don't change values, then the expression will never change, and the loop will either never iterate or it will never stop iterating.

If your loop never operates at all, then the **while** expression is always `False`. This means that your initialization isn't right. A few **print** statements can show the values of your variables so you can see precisely what is going wrong.

One rare situation is a loop that isn't supposed to operate. For example, if we are computing the average of 100 dice rolls, we'll iterate 100 times. Sometimes, however, we have the "degenerate case", where we are trying to average zero dice rolls. In this case, the **while** expression may start out `False` for a good reason. We can get into trouble with this if some of the other variables are not be set properly. This can happen when you've made the mistake of creating a new variable inside the loop body. To be sure that a loop is designed correctly, all variables should be initialized correctly, and no new variables should be created within the loop body; they should only be updated.

If your loop is inconsistent – it works for some input values, but doesn't work for others – then the body of the loop is the source of the inconsistency. Every **if** statement alternative in the suite of statements within the loop has establish a consistent state at the end of the suite of statements.

Loop construction can be a difficult design problem. It's easier to design the loop properly than to debug a loop which isn't working. We'll cover this in *A Digression On Design*.

6.6.2 Counting Sevens

Let's combine the loop in the previous section and a **for** loop to get 100 different samples of the number of rolls before we get a 7.

The "outer" loop will execute 100 times; each time *i* will have a different value between 0 and 99. Each of these 100 excursions through the loop will reset *rolls* to zero and then roll dice until a 7 comes up.

Each time we figure out how many rolls before a 7, we add this number of rolls to *total*.

When we're done, we divide *total* by 100, to get the average number of rolls before we get a 7.

countsevens.py

```
1  from __future__ import print_function
2  import random
3  total= 0
4  for i in range(100):
5      rolls= 0
6      d1,d2=random.randrange(6)+1,random.randrange(6)+1
7      while d1 + d2 != 7:
8          rolls += 1
9          d1,d2=random.randrange(6)+1,random.randrange(6)+1
10     total += rolls
11 print(total/100.0)
```

3. The initialization for the outer loop creates a counter which will hold the total number of rolls before getting a seven for all of the games. The loop uses range(100) to assure that we gather data "for all" 100 simulations. In effect, this loop is a mapping from simulation number (i) to a count of rolls before rolling a 7. This outer loop is also a reduction that uses 100 simulations to compute the total number rolls to get a 7.

6. The initialization for the inner loop creates a counter (rolls) which will hold the number of rolls before getting a seven in this game only. It also initializes a pair of dice (d1 and d2) to the first roll. This is the typical initialization for a reduction.

7. While we haven't rolled a 7, we'll count one non-7 roll, then we'll roll the dice again. Note that the body of the while statement starts with an unknown pair of dice. When the pair is evaluated and found to be a number other than 7, a new pair of dice is created, restoring this condition that the dice are unknown. This is a typical search loop: we are searching for a 7 and counting the number of rolls until we find one.

10. Once we've rolled a 7, we add the number of rolls to our total. Since we are computing a single value from 100 samples, this is a reduction.

6.6.3 While Statement Exercises

1. **Update** `countsevens.py`.

In *Simulating All 100 Rolls of the Dice*, a **for** statement is used to iterate through 100 samples of data gathering. Replace this **for** statement with the equivalent statements using **while**. Hint: you'll have to add two new statements in addition to replacing the **for** statement.

2. **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.

Greatest Common Divisor of two integers, *p* and *q*

```
**Loop**. Loop until $p = q$.
    **Swap**. If $p < q$ then swap $p$ and $q$, $p \leftrightarrows q$.
    **Subtract**. If $p > q$ then subtract $q$ from $p$, $p \leftarrow p - q$.
**Result**. Print $p$
```

3. **Square Root**.

   This is an approximation of 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

Square Root of a number, *n*

```
**Two Initial Guesses**.
    $g_1 \leftarrow 0$
    $g_2 \leftarrow n$
    At this point, $g_1 \times g_1 - n \leq 0 \leq g_2 \times g_2 - n$.
**Loop**. Loop until $|g_1 \times g_1 - n| \div n < 0.001$.
    **Midpoint**. $mid \leftarrow (g_1 + g_2) \div 2$
    **Midpoint Squared vs. Number**. $cmp \leftarrow mid \times mid - n$
    **Which Interval?**
        if $cmp \leq 0$ then $g_1 \leftarrow mid$.
        if $cmp \geq 0$ then $g_2 \leftarrow mid$.
        if $cmp = 0$, $mid$ is the exact answer!
**Result**. Print $g_1$
```

4. **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 $w \leq x \leq y \leq z$ and w, x, y, z are from W, X, Y, and Z.

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.

[If you have experience in other languages, you might be tempted to use a `list` for this. That's cheating and won't teach you much about designing **if** statements.]

5. **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 $64 \leq 100 < 128$. The highest power of 2 less than or equal to 100 is 64, 2^6.

   Given a number n, find a number p such that $2^p \leq n < 2^{p+1}$.

   This can be done with only addition and multiplication by 2. Multiplication by 2, by 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 = 1, c = 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.

6. **Computing** π.

   Each of the following series compute increasingly accurate values of π (3.1415926...)

   - $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$
   - $\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$
   - $\pi = \sum_{0 \leq k < \infty} \left(\frac{1}{16^k}\right)\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$
   - $\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \cdots$

   For each of these you'll need to construct a loop that develops each term and adds it in to the total. At some point the terms will be so small that they don't contribute significantly to the answer; this is when the loop should stop.

   The third form uses summation (Σ) notation, telling us that the variable k takes on values from 0 to infinity. As a practical matter, k will go from zero to a value large enough that the expression computed is about zero. For more information on the (Σ) operator, see *Translating From Math To Python: Conjugating The Verb "To Sigma"*.

7. **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:

Definition of e.

$$e = \sum_{0 \leq k < \infty} \frac{1}{k!}$$

For more information on the (Σ) operator, see *Translating From Math To Python: Conjugating The Verb "To Sigma"*.

The $n!$ operator is "factorial". Interestingly, it's a post-fix operator, it comes *after* the value it applies to.

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 1.$$

For example, $4! = 4 \times 3 \times 2 \times 1 = 24$. By definition, $0! = 1$.

If we add up the values $\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$ 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 $\frac{1}{k!}$ terms, and an inner loop to compute the $k!$.

However, if we have a temporary value of $k!$, then each time through the loop we can multiply this temporary by k, and then add $1/temp$ to the sum.

You can test by comparing your results against *math.e*, $e \approx 2.71828$ or `math.exp(1.0)`.

8. **Hailstone Numbers**.

   For additional information, see [Banks02].

   Start with a small number, n, $1 \leq n < 30$.

   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.

   Test: for 27, the path length is 111, and the maximum value is 9232.

6.6.4 Iteration FAQ

Why are there two iteration statements, for and while ? Fundamentally, we really only need the **if** and **while** statements. However, the most common kind of iteration is the **for** statement style of processing. It gets tedious to write this out the long way using the **while** statement. The **for** statement is a handy short-hand.

This concept of a summary or abstraction that embodies a number of standard details is an important tool for programmers. In future sections we'll talk about creating these kind of processing summaries. In effect, we'll add new verbs to the Python language.

When we look at our computer, the operating system, the Python program, we see this layering effect. Each layer adds features, and makes the lower layers easier to use. The **for** statement continues this layering by enabling us to write iterations in a single statement that would have taken three statements.

6.7 Becoming More Controlling

There are a few situations where a **while** or **for** statement can be difficult to construct. To simplify things, we have two additional statements: **break** and **continue**. We'll cover these in *More Iteration Control: break* and *Yet More Control: continue*.

Since a program makes progress through the change of variables, we should be able to assert that our variables have particular relationships at various points in our program. We can formalize this assertion with the **assert** statement. We'll look at this in *The assert Statement*. This is another tool used for testing and debugging our programs.

Because this kind of program logic can be very hard to develop, we'll provide some warnings and advice in *The Hidden Dangers of else* and *A Digression On Design*.

6.7.1 More Iteration Control: break

Python offers two statements for more subtle loop control. The point of these statements is to help simplify the loops that we use to implement filter designs. In all cases, these statements can be rewritten into a **while**-expression. However, the **while** statement looks more complex than is appropriate for these fairly common situations.

The break Statement. The **break** statement terminates a loop prematurely. This is used so that we can state one or more additional conditions that will terminate an iteration; usually these conditions are too complex to be expressed in the **while**-expression. This can help us implement a search where we stop iterating when we find the first match.

The syntax is trivial.

```
break
```

The **break** statement is always found within **if** statements within the body of a **for** or **while** loop. The surrounding **if** statement has the terminating condition. A **break** statement can, for example, end a **for** before the end of the sequence has been reached.

Here's a complex terminating condition: we want to simulate parts of a Craps game that ends when we roll a 7 or the game lasts more than five rolls of the dice. We initialize our loop by determining two random values for *d1* and *d2*. Our loop will use a the `range(5)` sequence of five values to provide an upper limit on the number of dice we will roll. Also, we'll break out of the loop if the dice total 7.

```
from __future__ import print_function
import random
d1,d2=random.randrange(6)+1,random.randrange(6)+1
for i in range(5):
    if d1+d2 == 7:
        break
    d1,d2=random.randrange(6)+1,random.randrange(6)+1
if d1+d2 == 7:
    print("rolled 7")
```

```
else:
    print("5 rolls without a 7")
```

Tip: Debugging the **break** Statement

The most common problem with the **break** statement is an incorrect condition on the surrounding **if** statement, or an incorrect condition on the **while** statement. Designing these repetitive statements can be a relatively difficult problem, so we have some additional material on just that subject in *A Digression On Design*.

The most important debugging tool is the `print()` function. You can print the values of relevant variables in various positions in your loop body to be sure that things work the way you expect.

6.7.2 Yet More Control: continue

The continue Statement. The **continue** statement skips the rest of the loop's suite. The **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. We use this when we have relatively complex processing in our filter and want to declare that we are done processing a value and we want to move on to the next value in the sequence.

The syntax is trivial.

```
continue
```

Here's a contrived example of using **continue** to gracefully ignore certain numbers in a sequence. In this case, when `i % 2 == 0`, we have a number that can be divided by 2 with no remainder; an even number. Since we continue the loop for even numbers, we will only accumulate odd numbers in *total*.

```
from __future__ import print_function
total = 0
for i in range(20):
    if i % 2 == 0:
        continue
    total += i
print("total", total)
```

Tip: Debugging the **continue** Statement

The most common problem with the **continue** statement is an incorrect condition on the surrounding **if** statement, or an incorrect condition on the **while** statement. Designing these repetitive statements can be a relatively difficult problem, so we have some additional material on just that subject in *A Digression On Design*.

More Complex Example. 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 we wind up spinning the Roulette wheel 100 times.

There is 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.

sixodds.py

```
1  from __future__ import print_function
2  import random
3  oddCount= 0
4  for s in range(100):
5      lastSpin= s
6      n= random.randrange(38)
7      # Zero
8      if n == 0 or n == 37: # treat 37 as 00
9          oddCount = 0
10         continue
11     # Odd
12     if n%2 == 1:
13         oddCount += 1
14         if oddCount == 6: break
15         continue
16     # Even
17     assert n % 2 == 0 and 0 < n <= 36
18     oddCount = 0
19 print(oddCount, lastSpin)
```

2. We import the `random` module, so that we can generate a random sequence of spins of a Roulette wheel.
3. 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.
4. The **for** statement will assign 100 different values to s, such that $0 \leq s < 100$. This will control our experiment to do 100 spins of the wheel.
5. 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 exit condition is handled by the for statement itself.
6. We set *n* to a random spin of the wheel. We've asked for a random number from a pool of 38 numbers. This is the size of the usual double zero Roulette wheel.
8. 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.
12. When we determine that the number is odd by testing to see if the remainder is 1 when the spin, *n*, is divided by 2. If the spin is odd, the *oddCount* variable is incremented by 1.
14. 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 to exit from the loop, winding up after 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*.
17. We threw in an **assert** (see the next section, *The assert Statement*, for more information on this statement) 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.
18. If the number is even, we also set the **oddCount** to 0.

19. 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 **varname** 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.

6.7.3 The assert Statement

One useful programing invention is the notion of an assertion. An assertion has a condition that summarizes the state of the program's variables. If the condition is true, the program continues as if nothing happened. If the assertion is not true, the assertion has failed, and it raises an exception, which stops the program.

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. Generally, we put assertions in programs as a kind of document that describes what *should* be true. If it isn't true, the program will break right there, with a message that we can use to diagnose what went wrong.

There are two forms for the **assert** statement:

```
assert condition
```

```
assert condition , expression
```

If the assertion *condition* is `False`, the program is in error, and raises an `AssertionError` exception. If the *expression* is given, the `AssertionError` exception is raised using the expression. We'll cover exceptions in detail in *The Unexpected : The try and except statements*. For now, the most important part of raising an exception is that the program stops.

Here's an example of using **assert** to prove that out program works. We're trying to set *max* to the larger of two values, *a* or *b*. We include an assertion with a formal definition of what value *max* should have. It should be either *a* or *b*, and the larger of the two values.

```
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.

Tip: Debugging the **assert** Statement

The **assert** statement is an important tool for debugging other problems in your program. It is rare to have a problem with the **assert** statement itself. The only thing you have to provide is the condition which must be true. If you can't formulate the condition in the first place, it means you may have a larger problem in describing what is supposed to be happening in the program in general. If so, it helps to take a step back from Python and try to write an English-language description of what the program does and how it works.

Clear **assert** statements show a tidy, complete, trustworthy, reliable, clean, honest, thrifty program. Seriously. If you can make a clear statement of what must be true, then you have a very tight grip on what should be happening and how to prove that it really is happening. This is the very heart of programming: translating the program's purpose into a condition, creating the statements that make the conditions true, and being able to back this design up with a proof and a formal assertion.

6.7.4 The Hidden Dangers of else

As additional syntax, the **for** and **while** statements permit an **else** clause.

```
for variable in sequence :
    suite
else :
    else-suite
```

If the **else** clause is provided, the *else-suite* of statements is 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.

Even in the **if** statement, an **else** clause raises a small question 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:
    # Implied: 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 **for** or **while** statements.

6.7.5 A Digression On Design

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** $m \geq a$ **and** $m \geq b$).

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** $a \geq a$ **and** $a \geq b$).

Removing the parts that are obviously true, we're left with $a \geq b$. 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** $b \geq a$ **and** $b \geq b$). Again, we remove the parts that are obviously true and we're left with $b \geq a$. 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
```

This **if** statement has the statements that will set m to the larger of a or b. Each assignment is associated with a condition under which that assignment statement solves the problem.

The Post-Condition. 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 some algebra to fill in any guards or loop conditions to make sure that only the correct statement is executed.

There are several considerations when using the **while** statement. This list is taken from David Gries', *The Science of Programming* [Gries81].

1. The variables changed in the body of the loop must be initialized properly. If the loop's while-expression is initially false, everything is set correctly.
2. At the end of the suite, the condition that describes the state of the body variables 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 invariant body condition is true and the while-expression is false, the loop will have completed properly.
4. When the while-expression 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-expression 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-expression 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.

An Example. 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.

```
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. We can see that $2^{1-1} = 2^0 = 1 < x$. 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 exists some n such that $n - 1 < \log_2 n \leq n$ 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()`.

We should add this information on successful termination as comments in our loop.

6.8 Comments and Scripts

Our purpose in programming is to create a new application program for our computer. We'd like our new program to be like other application programs. We'd like to run our programs without using **IDLE**.

Our operating system gives us broad spectrum of ways to run programs. We want to fit seamlessly into these patterns so that our Python program is just like any other binary application program.

1. From the prompt, where we start **Python** with our Python language file as input.
   - Explicitly. We'll see how to do this in *Let Python Run It*.
   - Implicitly. We'll see how to do this in *Giving the Shell a Hint in GNU/Linux or MacOS* and *Giving the Shell a Hint – Windows Detailing*.
2. From the GUI, by double-clicking an icon. This varies widely among the operating systems. We'll cover Windows and MacOS variations in *Double-Clicking Icons*. For GNU/Linux, there are too many GUI environments to cover.

It turns out that tackling item #1, working from the **Terminal** (or **Command Prompt**) will also get us almost all the way to having our own desktop icon. We'll focus on command-line operation first. Once we have the command-line program working, we can extend this to create an icon that runs the command.

Running Outside IDLE. Back in *IDLE Time : Using Tools To Be More Productive* we started using **IDLE** to both create and run our script modules.

It's time to move past that. Rather than using **IDLE** to run our new programs, we'll execute our programs directly in the **Command Prompt** or **Terminal** window.

6.8.1 Comments

Comments are notes and remarks to people who are looking at the program. They aren't used by Python program, but can be reminders and clarifications.

Comments can be placed anywhere in our program. They're covered by the syntax rules missing from *Instant Gratification : The Simplest Possible Conversation*.

Important: Syntax Rule Three

Everything from a # to the end of the line is ignored by the Python program.

If the # occurs inside a quoted string, it is just another character. The # for a comment must occur outside a string.

The GNU/Linux shell can sneak a look at the first line of a Python file. If the first line is the special #! comment, this defines the interpreter that will be used. Consequently, many Python files begin with the following line:

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

And just to be complete, here's rule four.

Important: Syntax Rule Four

Rule four is a historical anachronism from the very early days of Unicode. Not all operating systems were perfectly compatible with Unicode files. Rule 4 allows you to use a special comment with `#coding:` or `#coding=` indicate the coding.

6.8.2 Let Python Run It

The simplest way to execute a script is to tell Python to run it. We do this by providing our script file name as the first parameter to the **python** command. In this style, we explicitly name both the interpreter and the input script. We enter this command at the Windows **Command Prompt**, or the GNU/Linux/MacOS **Terminal**.

Here's an example that will provide the `example1.py` file to the Python interpreter for execution.

```
python example1.py
```

This is identical in GNU/Linux, MacOS and Windows.

There are really two parts to making this work flawlessly:

- The shell (or `cmd.exe` in Windows) must be able to find the **Python** program. In *Instant Gratification : The Simplest Possible Conversation* we looked at what you must do to assure that the **Python** program is on your shell's search path.
- The shell must be able to find your `example1.py` file. Since the shell doesn't search for this, you have to be sure that you provide the precise location of your file. For now, we're assuming that you have an `exercises` directory, and this is the current working directory. For Windows, use **cd** to see your current directory. For GNU/Linux or MacOS, use **pwd** to print the name of the working directory.

Tip: Debugging Explicit Python Scripts

There are two things which can go awry with a script: the operating system can't find Python or the operating system can't find your script file. Here's a debugging procedure which may help.

1. Start with the simplest possible operating system command: `python`. If this doesn't work, Python isn't installed correctly: reinstall Python. If this *does* work, then the shell or command prompt can find Python, and that is not the problem.
2. Use your operating system's directory and file commands (Windows: **CD** and **DIR**; GNU/Linux or MacOS: **pwd** and **ls**) to find your script file. If your file is not in the current directory, then you can either change directories, or include the directory name in your file.

- Change your current working directory to the correct location of your files. For Windows: use **CD**; for GNU/Linux and MacOS: use **cd**. For example, if your files are in an `exercises` directory, you can do **cd exercises**.
- Include the directory name on your file. For example, if your files are in an `exercises` directory, you can run the `script1.py` script with **python exercises/script1.py**.

3. If you can find Python, and you appear to be in the correct directory, the remaining problem is misspelling the filename for your script. This is relatively common, actually. First time GNU/Linux and MacOS users will find that the shell is sensitive to the case of the letters, that some letters look alike, it is possible to embed non-printing characters in a file name, and it is unwise to use letters which confuse the shell. We have the following advice.
   - File names in GNU/Linux should be one word, all lower case letters and digits. These are the standard Python expectations for module names. While there are ways around this by using the shells quoting and escaping rules, Python programs avoid this.
   - File names should avoid punctuation marks. There are only a few safe punctuation marks: `-`, `.` and `_`. Even these safe characters should not be the first character of the file name.
   - Some Windows programs will tack an extra `.txt` on your file. You may have to manually rename the file to get rid of this.
   - In GNU/Linux, you can sometimes embed a space or non-printing character in a file name. To find this, use the **ls -s** to see the non-printing characters. You'll have to resort to fairly complex shell tricks to rename a badly named file to something more useful. The `%` character is a wild-card which matches any single character. If you have a file named `script^M1.py`, you can rename this with **mv script%1.py script1.py**. The `%` will match he unprintable `^M` in the file name.

Looking Forward. In the long run, we don't always want to have our Python application program files and our data files all mixed up together in the same directory. We'd like to be able to put our programs in a directory like `/usr/local/myapp` or `C:\Program File\MyApp`. GNU/Linux and MacOS have many tricks for making programs easy to start. We'll look at some techniques for this in the next section.

6.8.3 Giving the Shell a Hint in GNU/Linux or MacOS

In general, the various OS shells have a handy trick that streamlines application startup for us. The shell can read the first line of a file to determine two things: is the file a script, and if the file is a script, which program should process the statements in that script.

Windows can also rely on a somewhat obscure operating system setting called a "file association". This is typically set by the Python installer. We'll cover the Windows settings in the next section.

To conform to the POSIX standards, a shell has to follow this procedure.

1. When you type a statement to the shell, the first word is the name of the program to be executed. The remaining words are arguments to that program.
2. If the first word isn't a special command built into the shell, the shell searches its `PATH` for a file name which matches this word. (In GNU/Linux and Mac OS, this file must also be marked executable with the **chmod +x** command.) If no such file can be found, the shell responds with a "not found" error message.
3. If the shell finds an appropriate file, the shell reads the first few characters of that file.
4. If the first two characters are `#!` (called "sharp bang" or "shebang"), then the shell uses the rest of the first line as a single command. The shell runs this one-line command, and provides the file it found as input to that command.

The very cool part of this trick is that `#!` is a *comment* to Python. Comments are simply ignored by Python. This first line of our Python file is, in effect, directed at the shell. The shell uses the first line of our file as a hint to see what the language is, and Python studiously ignores it.

A file that is going to be executable in would look like this:

celsius.py

```
#!/usr/bin/env python
# Convert 65F to Celsius
from __future__ import print_function
print(65, "F")
print((65-32) * 5 / 9, "C")
```

The path to Python varies by operating system

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

Mac OS X `#!/Library/Frameworks/Python.framework/Versions/2.6/bin/python`

Windows `#!C:\\Python26\\bin\\python`

For GNU/Linux and Mac OS, we need to use the following shell command just once to mark it executable.

```
chmod +x celsius.py
```

Once we've marked it executable, we can do the following to run it.

```
[slott@linux01 slott]$ ./example1.py
65 F
18 C
```

The last example shows the prompt from a RedHat GNU/Linux computer, named `linux01`.

Security Consideration

This last example depends on the way the shells assure security. The `./` prefix forces the shell to look in the current directory for the file name that matches the command. This is optional in some shell environments; it always works, but isn't always necessary.
The reason for this extra punctuation is to make it difficult to override the built-in shell command names with rogue viruses or spyware. Think of what could happen if you wrote your own version of **chmod** which could then add a virus to every file that was marked executable. The way to prevent this is to assure that the search path is tightly controlled and the user has to specifically add to the search path to run a program that isn't built in to the operating system. Since no shell script would ever say `./chmod`, a rogue **chmod** program would never get used.

Bottom Line. We have to do two things to make this implicit execution work. We have to make our file executable with **chmod**, and we have to include the magic `#!/usr/bin/env python` line as the very first line in the file.

We need to emphasize two parts of this recipe.

- The executable mode setting remains with a file forever, so we only need to set it once, when we create the file. When we forget to do this [*I don't say* if *we forget – everyone forgets*], we get an error that says our file can't be found as a command. This is a hint that we forget to mark the file as executable.
- The **chmod** command is done in the **Terminal**, not in **IDLE**. This is a command to the shell, and is not a Python statement. It helps to have an extra terminal window open for this kind of thing.

Tip: Debugging Implicit Python Scripts

If an implicit script doesn't work, you have two problems to resolve. First, would it work as an explicit command? If not, then fix that before doing anything else. Say you have a script `implicit.py` that won't run. The first thing to test is **python implicit.py**. If this doesn't work, see *Let Python Run It*.

The most common problem with implicit scripts is the hidden hand-shake between the shell and the first line of the script file. Most GNU/Linux variants will use `#!/usr/bin/env python`.

You may not have the **env** program in your UNIX. Try typing just **env** at the shell prompt. If this returns an error, you will need to know where Python installed. Enter **which python**. The shell will search its path for Python and report the directory in which it was found. This might be `/usr/local/bin/python`. Use this in the `#!` line. For example, `#!/usr/local/bin/python`.

6.8.4 Giving the Shell a Hint – Windows Detailing

For Windows users, the command prompt has a handy trick that streamlines application startup for us. Windows uses an operating system setting called a "file association" to determine what to do with certain kinds of files.

Windows uses the last few letters of the file name (the part after the final `.`) to indicate what program should interpret the script. If the last few letters are `.bat`, the shell processes the file. If the last few letters are `.py`, we can have windows run Python to process the file.

You can see if this is working correctly by opening a Command Prompt window. Change the directory to a place where you have some Python scripts. In my case, my scripts are in a folder named `E:\Writing\Technical\NonProgrammerBook\notes`. If you type the full name of a file, Windows should locate the interpreter correctly. It looks like this.

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
E:\Writing\Technical\NonProgrammerBook\notes> inputdemo.py
price: 12
12
shares: 24
24
$ 288
```

This file association is typically set by the Python installer. In the unlikely event that you don't have a proper association between Python files and the Python applications, you can create or modify this association with the **Folder Options** control panel. The **File Types** tab allows you to pair a file type with a program that processes the file. It is often simpler to uninstall and reinstall Python.

Setting the Python File Association

1. Open the Control Panel

   Use the **Start** menu, **Settings** sub menu to locate your **Control Panel**.

2. Open the Folder Options Control Panel

   Double-click the **Folder Options** Control Panel. This opens the *Folder Options* panel.

3. Open the File Types Tab of the System Control Panel

   Click the **File Types** tab on the **Folder Options** Control Panel.

   There are two areas: Registered File Types and Details for the selected type.

4. Find or Add the .PY Extension

   This dialog box has a title of *Registered File Types*. Scroll through the list of Extensions, looking for `PY`. If you don't find the PY extension, you'll have to create a new association. The easiest way do this is to uninstall and reinstall the current version of Python. See *Windows Installation*.

   If you do find the PY extension, click on the extension to highlight it. In the lower part of the window it will show details for 'PY' extension. It should open with Python. If you have installed Python more than once, it is possible that the files are associated with the wrong version or a non-existent version.

   Click the **Change...** button.

   This dialog box has a title of *Open With*. Locate the Python program, highlight it in the list and click **OK**. If Python is not on this list, it was not installed. See *Windows Installation* to install Python.

5. Finish Changing Your Folder Options

   The current dialog box has a title of *Folder Options*. Click **OK** to save your changes.

6.8.5 Double-Clicking Icons

Windows. For Windows programmers, the **Explorer** also uses the last letters of the file name to associate a script file with an interpreter. Windows will run the `pythonw.exe` program whenever you double-click a `.pyw` file, and run `python.exe` whenever you double-click a `.py` file. The Python installer should have set these file associations for you.

The down side of running programs this way is that the Python window vanishes as soon as the program is finished. This method leaves us wondering what happened when we run a character-mode program. In the long run, when we write advanced applications with a fancy GUI, this isn't a big problem. For now, however, it's a show-stopper. We prefer to work in the command prompt window because the window doesn't close automatically when a program finishes running.

MacOS. For MacOS programmers, we can associate a specific file with the **PythonLauncher**.

If we use the Mac OS Finder's **Get Info** item in the **File** menu, we get a window that shows us many things about our file. There's a section in this window named *Open With:*. In this section, we can associate our document with the PythonLauncher application.

After we set the association, when we double-click the icon, Mac OS will run the **PythonLauncher**. The PythonLauncher does two things for us: it creates a Terminal window and runs **Python** to execute our program.

GNU/Linux. For GNU/Linux and UNIX programmers, there are too many GUI environments, including KDE, GNOME and others. Each is slightly different, and it's too hard to cover all of the bases in this book.

6.8.6 Another Script Example

Here's an example script that we'll use to look at direct execution. We'll create a new file and write another small Python program. We'll call it `example2.py`.

example2.py

```
#!/usr/bin/env python
# Compute the odds of spinning red (or black) six times in a row
# on an American Roulette wheel.
from __future__ import print_function, division
print( (18/38)**6 )
```

Note that we included the `#!` code on the first line. This is a Python comment; it is really only used by GNU/Linux users, but it doesn't hurt for Windows or MacOS programmers to include this.

After we finish editing, we mark the file we made as *executable* using **chmod +x example2.py**. Since this is a property of the file, once we've set it, the executable status remains true no matter how many times we edit, copy or rename the file. We only have to make a file executable once, the first time we work with it.

When we run this in GNU/Linux or MacOS, we see the following.

```
[slott@linux01 slott]$ ./example2.py
0.0112962280375
```

When we run this in Windows, we see the following.

```
E:\Writing\Technical\NonProgrammerBook> example2.py
0.0112962280375
```

Which says that spinning six reds in a row is about a 1 in 89 probability. If we won all six spins in a row, we'd have made 64 times our bet.

6.8.7 Additional Flexibility

Another solution that works for all operating systems is to create a *shell script* (or Windows `.bat` file) that contains the full name of our application program.

Writing a shell script and a similarly-named `.bat` file allows us to put our application into a single directory (like `/usr/local/myapp/bin` or `C:\MyApp\bin`). We can then add that directory to the `PATH`.

Since our GNU/Linux shell script and our Windows `.bat` files have almost the same names, our application looks similar on all operating systems.

These extra shell script files have several purposes.

1. A shell script is the "lowest common denominator" between Windows and GNU/Linux. The Windows Command Prompt searches for `.bat` files that match the command you entered. The GNU/Linux shell can search for shell scripts, also.
2. It provides a kind of insulation between our Python program and the command a person types to make the program start running. We can pick a cool command name, and that's the name of the shell script and `.bat` file. We can use a different name for the Python script.
3. It's easy to create an icon that starts a shell script or starts a `.bat` file. Consistency between the command-line operation and the double-click-the-icon operation is very important.

GNU/Linux Shell Script. Here's the one line we need to put into our file for GNU/Linux folks who created a `/usr/local/myapp/example1.py`. If we make sure that this `example1` shell script file is on our path, we can then be in any working directory and run our example.

example1

```
python /usr/local/myapp/example1.py
```

The typical trick in GNU/Linux is to put our Python and shell files into a single directory, and then add this directory to the

`PATH` setting. We might put our files into `/usr/local/myapp`, then add `/usr/local/myapp` to the PATH.

Windows .BAT File. Here's the one line we need to put into our file for Windows folks who created a `C:\MyApp\bin\example1.py`. If we make sure that this `example1.bat` file is on our path, we can then be in any working directory and run our example.

Note that we've studiously avoided a filename with a space in it. We didn't put our application into `C:\Program Files` because we'd have to work around that pesky space.

example1.bat

```
python C:\MyApp\bin\example1.py
```

The typical trick in Windows is to put our Python and shell files into a single directory, and then add this directory to the

`PATH` setting. We might put our files into `C:\MyApp\bin`, then add `C:\MyApp\bin` to the `PATH`.

6.8.8 Script Exercises

Pick any two examples from *For Statement Exercises* or *While Statement Exercises* and make sure that they run from the command line. For GNU/Linux or MacOS users, you'll have to add the "shebang" line, and use the **chmod** command.

CHAPTER

SEVEN

ORGANIZING PROGRAMS WITH FUNCTION DEFINITIONS

Building A Solution in Pieces

Our initial programs have been sequences of statements. As our programs get more complex, we will find that this style of "long, flat" program is hard to work with. In *Adding New Verbs : The def Statement* we'll introduce the primary method for structuring and organizing our application programs, the function. It turns out that breaking a program into separate functions allows us to decompose a solution into several simpler parts. Functions are also a good intellectual tool to help us divide and conquer a complex problem.

We'll add several useful features in *Flexibility and Clarity : Optional Parameters, Keyword Arguments*. These will add flexibility so that it's easier to understand and use the functions we define.

In *A Few More Function Definition Tools*, we'll show a number of unique features that make Python's function definitions much cooler than other programming languages

7.1 Adding New Verbs : The def Statement

The heart of programming is the *evaluate-apply* cycle, where function arguments are evaluated and then a function is applied to those argument values. Even something like `3+5`, which seems simple, is a function applied to argument values. We write `3+5`, but Python treats it as if we said `operator.add(3,5)`.

In *What is a Function, Really?* we'll review the evaluate-apply cycle. In *Function Definition: The def and return Statements* we introduce the basics of defining and using our own unique functions.

We'll look at some design patterns in *Function Design Patterns – Rules of the Game*.

This chapter has some challenging exercises broken into two separate sections. The first, *Function Exercises* is more important. The second set of exercises, *Optional Function Exercises – Recurrence*, is optional.

The **return** statement is central to how functions work. The **yield** statement, which has a similar syntax, has quite different semantics; we'll look at the **yield** statement in *Looping Back : Iterators, the for statement and Generators*.

7.1.1 What is a Function, Really?

A function, in a mathematical sense, is a mapping from *domain* values to *range* values. Given a domain value, a function returns the appropriate range value. If we think of the square root function, it maps a positive number, n, to another positive number, r, such that $r^2 = n$. For example, the square root function applied to the value of 9 maps to 3. This fits the reqired condition $3^2 = 9$.

We can think of multiplication as a function, also. Multiplication maps a pair of values, a and b, to a new value, c, such that $c = a \times b$. The domain is pairs of numbers, a, b, the range is a number.

When we looked at the functions in the `math` module in *Better Arithmetic Through Functions*, they fit this mold perfectly. The `random` module, however, has a bit of a problem. In *The random Module – Rolling the Dice* we saw that many of these functions don't have a domain value, they only have a range value.

Oddly, `raw_input()` function that we looked at in *Can We Get Your Input?* allows the user to enter the range value. This doesn't seem to fit the strict mathematical sense of mapping from domain to range. There's no real domain and the user could enter just about anything they wanted.

The Language of Planet Python. Clearly, Python doesn't adhere to the letter of the formal mathematical definition. This is one of those cases where the computer science folks borrowed a word from mathematics, but had to stretch the meaning a bit to make it useful. While many Python functions *are* proper mathematical functions, Python allows us to use some additional patterns. We can define functions which do not need a domain value, but create new objects from scratch. Also, we can define functions that don't return values, but instead have some other side-effect, like creating a directory, or removing a file.

So what is a function in Python? A Python function is more like a verb than it is like a mapping. The mathematical functions, for example, have an implied sense of "compute". You can think of `sqrt()` as "compute the square root of". You can also think of it as "map", as in "map a number to it's square root."

Factory functions are a little different, in that less transformation is done. These are generally just a change in representation. You can think of the factory functions (in *Functions are Factories (really!)*) as wasy to "create from"; `int()` can be interpreted as "create the int value from".

Defining a Function. In Python, we define a function by providing three pieces of information:

- The name of the function. Hopefully this is descriptive; usually it is verb-like.
- A list of zero or more variables, called *parameters*; this defines the domain or input values. The phrase "zero or more" means that parameters are optional.
- A suite of one or more statements. If this contains a **return** statement, this defines the range or output value. The phrase "one or more" means that statements are not optional.

  Interestingly, the **return** statement is optional.

Typically, we create function definitions in script files because we don't want to type them more than once. We can then **import** the file with our function definitions so we can use them. **IDLE** helps us do this import with the **Run** menu **Run Module** item, usually `F5`.

Using a Function. When we used functions like `math.sqrt()` in an expression, we provided argument values to the function in `()`. The Python interpreter evaluates the argument values, then applies the function. When we use functions that we define, we'll use the name we gave to our function in front of the `()`. For more information on this evaluate-apply cycle, see *The Evalute-Apply Principle*

Evaluating and applying a function (sometimes termed "calling" the function) means that Python does the following:

1. Evaluate the argument expressions.
2. Assign the argument values to the function parameter variables.
3. Evaluate (or "call") the suite of statements that are the function's body. In this body, the **return** statement defines the result value for the function. If there is no **return** statement, the value `None` is returned.
4. Replace the function with the returned value, and finish evaluation of the expression in which the function was used.

We have to make a firm terminology 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 has a life before and after the function. The parameter is the name of a variable that is part of the function, and is a variable that exists only while Python is evaluating the function body.

7.1.2 Function Definition: The def and return Statements

Here, we'll cover definition of a new function. In the next section we'll review using a function and show how we use our newly-defined function.

Definition. 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. It must be a legal Python name; the rules are the same for function names as they are for variable names. The name must begin with a letter (or `_`) and can have any number of letters, digits or `_`. See *Python Name Rules*.

Each *parameter* is a variable name; these names are the local variables which will be assigned to actual argument values when the function is applied. We don't type the [and]'s; they show us that the list of names is optional. We don't type the ...; it shows us that any number of names can be provided. Also, the **,** shows that when there is more than one name, the names are separated by **,**.

The *suite* (which must be indented) is a block of statements that computes the value for the function. Any statements may be in this suite, including nested function definitions.

The first line of a function is expected to be a *document string* (called a *docstring*, and generally a triple-quoted `"""` string) that provides a basic description of the function. We'll return to this docstring in *Functions Style Notes*.

Returning a Result. A **return** statement specifies the result value of the function. This value will become the result of applying the function to argument values. This value is sometimes called the *effect* of the function.

```
return [ expression ]
```

The expression is the final result of the function. We don't type the [and]'s, they show us that the expression is optional. If we don't provide one, the Python value of `None` will be returned.

Let's look at a complete, although silly, example.

```
def odd( spin ):
    """Return "odd" if this spin is odd."""
    if spin % 2 == 1:
        return "odd"
    return "even"
```

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 $spin \div 2$ is 1; if so, we return `"odd"`. Otherwise, we return `"even"` .

Use. We would use our `odd()` function like this. This example will generate a random spin, *s* , between 0 and 36. (These are the rules for European Roulette, with a single zero.) We'll use our `odd()` function to determine if the spin was even or odd.

```
from __future__ import print_function
import random
s = random.randrange(37)
```

```
if s == 0:
    print("zero")
else:
    print(s, odd(s))
```

1. We generate a random number, *s*.
2. We break the generated number into two classes with a **if** statement. The if-clause handles the case where *s* is zero.
3. The else evaluates our `odd()` function and prints the result.

   When Python evaluates the function `odd()`, the following steps are followed:

   (a) Python evaluates the arguments. In this case, it's just the variable *s*. In other cases, it may be more complex.

   (b) Python assigns this argument value to the local parameter variable, *spin*.

   (c) Python applies `odd()`: the suite of statements is executed, ending with `return "odd"` or `return "even"`.

   (d) This result returned to the original statement which called the function. This original statement can now finish it's expression evaluation. In this case, it will be the expression in the **print** statement.

Tip: Debugging Function Definition

There are three areas for mistakes in function definition: the **def** statement itself, the **return** statement and using the function in an expression.

The syntax of the **def** statement contains three parts. If you have syntax errors on the definition, you've got one of these three wrong, or you're misspelling "def".

- The name, which is a Python name, following the name rules in *Python Name Rules*.
- The parameters, which is list of names, separated by commas. The `()`s around the parameter list is required, even if there are no parameters. The `,` to separate parameters is required.
- The indented suite, a block of Python statements.

The **return** statement is how the return value is defined. If you omit this, your function always returns `None`. The **return** statement also ends execution of the function's body; if you have this statement out of place, your function may not fully execute.

When you use the function, you have to pass actual argument values for each parameter variable. The matching is done by position: the first argument value is assigned to the first positional parameter.

7.1.3 Function Design Patterns – Rules of the Game

There are a couple of rules we need to clarify. These rules lead to a number of common design patterns for functions. First, we'll look at the importance of providing a docstring, then we'll look at three common design variations.

The Docstring. It's important to note that the docstring for a function must explain what kind of value the function returns, or if the function does not return anything useful. This information isn't obvious; there's only one way to make it obvious, and that's to write it down. Python is very helpful about making the docstring part of the function definition.

If you want too see the power of the docstring, look back at our `odd()` function. Here's what happens when we ask for help.

```
>>> help(odd)
Help on function odd in module __main__:

odd(spin)
    Return "odd" if this spin is odd.
```

If you're using Python directly, that is, you are *not* using **IDLE**, this will look a little different. See the sidebar for a little bit of information on the help viewer that may be used.

Tip: Direct Python and Help()

When executing `help()` while using Python directly (*not* using **IDLE**), you'll be interacting with a help viewer that allows you to scroll forward and back through the text.

For more information on the help viewer, see *Getting Help*.

On the Mac OS or GNU/Linux, you'll see an `(END)` prompt telling you that you've reached the the document; hit `q` to exit from viewing help.

Since our docstring shows up when we ask for help, we should be sure that we've put down everything we need to remember about the function.

Rules of the Game. There are two important rules that bracket what a function can be used for. These are constraints on what is a sensible definition of a function. Some functions will bend the second rule a bit.

1. **A function has no memory.** We call this *stateless*. We'd like to call this the *no hysteresis* rule because the word hysteresis is exactly what we're talking about; but hysteresis is a pretty obscure term for "influenced by previous events". When we look at a function like sine or square root, the answer doesn't depend on the previous requests for sines or square roots. The result only depends on the inputs.
2. **A function is idempotent.** The term *idempotency* means that a function, given the same inputs, always produces the same outputs. This is part of the standard mathematical definition of a function: the same input produces the same output.

   The `random` module has functions that bend this rule. Also `raw_input()` bends this rule.

These rules are so important that Python enforces them. The way Python enforces these rules is by automatically deleting any variables created inside a function when the function finishes.

You'll note that our random-number generating functions violate the idempotency rule. Each time you apply the `randrange()` function, you get a different value. Clearly, this random number generator function does something special and unusual to work around Python's enforcement of the rules. We'll return to this below.

When you seem to need a function that has a memory or a state change, you aren't really talking about a function anymore. To break the *no hysteresis* rule, you'll need to define an object, not a function. Defining object classes will require many more language features than we've seen so far, so we'll introduce this later, in *Data + Processing = Objects*.

Generally, any variable you use within a function body is private to that body. This is because all of a function's variable names exist in a *namespace* that is local to the function. This includes the parameter variables created by the definition and any local variables your statements create. The namespace (and the variable names) cease to exist when the function's processing is complete. We'll look at this more closely in *Keeping Track of Variable Names – The Namespace*.

Mathematical Functions. A mathematical function follows the standard definition of a transformation from a domain to a range. All of the functions in the `math` module are examples of these. We can copy this design pattern and create functions which transform an input to produce an output. Our example of `odd()` in *Function Definition: The def and return Statements* followed this pattern.

These functions have no hysteresis (no memory) and are idempotent (same results for the same input). These are well-behaved, and use a **return** statement to return a meaningful value.

The docstrings for these functions always look like this:

```
def myFunction( a, b ):
    """myFunction(a,b) -> someAnswer

    Some short, clear explanation of myFunction.
    """

    The suite of statements
```

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 the value returned.

These functions still have no hysteresis (no memory) and are idempotent. They just don't return a value. Instead, we expect that their side-effect is the same each time we call it.

Here's an example of a function that doesn't return a value, but carries out a procedure.

```
def report( spin ):
    """report(spin)

    Reports the current spin."""
    if spin == 0:
        print("zero")
        return
    print(spin, odd(spin))
```

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.

It turns out that any expression can be used as a complete statement. Since a function evaluation is an expression, and an expression is a statement, a function call is a complete statement. Because of this, a function definition can be like adding a new statement to the language.

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 (incorrectly) in an expression, and everything will still work out 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`.

Accessor Functions. A more specialized form is a function that creates a new value by accessing some concealed object. Often this concealed object is created when we import a module; we can describe the object as being *encapsulated* in the module. We'll look at this later, when we talk about modules in *Modules : The unit of software packaging and assembly*.

These functions can have hysteresis and may (or may not) be idempotent. In the case of random numbers, we don't want idempotency, otherwise, we'd just get the same number over and over again.

These functions broke the rules by using an object that is part of the module that contains the function. For example, our random number generators functions use an object that is part of the `random` module. This is almost the only example of this kind of accessor function that we'll use.

7.1.4 Function Example

Here's a big example of using a slightly `odd()`, `spinWheel()` and `report()` functions. We've refactored `odd()` and `report()`; they aren't exactly like the versions shown above.

This shows how we can break something quite large down into smaller pieces, each of which can be understood in isolation. Since each individual function is short and focused, we can define and test this complex program one function at a time.

functions.py

```python
#!/usr/bin/env python
# Report a dozen spins of a Roulette wheel
from __future__ import print_function
import random

def odd( spin ):
    """odd(number) -> True if the number is odd."""
    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 )
```

1. The `odd()` function is a simple mathematical function with a domain of numbers and a range of boolean (`True`, `False`). If the number is odd, this function returns `True`; otherwise it returns `False`.
2. The `report()` function uses the `odd()` function to determine if the number is even or odd and write an appropriate line to our final report. This function doesn't return a useful value, and is a kind of procedural function.

3. The spinWheel() function uses random.randrange() to simulate a spin of the wheel and return that value.
4. The "main" part of this program is this for loop at the bottom that uses the previous function definitions. It calls spinWheel(), and then report(). This generates and reports on a dozen spins of the wheel.

For most of our exercises, this free-floating main procedure is acceptable. When we cover modules, in *Modules : The unit of software packaging and assembly*, 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, since we put it in a function named main().

7.1.5 Hacking Out A Solution

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 statement gets too long, this can become a nuisance. We introduced this way of working in *Instant Gratification : The Simplest Possible Conversation*. On the other hand, we have scripted use of the interpreter where we present a file as a finished program to do the intended job. 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 way of working in *Turning Python Loose With a Script*.

In between the interactive mode and scripted mode, we have a third operating mode, best called *hacking mode*. The idea is to write most of our script and then exercise that script interactively. In this mode, we'll develop script files of definitions, but we'll exercise them in an interactive environment. This is handy for developing, testing and debugging our new function definitions.

The basic procedure is as follows.

1. In **IDLE**'s editor (or our favorite substitute), write a script with our function definitions. We save this file; don't quit the editor, leave the window open.
2. In the file's window, we run the module. This is the **Run** menu, the **Run Module** item. This resets Python and executes all of the **def** statements.
3. In the Python shell, we enter statements to test the functions interactively. If they work, we're making progress.
4. If the testing doesn't work, we go back to our editor, make any changes and save the file.
5. Go back to step 2, to reset Python, execute the file and test our definitions.

Here's the sample function we're developing. If you look carefully, you'll see a serious problem.

function1.py Initial Version

```
#!/usr/bin/env python
def odd( number ):
    """odd(number) -> boolean

    Returns True if the given number is odd.
    """
    return number % 2 == "1"
```

Here's our interactive testing session.

```
>>> ================================ RESTART ================================
>>>
>>> odd(2)
False
>>> odd(3)
False
>>>
```

1. We selected **Run Module** from the **Run** menu. Python imported our `function1.py` module to our Python Shell.
2. We entered `odd(2)` and Python's value for this function was False. That's correct.
3. We entered `odd(3)` and Python's value was also False. That can't be correct.

What's wrong? How do we fix it?. There aren't many things can be wrong in this function. We've made a common mistake and used a string where we should have used a number. Look closely at the **return** statement.

The `number % 2 == "1"` should be `number % 2 == 1`. We need to fix `function1.py`.

After we fix `function1.py`, we can loop back to step 2 in our procedure. This will remove the old definitions, re-import our function and rerun our test. This whole sequence is handled by the **Run Run Module**, available as `F5`. It clears out the old definitions by restarting Python and then importing our module.

In this case, we've got the function working correctly. Here's the corrected version.

function1.py Final Version

```
#!/usr/bin/env python
def odd( number ):
    """odd(number) -> boolean

    Returns True if the given number is odd.

    >>> odd(2)
    False
    >>> odd(3)
    True
    """
    return number % 2 == 1
```

Here's our interaction in the Python Shell window. The two function calls and their answers are a handy little summary of how this function is supposed to work. Notice that we did a cut and paste from the Python Shell window into the docstring inside the function. That's the clearest way to define the function's intended purpose.

```
>>> ================================ RESTART ================================
>>>
>>> odd(2)
False
>>> odd(3)
True
>>>
```

Next Steps. Once we have the `odd()` function working, we can move on to debugging the `spin()` function, then `report()` function and finally the main procedure that produces the report. We call this building and testing in pieces "iterative" or "incremental" development.

7.1.6 The Evalute-Apply Principle

Back in *Execution – Two Points of View*, we touched on the evaluate-apply process that Python uses to compute the value of an expression. This is central to understanding what a Python statement means.

Consider the following expression:

```
math.sqrt( 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 too cumbersome to write for real programs.

```
math.sqrt( 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.

Here's the illustration of what has to happen to evaluate these functions.

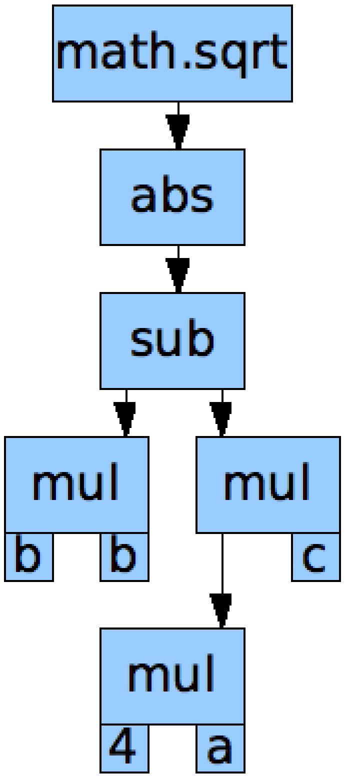

Figure 7.1: Evaluation of a Function

We read this picture starting from the top. Each box has an arrow showing the input it needs to work. Each function reaches down to get data from a lower-level function and passes the results back up to a higher-level function.

We're going to show this as a list of steps, with `>` to show how the various operations nest inside each other.

```
Evaluate the arg to math.sqrt:
> Evaluate the args to sub:
> > Evaluate the args to mul:
> > > Get the value of b
> > Apply mul to b and b, creating r3=mul( b, b ).
> > Evaluate the args to mul:
> > > Evaluate the args to mul:
> > > > Get the value of a
> > > Apply mul to 4 and a, creating r5=mul( 4, a ).
> > > Get the value of c
> > Apply mul to r5 and c, creating r4=mul( mul( 4, a ), c ).
> Apply sub to r3 and r4, creating r2=sub( mul( b, b ), mul( mul( 4, a ), c ) ).
Apply math.sqrt to r2, creating r1=math.sqrt( sub( mul( b, b ), mul( mul( 4, a ), c ) ) ).
```

Notice that a number of intermediate results were created as part of this evaluation. If we were doing this by hand, we'd write these down as steps toward the final result.

7.1.7 Function Exercises

1. **Temperature Conversions**.

   Package the Celsius to Fahrenheit conversion as a function. Similarly, package the Fahrenheit to Celsius conversion as a function. See *Simple Arithmetic : Numbers and Operators* for the basic formula.

   You'll create two functions, `c2f(tempC)` and `f2c(tempF)`.

   Use this function to prepare a handy little F to C conversion table. This should show temperatures from -10 F to 95 F in steps of 5 degrees, and the result of the `f2c()`.

2. **Mortgage Payment**.

   Package one of the mortgage payment calculations as a function. See *Simple Arithmetic : Numbers and Operators* for the basic formula.

   You'll create a function something like the following: `payment( principle, interestRate, numberOfPayments)`.

3. **Surface Air Consumption Rate (SACR)**.

   Package the SACR calculation as a function. See *Simple Arithmetic : Numbers and Operators* for the basic formula.

   You'll create one function like the following: `sacr( start, finish, depth, time )`.

4. **Wind Chill**.

   Package the Wind Chill calculation as a function. See *Simple Arithmetic : Numbers and Operators* for the basic formula.

   You'll create one function like the following: `chill( temperature, windSpeed )`.

   Using two **for** loops, you can make a wind chill table for temperatures from -20 up to 40, and wind speeds from 0 to 25.

5. **Roll Two Dice**.

See *Simulating All 100 Rolls of the Dice* for a simple loop that writes one hundred dice rolls. We can define a function which gets two random die values and returns the sum. You can replace the random-number generation with a slightly simpler-looking function call.

You'll replace the following two lines with a function call.

```
d1= random.randrange(6)+1
d2= random.randrange(6)+1
```

You'll create one function like the following: `dice()`.

7.1.8 Optional Function Exercises – Recurrence

These exercises demonstrate a technique called *recurrence* or *recursion*, in which a function is defined in terms of itself. This isn't a logical impossibility, since the definitions aren't completely circular.

As you can see from the procedures, each recursion involves a well-defined base case that isn't defined circularly, and then other cases that will eventually boil down to the base case. Since there is a well-defined base case, these functions aren't empty, circular definitions.

This recursion technique can, in some cases, out-perform a **for** loop. In theory, a recursive definition can express things which are more complex than can be expressed using loops. As a practical matter, many recursions are rather simple. We won't attempt to explain or justify it, instead we'll simply provide a batch of exercises. After working your way through these, you'll have as grasp on what is possible.

1. **Fast exponentiation**.

   This is a fast way to raise a number to an integer power. It requires the fewest multiplies, and does not use logarithms.

   Fast Exponentiation of integers, raises *n* to the *p* power

   (a) **Base Case**. If $p = 0$: return 1.0.

   (b) **Odd**. If p is odd: return $n \times \text{fastexp}(n, p-1)$.

   (c) **Even**. If p is even:

   compute $t \leftarrow \text{fastexp}(n, \frac{p}{2})$;

   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* for an alternate example of this exercise's algorithm. This version can be slightly faster than the loop we looked at earlier.

   Greatest Common Divisor of two integers, *p* and *q*

   (a) **Base Case**. If $p = q$: return p.

   (b) **p < q**. If $p < q$: return $\text{GCD}(q, p)$.

   (c) **p > q**. If $p > q$: return $\text{GCD}(p, p-q)$.

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, $1 \times 2 \times 3 \times \cdots \times n$.

   The formal definition is

   $$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

   $$0! = 1$$

   We touched on this in *Computing e*. This function definition can simplify the program we wrote for that exercise.

   Factorial of an integer, *n*

   (a) **Base Case**. If $n = 0$, return 1.

   (b) **Multiply**. If $n > 0$: return $n \times \text{factorial}(n-1)$.

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.

   The *n*th Fibonacci Number, F_n.

   (a) **F(0) Case**. If $n = 0$: return 0.

   (b) **F(1) Case**. If $n = 1$: return 1.

   (c) **F(n) Case**. If $n > 1$: return $\text{F}(n-1) + \text{F}(n-2)$.

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.

   Ackermann's Function of two numbers, *m* and *n*

   (a) **Base Case**. If $m = 0$: return $n + 1$.

   (b) **N Zero Case**. If $m \neq 0$ **and** $n = 0$: return $\text{ackermann}(m-1, 1)$.

   (c) **N Non-Zero Case**. If $m \neq 0$ **and** $n \neq 0$: return $\text{ackermann}(m-1, \text{ackermann}(m, n-1))$.

   Yes, this requires you to compute $\text{ackermann}(m, n-1)$ before you can compute $\text{ackermann}(m-1, \text{ackermann}(m, n-1))$.

7.2 Flexibility and Clarity : Optional Parameters, Keyword Arguments

Python gives us ways to add flexibility and clarity to our function definitions. We'll introduce how to add flexibility by using optional parameters in *Flexible Definitions with Optional Parameters*. Then, in *Adding Clarity with Keyword Argument* we show how Python can use named keyword parameters to add clarity.

In *Object Methods – A Cousin of Functions* we'll describe how to use *method functions* as a prelude to subjects in *Basic Sequential Collections of Data*. Methods are a kind of first cousin to functions.

There is even more sophistication in how Python handles function parameters. Unfortunately, this has to be deferred to *A Dictionary of Extra Keyword Values*, as it depends on a knowledge of dictionaries, which we won't get to until :ref'data.map'.

7.2.1 Flexible Definitions with Optional Parameters

One common design pattern could be called a "Function With A Special Case". We may have a function that covers 80% of our needs, but once in a while, we need to provide additional parameters to cover an unusual or special case.

We've seen some examples if this. For example, back in *Say It With Functions*, we saw that the `int()` function has an optional parameter of the *base* for the string to convert. Converting decimal strings covers most of the conversions; converting strings in another base is rare. Because of this pattern of use, the authors made the *base* parameter optional.

Python offers us two kinds of function parameters: required and optional.

- For required parameters, your function evaluation must provide the actual argument value.
- For optional parameters, you may provide a value; if you omit the value, a default value is used for you. In the case of the `int()` function, the *base* parameter has a default value of `10`.

Keeping Our Options Open. The way we make a parameter optional is by providing a default value for the parameter as part of the function definition. What we saw in *Adding New Verbs : The def Statement* was the syntax for defining functions with required parameters. We'll need some additional syntax to define optional parameters.

We can provide a parameter's default value to a function definition. Here we show that a parameter can have an optional initial value that is used if no argument value is supplied.

```
def name ( [ parameter [ = initializer ] ] , ... ):
    suite
```

The [*parameter* [= *initializer*]] tells us that a parameter, in general, is optional. Recall that we don't actually enter the [and]'s, they're markers to help us understand optional parts of the syntax. The [= *initializer*] tells us that a parameter may or may not have an initial value. While the [and]'s tell us that the initializer is optional, the `=` is essential punctuation for separating the parameter name from the initial value.

Many Options. When there are a number of optional elements we will have several forms of function definitions. We'll look some of the various combinations that are available.

- '`def myFunction():`' is the no-parameters version. When you evaluate this function, you don't provide any argument values.
- '`def myFunction(req):`' defines a required parameter. When you evaluate this function, you must provide an argument value for the required parameters.
- '`def myFunction(opt=value):`' uses an initializer to define an optional parameter. When you evaluate this function, you may provide a argument value for this optional parameter. If you don't provide an argument value, the default value will be used.
- '`def myFunction(req,opt=value):`' is a mixture of required and optional values. With one optional parameter, there are two ways to call this function: '`myFunction(r)`' and '`myFunction(r,o)`'.
- '`def myFunction(req,opt1=value,opt2=value):`' is a mixture of required and optional values. With two optional parameters, there are four ways to call this function.

Other Kinds of Dice. Here's a small example of the "most of the time with exceptions" design pattern. We've been talking on and off about the casino game of Craps, which uses 6-sided dice. If we were talking about role-playing games, we might introduce dice based on the Platonic solids which include 4-sided, 6-sided, 8-sided, 12-sided and 20-sided dice. We could introduce other dice with asymmetric sides that include 10-sided or even 100-sided dice. How can we define this in Python?

Here's a `roll()` function definition that has an optional parameter for the number of sides on the die. If no value is provided, a default is used, which simulates a 6-sided die. If a value is provided, this is the number of sides. Note that we don't require a specific kind of dice, and are perfectly willing to roll 11-sided dice if that's what the game calls for.

```
import random
def roll( sides= 6 ):
    return random.randrange(1,sides+1)
```

We can use this function two ways:

```
>>> roll()
1
>>> roll(8)
2
```

When you define a function like this in **IDLE**, you'll notice something very cool happens when you use the function. When you type `roll()` a pop-up window appears that says `sides=6`, displaying the parameter and the default value.

Rules of the Game. There's an additional rule about positional parameter syntax that can't easily be captured in our simple grammar depiction. Python requires us to place all of the required parameters before all of the optional parameters.

This "required-before-optional" rule can seem capricious. However, the Python program must assign argument values to the parameter variables by position, from left to right.

Imagine the following hypothetical scenario.

```
def aBadIdea( opt=123, req ):
    some function body
```

What's so confusing about that?

Imagine we evaluate this function using `aBadIdea( 12 )`. Which parameter gets the value `12`? Since the first parameter is optional, maybe `12` is really the second parameter.

If `12` is supposed to be the *second* parameter, which is required, then the function evaluation is confusing because it sure looks like `12` is the value for the first parameter.

One of the Python principles is "In the face of ambiguity, refuse the temptation to guess." In this case, ambiguity is eliminated by putting the required parameters first.

Parameter Assignment. With required parameters in the beginning positions, and optional parameters in the ending positions, the parameter assignment process can handle each of the following situations.

- **Evaluated With Too Few Values**. The function evaluation doesn't provide enough argument values for all parameters. There are two versions of this situation:
  - If there aren't enough values for all of the required parameters, this is an error.
  - If all the required parameters have values, do the following:
    1. Assign argument values to the required parameter variables in order from left to right.
    2. Assign remaining argument values are assigned to optional parameter variables until the argument values run out.

3. Finally, the optional parameter variables that didn't get an argument value will be assigned their default initializer values. Once the parameter variables all have a value assigned, the function suite can be executed.

- **Evaluated With The Right Number of Values**. The function evaluation provides one argument value for each parameter variable. This means each required parameter and each optional parameter will have a value set by an argument. Once the parameters variables all have a value assigned, the function suite can be executed.
- **Evaluated With Too Many Values**. The function evaluation provides more argument values than the allowed parameter variables. For now, we have to consider this as an error. Hint: there's a way to cope with this, but it requires some additional types of collection data that we haven't covered yet. The full set of rules is something that has to wait until *Mappings : The dict*.

The "Too Many Values" rule is open to some debate. On one hand, if the arguments don't match the parameters, something is clearly wrong. On the other hand, it can be useful to specify a function that will handle an arbitrary number of parameters. The Python language doesn't impose one view or the other, it allows you to pick a side in the debate. For now, we have to treat too many argument values as an error. We will, eventually, have the option of coping with this situation.

7.2.2 Optional Parameter Example

Here's an example that has both required and optional parameters. This is a function that computes the final payout from bets with different forms of odds. In Roulette, all odds are stated as "x to 1", a simple multiplication of the amount you bet. In Craps, however, some odds are "x to 2", a more complex multiplication. Some bets includes a "house rake", typically 5% of the winnings.

This is actually one calculation that has several forms. We'd like to be able to use our payout function as follows:

- `payoutFrom( bet, 5 )` for the simple cases in Roulette. This would be for the 5:1 payout; you get $5 for every $1 bet.
- `payoutFrom( bet, 6, 5 )` for the odds bets in Craps. The bet pays 6:5; it wins $6 for every $5 bet.
- `payoutFrom( bet, 5, 6, .05 )` for the "don't pass odds bets" in Craps. This is the odds for bets *layed against* a point of 6 or 8. The bet pays 5 to 6 with a 5% commission on the winnings. You would bet $24 to win $20 less 5%, which is $19. Since you keep your original $24, you'll have a total of $43.

```
def payoutFrom( bet, odds, to=1, rake=0 ):
    return bet*odds/to * (1-rake)
```

Here are some additional examples, using a $10 or $24 dollar bet.

```
>>> payoutFrom( 10, 5 )
50
>>> payoutFrom( 10, 6, 5 )
12
>>> payoutFrom( 24, 5, 6, 0.05 )
19.0
```

Common Errors. If a required parameter (a parameter without a default value) is missing, this is a basic `TypeError`.

Here's an example of a script where we define a function that requires two argument values. We call it with an incorrect number of arguments to see what happens.

badcall.py

```
#!/usr/bin/env python
from __future__ import print_function
def hack(a,b):
    print(a+b)

hack(3)
```

When we run this example, we see the following error message.

```
$ python badcall.py
Traceback (most recent call last):
  File "badcall.py", line 5, in ?
TypeError: hack() takes exactly 2 arguments (1 given)
```

Tip: Debugging Optional Parameters

There are three areas for the common mistakes in function definition: the **def** statement itself, the **return** statement and using the function in an expression.

The syntax of the **def** statement contains three parts. If you have syntax errors on the definition, you've got one of these three wrong, or you're misspelling "def".

- The name, which is a Python name, following the name rules in *Python Name Rules*.
- The parameters, which is list of names, separated by commas. The `()`s around the parameter list is required, even if there are no parameters. The `,` to separate parameters is required. If we use initializers, the `=` to separate the variable and the initial value is required punctuation. Also note that the initializer is evaluated when the **def** statement is executed, not when the function is evaluated.
- The indented suite, a block of Python statements.

In our syntax summaries, we use `...` to show things that can be repeated, we don't actually enter the `...`.

The required parameters (which have no initial values) must precede the optional parameters (which have initial values).

The **return** statement is how the return value is defined. If you omit this, your function always returns `None`. The **return** statement also ends execution of the function's body; if you have this statement out of place, your function may not fully execute.

When you use the function, you have to provide actual argument values for each parameter that doesn't have an initializer. Since positional parameters are simply matched up in order, the arguments you present when you use of the function must match parameters of the definition.

7.2.3 What About the `range()` Function?

The essential rule of optional parameters is that all required parameters must be first, all optional parameters have initial values and must come after the required parameters. But, when we look at the Python `range()` function, we see what appears to be a violation of this rule. What we're seeing, however, is a slightly more sophisticated way of handling the parameter values.

Here are two examples that can be confusing at first.

- `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 not y:
        low, high, step = 0, x, 1
    elif not z:
        low, high, step = x, y, 1
    else:
        low, high, step = x, y, z

    # Real work is done with low, high 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.

Bottom Line. There must be a value for all parameters. The basic rule is that the values of parameters are set in the order in which they are defined. If an argument values is missing, and the parameter has a default value, this is used.

These rules define *positional parameters*: the position is the rule used for assigning argument values when the function is evaluated.

7.2.4 Default Value Restriction

It's very important to note that you should not provide a mutable object as a default value.

In *Basic Sequential Collections of Data* and *More Data Collections* we'll look at objects with values that can change. These cannot be provided as default values for functions. What will happen is that each time the function is evaluated and a default value is used, the same mutable object will be reused.

For now, however, any kind of number or string can be provided as a default value without giving it a second thought.

We'll revisit this in *Defining More Flexible Functions with Mappings* when we've seen these mutable objects.

7.2.5 Adding Clarity with Keyword Argument

Initially, we provided argument values for a function's parameter variables by *position*. When we evaluate a function, we provide the argument values in the same order that the function parameters were defined. Most mathematical functions follow the same pattern as `sin()` and `sqrt()`: they only have a single parameter. With a single parameter, matching values with parameters by position is pretty easy. When we get to three or four parameters, we reach the edge of what is easy to understand.

To keep things easy to read, Python provides yet another way to provide an argument value for a parameter. We can use explicit keywords to make the association between argument value and parameter variable perfectly clear. We note that effort spent preventing problems saves time in debugging and repairing the problems. The traditional value proposition is that an ounce of prevention is a pound of cure, giving a relative value of 16:1. Spending a few minutes picking parameter names that are easy-to-understand has a huge benefit over the life of your program.

Here's an example of calling a function using keywords and positional parameters. We'll use our `payoutFrom()` function from the previous section, *Flexible Definitions with Optional Parameters*. Note that we can mix keywords and positions, but we have to stick to a few rules so that Python can properly assign arguments to parameter values.

```
>>> payoutFrom( 10, 5 )
50
>>> payoutFrom( 10, odds=6, to=5 )
12
>>> payoutFrom( rake=0.05, odds=5, to=6, bet=24  )
19.0
```

Positional and Keyword. We have a total of four variations: 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.

The syntax for providing argument values is very flexible. Here are the semantic rules Python uses to assign argument values to parameter variables.

1. **Keywords**. Assign values to all parameters given by name, irrespective of position. If the keyword on the function evaluation is not an actual parameter variable, raise a `TypeError`.
2. **Positions**. Assign values to all remaining parameters by position. It's possible to mistakenly assign a value by both keyword *and* position; if so, raise a `TypeError`.
3. **Defaults**. Assign defaults for any parameters that don't yet have values and have defaults defined; if any parameters still lack values, raise a `TypeError`.

Average Dice. Here's another example with a simple parameter list. We need to know how many samples to average. The number of sides on each die, however, has an obvious default value of six, because six-sided dice are so common. However, we'll allow a user to override the number of sides in case they want to simulate rolls of 4-sided or 12-sided dice.

```python
import random

def averageDice( samples, sides=6 ):
    """Return the average of a number of throws of 2 dice."""
    s = 0
    for i in range(samples):
        d1,d2 = random.randrange(sides)+1,random.randrange(sides)+1
        s += d1+d2
    return float(s)/float(samples)
```

Next, we'll show a number of different kinds of arguments to this function: keyword, positional, and default.

```python
test1 = averageDice( 200 )
test2 = averageDice( samples=200 )
test3 = averageDice( 200, 6 )
test4 = averageDice( sides=6, 200 )
```

When the `averageDice()` function is evaluated to set *test1*, the positional form is used for *samples*, and a default for *sides*. The second call of the `averageDice()` function uses the keyword form for *samples*, and a default for *sides*. The third version provides two values positionally. The final version supplies a keyword value for *sides*; the value for *samples* is supplied by position.

Tip: Debugging Keyword Parameters

When you use a function, you have to provide actual argument values for each parameter that doesn't have an initializer. Two things can go wrong here: the syntax of the function call is incorrect in the first place, or you haven't provided values to all parameters.

You may have fundamental syntax errors, including mis-matched `()`, or a misspelled function name.

You can provide argument values by position or by using the parameter name or a mixture of both techniques. Python will first extract all of the keyword arguments and set the parameter values. After that, it will match up positional parameters in order. Finally, default values will be applied. There are several circumstances where things can go wrong.

- A parameter is not set by keyword, position or default value
- There are too many positional values.
- A keyword is used that is not a parameter name in the function definition.

7.2.6 Optional and Keyword Parameter Exercises

[No, the exercises aren't optional, the parameters are optional.]

1. **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 evaluate the `win()` function seven 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.

2. **Which is Clearer?**.

   How do keyword parameters help with design, programming and testing? Which is clearer, positional parameter assignment or keyword assignment? Should one technique be used exclusively? What are the benefits and pitfalls of each variation?

7.2.7 Object Methods – A Cousin of Functions

This is a quick look forward toward *Basic Sequential Collections of Data*. This section is an introduction to some syntax we'll use extensively.

We've seen how we can create functions and use those functions in expressions. Python has a closely-related technique called *method functions* or just *methods*. The functions we've used so far are available anywhere. A method function, on the other hand, is part of a specific object. The object's *class* defines what methods and what properties the object has.

All of the Python data types we're going to introduce in *Basic Sequential Collections of Data* will use method functions. This section will focus on the basic principles of how you *use* method functions. As with ordinary functions, you need to know how to use them before you can design them.

The syntax for using (or "calling") a method function looks like this: `.(=,)`

```
someObject  aMethod  [ [ parameter  ] argument ]  ...
```

A single `.` connects the owning object (*someObject*) with the method name (`aMethod()`). As with a function, the `()` are essential to mark this as a method function evaluation.

The [[parameter=] argument]'s indicate that the parameter keywords are permitted. Also, the [and]'s indicate that – in general – argument values are optional. Some method functions will compute results based on the object itself, not on arguments to the function. The ... means that the argument values are repeated. The `,` is the separator between argument values.

We have to make an important distinction here between the syntax and the semantics of using a function:

- The syntax summary say that we can have any number of argument values.
- Semantically, however, the argument values will be matched against the declared list of parameter variables. If we provide too many values or too few values, we'll get an error.

It's important to note that we can't capture all of the semantics in our syntax summaries. Consequently, we have to watch out for any of Python's additional rules.

Two Small Examples. Here are two examples of how we apply these method functions to string objects.

```
>>> "Hi Mom".lower()
'hi mom'
>>> "The Walrus".upper()
'THE WALRUS'
```

In this example, we apply the `lower()` method function of the string object `"Hi Mom"`.

We apply the `upper()` method function of the string object `"The Walrus"`.

When we looked at the `math` and `random` modules in *Meaningful Chunks and Modules*, we were looking at module functions. These module functions are imported as part of a module; that's why their names are *qualified* by the name of the owning module. When we import `math`, use the qualified name `math.sqrt()`. The syntax of object method functions follows the module function pattern.

Modules and objects are two examples of the principle of *encapsulation*. There are numerous differences between objects and modules, and we'll look at these more closely when it's appropriate. The important similarity is that both modules and objects are containers of functions. Modules contain functions and objects contain method functions.

Bottom Line. We want to be able to use method functions starting with *Basic Sequential Collections of Data*. Once we've learned how to use method functions,We'll show how you create classes and method functions in *Defining New Objects*. We'll show how you create modules and module functions in *Modules : The unit of software packaging and assembly*.

7.2.8 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 _ 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. Some programmers will put in extra `()`s just to make line breaks neat.

Function names are typically `lower_case_with_underscores()` or `mixedCase()`. A few important functions were once done in `CapWords()` style with a leading upper case letter. This can cause confusion with class names, consequently, the recommended style is a leading lowercase letter for function names.

In some languages, it is traditional to name related functions with a common prefix. For example, a related group of "inet" functions may be `inet_addr()`, `inet_network()`, `inet_makeaddr()`, `inet_lnaof()`, `inet_netof()`, `inet_ntoa()`, etc. Because Python has classes (covered in *Data + Processing = Objects*) and modules (covered in *Modules : The unit of software packaging and assembly*), this kind of function-name prefix is not used in Python programs. The class or module name is the prefix.

Parameter names are also typically typically `lower_case_with_underscores()` or `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 built in function `range()`. We use a trailing `_` to sort this out.

```
def integrate( aFunction, aRange ):
    """Integrate a function over a range."""
    body of the function
```

In other languages (notably Visual Basic), it is common to prefix variables with complex codes that indicate the scope and type of the variable. This is sometimes called "Hungarian Notation" because there's a kind of family name given first.

Because Python is object-oriented, these kinds of prefix codes will be inaccurate or incomplete. Also, Python strives for an English-like look, and short, cryptic prefixes interfere with this look. Python parameter names should be clear, short words that work well as keywords.

Formatting. Blank lines are used sparingly in a Python file, generally to separate unrelated material. Typically, 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.

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))
```

Getting Help. The docsting can be retrieved with the `help()` function.

`help`(*object*) → string

Prints help on the specific object. For functions, classes or modules, this prints the object's docstring. For a variable, it prints the value of the variable.

When executing `help()` while using Python directly (*not* using **IDLE**), you'll be interacting with a help viewer that allows you to scroll forward and back through the text.

For more information on the help viewer, see *Getting Help*.

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.
```

7.3 A Few More Function Definition Tools

This chapter covers a few more function definition techniques.

We looked at multiple assignment back in *Combining Assignment Statements*. We'll look at how a function can return multiple values in *Returning Multiple Values*.

We'll talk about how the variables used inside a function's suite are kept private in *Keeping Track of Variable Names – The Namespace*. Finally, we'll look at other ways we interact with functions in *Talking About Functions Behind Their Backs*.

We'll also cover *The global Statement*, more as a cautionary note, because the **global** statement allows you to build programs which are intentionally confusing.

7.3.1 Returning Multiple Values

We looked at the multiple assignment statement in *Combining Assignment Statements*. In order to work with this, we need to define a function which can return multiple values. Python has some built-in functions that have this property. For example, `divmod()` returns the divisor and remainder in division. We can imagine defining a function, `rollDice()` that would return two values showing the faces of two dice.

In Python, we return multiple values by returning an object called a `tuple`. The following is a quick example. We'll wait for *Doubles, Triples, Quadruples : The tuple* for more complete information on the `tuple` class of objects.

This shows the `rollDice()` function, which returns two values. (To split a hair, I should say that it returns a two-valued `tuple`.) You'll recall from *Combining Assignment Statements* that Python is perfectly happy with multiple expressions on the right side of an **assignment** statement, and multiple destination variables on the left side. Returning multiple values from a function is one of the logical consequences of multiple assignment; it makes this particular kind programming task considerably simpler than in other languages.

rolldice.py

```
#!/usr/bin/env python
from __future__ import print_function
import random

def rollDice():
    return 1 + random.randrange(6), 1 + random.randrange(6)

d1,d2=rollDice()
print(d1, d2)
```

This can simplify a number of previous examples. In particular, look at *Roll Dice Until Craps* in *The while Statement* and *Counting Sevens* for examples that can be simplified by using this `rollDice()` function.

Important: Debugging

A function that returns multiple values is rather specialized. For now, it can only be used in a multiple assignment statement. When we learn more about tuples (in *Doubles, Triples, Quadruples : The tuple*), we'll see how we can do a few additional things with these kinds of functions.

The number of variables on the left-hand side of the multiple assignment statement must match the number of values on the **return** statement of the function. It helps to emphasize this in the function's docstring, so that it is perfectly clear how many values the function returns.

7.3.2 Keeping Track of Variable Names – The Namespace

This is the newbie's overview of how Python determines the meaning of a name. We'll omit some details and touch on a few important points. For more information, see section 4.1 of the *Python Language Reference* [PythonRef].

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 object names, they are called *namespaces*. The global namespace is available to all modules that are part of the currently executing Python script.

Additionally, each function and module has its own private namespace. Some structures we haven't covered yet, including classes and anonymous blocks of code given to the **exec** statement will also have private namespaces.

It is important to note that when one function calls another function, each function is evaluated in a private namespace. This means that the namespaces nest inside each other. When a function is evaluated, a namespace is created. When a function evaluates other functions, they have nested namespaces. When a function finishes at the end of the suite, or because of a **return** statement, the namespace is removed. The nested namespaces are unwound in reverse order from the way they were created.

Names are resolved using the nested collection of namespaces that define an execution environment. Python always checks the innermost, or most-local dictionary first, and then checks the global dictionary.

Preventing Collisions. This nesting of namespaces means that your function can use variable names that are also used by other functions, or are part of the global namespace without worrying about *collisions*. A collision occurs when a variable's value is changed unexpectedly. If Python only had once namespace, then we would have to look at every function and module to be sure that our functions didn't use variables that were changed by some other module or function.

Since the local namespace of the function is searched first, names are understood locally. Searching other, non-local namespaces, is a kind of fall-back plan when the variable is not found in the local namespace. Generally, we write our functions so that *all* the variables are either parameters or variables created inside the function. Rather than burn up brain calories trying to work out the namespace that provides needed variables, we strive to be sure all names are local.

Nested Functions. Consider the following incomplete script. This doesn't really do much except show an outline of how programs are often defined as multiple functions. We'll look at the three nested contexts from outermost to innermost.

```
def rolldice( dice, sides=6 ):
    do some work
def average( rolls, dice ):
    for i in range(rolls):
        r= rolldice( dice, sides )
    for i in range(rolls):
        r2= rolldice( 2*dice, sides )

rolls=10
sides=8
average( rolls*12, 2 )
```

1. The main script executes in the global namespace. It defines two functions, `rolldice()` and `average()`. Then it defines two global variables, *rolls* and *sides*). Finally, it evaluates one of those functions, `average()`.
2. The `average()` function has a local namespace, where five variables are defined. Two of these are parameter variables: *rolls*, *dice*. The rest are ordinary variables *i*, *r*, and *r2*. When `average()` is called from the main script, the local *rolls* will hide the global variable with the same name. The global *rolls* is `10`, but the local value is `10*12`. Can you see why?

   The reference to *sides* is not resolved in the local namespace, but is resolved in the global namespace. This is called a *free variable*, and is generally a symptom of poor software design.
3. The `rollDice()` function (which has it's suite of statements omitted) has a local namespace, where two parameter variables are defined: *dice* and *sides*. When `rollDice()` is called from `average()`, there are three nested scopes that define the environment: the local namespace for `rollDice()`, the local namespace for `average()`, and the global namespace for the main script.

   The local variables for `rollDice()` hide variables declared in other namespaces. The local *dice* hides the variable with the same name in `average()`. The local *sides* hides the global variable with the same name.

Functions for Looking At Namespaces. If you evaluate the built-in 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 evaluate the built-in function `locals()`, you'll see the same variables as you will from `globals()` because the top-level Python window interprets your input in the global namespace. However, if you evaluate the `locals()` function from within the body of a function, you'll be able to see the difference between local and global namespaces.

The following example shows the creation of a global variable *a*, and a global function, *q*.

```
>>> a=22.0
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': 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 0x6feb0>, '__doc__'
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'q': <function q at 0x6feb0>, '__doc__'
>>> q(22.0,7.0)
{'a': 3.1428571428571428, 'y': 7.0, 'x': 22.0}
```

1. When we evalate `globals()` initially, it has some `__builtin__` objects, plus our variable *a*.
2. In our function `q()`, we print the value of `locals()` to see what's in the local namespace while `q()` is being evaluated.
3. We show the result of `locals()` and `globals()`. At the top-level of Python, they're the same.
4. When we evaluate `q()`, we see that the locals inside `q()` are just the parameters.

A built-in function `vars()` 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. It turns out that the local variables are kept in a Python internal object named ___*dict*___. The `vars()` function retrieves this information.

The function `dir()` also examines the internal ___*dict*___ object for a specific object; it will 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.

Important: Debugging

There are two big problems people have with namespaces. First, they forget that variables belong to a specific namespace, and try to use variables as though they exist globally. Some languages (COBOL, original BASIC) assume that all variables are global. In languages like C and Pascal, it is relatively easy to declare a global variable. Python tries to avoid the kinds of problems that are caused by the hidden coupling that global variables cause.

The other problem is failing to include the module name to refer to an imported function definition. When we say `import math`, the `math` module is created with its own namespace, and all the **def** statements that are imported execute in `math` module's namespace. Because of this, we have to say `math.sqrt`, including the module name in front of the function name.

This second problem may stem from failing to note what happens with the **import** statement. If we type a definition directly at the `>>>`, it is defined in the global namespace. If, however, we import a module with the definition, the **def** statement executes in the *module*s namespace. Since the definition happens in the module's namespace, we have to qualify the function name with the module name.

7.3.3 Talking About Functions Behind Their Backs

This section has some additional notes that can help you read other Python programs. We don't particularly recommend these techniques for newbies. However, you'll need to know about this so you can read more sophisticated Python programs you find.

One interesting consequence of the Python world-view is that a function is an object of the class `function`. Other objects like this include all of the built-in functions. There are related objects called generators. They are all variations on the theme of function.

Each function defined in our program is an object that we create with the **def** statement. A string object is something we create with `""`s. A number object is created with a numeric literal.

It turns out that a function object can be used in three very different ways, depending on the context in which the name occurs.

- We can apply the function when we follow the name with `()`s.
- We can also create an alias for a function by slapping another variable name on the object.
- And, we can assign additional attributes to the function, above and beyond the name and the docstring.

Apply The Function. By far, the most common use for a function object is to use `()`s to apply the function to argument values. This is what we've seen in detail throughout this part. This is the ordinary use for functions.

This explains why a function with no argument values still needs empty `()`s. The `()`s are the syntax that tells Python to evaluate the function.

You can think of the `()`s as a kind of operator. This `()` operator *applies* a function object to the argument values.

```
function ( [ [ parameter = ] argument ] , ...  )
```

Alias The Function. When we use the name of a function without any `()`s, we are not applying the function to argument values. We're talking about the function; we're not asking the function to do anything. When we leave off the `()`s, we're making the function into a noun. It's the difference between talking about the verb "to write" and actually writing a note to someone.

One way that we talk about a function is to assign another name to the function. This creates an alias for the function. This can be dangerous, because it can make a program obscure. However, it can also simplify the evolution and enhancement of software. We have to cover it because it is a very common technique.

Imagine that the first version of our program had two functions named `rollDie()` and `rollDice()`. The definitions might look like the following.

rolldice.py – First Version

```
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 `rollDice()` function. Here's our new, slick, expanded function that rolls any number of dice.

```
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 the original `rollDie()` and `rollDice()` are just special cases of `rollNDice()`, we should replace them with something like the following.

rolldice.py – Second Version

```
def rollDie():
    return rollNDice( 1 )
```

```
def rollDice():
    return rollNDice()
```

This revised definition of `rollDice()` is really just an another name for the `rollNDice()`. We can see that our definition of `rollDice()` doesn't add anything new. Compare it with `rollDie()`, which supplies an argument value to the `rollNDice()` function.

Because a function is an object assigned to a name, we can have multiple names for a function. Here's how we create an alias to a function.

```
rollDice = rollNDice
```

An alias is simply a name change.

There are other uses for this technique. The most common variation on this technique is to make a local name out of a qualified name. For example, we may see something like the following:

```
rand= random.randrange
```

It turns out that evaluating this kind of local function variable is slightly faster than evaluating the qualified name. This is because the qualification requires Python to lookup the function name in the module's namespace, an operation that requires a tiny atom of additional time. Consequently, you'll see this little optimization technique in many Python programs.

Get Attributes of the Function. 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* [PythonRef] and section 2.3.9.3 of the *Python Library Reference* [PythonLib].

____doc____ Docstring from the first line of the function's body.

____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.author= "sfl"
```

7.3.4 More Function Exercises

These exercises ask you to define two functions. One of which is used by the other. In the "Maximum Value of a Function" exercise, you'll define some small function which is then used by the `maxFx()`. Similarly, in the "Integration" exercise, you'll create some small function which is used by the `integrate()` function.

And yes, the integration exercise is almost calculus. But really, it's just the sum of the areas of a bunch of rectangles, so it's inside the box of algebra.

1. **Maximum Value of a Function**.

   Given some integer-valued function `f()`, we want to know what value of x has the largest value for `f()` in some interval of values. For additional insight, see [Dijkstra76].

   Imagine we have an integer function of an integer, call it `f()`. 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()` in the interval $0 \leq x < 10$ occurs when x is 9.

   What about `f3()` in the range $-10 \leq x < 10$?

   ### Max of a Function, *F*, in the interval *low* to *high*

   ```
   (a) Initialize.
       $x \leftarrow low$;
       $max \leftarrow x$;
       $max_F \leftarrow \mathrm{F}(max)$.
   (b) Loop. While $low \leq x < high$.
       i. New Max? If $\mathrm{F}(x) > max_F$:
           $max \leftarrow x$;
           $max_F \leftarrow \mathrm{F}(max)$.
       ii. Next X. Increment $x$ by 1.
   (c) Return. Return $max$ as the value at which $\mathrm{F}(x)$ had the largest value.
   ```

2. **Integration**.

   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 $h = \frac{b-a}{s}$. 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.

Integrate a Function, *F*, in the interval *a* to *b* in *s* steps

```
(a) Initialize.
    $x \leftarrow a$
    $h \leftarrow \frac{b-a}{s}$
    $sum \leftarrow 0.0$
(b) Loop. While $a \leq x < b$.
    i. Update Sum. Increment $sum$ by $F(x) \times h$.
    ii. Next X. Increment $x$ by $h$ .
(c) Return. Return $sum$ as the area under the curve F() for $a \leq x < b$.
```

7.3.5 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, the variables are discarded.

Note that the namespace that will be used for evaluation of a function is distinct from the namespace in effect when we *define* that function. Our **def** statements are almost always executed in the global namespace.

When we looked at the **import** statement briefly in *The math Module – Trig and Logs* we glossed over this. When we say `import math`, this creates the name `math` in the global namespace. However, the functions we want to use (like `sqrt()`) are in the local namespace of module `math`. We need to say `math.sqrt()` to make this ownership clear.

When we move on to talk about classes (*Data + Processing = Objects*) and modules (*Modules : The unit of software packaging and assembly*), we'll see other contexts in which the local and global namespaces are different.

The standard rules, then, are these:

- Names are created in a local namespace.
- The interactive session (or the initial script) has the global namespace as it's local namespace.
- Every other context (e.g. within a function's suite or within a module) uses a distinct local namespace.

Python offers us the **global** statement to change these rules.

```
global name , ...
```

The **global** statement tells Python that the following names are part of the global namespace, not the local 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: Global Warning

The **global** statement has a consequence of creating a hidden coupling between pieces of software. This can lead to difficulty in maintenance and enhancement of the program. Classes and modules allow us to assemble complex programs without the hidden coupling of global variables.

As a general policy, we discourage use of the **global** statement. We present it here so that you can read someone else's programs.

CHAPTER

EIGHT

GETTING OUR BEARINGS

In sailing terms, we're *rounding the mark* : we've finished one leg of our journey and we're starting the next leg. Many sailing race courses are variations on a simple *out and back* design. When racing on one of these courses, you cover the same water going in opposite directions. Many courses are laid out so that you start the race going into the wind and finish the race going away from the wind.

Following this pattern, we're covering programming by first viewing it as procedural statements and then viewing it as data structures. Neither, by itself, is a complete picture. Each depends on the other: we need data to process with our procedural statements; we need procedural statements to process our data.

Data and processing are two sides to the same coin. This duality is central to all programming, and leads to a terrible dilemma when teaching programming: which comes first? We can't easily teach data without the statements to process it. Neither can we teach the processing statements without covering the associated data structures.

8.1 Where We've Been; Where We're Going

We'll look over our shoulder at the first part of the course and review the procedural statements. Then we'll survey the course as a whole so we can see the additional marks we'll be sailing around. Finally, we'll look at the next leg and see what we'll be doing in the next few chapters.

Looking Back. In the first parts (*Arithmetic and Expressions*, *Programming Essentials*, *Some Self-Control* and *Organizing Programs with Function Definitions*), we introduced almost all of the procedural elements of the Python language. We started with expressions using the numeric data types: integer, float, long integer and complex. We've covered fourteen of the twenty statements that make up the Python language.

- Expression statements – for example, a function evaluation where there is no return value.
- The **import** statement – to include a module into another module or program.
- The **assignment** statements, from simple to augmented – used to set the value of a variable.
- The **pass** statement – which does nothing, but is a necessary placeholder for an **if**, **while** or **class** suite that is empty.
- The **if** statement – for conditionally performing suites of statements. This includes **elif** and **else**.
- The **for** and **while** statements – for performing suites of statements using a list of values or while a condition is held true.
- The **break** and **continue** statements – for short-cutting loop execution
- The **def** statement – to create a function.
- The **return** statement – to exit a function, possibly providing the return value.

- The **assert** statement – to confirm the program is in the expected state.

To be complete, we looked at these statements, also.

- The **print** statement – to provide visible output. We prefered to use the `print()` function, however.
- The **global** statement – to adjust the scoping rules, allowing local access to global names. This can cause confusion, so we'd like to avoid it.
- The **del** statement – used to remove a variable, function, class, module or other object. This statement isn't much use to newbies.

Off In The Distance. There are a few topics that need to be deferred until later.

- We'll look at some more advanced statements of the Python language in *Additional Processing Control Patterns*; this will include **try**, **except** and **yield**.
- The **class** statement will be covered in detail in chapters on object oriented programming, starting with *Data + Processing = Objects*.
- We'll revisit the **import** statement in detail in *Modules : The unit of software packaging and assembly*. Additionally, we'll cover the **exec** statement in *Wrapping and Packaging Our Solution*.
- The **with** statement is handy for creating a context in which processing can occur. Working with a file, for example, is a common kind of context. A file must be opened and closed; these two operations define the context.

Looking Ahead. The next parts focus on adding various data collections that are part of the Python language. The subject of data representation and data structures are 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 interesting data by giving it a meaningful structure.

There's a world of difference between a random string of letters, and a meaningful poem. In poetry, the line breaks carry additional meaning. Sometimes the punctuation or even capitalization (or lack of capitalization) can carry more subtle shades of meaning. In music, the "song" structure with verse, chorus, bridge and repeats gives us a way to remember a bunch of words. Structure is the essential ingredient that lifts a piece of data above the background noise to make it meaningful and informative.

There's a rich family tree of data types in Python. This list will show how we're going to cover all of these various data types.

- Unstructured. We looked at most of these in *Simple Arithmetic : Numbers and Operators*. These objects have a single value that we could describe as "atomic" in the sense of being indivisible (not in the sense of radioactive).
  - Plain Integer.
  - Floating Point.
  - Long Integer.
  - Complex. This straddles the line with structured types; it's so simple it may as well be an unstructured type. These have more in common with the other numeric types than the collection types.
  - Boolean. We looked at this in *Truth and Logic : Boolean Data and Operators*.
- Structured. These are collections of items. This is where we're headed next.
  - Sequence. These kinds of collections keep items in order; the items can be identified by their position. We'll introduce the common features of these types in *Collecting Items in Sequence*.
    - String, Unicode String. *Sequences of Characters : str and Unicode*.
    - Tuple. *Doubles, Triples, Quadruples : The tuple*.

    * List. *Flexible Sequences : The list* and *Common List Design Patterns*.

- Set. This kind of collection doesn't identify items by position or a key; it simply collects the items. *Collecting Items : The set*.
- Mapping. This kind of collection identifies items by a key value; there's no particular order to the items.
    * Dictionary. Currently, this is the only type of mapping. *Mappings : The dict*.
- File. A file is what we use to make our data structures *persistent* by writing them to devices like hard disks or removable USB drives. *External Data and Files* and *Files, Contexts and Patterns of Processing*. Even something as remote-sounding as a file available in the Internet, identified by it's URL, can be used as if it were a simple file. *File-Related Library Modules*.
- Other.
    * Exception. *The Unexpected : The try and except statements*. An exception part of event-driven programming. These break us out of the strictly sequential mode that our programs normally use.
    * Generator Functions and Iterators. *Looping Back : Iterators, the for statement and Generators*. This chapter will give us a number of very cool techniques that we can use with the **for** statement.
    * Function. We started in *Organizing Programs with Function Definitions*. We'll add details in *Defining More Flexible Functions with Mappings*.
    * Class. *Data + Processing = Objects*. Since this gets an entire part, not just a chapter; you can guess that this is a big deal.
    * Module. *Modules : The unit of software packaging and assembly*. Likewise; modules will be a pretty big deal.

CHAPTER

NINE

BASIC SEQUENTIAL COLLECTIONS OF DATA

Strings, Lists and Tuples

Python has a rich family tree of collections. This part will focus on the sequential collections; *Collecting Items in Sequence* will introduce the features that are common to all of the types of sequences.

In *Sequences of Characters : str and Unicode* we describe the string subclass of sequence. The exercises include some challenging string manipulations.

We describe fixed-length sequences, called *tuples* in *Doubles, Triples, Quadruples : The tuple*. Because tuples are quite simple, they give us an opportunity to digress and introduce some basic kinds of algorithms commonly used for statistical processing. The exercises include *Translating From Math To Python: Conjugating The Verb "To Sigma"*, which describes how to approach writing programs for doing statistical calculations.

In *Flexible Sequences : The list* we describe the variable-length sequence, called a list. Lists are one of the cool features that set Python apart from other programming languages. The exercises at the end of the list section include both simple and relatively sophisticated problems.

We'll cover some advanced features of the list in *Common List Design Patterns*. This chapter includes some common techniques for creating useful data structures out of the basic tools we have at our disposal. It will cover the common need to sort a list into order. We'll also cover multi-dimensional structures: moving from mathematical vectors to matrices.

9.1 Collecting Items in Sequence

In this chapter we'll cover the common features of the various kinds of collections which keep items in sequence. This will set the stage for the following chapters:

- The String and Unicode String sequences in *Sequences of Characters : str and Unicode*.
- Fixed-length sequence in *Doubles, Triples, Quadruples : The tuple*.
- Variable-length sequences in *Flexible Sequences : The list*.

In this section we'll define what we mean by sequence in *Sequence means "In Order"*. We'll talk about designing programs that use sequences in *Working With a Sequence*. We'll compare the four kinds of sequences in *Subspecies of Sequences*. We'll look at the common features of sequences in *Features of a Sequence*.

9.1.1 Sequence means "In Order"

A *sequence* is a collection of individual items. A sequence keeps the items in a specific order, which means we can identify each item by its numerical position within the collection. Some sequences (like the tuple) have a fixed number of items, with static positions in the sequence. Other sequences (like the list) have a variable number of items, and possibly dynamic positions in the sequence.

Python has other collections which are not ordered. We'll get to those in *More Data Collections*.

Here's a depiction of a sequence of four items. Each item has a position that identifies the item in the sequence.

position	0	1	2	3
item	`3.14159`	`'two words'`	`2048`	`(1+2j)`

Sequences are used internally by Python. A number of statements and functions we have covered have sequence-related features. We'll revisit a number of functions and statements to add the power of sequences to them. In particular, the **for** statement is something we glossed over in *The for Statement*.

The idea that a **for** statement processes items in a particular order, and a sequence stores items in order is an important connection. As we learn more about these data structures, we'll see that the processing and the data are almost inseparable.

It turns out that the `range()` function that we introduced generates a sequence object. You can see this object when you do the following:

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> range(1,7)
[1, 2, 3, 4, 5, 6]
>>> range(2,36,3)
[2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35]
```

We'll look at the `range()` function and how it generates `list` objects in detail in *Flexible Sequences : The list*.

9.1.2 Working With a Sequence

The typical outline for programs what work with sequences is the following. This is pretty abstract; we'll follow this outline with a more concrete example.

1. Create the sequence. This may involve reading it from a file, or creating it with some kind of generator.
2. Transform the sequence. This may involve computing new values, using a filter to select values that match a condition, or reducing the sequence to a summary.
3. Produce a final result.

Let's say that we have a betting strategy for Roulette that we would like to simulate and collect statistics on the strategy's performance. The verb *collect* is a hint that we will have a collection of samples, and a sequence is an appropriate type of collection.

Let's work backwards from our goal and see how we'll use collections to do this simulation. Once we have all of the necessary steps that lead to our goal, we can just reverse the order of the steps and write our program.

- **Print Results**. We are done when we have printed the results from our simulation and analysis. In this case, the results are some simple descriptive statistics: the mean ("average") and the number of samples.

  To print the values, we must have computed them.

- **Compute Mean**. The *mean* is the sum of the samples divided by the count of the samples. The sum is a *reduction* from the collection of outcomes, as is the count.

  To compute the sum and the count, we must have a collection of individual results from playing Roulette.

- **Create Sample Collection**. To create the samples, we have to simulate our betting strategy enough times to have meaningful statistics. We'll use an iteration to create a collection of 100 individual outcomes of playing our strategy. Each outcome is the result of one session of playing Roulette.

  In order to collect 100 outcomes, we'll need to create each individual outcome. Each outcome is based on placing and resolving bets.

  – **Resolve Bets**. We apply the rules of Roulette to determine if the bet was a winner (and how much it won) or if the bet was a loser.

    Before we can resolve a bet, have to spin the wheel. And before we spin the whell, we have to place a bet.

  – **Spin Wheel**. We generate a random result. We increase the number of spins we've played.

    In order for the spin to have any meaning, of course, we'll need to have some bets placed.

  – **Place Bets**. We use our betting strategy to determine what bet we will make and how much we will bet. For example, in the Martingale system, we bet on just one color. We double our bet when we lose and reset our bet to one unit when we win. Note that there are table limits, also, that will limit the largest bet we can place.

When we reverse these steps, we have a very typical program that creates a sequence of samples and analyzes that sequence of samples.

Other typical forms for programs may include reading a sequence of data items from files, something we'll turn to in later chapters. Some programs may be part of a web application, and process sequences that come from user input on a web form.

9.1.3 Subspecies of Sequences

There are four subspecies of sequence:

- The `str`, which is a collection of the US-ASCII characters. The US-ASCII standard includes the 128 most commonly-used characters.
- The `Unicode string`, which is a collection of Unicode (or Universal Character Set) characters. The Unicode standard includes just about any character in any of the world's alphabets.
- The `tuple`, which is a collection of any kind of Python object. By "any kind of Python object", we mean *any* kind of object: numbers, strings, sequences, functions, anything.
- The `list`, which is a collection of any kind of Python object. The list collection can be altered after it's created.

When we create a `tuple`, `str` or `Unicode`, we've created an *immutable*, or static object. We can examine the object, looking at specific characters or values. We can't change the object. This means that we can't put additional data on the end of a `str`. What we can do, however, is create a new `str` 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. Objects can be removed from a `list`, also. The order of the objects can be changed.

We call these *subspecies* because, to an extent, they are interchangeable. It may seem like a sequence of individual characters has little in common with a sequence of complex numbers. However, these two sequence objects do have some common kinds of features. In the next section, we'll look at all of the features that are common among these sequence subspecies.

A great deal of Python's internals are sequence-based. Here are just a few examples:

- The **for** statement, in particular, expects a sequence, and we often create a list with the `range()` function.
- When we split a `str` using the `split()` method, we get a list of substrings.
- When we define a function, we can have positional parameters collected into a sequence, something we'll cover in *Mappings : The dict*.

9.1.4 Features of a Sequence

All the varieties of sequences (strings, tuples and lists) have some common characteristics. We'll look at a bunch of Python language aspects of these pieces of data, including:

- There is a syntax for writing the kind of sequence. Strings, for example, are surrounded by quotes.
- There are operations that we can apply to a sequence. Strings, for example, can be concatenated using the `+` operator.
- Some built-in functions are appropriate for different kinds of sequences. In particular, each kind of sequence has an appropriate factory function with obvious names like `str()`, `unicode()`, `list()`, and `tuple()`.
- There are rules for how the comparison operators apply between two sequences.
- A sequence object has specific methods. Some methods are generic, and all sequences offer them. Other methods are unique to that kind of sequence.
- Some of the Python statements interact with sequences. We'll have to revisit some statement descriptions to explain how the statements make use of sequences.
- In some cases, there are library modules that work with this kind of sequence.

Inside a Sequence. Our programs talk about sequences in two senses. Sometimes we talk about the sequence as a whole. Other times we talk about individual items or subsequences. Naming an item or a subsequence is done with a new operator that we haven't seen before. We'll introduce it now, and return to it when we talk about each different kind of sequence.

The `[]` operator is called a *subscription*. It puts a subscript after the sequence to identify which specific item or items from the sequence will be used. There are two forms for the `[]` operator:

- The single item format is `[]`

  ```
  sequence  index
  ```

  This identifies one item based on the position number.

- The *slice* format is `[:]`

  ```
  sequence  start  end
  ```

  This identifies a subsequence of items with positions from *start* to *end* -1. This creates a new sequence which is a *slice* of the original sequence; there will be *end* - *start* items in the resulting sequence.

Items are identified by their position numbers. The position numbers start with zero at the beginning of the sequence.

Important: Numbering From Zero

Newbies are often tripped up because items in a sequence are numbered from zero. This leads to a small disconnect between or cardinal numbers and ordinal names.

The ordinal names are words like "first", "second" and "third". The cardinal numbers used for these positions are `0`, `1` and `2`. We have two choices to try and reconcile these two identifiers:

- Remember that the ordinal names are always one too big. The "third" item is in position "2".
- Try to use the word "zeroth" (or "zeroeth") for the item in position 0.

In this book, we'll use conventional ordinal names starting with "first", and emphasize that this is position 0 in the sequence.

Positions are also numbered from the end of the sequence as well as the beginning. Position -1 is the last item of the sequence, -2 is the next-to-last item.

Important: Numbering In Reverse

Experienced programmers are often tripped up because Python identifies items in a sequence from the right using negative numbers, as well as from the left using positive numbers. This means that each item in a sequence actually has two numeric indexes.

Here's a depiction of a sequence of four items. Each item has a position that identifies the item in the sequence. We'll also show the reverse position numbers.

---

forward position	0	1	2	3
reverse position	-4	-3	-2	-1
item	`3.14159`	`'two words'`	`2048`	`(1+2j)`

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 items, the last item is in position 3. Rather than write `S[ len(S)-1 ]`, Python lets us simplify this to `S[-1]`.

Factory Functions. There are also built-in factory (or "conversion") functions for the sequence objects. These are ways to create sequences from other kinds of data.

`str`(*object*) → string
: Creates a string from the *object*. This provides a human-friendly string representation of really complex objects. There is another string factory function, repr, which creates a Python-friendly representation of an object. We'll return to this in *Sequences of Characters : str and Unicode*.

`unicode`(*object*) → unicode
: Creates a Unicode string from the *object*.

`list`(*sequence*) → list
: Return a new `list` whose items are the same as those of the argument sequence. Generally, this is used to convert immutable tuples to mutable lists.

`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. Generally, this is used to convert mutable lists into immutable tuples.

Accesssor Functions. There are several built-in accessor functions which return information about a sequence.

These functions apply to all varieties of lists, strings and tuples.

`min`(*iterable*) → item
: Return the item which is least in the iterable (sequence, set or mapping).

`max`(*iterable*) → item
: Return the item which is greatest in the iterable (sequence, set or mapping).

`len`(*iterable*) → number
: Return the number of items in the iterable (sequence, set or mapping).

These functions hint at a generalization. A sequence, it turns out, is a kind of iterable object. These functions apply to any iterable. We'll look at this generalization in *Looping Back : Iterators, the for statement and Generators*.

Aggregation Functions. The following functions create an aggregate value from a sequence of values. In the case of `sum()` it must be a sequence of numbers. In the case of `any()` and `all()`, it must be sequence of boolean values.

Applying `any()` or `all()` to a string is silly and always returns `True`.

Why? [Hint: do `bool('a')` to see what an individual character's truth value is.]

Similarly, applying `sum()` to a sequence that isn't all numbers raises a `TypeError` exception.

`sum`(*iterable*) → number
: Sum the values in the iterable (set, sequence, mapping). All of the values must be numeric.

`all`(*iterable*) → boolean
: Return `True` if all values in the iterable (set, sequence, mapping) are equivalent to `True`.

`any`(*iterable*) → boolean
: Return `True` if any value in the iterable (set, sequence, mapping) is equivalent to `True`.

9.1.5 Sequence 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 for Statement*, we had to dance around the sequence issue. Would it make more sense to introduce the various types of collections first, and then describe statements that process the collections 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?

9.1.6 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 `[]` or `()`. Spaces are not put before `,`.

9.2 Sequences of Characters : `str` and `Unicode`

A `str` is a sequence of characters. By "character" we mean any of the 128 US-ASCII characters: the digits, the punctuation marks, the letters.

In the case of a `unicode` object, we mean a sequence of any of the millions of Unicode characters.

We'll examine a number of aspects of strings.

- Some basic semantics in *What Does Python mean by "String?"*.
- Syntax for writing literal values in *Writing a String in Python*.
- Functions that create strings in *String Factory Functions*.
- Relevant operators and how they apply to strings in *Operating on String Data*. We'll look at a unique string operator, `%`, in *format() : The Format Method*.
- Some built-in function that apply to strings in *Built-in Functions for Strings*.
- Comparison operators in *Comparing Two Strings – Alphabetical Order*.
- Statements which interact with strings in *Statements and Strings*.
- This is followed by the numerous string method functions in *Methods Strings Perform*.

There is a `string` module, but it isn't heavily used. We'll look at it briefly in *Modules That Help Work With Strings*. Part 8 of the *Python Library Reference* [PythonLib] contains 11 modules that work with strings; we won't dig into these deeply. We'll return to the most important string module in *Text Processing and Pattern Matching : The re Module*.

We'll look at some common patterns of string processing in *Some Common Processing Patterns*.

9.2.1 What Does Python mean by "String?"

A string is an immutable sequence of characters. Let's look at this definition in detail.

- Since a string is a sequence, all of the common operations and built-in functions of sequences apply. This includes `+`, `*` and `[]`.
- Since a string is immutable, it cannot be changed. New strings can be built from other strings, but a string cannot be modified.
- Since strings are an extension to the basic sequence type, strings have additional method functions.

Here's a depiction of a string of 10 characters. The Python value is `"syncopated"`. Each character has a position that identifies the character in the string.

position	0	1	2	3	4	5	6	7	8	9
character	s	y	n	c	o	p	a	t	e	d

We get string objects from external devices like the keyboard, files or the network. We present strings to users either as files or on the GUI display. The **print** statement converts data to a string before showing it to the user. This means that printing a number really involves converting the number to a string of digits before printing the string of digit characters.

Often, our program will need to examine input strings to be sure they are valid. We may be checking a string to see if it is a legal name for a day of the week. Or, we may do a more complex examination to confirm that it is a valid time. There are a number of validations we may have to perform.

Our computations may involve numbers derived from input strings. Consequently, we may have to convert input strings to numbers or convert numbers to strings for presentation.

9.2.2 Writing a String in Python

We looked at strings quickly in *Strings – Anything Not A Number*. A String is a sequence of characters. We can create strings as literals or by using any number of factory functions.

When writing a string literal, we need to separate the characters that are in the string from the surrounding Python values. String literals are created by surrounding the characters with quotes or apostrophes. We call this surrounding punctuation *quote characters*, even though we can use apostrophes as well as quotes.

There are several variations on the quote characters that we use to define string literals.

Single-quote. A single-quoted string uses either the quote (`"`) or apostrophe (`'`). A basic string must be completed on a single line. Both of these examples are essentially the same string.

- Single-Apostrophe looks like this: `'xyz'`.
- Single-Quote looks like this: `"xyz"`.

Triple-quote. Multi-line strings can be enclosed in triple quotes or triple apostrophes. A multi-line string continues on until the matching triple-quote or triple-apostrophe.

- Triple-Apostrophe looks like this: `'''xyz'''`.
- Triple-Quote looks like this: `"""xyz"""`.

Here some examples of creating strings.

```
a= "consultive"
apos= "Don't turn around."
quote= '"Stop," he said.'

doc_1= """fastexp(n,p) -> integer
Raises n to the p power, where p is a positive integer.

:param n: a number

:param p: an integer power
"""

novel= '''"Just don't shoot," Larry said.'''
```

a A simple string.

apos A string using `"`. It has an `'` inside it.

quote A string using `'`. It has two `"` inside it.

doc_1 This a six-line string.

Use `repr(doc_1)` to see how many lines it has. Better, use `doc_1.splitlines()`.

novel This is a one-line string with both `"` and `'` inside it.

Non-Printing Characters – Really! [How can it be a character and not have a printed representation?]

ASCII has a few dozen characters that are intended to control devices or adjust spacing on a printed document.

There are a few commonly-used non-printing characters: mostly tab and newline. One of the most common escapes is `\n` which represents the non-printing newline character that appears at the end of every line of a file in GNU/Linux or MacOS. Windows, often, will use a two character end-of-line sequence encoded as `\r\c`. Most of our editing tools quietly use either line-ending sequence.

These non-printing characters are created using *escapes*. A table of escapes is provided below. Normally, the Python compiler translates the escape into the appropriate non-printing character.

Here are a couple of literal strings with a `\n` character to encode a line break in the middle of the string.

```
'The first message.\nFollowed by another message.'
```

```
"postmarked forestland\nconfigures longitudes."
```

Python supports a broad selection of \ escapes. These are printed representations for unprintable ASCII characters. They're called escapes because the \ is an escape from the usual meaning of the following character. We have very little use for most of these ASCII escapes. The newline (`\n`), backslash (\), apostrophe (`'`) and quote (`"`) escapes are handy to have.

Important: Escapes Become Single Characters

We type two (or more) characters to create an escape, but Python compiles this into a single character in our program.

In the most common case, we type `\n` and Python translates this into a single ASCII character that doesn't exist on our keyboard.

Since \ is always the first of two (or more) characters, what if we want a plain-old \ as the single resulting character? How do we stop this escape business?

The answer is we don't. When we type \\, Python puts a single \ in our program. Okay, it's clunky, but it's a character that isn't used all that often. The few times we need it, we can cope. Further, Python has a "raw" mode that permits us to bypass these escapes.

Escape	Meaning
`\\`	Backslash (\)
`\'`	Apostrophe (\ `'`)
`\“`	Quote (`"`)
`\a`	Audible Signal; the ASCII code called BEL. Some OS's translate this to a screen flash or ignore it completely.
`\b`	Backspace (ASCII BS)
`\f`	Formfeed (ASCII FF). On a paper-based printer, this would move to the top of the next page.
`\n`	Linefeed (ASCII LF), also known as newline. This would move the paper up one line.
`\r`	Carriage Return (ASCII CR). On a paper based printer, this returned the print carriage to the start of the line.
`\t`	Horizontal Tab (ASCII TAB)
`\ooo`	An ASCII character with the given octal value. The *ooo* is any octal number.
`\xhh`	An ASCII character with the given hexadecimal value. The `x` is required. The *hh* is any hex number.

We can also use a \ at the end of a line, which means that the end-of-line is ignored. The string continues on the next line, skipping over the line break. Here's an example of a single string that was so long had to break it into multiple lines.

```
"A manuscript so long \
that it takes more than one \
line to finish it."
```

Why would we have this special dangling-backslash? Compare the previous example with the following.

```
"""A manuscript so long
that it takes more than one
line to finish it."""
```

What's the difference? Enter them both into IDLE to see what Python displays. One string represent a single line of data, where the other string represents three lines of data. Since the \ escapes the meaning of the newline character, it vanishes from the string. This gives us a very fine degree of control over how our output looks.

Also note that adjacent strings are automatically put together to make a longer string. We won't make much use of this, but it something that you may encounter when reading someone else's programs.

`"syn" "opti" "cal"` is the same as `"synoptical"`.

Unicode Strings. If a `u` or `U` is put in front of the string (for example, `u"unicode"`), this indicates a Unicode string. Without the `u`, it is an ASCII string. Unicode refers to 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.

For the thousands of Unicode characters that are not on our computer keyboards, a special `\uxxxx` escape is provided. This requires the four digit Unicode character identification. For example, "日本" is made up of Unicode characters `U+65e5` and `U+672c`. In Python, we write this string as `u'\u65e5\u672c'`.

Here's an example that shows the internal representation and the easy-to-read output of this string. This will work nicely if you have an appropriate Unicode font installed on your computer. If this doesn't work, you'll need to do an operating system upgrade to get Unicode support.

```
>>> ch= u'\u65e5\u672c'
>>> ch
u'\u65e5\u672c'
>>> print(ch)
 日本
```

It's very important to note that Unicode characters are encoded into a sequence of bytes when they are written to a file. A sequence of bytes read from a file can be decoded to get the Unicode characters.

Once inside the computer's memory, in a Python program, there's no encoding. Just characters.

There are a variety of Unicode encoding schemes. The choice of encoding is based on assumptions about the typical number of bytes for a character. For example, the UTF-16 codes are most efficient when most of characters actually use two bytes and there are relatively few exceptions. The UTF-8 codes, on the other hand, work well on the internet where many of the protocols expect only the US ASCII characters.

For the most part, we can use the `io` module to control opening and closing files with specific encodings.

In the rare event that we need really fine control over the encoding, the `codecs` module provides mechanisms for encoding and decoding Unicode strings.

See http://www.unicode.org for more information.

Raw Strings. If an `r` or `R` is put in front of the string (for example, `r"raw\nstring"`), this indicates a raw string. This is a string where the backslash characters (\) are not interpreted by the Python compiler but are left as is. This is handy for Windows files names, which contain \. It is also handy for *regular expressions* that make heavy use of backslashes. We'll look at these in *Text Processing and Pattern Matching : The re Module*.

`"\n"` is an escape that's converted to a single unprintable newline character.

`r"\n"` is two characters, \ and `n` .

9.2.3 String Factory Functions

There is some subtlety to the factory functions which create strings. We have two conflicting interpretations of "string representation" of an object. For simple data types, like numbers, the string version of the number is the sequence of characters. However, for more complex objects, we often want something "readable" that doesn't contain every nuance of the object's value. Consequently, we have two factory functions for strings: `str()` and `repr()`.

`str`(*object*) → string
: Creates a string from the *object*. This is usually a human-friendly view of the object.

`repr`(*object*) → string
: Creates a representation of *object* in Python syntax. Typically, this is a detailed, complete view of the object. For most object types, `eval(repr( object )) == object`. This is true for the built-in sequence types that we'll look at in this part.

`unicode`(*object [,encoding] [,errors]*) → Unicode string
: Creates a new Unicode string from the given encoded string. *encoding* defaults to the current default string encoding. The optional *errors* parameter defines the error handling, defaults to `'strict'`. The `codec` module provides a more complete set of functions for encoding and decoding Unicode strings. Generally, you will be using 'UTF-8' or 'UTF-16' encodings, since these cover much of the data the passes around the Internet.

`eval`(*string*) → value
: Evaluate a string, which is expected to be a legal Python expression.

You can make use of `repr()` to get a detailed view of a specific sequence to help you in debugging. This can, for example, reveal non-printing characters in a character string.

The `str()` function converts any object to a string. Plus, we've seen other functions (like `hex()` and `oct()`) that produce strings.

```
>>> a= str(355.0/113.0)
>>> a
'3.14159292035'
>>> hex(48813)
'0xbead'
```

The `repr()` function also converts an object to a string. However, `repr()` 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.

Important: Python 3

In Python 2, the `repr()` function can also be invoked using the *backtick* ('`'), also called accent grave.

This '`' syntax is not used much and will be removed from Python 3.

Here are several version of a very long string, showing a number of representations.

```
>>> a="""a very
... long symbolizer
... on multiple lines"""
>>> repr(a)
"'a very\\nlong symbolizer\\non multiple lines'"
>>> a
'a very\nlong symbolizer\non multiple lines'
>>> print(a)
a very
```

```
10  long symbolizer
11  on multiple lines
```

1. We set *a* to a very long string with `\n` characters in it.
4. The `repr()` shows a Python expression that produces the string. Interestingly, the result is a string which evluates to a string.
6. The value of *a* is the string, with the `\n` characters shown explicitly.
8. When we print *a*, we see the value with the `\n` characters inpterpreted.

The `unicode()` function converts an encoded `str` to an internal Unicode String. There are a number of ways of encoding a Unicode string so that it can be placed into email or a database. The default *encoding* is called `'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 explicit Unicode conversions.

```
>>> unicode('\xe6\x97\xa5\xe6\x9c\xac','utf-8')
u'\u65e5\u672c'
```

The above example shows the UTF-8 encoding for 日本 as a string of bytes and as a Python Unicode string. The Unicode string character numbers (`u65e5` and `u672c`) are easier to read as a Unicode string than they are in the UTF-8 encoding.

9.2.4 Operating on String Data

There are a number of operations that apply to string objects. Since strings (even a string of digits) isn't a number, these operations do simple manipulations on the sequence of characters.

If you need to do arithmetic operations on strings, you'll need to convert the string to a number using one of the number factory functions `int()`, `float()`, `long()` or `complex()`. See *Functions are Factories (really!)* for more information on these functions. Once you have a proper number, you can do arithmetic on it and then convert the result back into a string using `str()`. We'll return to this later. For now, we'll focus on manipulating strings.

There are three operations (`+`, `*`, `[]`) that work with strings and a unique operation `%` that can be performed only with strings. The `%` is so sophisticated, that we'll devote a separate section to just that operator.

The + Operator. The `+` operator creates a new string as the concatenation of two strings. A resulting string is created by gluing the two argument strings together.

```
>>> "hi " + 'mom'
'hi mom'
```

The * Operator. The `*` operator between strings and numbers (*number* `*` *string* or *string* `*` *number*) creates a new string that is a number of repetitions of the argument string.

```
>>> print(2*"way " + "cool!")
way way cool!
```

The [] operator. The `[]` operator can extract a single character or a substring from the string. There are two forms for picking items or slices from a string.

This form extracts a single item.

```
string[index]
```

Items are numbered from `0` to `len(string)-1`. Items are also numbered in reverse from `-len(string)` to `-1`.

This extracts a slice, creating a sequence from a sequence.

```
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).

For more information on how the numbering works for the `[]` operator, see *Numbering from Zero*.

Important: The meaning of `[]`

Note that the `[]` characters are part of the syntax.

We use [and] for optional elements. This is not part of the syntax, but a description of optional syntactic elements. This can lead to confusion because there are two meanings for `[]` characters.

Since most technical documentation uses [and] for optional elements, we've elected to stick with that rather than try to adopt something more clear, but atypical.

Here are some examples of picking out individual items or creating a slice composed of several items.

```
>>> s="artichokes"
>>> s[2]
't'
>>> s[:5]
'artic'
>>> s[5:]
'hokes'
>>> s[2:3]
't'
>>> s[2:2]
''
```

The last example, `s[2:2]`, shows an empty slice. Since the slice is from position 2 to position 2-1, there can't be any characters in that range; it's a kind of contradiction to ask for characters 2 through 1. Python politely returns an empty string, which is a sensible response to the expression.

Recall that string positions are also numbered from right to left using negative numbers. `s[-2]` is the next-to-last character. We can, then, say things like the following to work from the right-hand side instead of the left-hand side.

```
>>> s="artichokes"
>>> s[-2]
'e'
>>> s[-3:-1]
'ke'
>>> s[-1:1]
''
```

9.2.5 % : The Message Formatting Operator

The % operator can be used to format a message. The argument values are a template string and a tuple of individual values. The operator creates a new string by folding together two elements:

- The literal characters in the template string.
- Characters from the values, which were converted to strings using conversion specifications in the template string.

Here we'll only look at a quick example. We prefer to use the format method of a string. We'll cover that in *format() : The Format Method*.

```
>>> "Today's temp is %dC (%dF)" % (3, 37.39)
"Today's temp is 3C (37F)"
```

The template string is `"Today's temp is %dC (%dF)"`. The two values are `(3, 37.39)`. You can see that the values were used to replace the `%d` conversion specification.

Our template string, then, was really in five parts:

1. `Today's temp is` is literal text, and appears in the result string.
2. `%d` is a conversion specification; it is replaced with the string conversion of `3`. Okay, it seems kind of silly, but `3` in Python is a number, not a string, and it has to be converted to a string. The `print()` function does this automatically. Also, when we work in the **IDLE** *Python Shell*, **IDLE** does this kind of string conversion automatically, also. We've been spoiled.
3. `C (` is literal text, and appears in the result string.
4. `%d` is a conversion specification; it is replaced with the string conversion of `37.49`. While it isn't obvious what happened, here's a hint: the `%d` specification produces decimal integers. To produce an integer from a floating-point number, two conversions had to happen.
5. `F)` is literal text, and appears in the result string.

For details, see the Python Library Reference.

http://docs.python.org/release/2.6/library/stdtypes.html#string-formatting-operations

We're going to focus on the `str.format()` method. We'll cover that in *format() : The Format Method*

9.2.6 Built-in Functions for Strings

The following built-in functions are relevant to working with strings and characters.

Perhaps the most important is the `print()` function.

The `print()` function must convert each expression to a string before writing the strings to the standard output file. Generally, this is what we expect.

For example, when we do `print( abs(-5) )`, the argument is an integer and the result is an integer. This integer result is converted to the obvious string value and printed.

If we do `print( abs )`, what happens? We're not applying the `abs()` function to an argument. We're just converting the function to a string and printing it.

All Python objects have a string representation of some kind. Therefore, the `print()` function is capable of printing anything.

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 that belongs to a Unicode number. `ord()` transforms an ASCII character to its ASCII code number, or transforms a Unicode character to its Unicode number.

`len`(*iterable*) → integer
Return the number of items of a set, sequence or mapping.

```
>>> len("restudying")
10
>>> len(r"\n")
2
>>> len("\n")
1
```

Note that a raw string (`r"\n"`) doesn't use escapes; this is two characters. An ordinary string (`"\n"`) interprets the escapes; this is one unprintable character.

`chr`(*i*) → character
Return a string of one character with ordinal i; $0 \leq i < 256$.

This is the standard US ASCII conversion, `chr(65) == 'A'`.

`ord`(*character*) → integer
Return the integer ordinal of a one character string. For an ordinary character, this will be the US ASCII code. `ord('A') == 65`.

For a Unicode character this will be the Unicode number. `ord(u'\u65e5') == 26085`.

`unichr`(*i*) → Unicode string
Return a Unicode string of one character with ordinal i; $0 \leq i < 65536$. This is the Unicode mapping, defined in http://www.unicode.org/.

```
>>> unichr(26085)
u'\u65e5'
>>> print(unichr(26085))
 日
>>> ord(u'\u65e5')
26085`
```

Note that `min()` and `max()` also apply to strings. The `min()` function will return the character closest that front of the alphabet. The `max()` function returns the character closest to the back of the alphabet.

```
>>> max('restudying')
'y'
>>> min('restudying')
'd'
```

9.2.7 Comparing Two Strings – Alphabetical Order

The standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`) apply to strings. These comparisons use character-by-character comparison rules for ASCII or Unicode. This will keep things in the expected alphabetical order.

The rules for alphabetical order include a few nuances that may cause some confusion for newbies.

- All of the digits come before any letters of the alphabet.
- All Uppercase letters come before any lowercase letters.
- The punctuation marks are intermixed with the letters and numbers in an obscure way. You'll have to get an ASCII character chart to see the punctuation marks and how they work with the other letters.

- Numbers aren't interpreted numerically, but as a string of characters; consequently `'11'` comes before `'2'`. Why? Compare the two strings, position-by-position: the first character, `'1'`, comes before `'2'`. They may look like numbers to you; but they're strings to Python.

Here are some examples.

```
>>> 'hello' < 'world'
True
>>> 'inordinate' > 'in'
True
>>> '1' < '11'
True
>>> '2' < '11'
False
```

These rules for alphabetical order are much simpler than, for example, the American Library Association Filing Rules. Those rules are quite complex and have a number of exceptions and special cases.

There are two additional string 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 in the string, `False` if the character is not found. The '`not in`' operator returns `True` if the character is not found in the string.

```
>>> "i" in 'microquake'
True
>>> "i" in 'formulates'
False
```

9.2.8 Statements and Strings

There are three statements that are associated with strings: the various kinds of **assignment** statements and the **for** statement deals with sequences of all kinds. Additionally the **print** statement is associated with strings.

The Assignment Statements. The basic assignment statement applies a new variable name to a string object. This is the expected meaning of assignment.

The += augmented assigned works as expected. `a += 'more data'` is the same as ``a = a + `more data'``. Recall that a string is immutable; something like `a += 'more data'` creates a new string from the old value of *a* and the string `'more data'`.

It turns out the *= also works for a string and an integer. It's a little surprising, though, when you have something like this.

```
>>> value= 3
>>> value*= 'hello '
>>> value
'hello hello hello '
```

When in doubt, break down the `*=` operator to it's component parts. It helps to think of the statment like this: `value = value * 'hello'`.

The for Statement. Since a string is a sequence, the **for** statement will visit each character of the string.

```
for c in "lobstering":
    print(c)
```

The print Statement. The **print** must convert each expression to a string before writing the strings to the standard output file. We prefer, however, to use the `print()` function.

9.2.9 Methods Strings Perform

A string object has a number of method functions. These can be separated into three groups:

- transformations, which create new strings from old.
- accessors, which access a string and return a fact about that string.
- parsers, which examine a string and create a different data object from the string.

We'll look at one of the most important transformations, the `string.format()` method, separately. Details are in *format() : The Format Method*, below.

Transformations. The following transformation functions create a new string from an existing string.

class str

`str.capitalize()` → string
Create a copy of the original string with only its first character capitalized.

`"vestibular".capitalize()` creates `"Vestibular"`.

`str.center(`*width*`)` → string
Create a copy of the original string centered in a new string of length *width*. Padding is done using spaces.

`"subheading".center(15)` creates `' subheading '`. With explicit spaces shown as · this is

`'···subheading··'`

`str.decode(`*encoding*[, *errors*]`)` → string
Return an decoded version of the original string. The default *encoding* is the current default string encoding, usually 'ascii'. *errors* may be given to set a different error handling scheme; default is 'strict' meaning that encoding errors raise a `ValueError`. Other possible values for *errors* are 'ignore' and 'replace'.

Section 4.9.2 of the Python library defines the various decodings available. One of the codings is called "base64", which mashes complex strings of bytes into ordinary letters, suitable for transmission on the internet.

\ `'c3RvY2thZGluZw=='.decode('base64')` creates `'stockading'`.

`str.encode(`*encoding*[, *errors*]`)` → string
Return an encoded version of the original string. The default *encoding* is the current default string encoding, usually 'ascii'. *errors* may be given to set a different error handling scheme; default is 'strict' meaning that encoding errors raise a `ValueError`. Other possible values for *errors* are 'ignore' and 'replace'.

Section 4.9.2 of the Python library defines the various decodings available. We can use the Unicode UTF-16 code to make multi-byte Unicode characters.

`'blathering'.encode('utf16')` creates :`'\xff\xfeb\x00l\x00a\x00t\x00h\x00e\x00r\x00i\x00n\x00g\x00'`.

`str.expandtabs(`*tabsize*`)` → string
Return a copy of the original string where all tab characters are expanded using spaces. If *tabsize* is not given, a tab size of 8 spaces is assumed.

`str.format(`*value, ...*`)` → string
Insert the values into the template string to create a new string.

`"pi = {0:=+10.5f}".format( math.pi )` creates `'pi = + 3.14159'`.

The value in the arguments (`math.pi`) is inserted into the template, following the conversion specificaion `{0:=+10.5f}`.

For details, see *format() : The Format Method*.

`str.join(sequence)` → string
Return a new string which is the concatenation of the original strings in the *sequence*. The separator between elements is the string object that does the join.

`" and ".join( ["ships","shoes","sealing wax"] )` creates `'ships and shoes and sealing wax'`.

`str.ljust(width)` → string
Return a copy of the original string left justified in a string of length *width*. Padding is done using spaces on the right.

`"reclasping".ljust(15)` creates `'reclasping '`. With more visible spaces, this is

```
'reclasping·····'
```

`str.lower()` → string
Return a copy of the original string converted to lowercase.

`"SuperLight".lower()` creates `'superlight'`.

`str.lstrip()` → string
Return a copy of the original string with leading whitespace removed. This is often used to clean up input.

`" precasting \n".lstrip()` creates `'precasting \n'`.

`str.replace(old, new[, count])` → string
Return a copy of the original string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The most common use is `"$HOME/some/place".replace("$HOME","e:/book")` replaces the `"$HOME"` string to create a new string `'e:/book/some/place'`.

Once in a while, we'll need to replace just the first occurance of some target string, allowing us to do something like the following: `'e:/book/some/place'.replace( 'e', 'f', 1 )`.

`str.rjust(width)` → string
Return a copy of the original string right justified in a string of length *width*. Padding is done using spaces on the left.

`"fulminates".rjust(15)` creates :`' fulminates'`.

With more visible spaces, this is

```
'·····fulminates'
```

`str.rstrip()` → string
Return a copy of the original string with trailing whitespace removed. This has an obvious symmetry with `lstrip()`.

`" precasting \\n".rstrip()` creates `' precasting'`.

`str.strip()` → string
Return a copy of the original string with leading and trailing whitespace removed. This combines `lstrip()` and `rstrip()` into one handy package.

`" precasting \n".strip()` creates `'precasting'`.

`str.swapcase()` → string
Return a copy of the original string with uppercase characters converted to lowercase and vice versa.

`str.title()` → string
: Return a titlecased version of the original string. Words start with uppercase characters, all remaining cased characters are lowercase.

  For example, `"hello world".title()` creates `'Hello World'`.

`str.upper()` → string
: Return a copy of the original string converted to uppercase.

Accessors. The following methods provide information about a string.

class str

`str.count(`*sub*[, *start*, *end*]`)` → integer
: Return the number of occurrences of substring *sub* in a string. If the optional arguments *start* and *end* are given, they are interpreted as if you had said *string* `[` *start* `:` *end* `]`.

  For example `"hello world".count("l")` is 3.

`str.endswith(`*suffix*[, *start*, *end*]`)` → boolean
: Return `True` if the string ends with the specified *suffix*, otherwise return `False`. With optional *start*, or *end*, the test is applied to *string* `[` *start* `:` *end* `]`.

  `"pleonastic".endswith("tic")` creates `True`.

`str.find(`*sub*[, *start*, *end*]`)` → integer
: Return the lowest index in the string where substring *sub* is found. If optional arguments *start* and *end* are given, than *string* `[` *start* `:` *end* `]` is searched. Return `-1` on failure.

  `"rediscount".find("disc")` returns 2; `"postlaunch".find("not")` returns `-1`.

`str.index(`*sub*`)` → integer
: Like `find()` but raise `ValueError` when the substring is not found.

  See *The Unexpected : The try and except statements* for more information on processing exceptions.

`str.isalnum()` → boolean
: Return `True` if all characters in the string are alphanumeric (a mixture of letters and numbers) and there is at least one character in the string. Return `False` otherwise.

`str.isalpha()` → boolean
: Return `True` if all characters in the string are alphabetic and there is at least one character in the string. Return `False` otherwise.

`str.isdigit()` → boolean
: Return `True` if all characters in the string are decimal digits and there is at least one character in the string, `False` otherwise.

`str.islower()` → boolean
: Return `True` if all characters in the string are lowercase and there is at least one cased character in the string, `False` otherwise.

`str.isspace()` → boolean
: Return `True` if all characters in the string are whitespace and there is at least one character in the string, `False` otherwise. Whitespace characters includes spaces, tabs, newlines and a handful of other non-printing ASCII characters.

`str.istitle()` → boolean
: Return `True` if the string is a titlecased string, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones, `False` otherwise.

`str.isupper()` → boolean
Return `True` if all characters in the string are uppercase and there is at least one cased character in the string, `False` otherwise.

`str.rfind(`*sub*[, *start*, *end*]`)` → integer
Return the highest index in the string where substring *sub* is found. Since this is the highest index, this looking for the right-most occurrence, hence the "r" in the name. If optional arguments *start* and *end* are provided, then *string* `[` *start* `:` *end* `]` is searched. Return -1 on failure to find the requested substring.

`str.rindex(`*sub*`)` → integer
Like `rfind()` but raise `ValueError` when the substring is not found.

`str.startswith(`*prefix*[, *start*, *end*]`)` → boolean
Return `True` if the string starts with the specified *prefix*, otherwise return `False`. With optional *start*, or *end*, test *string* `[` *start* `:` *end* `]`.

`"E:/programming".startswith("E:")` is `True`.

Parsers. The following methods create another kind of object, usually a sequence, from a string.

class str

`str.split(`*sep*[, *maxsplit*]`)` → sequence
Return a list of the words in the string the string, 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.

We can use this to do things like `aList= "a,b,c,d".split(',')`. We'll look at the resulting sequence object closely in *Flexible Sequences : The list*.

`str.splitlines(`*keepends*`)` → sequence
Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and `True`. This method can help us process a file: a file can be looked at as if it were a giant string punctuated by `\n` characters.

We can break up a string into individual lines using statements like `lines= "two lines\nof data".splitlines()`.

`str.partition(`*punctuation*`)` → tuple
Locate the left-most occurance of *punctuation*. If found, split the string into three parts. The part before, the punctuation that was found and the part after.

If the punctuation was not found, then the last two elements are zero-length strings.

```
>>> first, punct, last = "label :: several :: values".partition( "::" )
>>> first
'label '
>>> punct
'::'
>>> last
' several :: values'
```

`str.rpartition(`*punctuation*`)` → tuple
Similar to `str.partition()`, except it search for the right-most occurence of the punctuation.

Here's another example of using some of the string methods and slicing operations.

temperature.py

```
1  temp= raw_input("temperature: ")
2  if temp.isdigit():
3      unit= raw_input("units [C or F]: ")
4  else:
5      unit= temp[-1:]
6      temp= temp[:-1]
7  unit= unit.upper()
8  if unit.startswith("C"):
9      print(temp, c2f(float(temp)))
10 elif unit.startswith("F"):
11     print(temp, f2c(float(temp)))
12 else:
13     print("Units must be C or F")
```

2. The `str.isdigit()` method tells us if the string is all digits, or contains some extra characters. If the input string ends with C or F, we'll handle this small typing mistake gracefully.

5. This is the standard "break a string at a position" pattern. In this case, we are breaking at the last position of the string. The final character will be assigned to the unit variable, which we expect to be C or F.

7. We use the `str.upper()` method to create a new string which is only uppercase letters. In the long run, this is simpler and more reliable than messing around with unit.startswith("C") or unit.startswith("c").

8. We use the `str.startswith()` method to examine the first part of the user's input. This will allow the user to spell out "Celsius" or "Fahrenheit".

9.2.10 `format()` : The Format Method

The `str.format()` method is used for format a message. The method is applied for a formatting template. The arguments to this method are the values which are inserted into that template. The method creates a new string by folding together two elements:

- The literal characters in the template string.
- Characters from the values, which were converted to strings using *conversion specifications* in the template string. These are formatting rules surrounded by `{}` in the template.

First we'll look at a quick example, then we'll look at the real processing rules behind this method. This example has a template string and two values that are used to create a resulting string.

```
>>> "Today's temp is {0:d}C ({1:.2f}F)".format(3, 37.39)
"Today's temp is 3C (37.39F)"
```

The template string is `"Today's temp is {0:d}C ({1:.2f}F)"`. The two values are `(3, 37.39)`. You can see that the values were used to replace the `{0:d}` and `{1:.2f}` conversion specifications.

Our template string, then, was really in five parts:

1. `Today's temp is` is literal text, and appears in the result string.
2. `{0:d}` is a conversion specification; it is replaced with the string conversion of `3`. Okay, it seems kind of silly, but `3` in Python is a number, not a string, and it has to be converted to a string.
3. `C (` is literal text, and appears in the result string.
4. `{1:.2f}` is a conversion specification; it is replaced with the string conversion of `37.39`.

5. `F)` is literal text, and appears in the result string.

The `{}` conversion specifications include several important features. We'll look at a few of the most common options. For complete details, see the Python Library Reference.

http://docs.python.org/release/2.6/library/string.html#format-string-syntax

The Big Picture. Each specification has one mandatory and one optional part. This leads to two ways to specify a conversion.

```
{field_name}
```

```
{field_name:format}
```

The mandatory *field_name* specifices which piece of data is taken from the arguments.

The optional *format* specifies how that piece of data should be formatted. If there's no command::*format* then the object is converted to a string using default formatting rules.

The `{}` and `:` are part of the syntax.

In the `{0:d}` example, the *field_name* is 0 (the first argument value). The *format* is `d`.

In the `{1:.2f}` example, the *field_name* is 1 (the second argument value). The *format* is `.2f`.

This is just an overview of the most important parts. We've left quite a bit out.

Format Features Each format actually has a fairly large number of optional features. The full format has seven parts. All of these are optional.

```
[fill][sign][options][width][.precision][type]
```

The *fill* takes one or two characters. It's one of the four alignment characters with an optional prefix. This leads to 8 possibilities.

- `<`, '`fill<`'. Align to the left. Fill any extra positions on the right with spaces or the *fill* character.
- `>`, '`fill>`'. Align to the right. Fill any extra positions on the left with spaces or the *fill* character. Using `*>` will prepend `*` to a number.
- `^`, '`fill^`'. Center in the available space. Full extra positions on left and right with spaces or the *fill* character.
- `=`, '`fill=`'. Put the padding between sign and digits. The sign is specified separately, and it's common to use both fill and sign. For example, `=+` to explicitly show the sign followed by spaces. Another common use is `0=+` to show a sign followed by leading zeroes.

The *sign* is one character. `+` shows all signs. `-` shows only negative signs. A space uses a space for positive and a sign for negative.

One of the *options* characters is a `#`. If present, then a prefix (`0b`, `0o`, `0x`) is used for binary, octal or hexadecimal conversions.

The other *options* character is a `0`. If present, leading zeroes are padded. This is the same as a `0=` fill specification. In effect, it makes the `=` optional.

The *width* is the overall number of positions into which the number is converted. The default is to left-align with trailing spaces. The various fill and sign options, however, provide a great deal of control over how the number is fit into the available width.

The *.precision* is the number of decimal places to include. The `.` is required to show that this is the precision. `.` clearly separates precision from width.

The *type* is the kind of data conversion to apply. There are two broad categories of conversion: integer and floating-point.

The most common integer conversion codes are `d` and `n`. The `d` conversion is ordinary decimal numbers. Additional integer conversions include `d`, `o`, `x` and `X` for binary, octal and hexadecimal.

The common float conversions codes are `e`, `E`, *f'*, `g` and `G`. The `e` and `E` conversions give "scientific" notation (`3.739000e+01`). The `f` conversion gives ordinary-looking numbers. The `g` and `G` conversions choose between `f` and `e` formatting. An additional float conversion is `%` which multplies by 100 to provide a good-looking percentage value.

Also, there's an `n` conversion for localized numbers with proper `,` or `.` separators and decimal points.

Examples. Here are some examples of messages with more complex templates.

```
"{0}: {1} win, {2} loss, {3:6.3f}".format(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 {0:>3d}: {1:>2d}, {2}".format(spin,number,color)
```

This example does three conversions: one number is converted into a right-aligned 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.

```
"Win rate: {0:.1%}".format( win/float(spins) )
```

This example has one conversion using the `%` type.

```
"Pay: {0:*>8d} dollars".format( amount )
```

This example has one conversion using a leading `*` fill character.

9.2.11 Modules That Help Work With Strings

Perhaps the most useful string-related module is the `re` module. The name is short because it is used so often in so many Python programs. However, it is a little too advanced to cover here. We'll talk about it in *Text Processing and Pattern Matching : The re Module*.

The module named `string` has a number of public module variables which define various subsets of the ASCII characters. These definitions serve as a central, formal repository for facts about the character set. Note that there are general definitions, applicable to Unicode character sets, different from the ASCII definitions.

string.ascii_letters `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`

string.ascii_lowercase `abcdefghijklmnopqrstuvwxyz`

string.ascii_uppercase `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

string.digits `023456789`

string.hexdigits `0123456789abcdefABCDEF`

string.letters All Letters; for many locale settings, this will be different from the ASCII letters

string.lowercase Lowercase Letters; for many locale settings, this will be different from the ASCII letters

string.octdigits `01234567`

string.printable All printable characters in the character set

string.punctuation All punctuation in the character set. For ASCII, this is `!"#$%&'()*+,-./:;<=>?@[\]^_\`{|}~`

string.uppercase Uppercase Letters.

string.whitespace A collection of characters that cause spacing to happen. For ASCII this is `\t\n\x0b\x0c\r ·`; Tab (HT), Newline (Line Feed, LF), Vertical Tab (VT), Carriage Return (CR) and space.

You can use these for operations like the following. We often use this string classifiers to test input values we got from a user or read from a file. We use `string.uppercase` and `string.digits` in the examples below.

```
>>> from __future__ import print_function
>>> import string
>>> a= "some input"
>>> a[0] in string.uppercase
False
>>> n= "123-45"
>>> for character in n:
...     if character not in string.digits:
...         print("Invalid character", character)
...
Invalid character -
```

9.2.12 Some Common Processing Patterns

There are a number of common design patterns for manipulating strings. These includes adding characters to a string, removing characters from a string and breaking a string into two strings. In some languages, these operations involve some careful planning. In Python, these operations are relatively simple and (hopefully) obvious.

Adding Characters To A String. We add characters to a string by creating a new string that is the concatenation of the original strings. For example:

```
>>> a="lunch"
>>> a=a+"meats"
>>> a
'lunchmeats'
```

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 *Sequence FAQ's*.

Removing Characters From A String. Sometimes we want to remove some characters from a string. Python encourages us to create a new string that is built from pieces of the original string. For example:

```
>>> s="black,thorn"
>>> s = s[:5] + s[6:]
>>> s
'blackthorn'
```

In this example, we dropped the sixth character (in position 5), `,`. Recall that the positions are numbered from zero. Positions 0, 1 and 2 are the first three characters. Position 5 is the sixth character. Here's how this example works.

1. Create a slice of *s* using characters up to the fifth. This is positions 0 through 4, a total of five characters.
2. Create a slice of *s* using characters starting from position 6 (the seventh character) through the end of the string.
3. Assemble a new string from these two slices; the sixth character (position 5) will have been ignored when we created the two slices.

In other languages, there are sophisticated methods to delete particular characters from a string. Again, Python makes this simpler by letting us create a new string from pieces of the old string.

Breaking a String at a Fixed Position. Often, we will break a string into pieces based on a fixed format. Python gives us a very handy way to do this.

```
>>> fn="1985 Mar 19"
>>> year= fn[:4]
>>> month= fn[5:8]
>>> day= fn[-2:]
>>> month
'Mar'
>>> day
'19'
```

Breaking a String at a Punctuation Mark. There are numerous variations on the parsing theme. We'll look at just one: locating a punctuation mark to split a string.

```
>>> prop="name : value which has : in it"
>>> label, _, value = prop.partition( ":" )
>>> label.rstrip()
'name'
>>> value.lstrip()
'value which has : in it'
```

In this example, we assigned the punctuation mark to the variable `_`. This variable is sometimes used as a "don't care" variable. We know that `str.partition()` always provides three values, but we only want two of them.

9.2.13 String Exercises

1. **Is Each Letter Unique?**.

   Given a ten-letter word, is each letter unique? Further, do the letters occur in alphabetical order?

   Let's say we have a 10-letter word in the variable *w*. We want to know if each letter occurs just once in the word. For example, "pathogenic" has each letter occurring just once. On the other hand, "pathologic".

   To determine if each letter is unique, we'll need to extract each letter from the word, and then use the `count()` method function to determine if that letter occurs just once in the word.

   Write a loop which will examine each letter in a word to see if the count of occurrences is just one or more than one. If *all* counts are one, this is a ten-letter word with 10 unique letters.

   Here's a batch of words to use for testing:

   > patchworks patentable paternally pathfinder pathogenic

   The alphabetical order test is more difficult. In this case, we need to be sure that each letter comes before the next letter in the alphabet. We're asking that `w[0] <= w[1] <= w[2]...`. We can break this long set of comparisons down to a shorter expression that we can evaluate in a loop. We can use `w[0] <= w[1]`, and `w[1] <= w[2]` to examine each letter and its successor.

Write a loop to examine each character to determine if the letters of the word occur in alphabetical order. Words like "abhorrent" or "immortals" have the letters in alphabetical order.

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 '`amount%10`' and '`amount/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 symbols "C" and "D". For numbers between 1000 and 4000, using "M".

   Here are some examples. 1994 = MCMXCIV, 1956 = MCMLVI, 3888= MMMDCCCLXXXVIII

3. **Word Lengths**.

   Analyze the following block of text. You'll want to break into into words on whitespace boundaries. Then you'll need to discard all punctuation from before, after or within a word.

   What's left will be a sequence of words composed of ASCII letters. Compute the length of each word, and produce the sequence of digits. (no word is 10 or more letters long.)

   Compare the sequence of word lenghts with the value of `math.pi`.

```
Poe, E.
Near a Raven

Midnights so dreary, tired and weary,
Silently pondering volumes extolling all by-now obsolete lore.
During my rather long nap - the weirdest tap!
An ominous vibrating sound disturbing my chamber's antedoor.
"This", I whispered quietly, "I ignore".
```

   This is based on http://www.cadaeic.net/cadenza.htm.

9.3 Doubles, Triples, Quadruples : The `tuple`

Tuple is a generalization from words like double, triple, quadruple, quintuple. The common ending to all these words appears to be "tuple". The computer-science folks extracted the suffix and made a new word (I think they call these back-formations or neologisms), *tuple*, out of the suffix.

A tuple is an immutable sequence of items.

- Some basic semantics in *What Does "Tuple" Mean?*.
- Syntax for writing literal values in *How We Write Tuples*.
- Functions that create strings in *The Tuple Factory Function*.
- Relevant operators and how they apply to strings in *Operations on Tuples*.
- Some built-in function that apply to strings in *Built-in Functions for Tuples*.
- Comparison operators in *Making Comparisons Between Tuples*.
- Statements which interact with strings in *Statements and Tuples*.

We'll look at some common patterns of tuple processing in *Translating From Math To Python: Conjugating The Verb "To Sigma"*.

9.3.1 What Does "Tuple" Mean?

A tuple is an immutable sequence of Python objects.

Mathematicians commonly work with *ordered pairs*. For instance, most analytical geometry is done with Cartesian coordinates (x, y), an ordered pair, or 2-tuple. All vector math can be done with tuples. These are remarkably common in mathematics, and we can create a neat, easy-to-read implementation in Python.

Here are some of the properties of tuples in Python.

- Since a tuple is a sequence, all of the common operations and built-in functions of sequences apply. This includes `+`, `*` and `[]`. See *Basic Sequential Collections of Data* for more information on the these operations.
- Since a tuple is immutable, it cannot be changed. This parallels the way a number, like `5`, is immutable.
- While tuples are an extension to the basic sequence type, they don't have any additional method functions; they are the "basic" sequence.

An essential ingredient here is that a tuple has a fixed and known number of items. 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 does not change. Indeed, the size of the tuple is a matter of what the program was designed to do.

Here's a depiction of a tuple of 3 items, the Python value is the RGB color code for a nice midnight-blue: `(51, 0, 153)`. Each item has a position that identifies the item in the tuple.

position	0	1	2
item	51	0	153

Immutability of Tuples. When someone asks about changing a tuple, we have to remind them that the `list`, in *Flexible Sequences : The list*, is for dynamic sequences of items. A tuple is generally used when the number of items is fixed by the nature of the problem. For example, 2-dimensional geometry, or a 4-part internet address, or a Cyan-Mangenta-Yellow-Black color code. Using a tuple, with a fixed number of items, saves Python from all of the bookkeeping necessary when there is a dynamic number of items.

Another common use for tuples is to create a function that returns multiple values. When we put multiple values in a **return** statement, we are creating a tuple. An example would be a function that simulates rolling two dice and returns a tuple with two dice values.

9.3.2 How We Write Tuples

Tuples are created by surrounding the sequence of objects with `()` and separating the objects with commas (`,`). This matches the conventional mathematical notation for coordinates: `(2,3)` is two-dimensional, `(1,5,8)` is a three-dimensional point.

Tuple items do not have to be the same type. A tuple can be a mixture of any Python data types, including lists, tuples, strings and numeric types.

Examples:

```
xy= (2, 3)
personal= ('Hannah',14,5*12+6)
singleton= ("hello",)
zero_tuple = ()
p2= ( "Hannah", (3,8,85), u'G\xe4llivare', )
```

xy A typical 2-tuple.

personal A 3-tuple with name and two numbers.

singleton A 1-tuple. The `,` is mandatory. Without the `,`, this is just an expression in `()`.

zero_tuple A way to specify a tuple with no actual data in it.

p2 A 3-tuple with a string, another 3-tuple (`(3,8,85)`) and a Unicode string. The extra `,` at the end is quietly ignored.

Important: But Wait!

"But wait!" you say. The `()` characters are used to identify parts of an expression. And the identify the argument values to a function. How can they also be used to define a new tuple object?

In the case of `()`, the context helps Python determine how to interpret these characters.

- When you have something like `a(b)`, this is a function application.
- When you have `(b)` by itself, this is an expression.
- When there is at least one `,` (as in `(a,b)` or `(a,)`), this is a tuple.
- If we say just `()`, this is a tuple with zero items. It's a strange degenerate case, but might be useful as a placeholder in a complex data object.

A pleasant consequence of this is that an extra comma at the end of a tuple is legal; for example, `(9, 10, 56, )` is still a three-tuple.

9.3.3 The Tuple Factory Function

In addition to literal values, the following function also creates a tuple object out of another sequence.

`tuple`(*sequence*) → tuple

Creates a tuple from the items in *sequence*. If the sequence is omitted, an empty tuple is created.

```
>>> tuple()
()
>>> tuple( "hi mom" )
('h', 'i', ' ', 'm', 'o', 'm')
```

In the second example, a string, which is a kind of sequence, is transformed into a tuple of individual characters.

9.3.4 Operations on Tuples

There are three standard sequence operations (`+`, `*`, `[]`) that can be performed with tuples as well as lists and strings.

The `+` operator. The `+` operator creates a new tuple as the concatenation of the arguments. Here's an example.

```
>>> ("part",8) + ("strings","tuples","lists")
('part', 8, 'strings', 'tuples', 'lists')
```

The `*` operator. 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. The `[]` operator selects an item or a slice from the tuple. There are two forms for picking items or slices from a tuple.

This form extracts a single item.

```
tuple[index]
```

Items are numbered from `0` to `len(tuple)-1`. Items are also numbered in reverse from `-len(tuple)` to `-1`.

This extracts a slice, creating a new sequence from a sequence.

```
tuple[start:end]
```

Items from *start* to *end*-1 are chosen to create a new tuple as a slice of the original tuple; there will be *end* - *start* items 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).

For more information on how the numbering works for the `[]` operator, see *Numbering from Zero*.

Here are some examples of selecting items or slices from a larger 5-tuple.

```
>>> 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)
```

The `%` Operator. The string format operator works between string and tuple. We prefer to use `str.format()`, however.

9.3.5 Built-in Functions for Tuples

A number of built-in functions create or process tuples.

`len(`*iterable*`)` → integer

Return the number of items of a set, sequence or mapping.

```
>>> some_tuple = ("part",8) + ("strings","tuples","lists")
>>> len( some_tuple )
5
```

`max(`*iterable*`)` → value

Returns the largest value in the iterable (sequence, set or mapping).

```
>>> stats = ( (5,'zero'), (43,'red'), (52, 'black') )
>>> max( stats )
(52, 'black')
```

`min(`*sequence*`)` → value

Returns the smallest value in the iterable (sequence, set or mapping).

```
>>> stats = ( (5,'zero'), (43,'red'), (52, 'black') )
>>> min( stats )
(5, 'zero')
```

Some other functions which apply to sequences in general are available. However, they don't much much sense for tuples. The iteration functions, like `enumerate()`, `sorted()`, `reversed()` and `zip()` are valid, but aren't very meaningful.

Aggregation Functions. The following functions create an aggregate value from a tuple.

`sum(`*iterable*`)` → number

Sum the values in the iterable (set, sequence, mapping). All of the values must be numeric.

```
>>> sum( ( 1, 3, 5, 7, 9 ) )
25
```

`all(`*iterable*`)` → boolean

Return `True` if all values in the iterable (set, sequence, mapping) are equivalent to `True`.

```
>>> compare_1 = ( 2<=3, 5<7, 22%2 == 0 )
>>> all( compare_1 )
True
>>> compare_2 =  ( 2 > 3, 5<7, 22%2 == 0 )
>>> all( compare_2 )
False
>>> compare_2
(False, True, True)
```

`any(`*iterable*`)` → boolean

Return `True` if any value in the iterable (set, sequence, mapping) is equivalent to `True`.

```
>>> roll = 7
>>> any( (roll == 7, roll == 11) )
True
>>> any( (roll == 2, roll == 3, roll == 12) )
False
```

9.3.6 Making Comparisons Between Tuples

The standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`, `in` and `not in`) work the same with tuples as they do with strings. The tuples are compared item by item. If the corresponding items are the same type, ordinary comparison rules are used. If the corresponding items 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
>>> 3 in a
True
>>> 3 in b
False
```

Here's a longer example

```
from __future__ import print_function
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 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.

9.3.7 Statements and Tuples

There are two kinds of statements that are associated with tuples: the various kinds of **assignment** statements and the **for** statement deals with sequences of all kinds.

The Assignment Statements. There is a variation on the **assignment** statement called a **multiple-assignment** statement that works nicely with tuples. We looked at this in *Combining Assignment Statements*. We quietly slipped past the tuple-ness of the multiple assignment statement. Multiple variables can set by decomposing the items of a tuple.

```
>>> x,y=(1,2)
>>> x
1
>>> y
2
```

An essential ingredient here is that a tuple has a fixed and known number of items. For example a 2-dimensional geometric point might have a tuple with x and y. A four-part color code might be a tuple with c, m, y and b.

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 kind of sequence called a list. A tuple is also a kind of 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)
```

9.3.8 Translating From Math To Python: Conjugating The Verb "To Sigma"

If you already know about the *sigma* operator, Σ, you can skip this section. This is background for the basic statistical formulas that we'll implement on tuples of data values.

The sigma operator, Σ, is used in a number of common statistical algorithms. While there are a lot of flashy mathematical symbols here, the purpose of this section is to demystify the math. This information can help give you the necessary background to tackle the exercises.

We can think of Σ as a complicated verb with a few prepositional phrases. The basic meaning of Σ is "to sum". The reason why we use the Greek version of "S" for "sum" is because we're not talking generally about "summing". We have to provide three pieces of information as part of a summation:

- The function we're summing. Picking a specific value out of a tuple with the `[]` operator will be the function we're summing.
- A variable which occurs in the function. We'll call this the "bound" variable because it is bound to this sigma operation.
- A range of values for the bound variable.

Here's the basic summation operation, showing the typical form for the Σ operator.

$$\sum_{i=0}^{n} f(i)$$

The Σ operator has the three additional clauses written around it.

- Below are the bound variable, i, and the starting value for the range, written as $i = 0$.
- Above is the ending value for the range, usually something like n.
- To the right is some function to evaluate 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; one that includes the end values of 0 and n. This is not a helpful definition for software; therefore, we will use a "half-open interval". It has exactly n items, including 0 and n-1; mathematically, $0 \leq i < n$.

Consequently, we prefer the following notation. It has the bound variable and the range of values written below. It has the function we're evaluating written to the right.

$$\sum_{0 \leq i < n} f(i)$$

Since statistical and mathematical texts often used 1-based indexing, some care is required when translating formulae from textbooks to programming languages that use 0-based indexing.

Statistical Algorithms. Our statistical algorithms will be looking at data in lists (or tuples). In this case, the variable x is a sequence of some kind, and the index (i) is an index to select individual values from the sequence.

$$\sum_{0 \leq i < n} x_i$$

Sometimes, we'll apply some function, `f()`, to each value of an array.

$$\sum_{0 \leq i < n} f(x_i)$$

Translating to Python. 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 a sequence of integers.

This is the Python implementation of Σ. This computes two values, the sum, *sum* and the number of items, *n*.

Sigma Using a Numeric Index

```
sum= 0
for i in range(len(aTuple)):
    x_i= aTuple[i]
    # fxi = some function of x_i
    sum += x_i
n= len(aTuple)
```

1. Get the length of *aTuple*. Execute the body of the loop for all values of *i* in the range 0 to the number of items-1.
2. Fetch item *i* from *aTuple* and assign it to *x_i*.
3. For simple mean calculation, the *fxi* statement does nothing.

   For a standard deviation calculation, we'd add a statement a *fxi* to compute the measure of deviation from the average.
4. Sum the *x_i* (or *fxi*) values.

Simplification. 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 item of the list, without actually using an explicit numeric counter. The processing simplifies to the following.

Sigma Using an Iterator

```
for x_i in aTuple:
    # fxi = some function of x_i
    sum += x_i
n= len(aTuple)
```

1. Execute the loop assigning each item of *aTuple* to *x_i*.
2. For simple mean calculation, the *fxi* statement does nothing.

   For a standard deviation calculation, we'd add a statement a *fxi* to compute the measure of deviation from the average.
3. Sum the *x_i* (or *fxi*) values.

Example. Here's an example of computing a sum. We're using the 6:00 AM temperatures this week as our sample data. We created a tuple with the unimaginative name of *data* for holding this tuple of temperatures.

```
>>> data = ( 8, 10, 12, 8, 6 )
>>> sum= 0
>>> for d in data:
...     sum += d
...
>>> sum
44
>>> sum/len(data)
8
```

Our **for** statement iterated through the data. The suite within the **for** statement added the data values into our accumulator, *sum*. The sum divided by the count is the mean.

To get precise results, be sure to use `from __future__ import division`.

9.3.9 Tuple Exercises

1. **Blocks of Stock**.

   A block of stock as a number of attributes, including as 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. We can create a list of those tuples, as follows.

   ```
   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 a tuple which represents a block of stock, multiplies shares by purchase price and returns the value of that block. The sum of these values is the total purchase price of the portfolio.

   This function would have the following definition:

   ```
   def cost( aBlock ):
       compute price times shares
       return cost
   ```

   Develop a second function that examines a tuple which represents a block of stock, multiplies shares by purchase price and shares by a current price to determine the total amount gained or lost by this block.

   This function would have the following definition:

```
def roi( aBlock, priceToday ):
    use cost( aBlock ) to get cost

    compute priceToday times shares

    return the difference
```

2. **Computing the 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.

$$\mu_x = \frac{\sum\limits_{0 \le i < n} x_i}{n}$$

   [The cryptic-looking μ_x is a short-hand for "mean of variable x".]

   You can find the definition of the Σ mathematical operator in *Translating From Math To Python: Conjugating The Verb "To Sigma"*. From this, we can develop the following method for computing the mean:

Computing Mean

   (a) **Initialize**. Set sum, s, to zero

   (b) **Reduce**. For each value, i, in the range 0 to the number of values in the list, n:

   add item x $_i$ to s

   (c) **Result**. Return $s \div n$.

1. **Computing the 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.

$$\sigma_x = \sqrt{\frac{\sum\limits_{0 \le i < n} (x_i - \mu_x)^2}{n - 1}}$$

   [The cryptic-looking σ_x is short-hand for "standard deviation of variable x".]

   You can find the definition of the Σ mathematical operator in *Translating From Math To Python: Conjugating The Verb "To Sigma"*. From this, we can develop the following method for computing the standard deviation:

Computing Standard Deviation

   (a) **Initialize**. Compute the mean, m.

   Initialize sum, s, to zero.

(b) **Reduce**. For each value, x_{i} in the list:

Compute the difference from the mean, $d \leftarrow x_i - \mu_x$.

Add d^2 to s.

(c) **Variance**. Compute the variance as $\frac{s}{n-1}$. The n- 1 factor reflects the statistical notion of "degrees of freedom", which is beyond the scope of this book.

(d) **Standard Deviation**. Return the square root of the variance.

The `math` module contains the `math.sqrt()`. For some additional information, see *The math Module – Trig and Logs*.

9.4 Flexible Sequences : The `list`

A list is a mutable sequence of items. It differs in important ways from an immutable tuple. Yet, being a sequence, it shares a large number of common features with other data structures.

- Some basic semantics in *What Does Python Mean by "List?"*.
- Syntax for writing literal values in *How We Write Lists*.
- Functions that create strings in *List Factory Functions*.
- Relevant operators and how they apply to strings in *Operations We Perform On Lists*.
- Some built-in function that apply to strings in *Built-in Functions for Lists*.
- Comparison operators in *Comparing Two Lists*.
- Statements which interact with strings in *Statements and Lists*.
- Methods of a list object in ref:*data.seq.list.meth*.

We'll look at some common patterns of list processing in *From Outlines to Bank Lines – Stacks and Queues*.

We'll provide two sets of exercises, because lists are that important. The first set, *List Exercises*, covers the basics. The second set, *More Advanced List Exercises*, is a number of more challenging exercises to be sure you have a chance to explore all the things lists can do.

9.4.1 What Does Python Mean by "List?"

A list is a variable length sequence of Python objects. This is the most flexible kind of sequence; it can contain any kind of Python data object. It can grow or shrink as needed. We often use lists to collect a set of data elements like cards in a blackjack hand: we get two cards to start, and then more and more cards as we ask for a hit.

Let's look at this definition in detail.

- Since a list is a sequence, all of the common operations and built-in functions of sequences apply. This includes `+`, `*` and `[]`.
- Since a list is mutable, it can be changed. Items can be added to the list or removed from the list. Unlike strings and tuples, these operations do not create a new list, but change the state of the existing list.
- Since lists are an extension to the basic sequence type, lists have unique method functions.

Let's look at a list of Roulette wheel spins. Here's a depiction of a list of four items, the Python value is `["red", "red", "black", "red"]`. Each item has a position that identifies where it is in the list.

position	0	1	2	3
item	'red'	'red'	'black'	'red'

Because a list is mutable, new items can be added to the list. These new items can be inserted in any position. We can append to the end of the list. We can put elements into the list by inserting before any of existing positions. If we insert before position zero, we will extend the list at the beginning. In addition to extending the list, we can replace any of the items in the list.

9.4.2 How We Write Lists

Lists are created by surrounding the list of objects with `[]` and separating the objects with commas (`,`). 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 = [ ]
```

myList A simple list of seven values; all of which happen to be numbers.

history An empty list; a list can be expanded by appending new elements.

List elements 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.

Important: But Wait!

But wait! you say. The `[]` characters are used to pick items out of tuples and characters out of strings. How can they also be used to define a new list object?

While it is confusing to use a single punctuation mark in two ways, this isn't the only punctuation mark that suffers from this fate. When we introduced tuples, we looked at `()`s and how Python is able to distinguish functions, expressions, and a tuples: `math.sqrt(42)`, `(42)` and `(42,)` are a function expression, a simple expression and a tuple, respectively. The `.`, also serves several purposes: it punctuates floating-point numbers, and it also separates the module name from an object in that module, and it separates an object from the name of an object method.

In the case of `[]`s, the context defines precisely how to interpret these characters.

- When you have something like `a[b]`, this is item selection from a sequence.
- When you have `[b]` by itself, this is a one-element list.
- When you have `[9, 8, 7][2]`, the `[9, 8, 7]` is a list; The `[2]` selects an item from that list.

Lists permit a sophisticated kind of display called a *comprehension*. We'll revisit this in some depth in *List Construction Shortcuts*. We have to mention it now because a comprehension has syntax similar to a literal list.

As a teaser, consider the following:

```
>>> [ 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 using an expression (`2*i+1` in this example). A great deal of power is available in comprehensions.

This is a kind of literal list of valiues, using the `[]` syntax; it can be used anywhere a literal list is appropriate.

9.4.3 List Factory Functions

In addition to literal values, the following function also creates a list object.

`list`(*sequence*) → list
Creates a list from the items in *sequence*. If the sequence is omitted, an empty list is created.

```
>>> list()
[]
>>> list( ("black","red" ) )
['black', 'red']
```

In the second example, a two-element tuple `("black","red" ))` – a kind of sequence – is transformed into a list of individual elements.

** The `range()` function is used heavily, primarily to control the **for** statement. Technically, it generates a list, so we include it here, after we introduced it briefly in *The for Statement*.

`range`([*start*], *stop*[, *step*]) → list
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).

9.4.4 Operations We Perform On Lists

Because a list is a sequence, the three standard sequence operations (`+`, `*`, `[]`) can be performed with lists.

The + operator. The `+` operator creates a new list as the concatenation of the arguments.

```
>>> ["field"] + [2, 3, 4] + [9, 10, 11, 12]
['field', 2, 3, 4, 9, 10, 11, 12]
```

The * operator. 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. The `[]` operator selects an item or a slice from the list. There are two forms for picking items or slices from a list.

This form extracts a single item.

```
list[index]
```

Items are numbered from `0` to `len(list)-1`. Items are also numbered in reverse from `-len(list)` to `-1`.

This extracts a slice, creating a new sequence from a sequence.

```
list[start:end]
```

Items from *start* to *end*-1 are chosen to create a new list as a slice of the original list; there will be *end* - *start* items 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 (position -1).

For more information on how the numbering works for the `[]` operator, see *Numbering from Zero*.

In the following example, we've constructed a list, *rolls* where each of the six items in the list is a tuple object. Each of these tuple objects is a pair of dice. When we say `rolls[2]`, we're extracting the item at position 2, which is the third item from the list. In this example, it's a "hard 4", a pair of 2's.

```
>>> rolls=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> rolls[2]
(2, 2)
>>> print(rolls[:3], 'split', rolls[3:])
[(6, 2), (5, 4), (2, 2)] split [(1, 3), (6, 5), (1, 4)]
>>> rolls[-1]
(1, 4)
>>> rolls[-3:]
[(1, 3), (6, 5), (1, 4)]
```

The % Operator. The string format operator works between string and list. We prefer to use `str.format()`, however.

9.4.5 Built-in Functions for Lists

A number of built-in functions create or deal with lists. The following functions apply to all sequences, including tuples and strings.

`len`(*iterable*) → integer

Return the number of items of a iterable (sequence, set or mapping).

```
>>> rolls=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> len(rolls)
6
```

`max`(*sequence*) → value

Returns the largest value in the iterable (sequence, set or mapping).

```
>>> rolls=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> max(rolls)
(6, 5)
```

Recall that tuples are compared element-by-element. The tuple `(6, 5)` has a first element that is greater than all but one other tuple, `(6, 2)`. If the first elements are the same, then the second element is compared.

`min`(*sequence*) → value

Returns the smallest value in the iterable (sequence, set or mapping).

```
>>> rolls=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> min(rolls)
(1, 3)
```

Recall that tuples are compared element-by-element. The tuple `(1, 3)` has a first element that is less than all but one other tuple, `(1, 4)`. If the first elements are the same, then the second element is compared.

Iteration Functions. These functions are most commonly used with a **for** statement to process list items.

enumerate(*iterable*) → iterator
Enumerate the elements of a set, sequence or mapping. This yields a sequence of tuples based on the original list. Each of the tuples has two elements: a sequence number and the item from the original list.

This is generally used with a **for** statement. Here's an example:

```
>>> rolls=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> for position, roll in enumerate( rolls ):
...     print(position, sum(roll))
...
0 8
1 9
2 4
3 4
4 11
5 5
```

sorted(*iterable [,cmp] [,key] [,reverse]*) → iterator
This iterates through an iterable object like a list in ascending or descending sorted order. Unlike the `sort()` method function, this does not update the list, but leaves it alone.

This is often used with a **for** statement. It can also be used with the `list()` function to create a copy of a list in sorted order.

Here's an example:

```
>>> rolls=[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
>>> descending= list( sorted( rolls, reverse=True ) )
>>> descending
[(6, 5), (6, 2), (5, 4), (2, 2), (1, 4), (1, 3)]
>>> rolls
[(6, 2), (5, 4), (2, 2), (1, 3), (6, 5), (1, 4)]
```

Now there are two copies of the original list: *rolls* is in the original order; *descending* is in descending order.

reversed(*sequence*) → iterator
This iterates through a sequence in reverse order.

This is generally used with a **for** statement. Here's an example:

```
>>> stats = [ (43,'red'), (52, 'black'), (5,'zero') ]
>>> for count, color in reversed( stats ):
...     print(count, color)
...
5 zero
52 black
43 red
```

zip(*sequence, ...*) → sequence
This creates a new sequence of tuples. Each tuple in the new sequence has values taken from the input sequences.

```
>>> color = [ "red", "green", "blue" ]
>>> level = [ 20, 30, 40 ]
>>> zip( color, level )
[('red', 20), ('green', 30), ('blue', 40)]
```

Aggregation Functions. These functions reduce a list to a single aggregate value.

`sum`(*iterable*) → number

Sum the values in the iterable (set, sequence, mapping). All of the values must be numeric.

```
>>> range(1,8*2,2)
[1, 3, 5, 7, 9, 11, 13, 15]
>>> sum(_)
64
```

`all`(*iterable*) → boolean

Return `True` if all values in the iterable (set, sequence, mapping) are equivalent to `True`.

The `all()` function is often used with List Comprehension, which we'll look at in *List Construction Shortcuts*.

```
>>> compare_1 = [ 2<=3, 5<7, 22%2 == 0 ]
>>> all( compare_1 )
True
>>> compare_2 = [ 2 > 3, 5<7, 22%2 == 0 ]
>>> all( compare_2 )
False
>>> compare_2
(False, True, True)
```

`any`(*iterable*) → boolean

Return `True` if any value in the iterable (set, sequence, mapping) is equivalent to `True`.

The `any()` function is often used with List Comprehension, which we'll look at in *List Construction Shortcuts*.

```
>>> roll = 7
>>> test = [ roll == 2, roll == 3, roll == 12 ]
>>> any( test )
False
>>> test.append( roll == 7 )
>>> test.append( roll == 11 )
>>> any( test )
True
>>> test
[False, False, False, True, False]
```

9.4.6 Comparing Two Lists

The standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`, `in` and `not in`) work the same with all sequences: lists, tuples and strings. The list 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 distinguish between the field bets based on different payout odds.

We have to note that comparing two lists which have very different contents may not be sensible. When we compare two strings, we can use this to put them into alphabetic order. In the case of comparing tuples, we generally compare tuples of the same length. For example, we might compare some three-tuples that encode red-green-blue colors. This is consistent with the ways we use tuples to represent a piece of data that has a fixed number of individual items.

In the case of lists, however, we have to be sure that we have an obvious meaning for the comparison. Python will allow us to compare any two list objects. As designers of programs, we have to be sure we are making a sensible comparison between objects that should be compared in the first place. We don't want to have programs that do senseless things like compare a list of the 46 highest peaks in New York with the list of ingredients in Fettucini Alfredo.

9.4.7 Methods to Transform Lists

A list object has a number of member methods. These can be grouped arbitrarily into transformations, which change the list, and accessors, which returns a fact about a list.

Transformations. The following method functions make changes to the given list. With the exception of `pop()`, these method functions don not return a value.

Do not do this; it doesn't do anything useful.

```
a = ['some','list','of']
a = a.append( 'values' )
```

The `list.append()` method function does not return a new value. It modifies the object. The return value happens to be `None`.

class `list`

`list.append`(*object*)

Update list *l* by appending *object* to end of the list.

```
>>> a=["red","orange","yellow"]
>>> a.append("green")
>>> a
['red', 'orange', 'yellow', 'green']
```

`list.extend`(*sequence*)

Extend the list by appending *sequence* elements. Note the difference from `append(object)`, which treats the argument as a single `list` object.

```
>>> a=["red","orange","yellow"]
>>> a.extend(["green","blue"])
>>> a
['red', 'orange', 'yellow', 'green', 'blue']
```

`list.insert`(*index*, *object*)

Update list *l* by inserting sequence‘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.

```
>>> a=["red","yellow","green"]
>>> a.insert(1,"orange")
>>> a
['red', 'orange', 'yellow', 'green']
```

`list.pop`(*index*) → item

Remove and return item at *index* (default is the last element, with an index of -1). An exception is raised if the list is already empty. This is the opposite of append(). Further, this is both a transformation of the list as well as an accessor that returns an item from the list.

```
>>> a=["red","yellow","green","blue"]
>>> a.pop()
'blue'
>>> a
['red', 'yellow', 'green']
```

`list.remove(`*value*`)`

Remove first occurrence of *value* from list *l*. An exception is raised if the value is not in the list.

This example has a list of four initial values, a string, a number, the result of an expression (which will be a number), and a tuple. We'll remove the tuple `(4,3,"craps")` from the list.

```
>>> a=["red",21,6*6,(4,3,"craps")]
>>> a.remove( (4,3,"craps") )
>>> a
['red', 21, 36]
```

`list.reverse()`

Reverse the items of the list *l*. This is done "in place", it does not create a new list.

```
>>> a=["red","yellow","green","blue"]
>>> a.reverse()
>>> a
['blue', 'green', 'yellow', 'red']
```

`list.sort(`[*cmp, key, reverse*]`)`

Sort the items of the list . This is done "in place", it does not create a new list. This does not return a value, either; it transforms the list.

If no comparison function (*cmp*) or key function (*key*) is provided, the items are simply compare and sorted into the desired order.

You can either provide a compare function, or a key function. It doesn't make sense to provide both.

If a key function, *key* is given, it must extract some comparable value from the list element. We'll return to this when we address lists of lists in *Sorting a List: Expanding on the Rules*.

If the reverse keyword, *reverse*, is `True`, the list is sorted in descending order. If *reverse* is omitted or set to `False`, the list is sorted in ascending order.

```
>>> a=["red","yellow","green","blue"]
>>> a
['red', 'yellow', 'green', 'blue']
>>> a.sort()
>>> a
['blue', 'green', 'red', 'yellow']
```

The list `sort()` transformation is very powerful. We'll look at more sophisticated sorting options in *Sorting a List: Expanding on the Rules*. For now, let's just look at the following simple examples. We'll sort simple lists of numbers and strings just to show you how this works.

```
>>> a=  [ 10, 1, 3, 9, 4 ]
>>> a.sort()
>>> a
[1, 3, 4, 9, 10]
>>> b= [ "word", "topic", "subject", "part", "section", "chapter" ]
>>> b.sort()
>>> b
['chapter', 'part', 'section', 'subject', 'topic', 'word']
```

Accessors. The following method functions determine a fact about a list and return that as a value.

class list

`list.count(`*value*`)` → integer

Return number of occurrences of *value* in list *l*.

```
>>> a=["red","red","black","red"]
>>> a.count("red")
3
>>> a.count("green")
0
>>> a.count("black")
1
```

`list.index(`*value*`)` → integer

Return index of first occurrence of *value* in the list. If the item is not found, this will raise a `ValueError`.

If the given value is in the list, then '`list[ list.index(value) ] is value`'.

```
>>> a=["red","yellow","green","blue"]
>>> a.sort()
>>> a.index('red')
2
>>> a[2]
'red'
>>> a
['blue', 'green', 'red', 'yellow']
```

`list.pop(`*index*`)` → item

Remove and return item at *index* (default is the last element, with an index of -1). An exception is raised if the list is already empty. This is the opposite of `append()`. Further, this is both a transformation of the list as well as an accessor that returns an item from the list.

```
>>> a=["red","yellow","green","blue"]
>>> a.pop()
'blue'
>>> a
['red', 'yellow', 'green']
```

9.4.8 Statements and Lists

There are three kinds of statements that are associated with tuples: the various kinds of **assignment** statements, and the **for** statement deals with sequences of all kinds. Additionally the **del** statement can update a list by removing an element.

The Assignment Statements. The variation on the **assignment** statement called **multiple-assignment** statement works with lists as well as tuples. We looked at this in *Combining Assignment Statements*. Multiple variables are set by decomposing the items in the list.

```
>> x,y = [1,"hi"]
>>> y
'hi'
>>> x
1
```

This will only work of the list has a fixed and known number of elements. This kind of multiple assignment makes more sense when working with tuples, which are immutable, rather than lists, which can vary in length.

The for Statement. The **for** statement works directly with sequences. When we first looked at **for** statements, we used the `range()` function to create a list for us. We can also create lists other ways. We'll see still more list construction techniques in the next chapter.

Here's the basic syntax for providing a literal sequence of values. We provide the list object that we want the **for** statement to use as the sequence of values. In this example, the variable *i* will be set to each value in the list, the prime numbers between `2` and `19`.

```
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].

9.4.9 From Outlines to Bank Lines – Stacks and Queues

Stacks and Queues are two ways that we can use lists. Stacks and queues are mutable sequences: items are put into them and removed from them. They are slight specializations of our Python's more general list. The difference between a stack and queue is the rules for putting things into the list and getting things out of the list.

Stack. A *stack* can be described as a last-in-first-out (LIFO) list. The last element inserted into the stack is the first element to be removed from the stack. When you stack dishes: the last dish is on the top of the stack; it will be the first dish removed from the stack.

A number of algorithms use a stack to keep track of nested processing contexts. For example, an outline is a nested structure: parts contain chapters, which contain sections, which contain sub-sections. A part is deeply nested, and has chapters stacked on "top" of it. A chapter, in turn, has sections stacked on top of it.

When we read (or write) we begin the part and set it on our mental stack. We then look at the chapter, opening it and putting it on our mental stack. We look at a section, putting it on the stack while we read the paragraphs, and then removing it from the stack when we are done with the section. The last section onto the stack is the first section off the stack when we get to a new title.

Queue. A *queue* can be described as a first-in-first-out (FIFO) list. Elements are stored temporarily in a queue, and processed in the order they were received. The line at the coffee shop is a queue: the first person in line is the first person to get their Cappucino.

Queues are often used as buffers to match processing speeds between fast and slow operations. For example, it takes less than a minute for my computer to generate a document with 402 pages, but my printer will take almost an hour to print the document. To balance this speed difference, the operating system creates a queue of print jobs.

Using Lists. Both the stack and queue are essentially a list. In the case of a stack, it is a list that has items added and removed at the last position only. A queue, on the other hand, has items appended at the end, but removed from the front of the list.

The `append()` and `pop()` method functions can be used to create a standard stack. 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("part I")
>>> stack.append("chapter 1")
>>> stack.append("intro section" )
>>> stack.pop()
'intro section'
>>> stack.append("another section" )
>>> stack.pop()
'another section'
>>> stack.pop()
'chapter 1'
>>> stack.pop()
'part I'
>>> stack
[]
```

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. Evaluating `pop(0)()` removes the first item from the queue it and returns it.

```
>>> queue=[]
>>> queue.append("part I")
>>> queue.append("part II")
>>> queue.pop(0)
'part I'
>>> queue.append("part III")
>>> queue.pop(0)
'part II'
>>> queue.append("part IV")
>>> queue.pop(0)
'part III'
>>> queue.pop(0)
'part IV'
>>> queue
[]
```

9.4.10 List Exercises

1. **Creating a Sequence of Outcomes**.

   Evaluating a betting strategy amounts to collecting a sequence of outcomes from playing a number of games. We want a betting strategy that is better than betting at random. One basis for comparison, then, is truly random results with a very simple betting strategy like “always bet on black”.

   We’ll assume that a single round of play in a casino game is an elaborate coin toss. In the case of

Roulette, there are a large number of outcomes, but we can focus on just betting red and black. This makes the game almost a coin toss. There are a total of 38 outcomes on an American table, composed of 18 red, 18 black and 2 green. If we play a number of rounds we'll win some and lose some. If we stay at the table for 200 spins, our results could vary from the really unlikely 200 wins to the equally unlikely 200 losses. In the middle are mixes of wins and losses. For a truly fair coin toss, this range of values is called the Gaussian or normal distribution. The question we need to have answered, is what is the average result of sessions that last for 200 spins of the wheel?

We can create a sequence of values that represents the wheel by assembling a list that has 18 copies of `'red'`, 18 copies of `'black'` and two copies of `'green'`. We can then use the `random.choice()` function to pick one of these values as the result of the spin.

If the chosen result is `'black'`, we've won, and our stake increases by one bet. Otherwise, we've lost and our stake decreases by one bet.

To simulate a session of betting, we initialize our table stakes to 100 betting units. This means we go to a $5 Roulette table with $500 in our pocket. We can create a loop which does the following 200 times:

(a) Use the `random.choice()` function to pick one of 38 values as the result of the spin.

(b) Increase or decrease the stake depending on the color chosen.

Each session, therefore, will have a result that is a single number, the final amount we left the table with. You can check your result by simulating a few thousand sessions and accumulating a sequence of final amounts.

Compute the average of your sequence of final amounts. You should have an average result of about 89. The standard deviation should be around 14. What does this mean? We can expect to lose 11 betting units over 200 spins of the wheel.

2. **Creating a Different Sequence of Outcomes**.

   In the previous exercise, we created a random sequence of outcomes for Roulette using a simple "always bet on black" betting strategy. What if we want to use a bet with a different payout? For example, the three column bets pay 2:1 when they win. How does this change our results?

   In Roulette there are 12 column 1, 12 column 2, 12 column 3, and 2 zero results on an American table. As with the previous exercise (*Creating a Sequence of Outcomes*), we can construct a sequence that represents the wheel by assembling a list of 38 elements that have the proper number of `"col1"`, `"col2"`, `"col3"` and `"zero"` values. We can then use the `random.choice()` function to pick one of these values as the result of the spin.

   We'll assume a consistent bet on `"col3"`. We'll choose a random result from the wheel sequence; if this result is `"col3"`, we've won, and our stake increases by two bets. Otherwise, we've lost and our stake decreases by one bet.

   We can revise our previous example to use this wheel, bet and result.

   Compute the average of your sequence of final amounts. You should have an average result of about 89. The standard deviation should be around 19. What does this mean? We can expect to lose 11 betting units over 200 spins of the wheel.

3. **Creating a Sequence of Really Bad Outcomes**.

   In the previous exercises, we created a random sequence of outcomes for Roulette using some simple "always bet on black" or "always bet on column three" betting strategy. What if we want to use a bet with a really bad payout? For example, there is a bet that covers zero, double zero, one, two and three. This bet will win 5/38th of the time, but pays as if it won 6.33/38 of the time. How does this change our results?

In Roulette there are 5 `'5bet'` results and 33 `'other'` results on an American table. As with the previous exercises (*Creating a Sequence of Outcomes*), we can construct a sequence that represents the wheel by assembling a list of 38 elements that have the proper number of `"5bet"`, `"other"` values. We can then use the `random.choice()` function to pick one of these values as the result of the spin.

We'll assume a consistent bet on `"5bet"`. We'll choose a random result from the wheel sequence; if this result is :`"5bet"`, we've won, and our stake increases by six bets. Otherwise, we've lost and our stake decreases by one bet.

We can revise our previous example to use this wheel, bet and result.

Compute the average of your sequence of final amounts. You should have an average result of about 83. The standard deviation should be around 34. What does this mean? We can expect to lose 17 betting units over 200 spins of the wheel.

4. **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.

If we generate thousands of random numbers between 0 and 9, we expect that we'll have the name number of 0's as 9's. Specifically, we expect that 1/10th of our numbers are 0's, 1/10th are 1's, etc. Actually random numbers are – well – random, so they will deviate slightly from this perfection.

This difference between actual and expected can be used for a more sophisticated statistical test called a Chi-Squared test. The formula is pretty simple, but the statistics beyond this book. The idea is, however, that the Chi-Squared test can help us tell whether our data is too well organized, meets our expectation for randomness, or is too disorganized.

What we'll do is generate random numbers, and assign them to one of ten different bins. When we've done this for a few thousand samples, we'll compare the count of numbers in each bin with our expectation to see if we've got a respectable level of randomness.

Use `random.random()` to generate an array of 1000 random samples; assign this to the variable *u*. These numbers will be uniformly distributed between 0 and 1.

Distribution test of a sequence of random samples, *U*

(a) **Initialize**. Initialize *count* to a list of 10 zeros.

(b) **Examine Samples**. For each sample value, *v*, in the original set of 1000 random samples, *U*.

  i. **Coerce Into Range**. Set $x \leftarrow \lfloor v \times 10 \rfloor$. Multiply by 10 and truncate and integer to get a a new value in the range 0 to 9.

  ii. **Count**. Increment *count* $[x]$ by 1.

(c) **Report**. We expect each count to be 1/10th of our available samples. We need to display the actual count and the % of the total. We also need to calculate the difference between the actual count and the expected count, and display this.

1. **Accumulating Unique Values**.

We'll use the *Bounded Linear Search* algorithm to locate just the unique values in a sequence of values. This could, for example, be a sequence of words extracted from a document. We might also use a procedure like this to evaluate a sequence of random numbers to be sure that all of the expected values were actually found in the sequence somewhere.

Let's assume we have a sequence, *seq* with 1000 values that are supposed to be random numbers between 0 and 37, inclusive. We created these numbers with something like the following.

```
import random
[ random.randrange(0,37) for i in range(1000) ]
```

[You may have already noticed the error in the above statement.]

We can use the following procedure to do a complete evaluation.

Unique Values of a Sequence, *seq*

```
(a) Initialize. Set $uniq \leftarrow$ list().
(b) Loop. For each value, $v$, in $seq$.
    We'll use the Bounded Linear Search to see if $v$ occurs in $uniq$.
    i. Initialize. Set $i \leftarrow 0$.
       Append $v$ to the list $uniq$.
    ii. Search. while $uniq[i] \neq v$: increment $i$.
       At this point $uniq[i] = v$. The question is whether $i = \text{len}(uniq)$ or not.
    iii. New Value?. if $i = \text{len}(uniq)$: $v$ is unique.
    iv. Existing Value?. if $i \neq \text{len}(uniq)$: $v$ is a duplicate of $uniq[i]$.
       Delete $uniq[-1]$, the value we added.
(c) Result. Return array $uniq$, which has unique values from $seq$.
```

You may also notice that this fancy Bounded Linear Search is suspiciously similar to the `index()` method function of a list. Rewrite this using `uniq.index()` instead of the Bounded Linear Search in step 2.

When we look the `set` collection, you'll see another way to tackle this problem.

9.4.11 More Advanced List Exercises

1. **Binary Search**.

   This is not as universally useful as the Bounded Linear Search (above) because it requires the data be sorted.

Binary Search a sorted Sequence, *seq*, for a target value, *tgt*

```
(a) Initialize. $l, h \leftarrow 0, \text{len}(seq)$.
    $m \leftarrow (l + h) \div 2$. This is the midpoint of the sorted sequence.
(b) Divide and Conquer. While $l + 1 < h$ and $seq[m] \neq tgt$.
    If $tgt < seq[m]$: $h \leftarrow m$. Move $h$ to the midpoint.
    If $tgt > seq[m]$: $l \leftarrow m + 1$. Move $l$ to the midpoint.
    $m \leftarrow (l + h) \div 2$. Compute a midpoint of the new, smaller sequence.
(c) Result. If $tgt = seq[m]$: return $m$
    If $tgt \neq seq[m]$: return -1 as a code for "not found".
```

2. **Quicksort**.

   The super-fast sort algorithm.

   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.

   Quicksort a List, *a* between elements *lo* and *hi*

   (a) **Partition**

      i. **Initialize**. $ls, hs \leftarrow lo, hi$. Setup for partitioning between *ls* and *hs*.

         $middle \leftarrow (ls + hs) \div 2$.

      ii. **Swap To Partition**. while $ls < hs$:

         If $a[ls].key \leq a[middle].key$: increment *ls* by 1. Move the low boundary of the partitioning.

         If $a[ls].key > a[middle].key$: swap the values $a[ls] \leftrightarrows a[middle]$.

         If $a[hs].key \geq a[middle].key$: decrement *hs* by 1. Move the high boundary of the partitioning.

         If $a[hs].key < a[middle].key$:, swap the values $a[hs] \leftrightarrows a[middle]$.

   (b) **Quicksort Each Partition**.

      QuickSort(*a* , *lo*, *middle*)

      QuickSort(*a* , *middle*+1, *hi*)

3. **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 recursive function instead of a loop. This can often be faster than the looping version shown above.

   Recursive Search a List, *seq* for a target, *tgt*, in the region between elements *lo* and *hi*.

   (a) **Empty Region?** If $lo + 1 \geq hi$: return -1 as a code for "not found".

   (b) **Middle Element**. $m \leftarrow (lo + hi) \div 2$.

   (c) **Found?** If $seq[m] = tgt$: return *m*.

   (d) **Lower Half?** If $seq[m] < tgt$: return recursiveSearch (*seq*, *tgt*, *lo*, *m*)

   (e) **Upper Half?** If $seq[m] > tgt$: return recursiveSearch(*seq*, *tgt*, *m*+1, *hi*)

4. **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.

Sieve of Eratosthenes

```
(a) Initialize. Create a list, prime of 5000 booleans, all True, initially.
    $p \leftarrow 2$.
(b) Iterate. While $2 \leq p < 5000$.
      i. Find Next Prime. While not $prime[p]$ and $2 \leq p < 5000$:
            Increment $p$ by 1.
     ii. Remove Multiples. At this point, $p$ is prime.
         Set $k \leftarrow p + p$.
         while $k < 5000$.
            $prime[k] \leftarrow False$.
            Increment $k$ by $p$.
    iii. Next p. Increment $p$ by 1.
(c) Report. At this point, for all $p$ if prime [ $p$ ] is true, $p$ is prime.
    while $2 \leq p < 5000$:
         if $prime[p]$: print $p$
```

The reporting step is a "filter" operation. We're creating a list from a source range and a filter rule. This is ideal for a list comprehension. We'll look at these in *List Construction Shortcuts*.

Formally, we can say that the primes are the set of values defined by $primes = \{p|_{0 \leq p < 5000} \textbf{ if } prime_p\}$. This formalism looks a little bit like a list comprehension.

5. **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 )`.

   The polynomial $2x^2 - 3x - 4$ is represented as `( 2, -3, -4 )`.

   The sum of these is $4x^3 + 2x^2 - 3$; `( 4, 2, 0, -3 )`.

   The product these 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 10, and the coefficients must all be between 0 and x-1. For example, $1987 = 1x^3 + 9x^2 + 8x + 7, \text{ when } x = 10$.

Add Polynomials, *p*, *q*

```
(a) Result Size. $r_{sz} \leftarrow$ the larger of len($p$) and len($q$).
(b) Pad P? If len($p$) $< r_{sz}$:
         Set p1 to a tuple of $r_{sz}$ − len($p$) zeros + $p$.
    Else: Set p1 to $p$.
(c) Pad Q? If len($q$) $< r_{sz}$:
         Set q1 t a tuple of $r_{sz}$ − len($q$) zeroes + $q$.
    Else, Set q1 to $q$.
```

(d) **Add**. Add matching elements from *p1* and *q1* to create result, r.

(e) **Result**. Return r as the sum of p and q.

Multiply Polynomials, *x*, *y*

```
(a) Result Size. $r_{sz} \leftarrow \text{len}(x) + \text{len}(y)$.
    Initialize the result list, $r$, to all zeros, with a size of $r_{sz}$.
(b) For all elements of x. while $0 \leq i < \text{len}(x)$:
        For all elements of y. while $0 \leq j < \text{len}(y)$:
            Set $r[i + j] = r[i + j] + x[i] * y[j]$.
(c) Result. Return a tuple made from $r$ as the product of $x$ and $y$.
```

6. **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.

9.5 Common List Design Patterns

This chapter presents some common processing patterns for lists. In *The One-Two Punch: 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 *List Construction Shortcuts*. In *Sorting a List: Expanding on the Rules* we cover some advanced sequence sorting. In *Tables and Matrices – More Multi-Dimensional Loop-the-Loops* 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).

9.5.1 The One-Two Punch: Lists of Tuples

Lists of tuples are surprisingly common. In other languages, like Java, we are forced to either use the too-complex built-in arrays or create an even more complex class definition to simply keep a few values together. Our canonical examples involve simple coordinate pairs for 2-dimensional or 3-dimensional geometries. Additional examples might includes the 3 codes 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. Here, we've defined three colors – black, a dark grey, a purple – and assigned this list of colors to the variable *colorScheme*.

```
colorScheme = [ (0,0,0), (0x20,0x30,0x20), (0x80,0x40,0x80) ]
```

A interesting form of the **for** statement uses multiple assignment to work with a list of tuples. Consider the following example which assigns *r*, *g* and *b* from each element of the 3-tuple in the list. We can then do calculations on the three values independently.

```
colorScheme = [ (0,0,0), (0x20,0x30,0x20), (0x80,0x40,0x80) ]
for r,g,b in colorScheme:
    print("color ({0:d},{0:d},{0:d})".format( r, g, b ))
    print("opposite ({0:d},{0:d},{0:d})".format( 255-r, 255-g, 255-b ))
```

This is equivalent to the following. In this example, we have the **for** statement assign each item in the list to the variable *color*, and then we use a separate multiple assignment to decompose the **for** tuple in *r*, *g* and *b*.

```
colorScheme = [ (0,0,0), (0x20,0x30,0x20), (0x80,0x40,0x80) ]
for color in colorScheme:
    r, g, b = color
    print("color ({0:d},{0:d},{0:d})".format( r, g, b ))
    print("opposite ({0:d},{0:d},{0:d})".format( 255-r, 255-g, 255-b ))
```

The `items()` function of a dictionary transforms a dictionary to a sequence of tuples. We'll cover dictionaries in *Mappings : The dict*. This is a teaser for some of what we'll see there.

```
from __future__ import print_function, division.
d = { 'red':18, 'black':18, 'green':2 }
for c,f in d.items():
    print("{0} occurs {1:f}".format(c, f/38))
```

The `zip()` built-in function interleaves two or more lists to create a list of tuples from the two lists. This is not terribly useful, but we'll use it to build dictionaries.

9.5.2 List Construction Shortcuts

Python provides a mechanism to construct a list called a *list comprehension* or *list maker*. A list comprehension uses a generator expression (similar to the **for** and **if** statements) to create a new list. The generator expression allows us to write a rule rather than write each individual value in a list.

Comprehensions implement the basic *map* and *filter* iteration patterns. See *Patterns of Iteration* for more information on these iteration design patterns. A comprehension doesn't implement the *reduction* pattern very well.

Map Processing. A list comprehension looks like list literal. It does this by enclosing a generator expression in `[]`s. Here's the simplest form, used to do a mapping.

```
[ expression for-clause ]
```

The *expression* is any expression.

The *for-clause* mirrors the **for** statement. The big difference is that a it doesn't have a complete suite of statements; it is just the target variable and the iterable sequence of values.

```
for variable in iterable
```

The overall generator expression executes the **for** loop; for each iteration, it evaluates the expression and yields value. The list comprehension uses that sequence of values to create the resulting list.

Here are some examples.

```
>>> import random

>>> [ 3*x+2 for x in range(12) ]
[2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35]
>>> [ (x,x) for x in (2,3,4,5) ]
[(2, 2), (3, 3), (4, 4), (5, 5)]
>>> [ random.random() for x in range(5) ]
[0.4527184178006578, 0.84888059794845783, 0.21016399448987311, 0.80816095098407259, 0.87693626640363287]
```

A list comprehension behaves like the following loop:

```
r= []
for  target in  sequence :
    r.append(  expr )
```

The basic process, then, is to iterate through the sequence in the *for-clause*, evaluating the expression, *expression*. The values that result are assembled into the list.

If the expression depends on the *for-clause* target variable, the expression is a map from the *for-clause* variable to the resulting list. If the expression doesn't depend on the *for-clause* target value, each time we evaluate the expression we'll get the same value.

Here's an example where the expression depends on the for-clause. This is a mappings from the `range(10)` to the final list.

```
>>> 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* assigns each value in this sequence to the target 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 zeros. Because the expression doesn't depend on the *for-clause*, this could also be done as

```
b= 10*[0]
```

Filter Processing. A comprehension can also have an *if-clause*. This acts as a filter to determine which elements belong to the list and which elements do not belong.

The more complete syntax for a list comprehension is as follows:

```
[ expr for-clause [ for-clause | if-clause ] ... ]
```

The *expr* is any expression. The *for-clause* mirrors the **for** statement:

```
for variable in sequence
```

The *if-clause* mirrors the **if** statement.

```
if filter
```

This syntax summary shows that the first *for-clause* is required. This can be followed by either *for-clause*s or *if-clause*s. The | means that we can use either a *for-clause* or an *if-clause* .

This syntax summary shows a **...** which means that you can repeat as many *for-clause*s and *if-clause*s as you need. We'll stick to the most common form, which is a single *if-clause* to create a filter.

Note that there's no **,** or other punctuation; the various for-clauses and if-clauses are simply separated by spaces.

Here is an example that creates the list of hardways rolls, which excludes two 2's and two 12's. The **for** loop creates a sequence of six numbers (from 1 to 6), assigning each value to *x* . The **if** filter only keeps values where `x+x` is not 2 or 12. All other values are used to create a tuple of `(x,x)`.

```
hardways = [ (x,x) for x in range(1,7) if x+x not in (2, 12) ]
```

These more complex list comprehensions behave like the following loop:

```
r= []
for  target 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 *if-clause* filter 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 function `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*.

Tip: Debugging List Comprehensions

List comprehensions have rather complex syntax. There are a number of ways to get `SyntaxError` messages. The expression, the for-clause and the overall structure of the expression, including balancing the `[]` are all sources of problems. Debugging a list comprehension is most easily done by building up the list comprehension one clause at a time.

A simple comprehension has two clauses: the expression clause and the for-clause. You can try each part out in IDLE individually.

- If the for-clause is wrong, the original sequence will not be correct.
- If the expression clause is not correct, the resulting sequence will be incorrect.

The for-clause of a list comprehension can be seen by entering just the for-clause as a separate statement. The expression clause can be evaluated for specific values to be sure that it works correctly.

For example, `[ 2*i+1 for i in range(5) ]` can be debugged in two parts. First, assure that `range(5)` produces the source sequence you expected. Second, assure that `2*i+1` works for values of *i* from 0 to 4.

9.5.3 Sorting a List: Expanding on the Rules

Let's look at a common processing problem. Our source is a table of raw data in a spreadsheet. We want to do some processing that is a pain in the neck to do in the spreadsheet. We can transform this spreadsheet into a list of tuples for processing by Python. We can then write Python programs to manipulate this data, doing mappings, filterings and reductions as well as sorting and presentation in an easy-to-read report or summary.

For example, we have a spreadsheet with raw census data that looks like the following:

Code	County	State	Jobs
001	Albany	NY	162692
002	Allegany	NY	11986
...			
121	Wyoming	NY	8722
123	Yates	NY	5094

We can easily transform this raw data 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),
]
```

Simple Sorting. Sorting this list can be done trivially with the list `sort()` method.

```
jobData.sort()
```

Note that this updates the *jobData* 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. This makes it very easy to use, *if* your tuple's elements are in the right order. If you want to compare the elements of your tuple in a different order, however, you'll need to do something extra.

Sorting By Another Column. Let's say we wanted to sort by state name, the third element in the tuple. We want don't want the naive comparison among tuples. We want a smarter comparison that looks at the elements we choose, not the first element in the tuple. We do this by giving a key function to the `sort()` method.

The key function returns an object or a simple sequence of the key values selected from each element to be sorted. In this case, we want the key function to return the third elements of our county jobs tuples.

```
def by_state( row ):
    return row[2]
jobData.sort( key=by_state )
```

Note that we pass the function object to the `sort()` method. A common mistake is to say `jobData.sort( by_state() )`. If we include the `()`s, we evaluate the function `by_state()` once, which is a mistake.

We don't want to evaluate the function; we want to provide the function to `sort()`, so that `sort()` can evaluate the function as many times as needed to sort the list.

Note that if we say `by_state()`, we evaluate `sort3()` without any argument values, which is also a type error. If we say `by_state` – naming the function instead of evaluating it – then `sort()` will properly call the function with the expected single argument.

Sorting By Multiple Fields. 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. Within each state, we want to use the number of jobs to sort the data.

We do this by creating a tuple of the fields we want to use for sorting.

```
def by_state_jobs( row ):
    return ( a[2], a[3] )
jobData.sort( key=by_state_jobs )
```

The `sort()` method must compare elements of the sequence against each other. If the `sort()` method is given a key function, this function is called to create the sort comparison key for each element.

In our case, we've provided a function (`by_state_jobs()`) that extracts a tuple as the key. The tuple contains the state and the number of jobs from each row.

Tip: Debugging List Sorting

There are three kinds of problems that can prevent a customized sort operation from working correctly.

- Our key function doesn't have the right form. It must be a function that extracts the key from an item of the sequence being sorted.

  ```
  def key( item ):
      return something based on the item
  ```

- The data in your list isn't regular enough to be sorted. For example, if we have dates that are represented as strings like `'1/10/56'`, `'11/19/85'`, `'3/8/87'`, these strings are irregular and won't sort very nicely. As humans, we know that they should be sorted into year-month-date order, but the strings that Python sees begin with `'1/'`, `'11'` and `'3/'`, with an alphabetic order that may not be what you expected.

  To get this data into a usable form, we have to *normalize* it. Normalizing is a computer science term for getting data into a regular, consistent, usable form. In our example of sorting dates, we'll need to use the `time` or `datetime` modules to parse these strings into proper Python objects that can be compared.

Ascending vs. Descending. The default sort is ascending order. We can sort into descending order by adding the *reverse* keyword parameter to the sort.

```
jobData.sort( key=by_state_jobs, reverse=True )
```

By default, *reverse* is `False`, giving us ascending order. When we set it to true, the list is sorted in reverse order; that is, descending.

The Lambda Shorthand. In reading other programs, you may see something like the following:

```
jobData.sort( key=lambda row: row[2] )
```

This **lambda** is a small, anonymous function definition. These are used sometimes because it saves having to create a function which is only used once in a single `sort()` operation. Mentally, you can rewrite this to the following:

```
# some unique name for the replacement
def aLambda( row ):
    # return followed by the original lambda expression
    return row[2]

# the original statement replaced with the new function
jobData.sort( key=aLambda  )
```

9.5.4 Tables and Matrices – More Multi-Dimensional Loop-the-Loops

Some situations demand multi-dimensional sequences. In a business we might have a budget with cost centers and months as two dimensions of a large table. We can put the months across the top of the table, and fill in the cost centers down the rows of the table.

One need for multi-dimensional sequences is the mathematical matrix operations. These are not obvious to the non-mathematical audience, so we'll cover this later in the section and in some of the exercises.

Let's look at a simple two-dimensional example that doesn't involve a matrix. Instead it involves a 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 can be done as a sequence of sequences. This table is a sequence of rows.

Each individual row, in turn 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 say `A[i][j]`. We want the row i from table A, and column j from that row.

Building a Table. 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.

```
from __future__ import print_function
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)
```

1. Use a list comprehension to create a six by six table of zeros. Actually, the table is six rows. Each row has six columns.

   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 zeros. The outer list, `[ [...] for j in range(6) ]` creates six copies of these inner lists.

2. Print the grid of zeroes.

3. Fill this list of lists 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.

4. Iterate 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 **for** loop enumerates all values of one die, *d1*. The inner **for** 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`. 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]]
```

Better-Looking Output. 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 `"{0:2d}".format(value)` string operation might be useful to get fixed-size numeric conversions.

Summarizing A Table. Let's summarize this two-dimensional table into a frequency table. The values of two dice range from 2 to 12. If we use a list with 13 elements, these elements will be identified with indexes from 0 to 12, allowing us to accumulate counts in this list.

```
fq= 13*[0]
print(fq)
for row in table:
    for c in row:
        fq[c] += 1
print(fq[2:])
```

1. We initialize the frequency table, *fq*, to be a list of 13 zeros.
2. The outer **for** 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.
3. The inner **for** 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 occurrences of each value, *c* by using the value as an index into the frequency table, *fq*. The increment the frequency value by 1.

This program produced the following output.

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1]
```

Using Indexes. 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.

```
for i in range(6):
    for j in range(6):
        c= table[i][j]
        fq[ c ] += 1
```

The outer loop sets the variable *i* to the values from 0 to 5. The inner loop sets the variable *i* 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 *i* to select a column from that row. This is the value, *c*. We then accumulate the frequency occurrences in the frequency table, *fq*.

The first version has the advantage of directly manipulating the Python objects, it is somewhat simpler. The second version, however, is more like common mathematical notation, and more like other programming languages. It is more complex because of a level of indirection. Instead of manipulating the Python sequence, we access the objects indirectly via their index in a sequence.

Matrix Addition. We use this latter 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 multiplication 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 zeros, 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*.

9.5.5 List Processing Exercises

List Comprehension 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**.

   Write a list comprehension that creates a list of tuples. Each tuple has two values, a temperature in Fahrenheit and a temperature in Celsius.

   Create one list for Fahrenheit 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 Fahrenheit values.

Sorting Exercises

1. **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.

Unique Values of a Sequence, *seq*.

(a) **Initialize**. Set $result \leftarrow list()$.

Sort the input sequence, *seq*.

(b) **Loop**. For each value, v in *seq*.

Already in *result* **?** If v is the last element in *result*: ignore it.

If v is not the last element in *result*:

Append v to the sequence *result*.

(c) **Result**. Return list *result*, which has unique values from *seq*.

2. **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 portfolio 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.

Multidimensional Exercises

1. **Matrix Formatting**.

   Given a 6×6 matrix of dice rolls, produce a nicely formatted result. Each cell should be printed with a format like `"| {0:2s}"` 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.

2. **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, $6 \times 6 \times 6$, 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.

9.5.6 Sequence FAQ's

How can there even be an immutable data structure? That sounds like a contradiction. Let's be sure to separate the data object's immutability from setting the value of a variable. A variable's value can be a series of different immutable objects. In many respects, changing the value of a variable is what defines the state of our program, and switching that value from one object to another object *is* what our program is supposed to do. Other times, the object is large, or complex, and it is somewhat more efficient to alter the object rather than create a new one.

Look at the following example. Here, the variable b changes from `"some long"` to `"some long string"`.

```
a= "some"
b= a + " long"
b= b + " string"
```

None of the string objects (`"some"`, `" long"` or `" string"`) change. There are two new strings that are built by this program: `"some long"` and `"some long string"`. Neither of these change after they are built as the program runs.

When the program ends, two strings (`"some"` and `"some long string"`) are associated with variables *a* and *b*. The remaining strings are quietly removed from memory, since they are no longer needed.

While the strings themselves are immutable, the values assigned to our variables reflect our intent to assemble a long string from smaller pieces.

Since lists do everything tuples do and are mutable, why bother with tuples? 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 two-tuples to represent (*x* , *y*) coordinate pairs. Another example might be a program that works with colors as three-tuples, (*r* , *g* , *b*), of red, green and blue levels. A variable-length list is not appropriate for these kinds of fixed-length tuple.

Wouldn't it be more efficient to allow mutable strings? Variable length strings are most commonly implemented by imposing an upper limit on a string's length. Having this upper limit is unappealing because it leads to the possibility of a program having data larger than this upper limit. Indeed, this "buffer overflow" problem is at the root of many security vulnerabilities.

This fixed upper limit model is embodied in the C string libraries. Strings can vary in length, but require the programmer set a fixed upper bound on the length. This amount of storage is allocated, and the string can vary up to that limit. While this provides excellent performance, it does impose an arbitrary restriction. Some languages (Java for example) stop gracefully when the string limit is exceeded, others (C for example) behave badly when strings exceed their declared length.

In effect, Python has strings of arbitrary size. Python does this by creating new strings instead of attempting to modify existing strings. Python is freed from this security issues associated with variable length strings and the resulting buffer overflow problem.

I noticed map, filter and reduce functions in the Python reference manual. Shouldn't we cover these? These functions are actually rather difficult to describe in this context because they reflect a view of programming that is fundamentally different from the approach we've taken in this book. We're covering programming from an imperative point of view. These three functions reflect the *functional* viewpoint. Both approaches are suitable for newbies. We had to pick one, and the coin toss came up imperative.

In the long run, these functions aren't that useful. Why? Because the List Comprehension (see *List Construction Shortcuts*) does everything that the `map()` and `filter()` functions do, making them unnecessary. The reduce design is often much more clearly expressed with an explicit **for** or **while** statement than with the `reduce()` function.

CHAPTER

TEN

ADDITIONAL PROCESSING CONTROL PATTERNS

Exceptions and Iterators

Exception processing is a way to alter the ordinary sequential execution of the statements in our program. Additionally, an `Exception` is an object that is raised internally by Python when our program does something illegal. We can make considerable use of exceptions and exception-handling statements to create event-driven programs. We'll cover this in *The Unexpected : The try and except statements*.

In *Looping Back : Iterators, the for statement and Generators* we'll look closely at some advanced procedural processing. We'll look at a Python object called an iterator and how we can create generator functions. These will allow us to define some more sophisticated processing; processing that will help us cope with the kinds of files we often encounter in the real world.

10.1 The Unexpected : The try and except statements

A well-written program should produce valuable results even when exceptional conditions occur. A program depends on numerous external resources: memory, files, other packages, input-output devices, to name a few. A problem with any of these resources is an exceptional situation that can interrupt the normal, sequential flow of the program.

When you're surfing the web and you see a "page not found" kind of error message, this can be an example of exception handling. The web server tried to find the file named in your web request. The attempt to open the file failed and raised an exception. The web server handled the exception, and sent back an error page instead of the page you asked for.

In *What Does Python Mean By "Exception"?* we define how an unusual event becomes an exception. We show the basic exception-handling features of Python in *How Do We Handle Exceptional Events?* and the way exceptions are raised by a program in *Raising The White Flag in Exceptional Situations*. We describe most of the built-in exceptions in *What the Built-in Exceptions Really Mean*. We'll also include style notes in *Style Notes* and a digression on problems that can be caused by poor use of exceptions in *Exception FAQ's*.

10.1.1 What Does Python Mean By "Exception"?

An *exception* is an event that interrupts the ordinary sequential processing of a program. Once an exception is *raised*, it must be handled immediately. Python examines the *exception handlers* to determine if exception processing should occur. If there is no handler for an exception, the program stops running, and a message is displayed on the standard error file.

We see this happen when we do something as simple as provide improper argument values to a function. This includes dividing by zero, or using `math.sqrt()` on a negative number. Here's a common kind of exception: we'll provide improper values to the `int()` function; it will raise an exception. Since the exception isn't handled, our one-line program will stop.

```
>>> int('not a number')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

Everything has a data and a processing side to it. Exceptions are no exception. *[Well, I thought it was funny.]*

- **Processing**. An exception is an event that changes the sequence of statement execution.
  - A **raise** statement interrupts the sequential processing of statements. This statement will also create an `Exception` object.
  - Handlers can process the exception, and use the `Exception` object.
- **Data**. An `Exception` object contains information about the exceptional situation. The data object is created by a **raise** statement and used by handlers. An exception, at the minimum has a name, but it can have a tuple of argument values, also.

The use of exceptions has a few important consequences.

1. The places in a program that raise exceptions may be hidden deep within a function or class. They should be exposed by describing them in the docstring. A phrase like "raises MySpecialException" is sufficient to alert readers of where exceptions originate.
2. Parts of a program will have handlers to cope with the 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.

Good and Bad Uses. Exceptions can be overused. Because exceptions change the sequence of statements that get executed, they can make a program murky and hard to follow.

Exceptions are best used to manage rare, atypical conditions. Exceptions should be considered as different from expected or ordinary conditions that a program is desinged to handle.

Here's one example: accepting input from a person. Exception processing is not typically used to validate the person's inputs. People make mistakes all the time trying to enter numbers or dates, and these kinds of errors are not *exceptional*.

On the other hand, unexpected disconnections from network services are good candidates for exception processing. These are rare and atypical. Exceptions are best used for handling problems with physical resources like files and networks.

While exceptions are best applied to rare situations, there is an example in Python where an exception is used for what appears to be a common situation. In the case of a **for** statement, there are times when the loop is ended by a `StopIteration` exception. The `StopIteration` exception is not something that your programs would ever deal with, so this use of exceptions is – well – exceptional.

Python has a large number of built-in exceptions, and you can create new exceptions. Generally, it is better to create a new exception that precisely captures the situation rather than attempt to bend the meaning of an existing exception.

10.1.2 How Do We Handle Exceptional Events?

Exception handling is done in the **try** statement. The basic form of a **try** statement looks like this:

```
try:
    suite
```

```
except  exception [ , target ] :
    handler suite
```

```
except:
    handler suite
```

Each *suite* is an indented block of statements. Any statement is allowed in the suite, including additional **try** statements.

If any statement in the **try** suite raises an exception, each of the **except** clauses are examined for a clause that matches the exception raised.

The "normal" course of events is that no statement in the **try** suite raises an exception. If there is no exception, then the **except** clauses are silently ignored.

Each **except** suite is designed to handle specific exceptions. Additionally, a final **except** suite, with no specific exception, can be provided that is a catch-all. This final non-specific **except** suite will be used if no other **except** suite matched the exception.

The structure of the **try** and **except** statements follow this basic philosophy of exceptions.

1. Attempt the intended suite of statements, expecting them work.
2. In the unlikely event that an exception is raised in the **try** suite, find an **except** clause to handle the exceptional situation.
3. If no **except** clause matches the exception raised by the **try** suite, and there is a generic **except** clause, execute that suite to handle the exceptional situation.
4. If there is no handler for the exception that was raised, the program stops with an error. This is what would have happened if there had been no **try** statement in the first place.

Working with IDLE. Here's an simplified example that will show the indentation that IDLE does for us automatically. This **try** statement has multiple suites. IDLE will indent automatically after the **try** clause. We'll have to use the `delete` key to outdent one level to enter the **except** clause.

```
>>> try:
...     a = int("hi mom")
...     print(a)
... except Exception, e:
...     print("Error:", e)
...     a = 42
...
Error: invalid literal for int() with base 10: 'hi mom'
```

1. You start to enter the **try** statement. When you type the letter `y`, the color of **try** changes to orange, as a hint that **IDLE** recognizes a Python statement. After the `:` (which is black), you hit `enter` at the end of the first line of the **try** statement.
2. **IDLE** indents for you. You type the two statements of the suite of statements. The assignment statement (`a=int("hi mom")`) is going to fail when the whole statement is executed. When it raises an exception, Python will start examining **except** clauses for a matching exception; the `print(a)` statement will never get executed.

3. To outdent, you use the `backspace` (Macintosh users will use the `delete` key). Notice that when you finish spelling **except**, that it changes color. Similarly, when you finish spelling `Exception`, it also changes color. Since this statement ends with a `:`, IDLE will automatically indent for you, so you can put in the exception-handling suite.

4. At the end of the suite, you don't have any more statements, so you hit `enter` on a blank line. The **try** statement is complete, so **IDLE** executes the statement. The exception is raised, it matches the first exception clause, a message is printed, and then variable *a* is set.

5. This is the output. This is the text of the `ValueError` which was raised by the attempt to create an integer value from the string `"hi mom"`.

6. When we ask for the value of *a*, we see that it has the value assigned in the exception clause, `42`.

Since the suite of statements in the **try** clause always raises an exception, this example is a little contrived. Let's look at some more typical examples.

10.1.3 Exception Handling Example

Here's an example of exception handling that shows a number of things which can go wrong. This example computes the average of numbers in a tuple. But what if the values in the tuple aren't all numbers? Or, what if the tuple is empty? These are exceptional situations which will raise specific Python exceptions, which our program can handle.

```
1   from __future__ import division, print_function
2   #data= ( 1, 2, 3 ) # Works
3   #data = () # raises ZeroDivisionError
4   data = ( "hi", "mom" ) # raises TypeError
5
6   sum= 0
7   try:
8       for d in data:
9           sum += d
10      print(sum/len(data))
11  except ZeroDivisionError:
12      print("No values in data")
13  except TypeError:
14      print("Some value in data is not a number")
```

4. We set *data* to define a set of data that we'll average. If we set *data* to an empty tuple, or a tuple with non-numeric data, we'll can see different types of exceptions.

7. In the **try** suite, we attempt to compute the sum of the values in the tuple. For certain kinds of inappropriate input, these statements will raise exceptions.

   - If *data* is `()`, an empty tuple, the **try** clause will attempt to divide by zero. This will raise an exception.

   - If *data* has a non-numeric element, the **try** clause will attempt to do a numeric operation on a string, and raise an exception.

11. We have an **except** clause to handle a `ZeroDivisionError`. If this exception is raised, it indicates that we were given an empty tuple.

13. We also have an **except** clause to handle a `ValueError`. If this exception is raised, indicates that we attempted to sum a value which was not a number.

You can run the above example three different ways and see the different kinds of exception handling. You do this by moving the comment (`#`) to choose which value of *data* you want to use.

There are two common design patterns for exception handlers. The most common kind of exception handling will clean up in the event of some failure; it might delete useless files, for example. A slightly less common kind of exception will compute an alternate answer; it might return a complex number instead of a floating-point number, for example. These choices aren't exclusive and some handlers will both delete resources and compute an alternate answer.

10.1.4 Raising The White Flag in Exceptional Situations

We've seen how to handle exceptions when they are raised automatically by Python. We've seen a number of exceptional situations: including attempts to divide by zero, take the square root of a negative number and do arithmetic on strings. Any of these situations will lead Python to raise an exception.

We can raise exceptions in our programs, also. We would do this when our program reaches a situation where we need to "throw in the towel". We may, for example, have a series of **if** statements to handle all the normal situations. If none of those are appropriate, it is often best to raise an exception.

The **raise** statement raises a new exception. Python immediately leaves the expected program execution sequence – the statement after a **raise** is *never* executed – and resumes execution by searching the enclosing **try** statements for a matching **except** clause. The effect of a **raise** statement is to either begin execution in an **except** suite, or the program will stop.

In addition to diverting execution to an exception handler, the **raise** statement also creates a new `Exception` object. In the first syntax example, the **raise** statement creates a new `Exception` with no additional information.

```
raise class
```

This second syntax example shows us that the `Exception` object is a kind of container, and you can stuff a value into the `Exception` that provides additional information. Usually the value is a String that amplifies the exception by providing details about the exceptional condition.

```
raise  class ( value )
```

Built-in exceptions can be raised by giving the exception class name for the *class*.

```
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.

It is possible to define a new class of exception objects. We'll return to this as part of the object oriented programming features of Python, in *Defining New Objects*. Here's the short version of how to create your own unique exception class. In this example, we've created a new family of exceptions called `MyError`.

```
class MyError( Exception ): pass
```

This single line 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 how you raise an exception of your own invention.

```
class MayNotBeNone( Exception ): pass

def someFunction( param ):
    """Does some processing or raises MayNotBeNone if param is None."""
    if param is None:
        raise MayNotBeNone( "{0!r} is invalid".format(param) )
    # Some Processing
    return "Some Result for {0!r}".format(param)
```

Here are two examples of using this function. The first example provides a valid argument value, and no exception is raised. The second example, however, provides an illegal input and an exception is raised by the function.

```
>>> someFunction( complex(1,.5) )
'Some Result for (1+0.5j)'
>>> someFunction(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in someFunction
__main__.MayNotBeNone: None is invalid
```

Exceptions can be raised anywhere, including in an **except** clause of a **try** statement. Raising an exception in an exception handler is a way to translate an exception from an internal Python exception to one our of own exceptions.

```
class MyError( Exception ): pass

try:
    attempt something risky
except FloatingPointError:
    raise MyError("something risky failed")
```

This example will transform a `FloatingPointError` into a user-defined `MyError`.

When the simple **raise** statement is used in an **except** clause, it re-raises the original exception.

```
try:
    attempt something risky
except Exception, ex:
    do_something( ex )
    raise
```

This example does some initial processing of the exception in the function `do_something()` and then re-raise the original exception again for processing by any enclosing **try** statements. This kind of two-step processing is often done to do cleanup of the risky statement, and then re-raise the exception so that the overall application to then log the error or stop running gracefully.

10.1.5 A More Complete Example

The following example uses a uniquely named exception to indicate that the user wishes to quit rather than supply input. This is how we make the `raw_input()` function into something more usable. Our approach is to separate the good data from other situations that arise when people interact with software like asking for help or trying to quit.

The Problem. We can't simply use the `raw_input()` function because it doesn't give us helpful feedback on errors, offer useful help, or gracefully handle a request to quit the application.

We'll lift up just one common interaction as an example of this. Let's say we want to get a "yes/no" answer from the person using our program. We want just the final answer, encoded as a string, either `"Y"` or `"N"`. If the user asks for help, we want them to see the help, but that isn't a result from our function. Similarly, if the user wants to quit the program, we want an exception to be raised.

Forces and Alternatives. Using `raw_input()` means that we have a lot of programming to validate the input, provide help and handle exceptions. Since this programming is almost always the same, we need to package this as a function.

A single function to handle all kinds of input seems rather complex. Validating a time or date is different from validating a "yes/no" answer. It will be easiest to have a family of functions: one for "yes/no", another for dates, another for file names.

A common way to provide help is to reserve an additional keyboard key for this. Apple keyboards have a `help` key, but most other computers lack this. Consequently, machines that are designed to run Windows traditionally use `F1`. However, when running in **IDLE**, `F1` is captured by **IDLE**, not by our script. When running from the command window, `F1` is captured by the command window itself as part of command history processing. Consequently, we'll use the `?` key for help.

There is no standard way for a user to say they want to exit from a script. While a character sequence like `ctrl-Q` is often used by GUI applications, this doesn't work very well for our scripts. IDLE captures this key sequence before our script sees it as input. Consequently, we'll use end-of-file or a simple `Q` to signal that we want to quit.

A Solution. We'll define a more focused function to get user input. This function will validate the input, providing useful error messages. It will also provide help when the user asks, and raise an exception when the user wants to quit.

Since we stole this idea, we'll also steal the name for this function. We'll call it `ckyorn()` as in "check for y or n". We can imagine defining a whole flock of these kinds of functions to check for numbers, check for dates, check for valid directories or file names.

The Contract. In this case, we're going to define a function that has a proper return value that will always be either `"Y"` or `"N"`. A request for help (`"?"`) is handled automatically inside this function. A request to quit is treated as an exception, and leaves the normal execution flow. This function will accept `"Q"` or end-of-file (via `Ctrl-D`; `Ctrl-Z Enter` on Windows) as the quit signal.

We'll define a new `UserQuit` exception to signal that the user wants to quit. In a longer program, this exception permits a short-circuit of all further processing, omitting some potentially complex **if** statements. If the user enters "Q", we'll raise this exception. What about end-of-file?

We can run a quick experiment to see what exception is produced by the `raw_input()` function when we sent it an end-of-file signal. We'll show the normally invisible `Ctrl-D` as `^D`.

```
>>> a=raw_input('test:')
^D
test:Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
EOFError
```

Our input function must transform the built-in `EOFError` exception into our `UserQuit` exception. We can do this by handling the `EOFError` exception and raising a `UserQuit` exception.

ckyorn.py

```
from __future__ import print_function

class UserQuit( Exception ): pass

def ckyorn( prompt, help="" ):
    """ckyorn( prompt, help ) -> string
    Raises UserQuit if the user is trying to quit.
    """
```

```
9       a= ""
10      ok= False
11      while not ok:
12          try:
13              a=raw_input( prompt + " [y,n,q,?]: " )
14          except EOFError:
15              a= "Q"
16              raise UserQuit
17          if a.upper() in [ 'Y', 'N', 'YES', 'NO' ]:
18              ok= True
19          elif a.upper() in [ 'Q', 'QUIT' ]:
20              raise UserQuit
21          elif a.upper() in [ '?' ]:
22              print(help)
23          else:
24              pass
25      return a.upper()[:1]
```

3. We define our own exception, `UserQuit`. We'll use this to signal one of two events: the user entered a "Q", or the user signaled and end-of-file to the operating system.

5. The `ckyorn()` function 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.

11. We establish 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"`).

12. 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 GNU/Linux it is `Ctrl-D`; for Windows it is `Ctrl-Z`.

14. 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.

17. If no exception was raised, we examine the input character to see if we can interpret it.

    If the user entered an expected answer, we set *ok*. The user's input is in *a*, which we can return.

    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 in a completely uniform manner.

    If the user enters `?`, we can provide a help message and prompt for input again.

25. 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.

We can use this function as shown in the following example. Here's a line of a script that uses our new `ckyorn()`.

```
allDone= ckyorn(
    help= "Enter Y if finished entering data",
    prompt= "All done?")
```

Here's the results from running this little script to get a value for *allDone* .

```
All done? [y,n,q,?]: ?
Enter Y if finished entering data
All done? [y,n,q,?]: q
Traceback (most recent call last):
    File "<stdin>", line 3, in <module>
    File "<stdin>", line 17, in ckyorn
  __main__.UserQuit
```

This example shows how we use exceptions to handle unexpected situations that arise. The most common source for these unexpected situations are the operating system and the human user. In the operating system case, there are resource limits that may lead to unexpected problems: we might be out of disk space or out of memory. In the human user case – well – people are unpredictable.

10.1.6 Debugging an Exception Handler

Debugging exceptions can be challenging. There are a number of things that can go wrong with exception-handling clauses. In all of these cases, the root cause is the same: an exception was reported that we didn't want to see, or an exception was handled by the wrong exception handler. We'll look at a number of the potential problems.

We'll break debugging down into two parts: the design problems and the syntax problems.

Tip: Debugging Exception Design

The try statement has at least two suites: the **try** suite and at least one **except** suite. Each of these can have `:`s missing, or be indented incorrectly. Since these are large, composite statements, there are a lot of places where problems can occur.

One other problem is that we may have put the wrong statements in the **try** suite. If we evaluate a statement that raises an exception, but that statement is not in a **try** suite, the exception won't get handled. If our **try** statement doesn't seem to catch the exception, one possibility is that we didn't enclose correct statement in the **try** statement.

Since Python reports the line number where the exception was raised, we can see where the exception originated and adjust the location of the **try** or **except** clauses to include the proper statements.

Another problem is that the exception is raised and the exceptions on our **except** statements don't match. We'll address this in *Debugging Exception Handling*.

Including too many statements in the **try** suite is just as bad as having too few statements. Including statements which cannot raise an exception in the first place can lead to confusion when reading the program. When we look at a program we wrote two weeks ago, we don't want to struggle to understand what it means. We'd like to be reasonably clear. To this end, a **try** suite should be as small as possible to handle the exception.

Second, we may be raising an exception for the wrong reason. Since a **raise** statement is always associated with an **if**, **elif** or **else** suites, the conditions on the **if** statement define the exceptional condition. We should be able to clearly articulate the condition that leads to the **raise** statement. Problems in the **if** statement will surface as errors in exception processing.

Tip: Debugging Exception Handling First, we may have the wrong exceptions named in the **except** clauses. If we evaluate a statement that raises an exception, but that exception is not named in an **except** clause, the exception won't get handled.

Since Python reports the name of the exception, we can use this information to add another **except** clause, or add the exception name to an existing **except** clause. We have to be sure we understand why we're getting the exception and we have to be sure that our handler is doing something useful. Exceptions like `RuntimeError`, for example, shouldn't be handled: they indicate that something is corrupt in our Python installation.

You won't know you spelled an exception name wrong until an exception is actually raised and the **except** clauses are matched against the exception. The **except** clauses are merely potential statements. Once an exception is raised, they are actually evaluated, and any misspelled exception names will cause problems.

Second, we may be raising the wrong exception. If we attempt to raise an exception, but spelled the exception's name wrong, we'll get a strange-looking `NameError`, not the exception we expected.

As with the **except** clause, the exception name in a **raise** clause is not examined until the exceptional condition occurs and the **raise** statement is executed. Since **raise** statements almost always occur inside **if**, **elif** or **else** suites, the condition has to be met before the **raise** statement is executed.

Generally, we prefer to minimize our use of the built-in Python exceptions. There are times when an existing exception clearly captures the nature of the condition. More often, however, our program has a unique exception, and we should have a uniquely named exception. By using our own exceptions, rather than Python exceptions, we avoid conflating our exceptional conditions with Python's own internal exceptional conditions.

Typically, we only define one or two new exceptions for our own modules. We don't want to define a large, complex group of exception classes. The typical approach is to define our own general-purpose `Error` exception in our module.

10.1.7 What the Built-in Exceptions Really Mean

Python has over two dozen built-in exceptions. These exceptions can be organized into three logical groups. The first group are exceptions that may actually occur in our programs. The second group are exceptions that can occur, but when they do occur, it means our program is very badly broken. The third group are supporting definitions that we newbies won't use.

This first group of exceptions can happen to our program. In some cases, an exception means that our program has a serious design or programming problem. In other cases, it may mean that we have to cope with unexpected situations like invalid input from the user, or running out of memory.

`AssertionError` Assertion failed. This exception is raised by one of our program's **assert** statements. See *The assert Statement* for more information.

`AttributeError` Attribute not found in an object. This exception indicates a serious design problem in an object class definition. Generally, it means that we forgot to assign an initial value to one of the attributes of an object. See *Defining New Objects* for more information.

`EOFError` Read beyond end of file. This happens when we read a file from the disk, or any file-like network resource. This isn't really "unexpected"; since the file is of a finite size, it has to end eventually. However it is a rare situation that happens just once at the end of the file. Generally, this exception is handled for us. See *External Data and Files* for more information.

`FloatingPointError`, `OverflowError` A floating-point operation failed or a numeric result was too large to be represented. This is rare, and often means that we've exceeded one of the internal limits on float-point operations. The first thing to suspect is an error in translating a mathematical formula into Python. If you've translated the formula correctly, you can prove that it works by supplying some known input values and getting a proper result. If your program works for some values, but doesn't work for other values, you have more serious mathematical issues, which are beyond what we can cover in this book.

`IOError` I/O operation failed. This can happen any time we deal with disk files or network resource. This general exception covers all of the various kinds of problems that can occur. When you print the exception, you'll see some additional details on what the real problem is. See *External Data and Files* for more information.

`IndexError` Sequence index out of range. This comes from trying to find an item beyond the end of a sequence – a tuple, list or string. This is always a design error: we shouldn't try to find items that don't actually exist. One of the most common occurrences is trying to find the first item of a sequence with no items at all. See *Basic Sequential Collections of Data* for more information.

This exception almost always means that an **if** statement is needed so that something more useful can be done when the sequence is empty.

`KeyError` Mapping key not found. This comes from trying to find a key that doesn't exist in the map. This is always a design error: we shouldn't try to find items that don't actually exist. See *Mappings : The dict* for more information.

We have a number of solutions: we can fix our program to put the element into the map correctly. Or, we can use the `get()`, `setdefault()` or `has_key()` method functions to determine if the key exists or to provide a suitable default value when the key doesn't exist.

`KeyboardInterrupt` Program interrupted by user. This happens when the user hits `Ctrl-C`. The user wants to exit from our program. Generally, we should not handle this exception. It's better to let our program stop running when the user wants it to stop.

`MemoryError` Out of memory. This may be a design problem in our program, or it may be the user's problem for buying a computer which is too small. If our program consistently runs out of memory, it could be designed to create too many objects. Almost all algorithms have two variations: one which operates in less time, and another which uses less memory. These design considerations are beyond the scope of this book.

`OSError` OS system call failed. This can happen any time we deal with any operating system resource. This general exception covers all of the various kinds of problems that can occur. When you print the exception, you'll see some additional details on what the real problem is.

`RuntimeError`, `SystemError` Unspecified run-time error or an internal error in the Python interpreter. When this happens, Python simply can't cope with something. This is rarely the fault of your program. More likely, you've got some complex problem with your operating system, Python or some add-on modules. If the problem is consistent, you should consider that you may have more serious problems with your computer. You may have viruses, spyware or other corrupt files.

`SystemExit` Request to exit from the interpreter. This exception is raised by the `sys.exit()` function.

`TypeError` The types of data don't make sense with the function or operator. This is a more serious design error. For example, :`"2"+3` is an example of a `TypeError`. If we mean to perform arithmetic, one of the values needs to be converted to a number. If we mean to concatenate strings, one of the values needs to be converted to a string.

`UnicodeError` Unicode related error. This happens when we attempt to process a Unicode string that isn't properly encoded. This often happens when reading Unicode data from files or other network sources. In this respect, it is like an `IOError` exception, and should be handled similarly.

`ValueError` A function was given an inappropriate argument value of a valid data type. The most notable example is attempting to take the square root of a negative number. Because you provided a number, the data type is valid. However the value of the data was not valid.

Compare `math.sqrt(-1)` with `math.sqrt("Hello Dolly")`. The first is sometimes reported a `ValueError` because the type is right, by the value's range is inappropriate. The second is a `TypeError`.

`ZeroDivisionError` The second argument to a division or modulo operation was zero. This is a design error, also. It is easy to check for this situation in an **if** statement and do something more useful than raise an exception.

The following exceptions are more typically returned at compile time – before your program can even execute. These errors indicate an extremely serious error in the basic construction of your 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 the module, or can't find a requested name within the module.

`IndentationError` Improper indentation.

`NameError` Name not found either locally (inside the function) or 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 the internal definitions on which `Exception` objects are based. Normally, these never occur directly. You would use these when designing a new exception of your own.

`Exception` Common base class for all exceptions.

`StandardError` Base class for all standard Python exceptions.

`ArithmeticError` Base class for arithmetic errors.

`EnvironmentError` Base class for I/O related errors.

`LookupError` Base class for lookup errors.

10.1.8 Exception Exercises

Background

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: `[,?,q]:`

```
your prompt  valid input hints
```

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.

General User Input Function Algorithm

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.

Prompt and Read. Use raw_input to prompt and read a reply from the user.

Help? If the user entered ”?”, provide the help message.

Quit? If the user entered “q” or “Q”, raise a UserQuit exception.

Try the following suite of operations

Convert. Attempt any conversion.

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.

Nothing?. If the user entered nothing, and there is a default value, return the default value.

3. **Result**. Return the validated user input.

   In the event of an exception, this function should generally raise a `UserQuit` exception.

Exercises

1. **ckdate**

   Prompts for and validates a date. The basic version can require dates have a specific format, for example `mm/dd/yy`. A more advanced version can 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 *Time and Date Processing : The time and datetime Modules*. This `ckdate()` function must not return bad dates or other invalid input.

2. **ckint**

   Display a prompt; verify and return an integer value. This version has no range checking, that is done by a separate function that gets an integer value in a given range.

3. **ckitem**

   Build a menu; prompt for and return an item from the menu of choices. A menu is a numbered list of 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, in addition to writing the help message, this additional help will also redisplay the menu of choices.

4. **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.

5. **ckpath**

   Display a prompt; verify and return a pathname. An advanced version can use the `os.path` module for information on construction of valid paths. This should check the user input to confirm that the path actually exists. See *Modules : The unit of software packaging and assembly* for more information.

6. **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.

7. **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.

8. **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.

9. **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.

10.1.9 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 *Modules : The unit of software packaging and assembly*. It is not typical to have a complex hierarchy of exceptional conditions defined by a module.

10.1.10 Exception FAQ's

Isn't raise just a glorified goto statement?

For that matter, aren't break and continue just glorified goto?

Isn't the goto statement harmful? 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 exception-handling suite. Sometimes these exception handlers are hard to find. This is a correct description of the construct, which leads to some difficult decision making.

Exceptions should reflect truly unusual situations. In this way, the normal course of events is clearly stated in the main path of execution. The presence of exceptions should simplify and clarify the program.

The same considerations apply to **break** and **continue**. Their use should clarify and simplify a program by eliminating the clutter of rare and unusual complications.

Future examples, which use I/O and OS calls, will show simple exception handling. However, exception laden programs are a problem to comprehend. Exception clauses are relatively expensive, measured by the time spent to understand their intent.

Aren't some exceptions the result of something that could be checked with an if statement? Some exception-causing conditions are actually predictable states of the program. The notable exclusions are I/O Error, Memory Error and OS Error, which 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.

For example, the quadratic equation example we have been using for this chapter can create two exceptions, each of which is much more easily and clearly checked with simple **if** statements.

10.2 Looping Back : Iterators, the for statement and Generators

We've looked at iteration from several points of view. Initially, we looked at the processing side of the coin when we looked at the **for** and **while** statements in *While We Have More To Do : The for Statement*. At that time, we lifted up the `range()` function as a good way to control the iterative processing.

We looked at the data side of the coin when we looked at sequences in *Flexible Sequences : The list* and *Common List Design Patterns*. In those sections we looked more closely at the `range()` function and how it creates a sequence that is used by a **for** statement.

A sequence (including the list created by `range()`) is intimately associated a design pattern we call a *iterator* and it's associated *generator function*. A generator helps us *generate* a sequence of values. This chapter will describe how to build generators.

In *The Iteration Contract – What the for Statement Expects* we'll look more closely at the **for** statement and how it makes use of an iterator. We'll use this to show more complete semantics of generators in *Customizing Iteration*. We can then look at the syntax for defining generator functions in *Generator Definition: The def and yield Statements*.

We'll provide two examples of how to make use of generators in *Putting Generators To Use*, *Geeky Generator Example: Web Server Logs*. and *Generator Example: Roulette Spins*.

10.2.1 The Iteration Contract – What the for Statement Expects

When we introduced iteration, we skipped over a few details that weren't helpful at the time. In this section, we'll take a close look at how iteration really works. We'll look at an example of an *iterator* and how the **for** statement uses this.

An iterator has four interesting events when it is evaluated in a **for** statement. This forms a contract between the **for** statement and the iterator. We'll provide names for each clause in the contract, what the **for** statement does, and what the iterator does to cooperate with the **for** statement.

Table 10.1: The Iterator Contract

Clause	The **for** statement	The Iterator
init	Initialize the Iterator.	The iterator does any initial calculations – similar to the way a function is evaluated. A **yield** statement provides an *initial* value.
next	Request the next value from the Iterator. If the Iterator yields a value, execute the **for** suite with this value.	The iterator is resumed where it left off; it yields the next value. Being resumed right after the **yield** statement is the unique feature of an iterator.
stop	When the iterator raises an exception, stop processing. Silently consume the `StopIteration` exception. Raise all others.	The iterator suite ends or executes **return**. This raises a `StopIteration` exception to indicate there are no more values.
break	At any time, the **for** statement may stop asking for values; this can happen when a **break** statement is executed.	The iterator may never get a chance to finish processing normally.

Looking ahead, the names of these clauses point toward the names of the method functions of `iterator` objects. This is not something we'll dwell on here, but that's how we chose those names for the clauses in the contract.

Sources of Iterators. A sequence object is the most common source for an iterator. When we say something as simple as the following, we are asking a list object to secretly hand off an iterator to the **for** statement. In this example, the list `[1, 2, 3, 4, 5]` gives a hidden iterator object the **for** statement.

```
for i in [1, 2, 3, 4, 5]:
    print(i)
```

Here's the secret hand-off that happened under the hood. This little example will show you that an iterator object is created. In the next section we'll see how to make use of that iterator object.

```
>>> iter( [1,2,3,4,5] )
<listiterator object at 0x70ef0>
```

We can look at our contract and see what happens under the hood when a **for** statement uses the list `[1,2,3,4,5]`.

- The **for** statement implicitly uses the `iter()` function to request an iterator from the list `[1,2,3,4,5]` and saves this iterator in a private variable somewhere.

  Conceptually: `forIter= iter( [1,2,3,4,5] )`.

- The **for** statement calls the iterator's `next()` method; the iterator yields the individual items in the list so that the **for** statement can execute the suite.

  Conceptually: `i= next(forIter)`.

- When the iterator runs out of values it raises an exception. The **for** statement handles this exception and finishes normally.

  Conceptually, there's a **try** block that quietly handles the `StopIteration` exception.

Explicit Iterators. In addition to Python's implicit use of iterators, we can explicitly ask for an iterator object. We can then manipulate that iterator to do more sophisticated processing on the underlying sequence.

Let's look at this **for** statement.

```
total=  0
for j in range(1,21,2):
    total += j
print(total)
```

Here is the equivalent program, written with an explicit iterator object and a **while** statement. From this, we can see precisely what the **for** statement does for us.

```
1  total= 0
2  try:
3      forIter= iter( range(1,21,2) )
4      while True:
5          j= next(forIter)
6          # The original suite
7          total += j
8  except StopIteration:
9      pass
10 print(total)
```

3. Initially, we get the iterator object and save it in a local variable, *forIter*.
5. We get the next value from the iterator object. We execute the suite of statements that are the suite of statements in the **for** statement.
8. When the iterator raises `StopIteration`, there are no more values to process.

The iter() Function. Here is the formal definition for the `iter()` function which exposes the iterator object to us. This is what the **for** statement uses under the hood to get the iterator for a sequence.

`iter`(*sequence*) → iterator
    Returns an iterator object from the given sequence.

10.2.2 Head-Tail Design

The most important use for detailed control over an iterator is to handle a common problem where we need to skip over the first items in a sequence. When we get to file processing, we'll find that files often have column titles or header records that must be processed differently from the rest of the body. By using the iterator object explicitly, we can gracefully skip over header records.

This is easy to write in Python because the **for** statement can accept either a sequence, an iterator or a generator function. Both an iterator and a generator function that we create adhere to the iterator contract. In the case where we use a sequence, the **for** statement will request the iterator object automatically.

Here's an example that skips over the first two values of a sequence.

```
someSequence= range(5)
theIterator= iter(someSequence)
headItem0= next(theIterator)
headItem1= next(theIterator)
for i in theIterator:
    print(i)
```

We'll see that a variety of collections have a "head-tail" structure. There is a header (usually a fixed number of items) and a tail that comprises all the rest of the items. When we create a spreadsheet, for example, we often have a fixed number of rows of column titles and an indefinite number of rows of data.

Tip: Debugging Iterators

There are several common problems with using an explicit iterator.

- Skipping items without processing them.
- Processing the same item twice
- Getting a `StopIteration` exception raised when trying to skip the first item.

Generally, the best way to debug a generator is to use it in a very simple iteration statement that prints the result of the iteration. Printing the items will show us precisely what is happening. We can always change the **print** statement into a comment, but putting a # in front of print.

Here's a good design pattern for skipping the first item in a sequence.

```
i = iter( someSequence )
next(i)         Skips an item on purpose
while True:
    a= next(i)
    some processing
    print(a)
```

Skipping items happens when we ask for the next() method of the iterator one too many times.

Processing an item twice happens when we forget to ask for the next() method of the iterator. We see it happen when a program picks off the header items, but fails to advance to the next item before processing the body.

Another common problem is getting a StopIteration exception raised when trying to skip the header item from a list or the header line from a file. In this case, the file or list was empty, and there was no header. Often, our programs need the following kind of **try** block to handle an empty file gracefully.

```
i = iter( someSequence )
try:
    next(i)  Skips an item on purpse
except StopIteration:
    No Items -- this is a valid situation, not an error
```

10.2.3 Customizing Iteration

A *generator function* is closely related to an iterator. It's something we can write. It follows the iterator contract so it can work with a **for** statement nicely.

First we'll look at a built-in generator function, then we'll look at how we create our own generator functions.

The xrange() generator. The xrange() function is an example of a generator that we can use instead of an implicit iterator. Recall that the range() function creates a list, and that the **for** statement must request an iterator from that list.

We can slim down this two-step operation by giving the **for** statement an xrange() generator instead of a list object. Our program will run slightly faster because we aren't creating a complete list and then creating an iterator; instead, we'll directly use a generator to yield individual values.

Here's what a program looks like that uses xrange() instead of range().

```
total= 0
for i in xrange(1,19,2):
    total += i
print(total)
```

Yes. It looks the same. Here's the small, but profound difference:

- xrange() is an iterator.
- range() creates a list from which an iterator is then created.

This sets the stage for us writing functions that are more like xrange() instead of range().

You can think of the `range()` function as having a definition like the following. The `range()` function result is the list created by iterating through the the `xrange()` generator.

```
def range(start,stop,step):
    return list( xrange(start,stop,step) )
```

Here's the formal definition for the `xrange()`. It looks a lot like `range()`.

`xrange`(*[start]*, *stop*[, *step*]) → generator
: Returns a generator (also known as a "generator iterator") that yields the same list of values that the `range()` function would return. However, since this is a generator, a list is not actually created in advance, making this faster and more memory efficient.

Important: Python 3

There will be a slight change in Python 3.

The `xrange()` function is actually much more useful than `range()`. `xrange()` is so much more useful that than the Python 3 `range()` function will be an iterator (exactly like Python 2 `xrange()`).

The Python 2 `range()` function – which creates a list object – will be removed from Python 3.

Why? We rarely want the actual list object from the `range()` function. It's far more common to want the iterator. The few times we need the list object, we can use the `list()` factory function to build the list.

```
someList = list( range(1, 21, 2) )
```

10.2.4 Generator Definition: The def and yield Statements

We define a generator using syntax that looks similar to the syntax for defining a function. The primary difference is that a generator will include at least one **yield** statement.

A generator, unlike an ordinary function, will work with a **for** statement. It has a very different life cycle from an ordinary function, even though they look alike.

We define a generator function by providing three pieces of information:

- The name of the generator.
- A list of zero or more variables, called *parameters*, with the domain or input values.
- A suite of statements. This must contains at least one **yield** statement, this will yield each individual output or range value to the **for** statement that uses this generator.

When the generator is initialized by a **for** statement, it will be called much like a function, and will execute until it reaches a **yield** statement. This yielded value will be given to the **for** statement for processing.

After each value is consumed, the **for** statement will resume execution of the generator on the next statement after the **yield** statement. Python handles all of the bookkeeping to make this happen.

When the generator exits normally (through a **return** statement, or by finishing all of the statements in its suite) it will raise a special `StopIteration` exception to notify the **for** statement that everything has finished normally.

def Syntax. We create a generator with a **def** statement. This provides the name, any parameter variables and the suite of statements that yields the generator's results. The definition of a generator looks nearly identical to the definition of a standard function. The one notable difference is the use of a **yield** instead of **return**.

```
def  name  (  [ parameter [ = initializer ] ] , ... ):
    suite
```

The *name* is the name by which the generator function is known. It must be a legal Python name; the rules are the same for function names as they are for variable names. The name must begin with a letter (or _) and can have any number of letters, digits or _. See *Python Name Rules*.

Each *parameter* is a variable name; these names are the local variables to which actual argument values will be assigned when the function is applied. We don't type the [and]'s; they show us that the list of names is optional. We don't type the **...**; it shows us that any number of names can be provided. Also, the **,** shows that when there is more than one name, the names are separated by **,**s.

The *suite* (which must be indented) is a block of statements that *must* include a **yield** statement to yield values for the generator. Any statements may be in this suite, including nested function definitions.

As with functions, the first line of a generator is expected to be a document string (generally a triple-quoted string) that provides a basic description of the function. See *Functions Style Notes*.

yield Syntax. The **yield** statement provides each value to the **for** statement. **,**

```
yield  expression [  ... ]
```

The **for** statement must have the proper number of variables to match the number of expressions in the **yield** statement.

The presence of the **yield** statement in a function body means that the function is actually a generator object. The generator will have the complete interface necessary to work with the **for** statement.

A **return** statement can be used to end the iteration. If used, the **return** statement doesn't return anything, and cannot have an expression. In a generator, the **return** statement raises the `StopIteration` exception to signal to the **for** statement that we are finished.

A Goofy Example. Here's an example that uses a sequence of **yield** statements to yield a fixed sequence of values. While not terribly practical, this shows how the **yield** statement fulfills the iterator contract with a **for** statement.

```
def primeList():
    yield 2
    yield 3
    yield 5
    yield 7
    yield 11
    yield 13
```

After defining this generator, here's what we see when we use it. This behaves as if we had the list `[2, 3, 5, 7, 11, 13]`. For a small list like this, the difference is invisible. However, for very large lists, the generator doesn't use as much memory.

```
>>> for i in primeList():
...     print(i)
...
2
3
5
7
11
13
```

Generate Craps Dice Rolls. Here's a slightly more sophisticated generator that yields a sequence of dice throws ending with a seven or the desired point. This generator creates pairs of random dice. If the pair of dice are 7 or the point, the generator yields the final roll to the **for** statement, and then finishes.

While the pair of dice is not 7 and not the point, then the pair will be yielded to the **for** statement. Each iteration of the **for** statement suite will generate the next pair of dice.

```
import random
def genToPoint( point=None ):
    d1,d2= random.randrange(1,7),random.randrange(1,7)
    while d1+d2 != 7 and d1+d2 != point:
        yield d1, d2
        d1,d2= random.randrange(1,7),random.randrange(1,7)
    assert d1+d2 == 7 or d1+d2 == point
    yield d1,d2
```

Here are two examples of using this generator function in a **for** statement. Since these examples depend on random numbers, your mileage may vary. In the first case, the generator yielded three spins, ending with a 7. In the second case, it yielded four spins before stopping. In both cases, we finished with Craps.

```
>>> for r in genToPoint(10):
...     print(r)
...
(3, 3)
(3, 3)
(4, 1)
(4, 1)
(1, 6)

>>> for r in genToPoint(10):
...     print(r)
...
(5, 5)
```

Tip: Debugging Generator Functions

One of the most common problems people have with writing a generator is omitting the **yield** statement. Without the **yield** statement, you have written a simple function: the **for** statement can't initialize it and get individual values from it.

If you get a `TypeError: iteration over non-sequence` error, you have omitted the **yield** statement from your generator function.

Additionally, all of the iterator problems are applicable when creating a generator function. We could have problems that cause us to skip items, process items twice or get an unexpected `StopIteration` exception.

10.2.5 Putting Generators To Use

We've dropped several hints about the need for generator functions. There are several classes of algorithms that work well with generators.

- We'll use generators when we have data that involves what we call the "Head-Tail" pattern. We have some data at the beginning which has to be treated specially.
- We may have a "Look-Ahead" problem. In this case, we have a sequence of data that forms several sequential "blocks" of data with a head or tail delimiter. Life is simpler if we can look ahead to see what's coming next in the sequence.

- We may have a "Reduce" problem where we're summarizing (or "reducing") a larger sequence into a summarized sequences. A sequence of stock transaction details that must be added up to create a sequence subtotals, for example.

Head-Tail. In the "Head-Tail" pattern, we have one or more items which are a preamble or heading. The most common example of this is data that we get from spreadsheets with column titles. We may, for example, want to download stock quotes from the internet; these files often have column titles or other preambles in front of the real data. Generally, the preamble is of a fixed size, and we can look at sample data to see how many lines of column titles need to be skipped.

The solution to the head-tail problem uses an explicit iterator. These solutions have this general pattern:

```
iterator = iter(  sequence )
# Consume the a heading item.
head = next(iterator)
# Process the tail items.
for  variable in  iterator :
    process a tail item
```

Sometimes, the heading is more complex than a fixed number of lines. In this case, we may have to do more sophisticated processing to skip the header. For example, the header may end with an item that has a long string of -s.

In this case, we may want to use an explicit iterator object. If we provide a sequence to the **for** statement, it will request an iterator from the sequence. If, on the other hand, we provide the **for** statement with an explicit iterator, the **for** statement won't reinitialize it.

```
myFakeData= """some title
some other title
----------------
the real data
more real data"""

# Create an iterator
myIter= iter( myFakeData.splitlines() )

# Find the last line of the header
for line in myIter:
    if line.startswith("--"):
        break

# Process the data after the header
for data in myIter:
    print(data)
```

1. We create an iterator, *myIter* over our fake data.
2. The first **for** statement uses this iterator and stops iteration when it finds a line that starts with `"--"`.
3. The second **for** statement uses the iterator to process the lines after the the heading.

Look-Ahead. The "Look-Ahead" pattern occurs when we have a number of items that need to be grouped together. We can't identify the last line of the group, but we can identify the first line of the next group. In this case, we have to look ahead one line to see if it is part of the group we are currently gathering, or is the first line of the next group.

A solution to the look-ahead problem requires a generator. These solutions have this general pattern:

```
def  generator (  parameters ):
    Create an empty accumulator
    for i in some sequence :
        Look at item i
        if not part of the current group? :
            yield the accumulated response
            Create a new, empty accumulator
        Accumulate  i in the accumulator
    yield the final accumulated response
```

There is a common variation on this theme. This combines the head-tail pattern with the look-ahead pattern. This is slightly less desirable because there are two copies of the *Seed the group accumulator with i* statement(s). However, for some kinds of complex processing, this may be difficult to avoid.

```
def  generator (  parameters ):
    iterator = iter(  some sequence )
    Skip the heading
    i =  iterator .next()
    Seed a new accumulator with  i
    for i in  iterator :
        Look at item  i
        if  part of the group? :
            Accumulate  i in the accumulator
        else:
            yield the accumulated response
            Seed a new accumulator with  i
    yield the final accumulated response
```

10.2.6 Geeky Generator Example: Web Server Logs

Here's an example of a look-ahead generator. It's very geeky – reading web server logs – but very common. We won't talk about the job of webmaster and web server administration. We'll talk about the Python programming needed to solve this common problem.

While a log analyzer traditionally reads a file (something we'll get to in *External Data and Files*), it could just as well process a triple-quoted string. Using a triple-quoted string is a great way to design and debug Python programs. We can easily create a triple-quoted string with a cut and paste of real data. We can then write our program using this small amount of data for testing purposes.

Here, for example, is a snip from a web server's log file. Most lines have a time stamp, a severity code (`INFO`, '`ERROR`' and `WARNING` are possible values), a process name, a username, a `-` and then some text. Some of these log messages continue on to following lines. Since each message occupies a variable number of lines, this log is rather difficult to process.

```
log= """
2003-07-28 12:46:42,843 INFO  [main] [] -
-------------------------------------------------------------------
XYZ Management Console initialized at: Mon Jul 28 12:46:42 EDT 2003
Package Build: 452
-------------------------------------------------------------------

2003-07-28 12:46:50,109 INFO  [main] [] - Export directory does not exist
2003-07-28 12:46:50,109 INFO  [main] [] - Export directory created successfully
2003-07-28 12:46:50,125 INFO  [main] [] - Starting Coyote HTTP/1.1 on port 9842
2003-07-28 12:57:14,046 INFO  [Thread-11] [] - request.getRequestURI =...
2003-07-28 12:57:18,875 INFO  [Thread-11] [admin] - Logged in
2003-07-28 12:57:19,625 INFO  [Thread-11] [] - request.getRequestURI =...
"""
```

We can process this more conveniently if we change each complete message into a tuple of lines. This makes multi-line messages (like the very first one in the log) and single-line messages (like the remaining lines) similar enough that processing is much easier.

We'd like to rearrange this text into a list of tuples. Each tuple is a complete log message.

- Item 0 of a message tuple should be the decomposition of the message header line. We can break it down into a 5-tuple with the time stamp, the severity, the process name, the user name and all of the following text. The time stamp is, itself, can be 7-tuple of year, month, day, hour, minute, second and millisecond.
- The remaining items of a message (if any) are simply additional lines of text from the message in the log file.

We'd like the first message in the log, which occupies the first 5 lines, to become the following Python list. Item 0 is a tuple which describes the header line, items 1 to 4 are the extra text after that header line. When we look at item 0 of the tuple, we see that it is a tuple with the time stamp, the severity, the process name and two empty strings for the missing items on the first line. The time stamp is also a tuple.

```
[ ( (2003,7,28,12,46,42,843), 'INFO', 'main', '', ''),
"------------------------------------------------------------------------------",
"XYZ Management Console initialized at: Mon Jul 28 12:46:42 EDT 2003",
"Package Build: 452",
"------------------------------------------------------------------------------" ]
```

We'd like the second message in the log, which occupies the next 1 line, to become the following Python list. Item 0 is a tuple which describes the header line, and this is the only item in the list. When we look at item 0 of the tuple, we see that it is a tuple with the time stamp, the severity, the process name and an empty strings for the empty `[]` which would normally have a username. The time stamp is also a tuple.

```
[ ( (2003,7,28,12,46,50,109), 'INFO', 'main', '',
'Export directory does not exist' ) ]
```

The Goal. Once we've done this transformation from the original text to these Python structures, we can then easily scan the log for interesting messages. Item 0 of each message is the header tuple, item 1 of this header tuple is the severity. If the log is transformed to a list, this processing can be a simple filter. Our goal, then, is to use simple filters to find interesting log messages.

Assume our log is transformed into a variable named *logList* and we want to see all messages where the severity is `ERROR`. A filter that keeps just the headers of these messages could look like this.

```
[ message[0] for message in logList if message[0][1] == 'ERROR' ]
```

Generator Design. Here's the start of a generator which will collect a message and all of the following lines into a tuple of strings. This has the basic pattern of a "look-ahead" generator. We'll accumulate a complete message by looking ahead to the first line of the next message. This first line of the next message is our look-ahead. We can yield the previous message, and then reset our processing to begin with this first line.

logScanGenerator.py

```
def logScan( the_lines ):
    currentMessage = []
    for line in the_lines:
        if not line:
            continue
        if line[:4] == '2003':
            if currentMessage:
                # Parse currentMessage[0] to create a 5-tuple.
```

```
9                   # Parse date to create a 7-tuple.
10                  yield currentMessage
11                  currentMessage= []
12          currentMessage.append( line )
13      if currentMessage:
14          # Parse currentMessage[0] to create a 5-tuple.
15          # Parse date to create a 7-tuple.
16          yield currentMessage
```

2. We create an empty list into which we will accumulate a single message.
3. For this example, `the_lines` is a triple-quoted string into lines. For example, `log.splitlines()`.
4. We are filtering out blank lines. We simply skip over them, getting the next line from our source.
6. We are identifying header lines. Clearly, this is not going to work in general, since it is tied to this example file. When we look at the `re` module, we will see some far better techniques for identifying a header line.
7. When we find a header line, there are one of two conditions.
   - If we have data in *currentMessage*, we have looked ahead to the next message: we are done with the current message. We don't show how to break up the header into tuples, that's left as an exercise. We then yield the resulting list as a single message. When the **for** statement returns, we can use this look-ahead line to begin accumulating the next message.
   - Otherwise, if there is no data in *currentMessage*, we have found our first header and can begin to accumulate a message.
16. We yield the final accumulated message. There is a circumstance in which there will be no final message.

We've left off the details of parsing the first line and also parsing the timestamp. We can add these features later, once we get the basic log analyzer to work.

10.2.7 Generator Example: Roulette Spins

Let's look at a less geeky example of a generator. This will summarize some details of Roulette spins and yield the summaries for use.

Assume we have the a list of tuples that show number and color. This could be a record of actual spins in a casino. Or it could be created by a random number function.

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 evaluate this function repeatedly to get the sequence of gaps.

```
from __future__ import print_function, division

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
```

```
14          else:
15              count += 1
16      yield count
```

3. The *spins* variable defines our sample data. This might be an actual record of spins, or it could be created by another program.
8. We define our `countReds()` generator. This generator 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.
11. For black spins, however, we yield the length of the gap between the last black. This value is given to the **for** statement to be processed.

    When **for** statement's suite is done, it will resume the generator right after the **yield** statement: the count will be reset, and the **for** loop will advance to examine the next number in the sequence.
16. When the sequence is exhausted, we also yield the final count. This last gap counts may have to be discarded for certain kinds of statistical analysis because it doesn't represent an actual black spin.

This program shows how we use the `countReds()` generator function. In this case, we reduce the values by accumulating the total of all gaps and the number of gaps. We can then compute the average gap size.

```
total= 0
count= 0
for gap in countReds(spins):
    total += gap
    count += 1
print(count,"gaps")
print("average size",total/count)
```

Here's the results I got from running this script.

```
7 gaps
average size 0.857142857143
```

10.2.8 Generator Functions

These functions will generate a new iterable from an existing iterable, often a sequence.

`enumerate`(*iterable*) → iterator

Enumerate the items of a sequence, set or mapping. This yields a sequence of tuples based on the original iterable. Each of the tuples has two items: a sequence number and the item from the original iterable.

This kind of iterator is generally used with a **for** statement.

`sorted`(*iterable [,cmp] [,key] [,reverse]*) → iterator

This iterates through a iterable (sequence, set or mapping) in ascending or descending sorted order. Unlike a list's `sort()` method function, this does not update the list, but leaves it alone.

This kind of iterator is generally used with a **for** statement.

`reversed`(*iterable*) → iterator

This iterates through an iterable (sequence, set or mapping) in reverse order.

This kind of iterator generally used with a **for** statement. Here's an example:

```
>>> the_tuple = ( 9, 7, 3, 12 )
>>> for v in reversed( the_tuple ):
...     print(v)
...
12
3
7
9
```

zip(*sequence*, *...*) → sequence
This creates a new sequence of tuples. Each tuple in the new sequence has values taken from the input sequences.

```
>>> color = ( "red", "green", "blue" )
>>> level = ( 20, 30, 40 )
>>> zip( color, level )
[('red', 20), ('green', 30), ('blue', 40)]
```

10.2.9 Generator Exercises

1. **Finish The Log Analyzer**.

   Improve the log analyzer in *Geeky Generator Example: Web Server Logs* by adding a step to parse the header line, transforming the string into a 5-tuple. This will break out the date, severity, thread, username and "rest of the line" message as five separate strings and assemble a tuple from these strings.

   This problem isn't solved by writing yet another generator, it's basic string manipulation inserted into the generator example.

   Further process the date to break it into a 7-tuple.

   In *Text Processing and Pattern Matching : The re Module* we'll look at the `re` moodule; in *Time and Date Processing : The time and datetime Modules*, we'll look at other modules that can improve how these parsers work.

2. **Better Pattern Matching**.

   The log analyzer in *Geeky Generator Example: Web Server Logs* has a perfectly awful pattern-matching procedure. The code `line[:4] == "2003"` matches header lines against the year to see if it is a proper header line or not. This is a terrible dependency between program and data. This program will only work on logs created in 2003. Worse still, this code is repeated: when we change one, we may forget to change the other. What can we do to fix that?

   What we need is a function named `isHeader()` which examines a line to see if it has the proper punctuation to be a header. If it has the right pattern of digits, `- :` and `,`, the function returns `True`. Here's what we are looking for:

   ```
   2003-07-28 12:46:50,109
   ```

   Write a generator named `punctuation()` which will examine each character of a string, separating it into sequences of characters and individual punctuation marks. A simple version can separate the line into two kinds of tokens:

   - Punctuation marks (especially `-`, `:`, `,` and space).
   - Sequences of digits and sequences of letters.

   You'll be following the "look-ahead design" to accumulate sequences of digits and letters until your next character is a punctuation mark; then you'll yield the sequence of digits or letters you found. You can then yield the punctuation mark. Then you can reset your accumulator to be an empty string.

The pattern match can be a function that is something like the following:

```
def isHeader( aLine ):
    sequence = [ token for token in punctuation( aLine ) ]
    return (sequence[1] == '-' and sequence[3] == '-' and sequence[5] == ' '
    and sequence[7] == ':' and sequence[9] == ':' and sequence[11] == ',')
```

The final step is to rewrite the `logScan()` function to use `isHeader()` instead of `line[:4] == "2003"`.

In *Text Processing and Pattern Matching : The re Module* we'll look at the `re` moodule that can improve how this parser works.

3. **Two For One**.

   The pattern matching function, `isHeader()` broke down a header into individual data elements, including all of the punctuation marks and all of the words and digits. Once we recognized a header, we then parsed the header line a second time to break out the various fields. Can't we do both operations at once?

   Take a closer look at the `isHeader()` function. Does it have to return `True`, or can it return any value that is equivalent to true? What if it returned the nested tuples of `( (year, month, day, hour, minute, second, millisecond), severity, thread, user, string)` for headers?

   This would mean that the first 25 or so tokens would participate in this parsing. The remaining tokens were simply the text at the end of the message line. We can transform the tail end of the line from a sequence of strings to a single string with something line `"".join( line[25:] )`. This will begin with the 26th item (in position 25) and recreate the original string from this sequence.

   Rewrite `isHeader()` to return a proper tuple for headers and `False` for lines that can't be recognized as a header.

   The final step is to rewrite the `logScan()` function to use this revised `isHeader()` function.

4. **Improve Generator Efficiency**.

   In *Geeky Generator Example: Web Server Logs* we have a serious performance problem. Look at the `if currentMessage:` statement on the 7th line of the example. This condition is always true except the very first time through this loop. How can we avoid that needless condition checking?

   Rather than ask if we have found a valid header line, we should assure that we have already seen the valid header line. We'll need to change the initialization of our loop to set *currentMessage* to a valid message header instead of an empty list.

   We will need to rewrite this generator in three steps:

5. **Use the head-tail design and get an explicit iterator for individual lines**.

   In the example, we used the `splitline()` method of a string (or the `readline()` method of a file), and gave this sequence to a **for** statement. Instead of this, we need to get the iterator for the sequence of lines using the `iter()` function. Once we have the iterator, we can locate the first valid line.

6. **Find the first valid header and initialize currentMessage**.

   We are going to use the iterator's `next()` function to get lines, looking for one that matches the header line pattern. In the situation where we can't initialize *currentMessage*, we have an empty log file and we're done without yielding anything.

7. **Simplify the for loop**.

   Now that *currentMessage* is initialized, we can have a proper loop that examines the next line to see if it is a header. If so, yield the *currentMessage* knowing that it is a valid message, reset *currentMessage* to have the next header line. Otherwise, accumulate the line in the current message. The change to the **for** statement is minor, we use the iterator instead of the list of lines.

We can eliminate the `if currentMessage:` statement. This processing loop will now run considerably faster.

10.2.10 Iterator and Generator FAQ's

Why use iterators explicitly? The explicit use of iterators helps us separate a sequence into head and body. We have two approaches to separating header from body:

- We can use an **if** statement. Essentially, we ask "if this is the header" on every single line. This is a relatively simple filter, but it introduces a considerable amount of additional processing. After all, we knew which line was first, and we knew that every line after the first was not the first.
- We can use an iterator, process the header items initially, and then process the rest of the rows in a **for** statement. This can clarify the processing difference between the header and the rest of the data. However, it isn't a simple filter, reduce or map processing pattern. This is another pattern, called a partition.

What is the advantage of introducing this thing called a "generator?" The explicit use of generators helps us look ahead to the next element of a sequence. In effect, a generator is appropriate when our input has to be reduced from complex structures to simpler summaries. When a program reads complex structures, it reads the input in detail, summarizes (reducing the complexity), and then processes the summaries.

We use a divide and conquer strategy to divide the problem into the summarization part and the processing the summaries part. The summarization is handled by a generator which yields summaries. The summary processing can then be separated cleanly from the summarization.

In the Exceptions FAQ you said that exceptions were for rare events, but here you're using them for the in
The general statement still stands: exceptions are for rare or exceptional situations. A clutter of exceptions is more confusing than a well-planned set of ordinary **if** statements.

Generator functions *must* use an exception to end the processing loop. This is an example of the "out-of-band signal" design pattern. We need some kind of *signal* that isn't a piece of data. The alternative is to define some particular data value as an end-of-iteration sentinel. Doing this makes that sentinel value sacred, and limits our flexibility. Rather than pick a sentinel value that would, in effect, be an illegal value for any program, we raise an exception instead.

In the C programming language, the designers elected to use a particular ASCII character as the end-of-string sentinel value. One consequence of that is that files which contain that specific ASCII character cannot be processed easily by a C program. Some C-language programs discard that character, other C-language programs can't read files with that character.

CHAPTER

ELEVEN

MORE DATA COLLECTIONS

Sets and Mappings (called "Dictionaries")

First, we'll cover a simple un-ordered collection, called a set, in *Collecting Items : The set*.

In *Mappings : The dict*, we'll cover the mapping collection, called a dictionary. This type of collection *maps* a label (or key) to a value.

We can show how Python uses dictionaries and sequences to handle arbitrary lists of parameters to functions in *Defining More Flexible Functions with Mappings*.

11.1 Collecting Items : The `set`

A set is a collection of items without a defined order. We don't refer to items by their position in the sequence, since there's no order. We can test for presence or absence, however.

We'll look at what Python means by a Set in *What does Python mean by "Set"?*. We'll show how to create a Set in *How Do We Create a Set?*. We'll look at the various operations we can perform on a Set in *Operations We Can Perform On A Set*.

We'll look at the comparison operators for sets in *Comparing Sets: Subset and Superset*. There are a large number of method functions, which we'll look at in *Method Functions of Sets*. Some of the statements we've already seen interact with sets; we'll look at this in *Statements and Sets*. We'll look at some built-in functions in *Built-in Functions For Sets*. We'll present a moderately complex example in *Example of Using Sets*.

11.1.1 What does Python mean by "Set"?

A `set` is a collection of values. In a way, a `set` is the simplest possible collection. Recall that a `list` or `tuple` keeps the values in sequential order; items are identified by their position in the sequence. A `set`, on the other hand, doesn't keep the values in any particular order, and you don't identify the values by their position. A set, since it doesn't have sequential positions, can only have one instance of each distinct object.

This definition of set parallels the mathematical definition used in set theory. This means that we have operators available to determine the common elements ("intersection") between two sets, the union of two sets, and the differences between two sets.

A set is sometimes handy because we have to work with a collection of things where order doesn't matter, but we want to be sure to avoid duplicates. For example, if we are sending letters to families of children in school, each child contributes one family to the set. If we have siblings in the school, we don't want to include their family twice. This is the central idea behind accumulating a set: some elements may be mentioned more than once, but once is enough to be a member of a set.

Mutability. We have to look at two aspects of mutability. The items in a set must be immutable: strings, numbers and tuples. We can't easily create a set which contains a bunch of list objects.

Why not?

Let's say we had two lists:

```
list_one = [1, 2]
list_two = [1]
```

Assume, for the moment, that we could somehow create a set from these two lists.

What happens when we do this?

```
list_two.append( 2 )
```

Oops. Now we have two lists in the set which appear identical.

This – clearly – must be forbidden. That's one aspect of mutability: all items in a set must be immutable.

The second aspect of ummutability reflects sets themselves. There are two flavors of sets: `set` and `frozenset`. The ordinary `set` is mutable, in the same way that a list is mutable. A `frozenset`, on the other hand, is immutable, more like a tuple.

As with tuples, we can create a new, larger frozenset from the union of two other frozensets. The original sets doesn't change, but we can use them to create a new set.

11.1.2 How Do We Create a Set?

Sets are created using the `set()` or `frozenset()` factory functions. Unlike sequences, there's no way to write down a literal value for a `set`. We can make sets out of lists or tuples using the `set()` factory function.

set(*sequence*) → set

Creates a set from the items in *sequence*. If the sequence is omitted, an empty set is created.

Duplicates will be removed, for example, `set([1,1,2,3,5]) == set([1, 2, 3, 5])`.

Also, the original order may not be preserved.

frozenset(*sequence*) → set

Creates a set which can no longer be updated from the items in *sequence*. This set is immutable and can be used like a tuple.

Here are some examples of creating sets.

```
fib=set( [1,1,2,3,5,8,13] )
prime=set( [2,3,5,7,11,13] )
_= "now is the time for all good men to come to the aid of their party"
words=set( _.split() )
craps=set( [(1,1), (1,2), (2,1), (6,6)] )
```

fib This is a set of Fibonacci numbers. The value 1 is duplicated on the input sequence. The set can't have duplicates, so the resulting set value will be set([1,2,3,5,8,13]).

prime This is a set of prime numbers. There are no duplicates in the input sequence, so the set has the same number of elements.

words This is a set of distinct words extracted from the phrase. The `len(_.split())` is 16. Then `len(words)` is 14. If you check carefully, you'll see that the strings `'to'` and `'the'` are duplicated in the input sequence.

craps This is a set of pairs of dice. On the first roll of a Craps game, if the shooter rolls any of these combinations, totalling 2, 3 or 12, the game is over, and the shooter has lost. Each element in the set is a 2-tuple made up of the two individual dice.

Tip: Debugging `set()`

A common mistake is to do something like `set( 1, 2, 3 )`, which passes three values to the `set()` function. If you get a `TypeError: set expected at most 1 arguments, got n`, you didn't provide proper tuple to the set factory function.

Another interesting problem is the difference between `set( ("word",) )` and `set( "word" )`.

- The first example provides a 1-element sequence, `("word,")`, to `set()`, which becomes a 1-element set.
- The second example passes a 4-character string, `"word"`, which becomes a 4-element set.

In the case of creating sets from strings, there's no error message. The question is really "what did you mean?" Did you intend to put the entire string into the set? Or did you intend to break the string down to individual characters, and put each character into the set?

11.1.3 Operations We Can Perform On A Set

Sets have a large number of operators. Sets are widely-studied mathematical objects, and a number of those mathematical operations are defined in Python. There are four operations we can perform on sets: union (`|`), intersection (`&`), difference (`-`) and symmetric difference (`^`).

Important: But Wait!

You may recognize these operators (`|`, `&`, `-` and `^`) from *Special Ops : Binary Data and Operators*. These symbols *also* stand for operators that apply to individual bits in an integer value.

Remember that Python examines the objects on either side of the operator to see what type of data object they are. When you write an expression that involves two sets, Python will do the set operations. When presented two integers, Python will do the special binary operations.

The `|` operator. The `|` operator computes the union of two sets; it computes a new set which has all the elements from the two sets which are being unioned. In essence, an element is a member of '`s1 | s2`' if it is a member of s1 *or* a member of s2.

Here's the Venn diagram that uses shading to show the elements which are in the union of two sets.

Here are some examples.

```
>>> fib | prime
set([1, 2, 3, 5, 7, 8, 11, 13])
>>> fib | words
set([1, 2, 3, 5, 8, 'is', 'men', 13, 'good', 'aid', 'now', 'come', 'to', 'for', 'all', 'of', 'their', 'time', 'part
```

In the first example, we created a union the *fib* set and the *prime* set. In the second example, we computed a fairly silly union that includes the *fib* set and the *words* set; since one set has numbers and the other set has strings, it's not clear what we would do with this strange collection of unrelated things.

The union operator can also be written using method function notation.

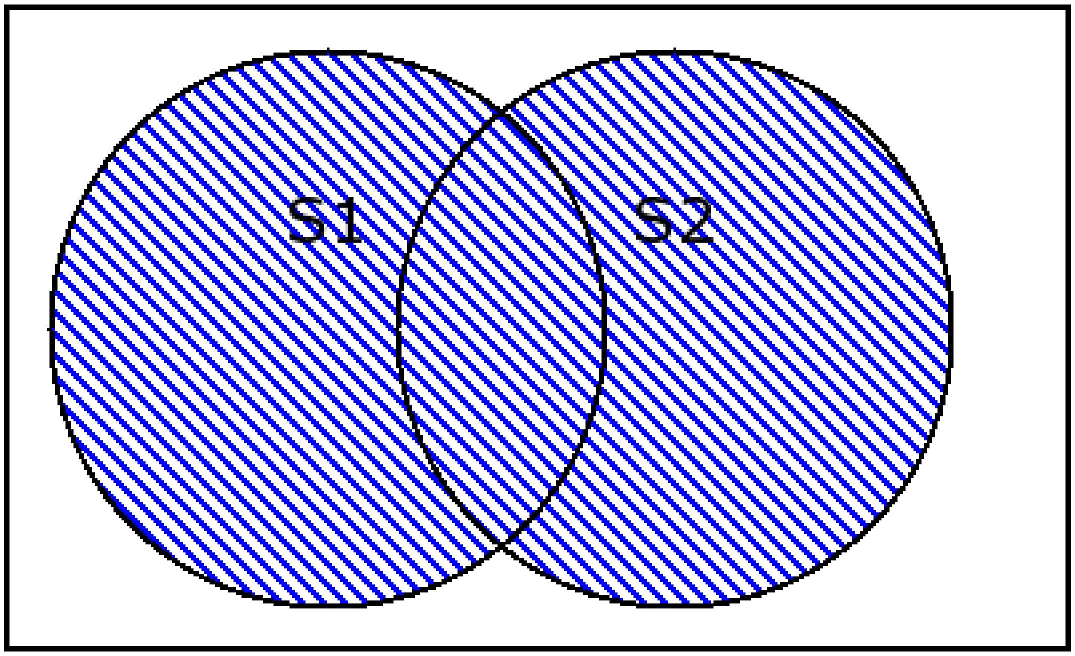

Figure 11.1: **Union of Sets, S1|S2**

```
>>> fib.union( prime )
set([1, 2, 3, 5, 7, 8, 11, 13])
>>> words.union( fib )
set([1, 'all', 'good', 5, 'for', 'to', 8, 'of', 'is', 'men', 2, 13, 'their', 3, 'time', 'party', 'the', 'now', 'com
```

Note that the two results of `fib | words` and `words.union(fib)` have the same elements in a different order. We can assure that this is true with something like the following:

```
>>> fib | words == words.union(fib)
True
>>> fib | words == words | fib
True
```

The above two expressions show us that the essential mathematical rules are true, even if the order of the elements is sometimes different.

The `&` operator. The `&` operator computes the intersection of two sets; it computes a new set which has only the elements which are common to the two sets which are being intersected. In essence, an element is a member of `s1 & s2` if it is a member of s1 *and* a member of s2.

Here's the Venn diagram that uses shading to show the elements which are in the intersection of two sets.

Here are some examples.

```
>>> fib & prime
set([2, 3, 5, 13])
>>> fib & words
set([])
```

In the first example, we created an intersection of the *fib* set and the *prime* set. In the second example, we computed a fairly silly intersection that shows that there are no common elements between the *fib* set and the *words* set.

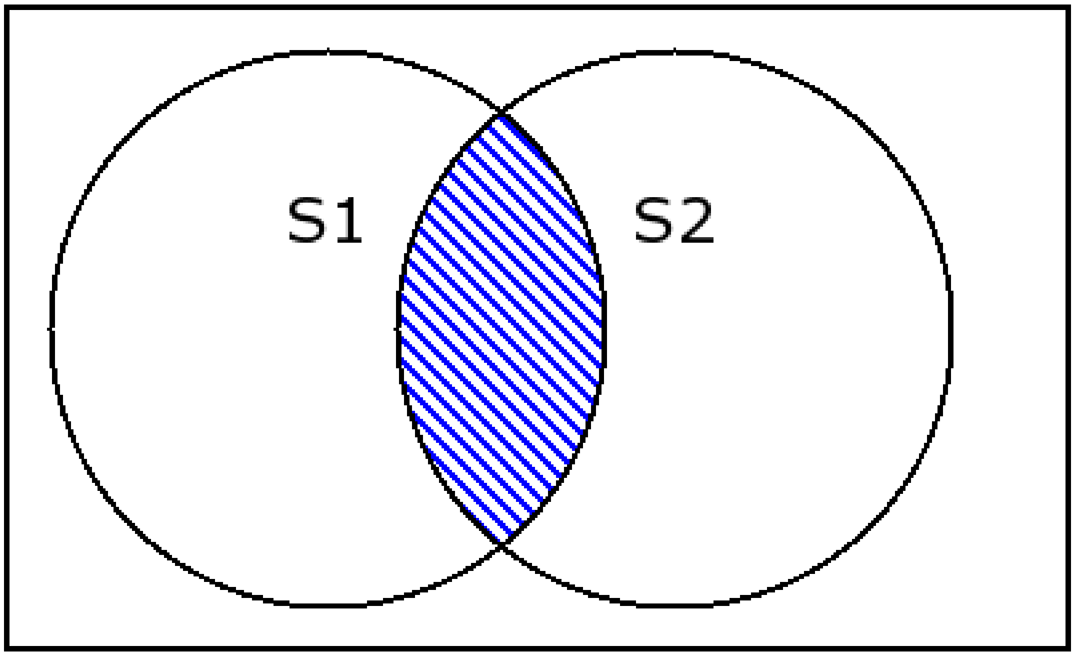

Figure 11.2: **Intersection Of Sets, S1&S2**

The intersection operator can also be written using method function notation.

```
>>> prime.intersection( fib )
set([2, 3, 5, 13])
>>> words.intersection( fib )
set([])
```

The `-` operator. The `-` operator computes the difference between two sets; it computes a new set which starts with elements from the left-hand set and then removes all the matching elements from the right-hand set. It fits well with the usual sense of subtraction. In essence, an element is a member of `s1 - s2` if it is a member of s1 *and not* a member of s2.

Here's the Venn diagram that uses shading to show the elements which are in the difference, `s1-s2`.

Here are some examples.

```
>>> fib-prime
set([8, 1])
>>> prime-fib
set([11, 7])
>>> fib-words
set([1, 2, 3, 5, 8, 13])
```

In the first example, we found the elements which are in the *fib* set, but not in the *prime* set. We can think of this as starting with the *fib* set and removing all the values that are in the *prime* set. In the second example, we found the elements which are in the *prime* set, but not in the *fib* set.

The third example shows the *fib* set with the word set removed. In this case, it's still the same *fib* set. We can prove this evaluating `fib-words == fib`.

The difference operator can also be written using method function notation.

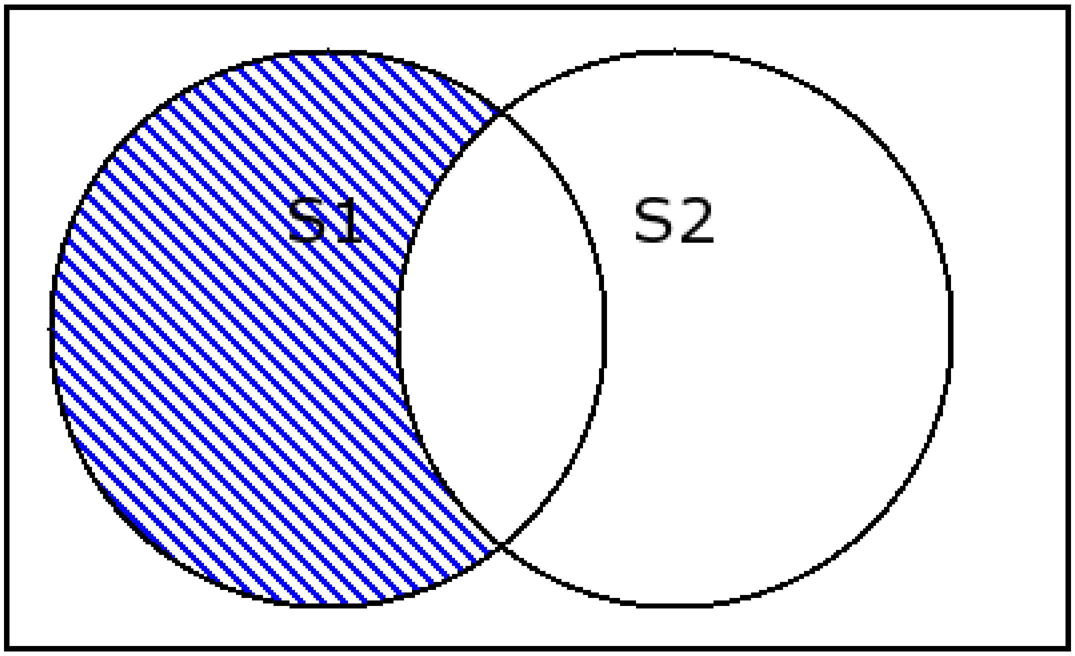

Figure 11.3: **Difference of Sets, S1-S2**

```
>>> prime.difference( fib )
set([11, 7])
>>> fib.difference( prime ) == fib-prime
True
```

The ^ operator. The ^ operator computes the "symmetric difference" between two sets; it computes a new set which elements that are in one or the other, but not both. Since a union is elements which are in one set or the other, and an intersection is elements which are in both, the symmetric difference of two sets is `(s1|s2)-(s1&s2)`. Rather than have to write this out, we have a pleasant short-hand operator.

Here's the Venn diagram that uses shading to show the elements which are in the symmetric difference of two sets.

Here are some examples.

```
>>> fib^prime
set([1, 7, 8, 11])
>>> fib^words
set([1, 'all', 'good', 5, 'for', 'to', 8, 'of', 'is', 'men', 2, 13, 'their', 3, 'time', 'party', 'the', 'now', 'com
```

In the first example, we found the elements which are in the *fib* set or the *prime* set, but not both. In effect, a union is computed and the common elements removed from that union. In the second example, we found the elements which are in the *fib* set or the *words* set, but not both. In this case, there are no common elements, so the symmetric difference is the same as the union.

The symmetric difference operator can also be written using method function notation.

```
>>> prime.symmetric_difference( fib )
set([1, 7, 8, 11])
>>> prime.symmetric_difference( fib ) == prime ^ fib
True
>>> prime ^ fib == (prime|fib)-(prime&fib)
```

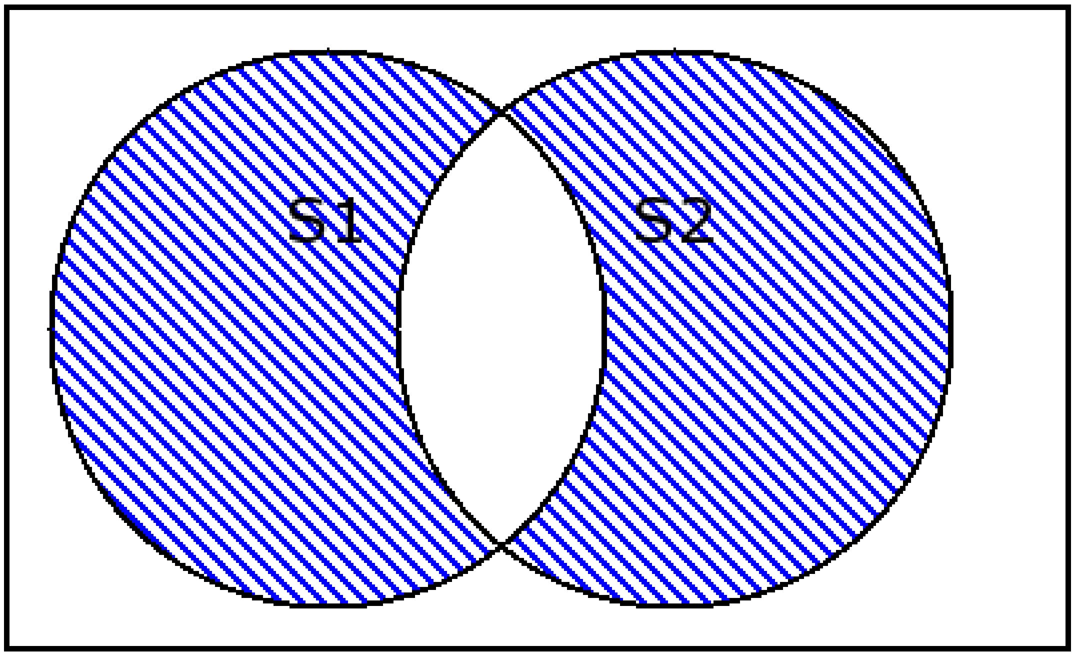

Figure 11.4: **Symmetric Difference, S1^S2**

```
True
```

11.1.4 Comparing Sets: Subset and Superset

Some of the standard comparisons (`<=`, `>=`, `==`, `!=`, `in` and `not in`) work with sets, but some of these operators have a meaning that's appropriate to sets. For tuples and strings, where the order of the elements matters, the collections are compared element by element. For sets, the order of the elements doesn't matter, so the comparisons have slightly different semantics.

The `in` and `not in` operators are the same as for other collections. They check to see if a given element is in the set or not in the set.

The following Venn diagram illustrates *s2* being a subset of *s1*.

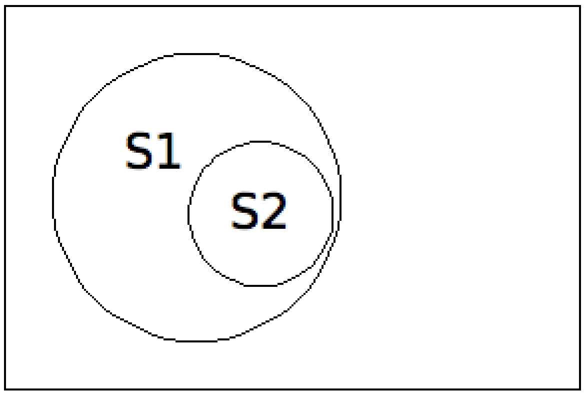

Figure 11.5: **Subset, S2 in S1**

The set comparisons are equality and subset comparisons. Therefore, `s1 <= s2` asks if set *s1* is a subset of *s2*. The `==` and `!=` operations do what you'd expect, comparing to see if the two sets have the same collection of elements.

```
>>> diff = prime & fib
>>> diff <= prime
True
>>> diff <= fib
True
```

In this example, we computed the intersection of prime and fib, which was the small set of numbers common to both sets, `set([2, 3, 5, 13])`. This set, by definition, has to be a subset of both of the original sets.

As with other set operators, we also have method function notation for these operations.

```
>>> "to" in words
True
>>> diff.issubset( prime )
True
>>> prime.issuperset( diff )
True
>>> (1,2) in craps
True
```

11.1.5 Method Functions of Sets

We've already seen a large number of method functions that apply to sets. These method functions all compute new sets from two existing sets. In addition to these, there methods functions to change the elements in a set. Finally, there are also method functions for updating a set based on another set.

We'll review the set operators first, since we've already seen them. We'll presume that we have two sets, *s1* and *s2*, for each of these functions.

Operators. These method functions are the same as the various set operators. They apply an operation between two sets and create a new set.

class set

`set.union`(*s2*) → set

Returns a new set which is the union of the distinct elements of *s1* and *s2*. This can also be written :s1|s2.

```
>>> set( [ "now", "is" ] ).union( set( [ "is", "the" ] ) )
set(['is', 'now', 'the'])
```

`set.intersection`(*s2*) → set

Returns a new set which is the intersection of the elements of *s1* and *s2*. This is only the common elements to both sets. This can also be written `s1&s2`.

```
>>> set( [ "now", "is" ] ).intersection( set( [ "is", "the" ] ) )
set(['is'])
```

`set.difference`(*s2*) → set

Returns a new set which has only the elements from *s1* that are not also elements of *s2*. The new set is effectively a copy of *s1* with elements from *s2* removed. This can also be written `s1-s2`.

```
>>> set( [ "now", "is" ] ).difference( set( [ "is", "the" ] ) )
set(['now'])
```

`set.symmetric_difference(`*s2*`)` → set

Returns a new set which has elements that are unique to *s1* and *s2*. The new set is effectively the union of *s1* and *s2* with the intersection elements removed. This can also be written as `s1^s2`.

```
>>> set( [ "now", "is" ] ).symmetric_difference( set( [ "is", "the" ] ) )
set(['now', 'the'])
```

Accessors. These method functions comparison operators. They apply a comparison between two sets and create a boolean value.

class set

`set.issubset(`*s2*`)` → boolean

Returns `True` if *s1* is a subset of *s2*. To be a subset, all elements of *s1* must be present in *s2*. This can also be written as `s1 <= s2`.

```
>>> set( [ "now", "is" ] ).issubset( set( [ "is", "now", "the" ] ) )
True
```

`set.issuperset(`*s2*`)` → boolean

Returns `True` if *s1* is a superset of *s2*. To be a superset, all elements of *s2* must be present in *s1*. This can also be written as `s1 >= s2`.

```
>>> set( [ "now", "is" ] ).issuperset( set( [ "is", "now", "the" ] ) )
False
```

Manipulators. This next group of methods manipulate a set by adding or removing individual elements. These operations do not apply to a `frozenset`.

class set

`set.add(`*object*`)`

Adds the given *object* to set *s1*. If the object did not previously exist in the set, it is added. If the object was already present in the set, the *s1* doesn't change.

```
>>> craps=set()
>>> craps.add( (1,1) )
>>> craps.add( (6,6) )
>>> craps.add( (1,2) )
>>> craps.add( (2,1) )
>>> craps
set([(1, 2), (1, 1), (2, 1), (6, 6)])
```

`set.remove(`*object*`)`

Removes the given *object* from the set *s1*. If the object did not exist in the set, an `KeyError` exception is raised.

```
>>> colors= set( [ "red", "black", "green" ] )
>>> colors.remove( "green" )
>>> colors
set(['black', 'red'])
```

`set.pop()` → object

Removes an object from set *s1*, and returns it. Since there is no defined ordering to a set, any object is eligible to be removed. If the set is already empty, a `KeyError` is raised.

```
>>> colors= set( [ "red", "black", "green" ] )
>>> while len(colors):
...     print(colors.pop())
...
green
```

```
black
red
>>> colors
set([])
```

set.clear()

Removes all objects from the set. After this method, the set is empty.

```
>>> colors= set( [ "red", "black", "green" ] )
>>> colors
set(['green', 'black', 'red'])
>>> colors.clear()
>>> colors
set([])
```

Updates. The following group of methods update a `set` using another `set` of elements. Each of these method functions parallels the operator method functions, shown above.

There is a significant difference, however. These methods actually mutate the set object to which they are attached. Each of these functions is available as an augmented assignment operator, which emphasizes the change to an set.

class set

set.update(*s2*)

Adds all the elements of set *s2* to set *s1*. This can also be written as `s1 |= s2`.

```
>>> two=set( [ (1,1) ] )
>>> three=set( [ (2,1), (1,2)] )
>>> twelve=set( [ (6,6) ] )
>>> craps=set()
>>> craps.update( two )
>>> craps.update( three )
>>> craps.update( twelve )
>>> craps
set([(1, 2), (1, 1), (2, 1), (6, 6)])
```

set.intersection_update(*s2*)

Updates *s1* so that it is the intersection of `s1&s2`. In effect, this removes elements from *s1* which are not also found in *s2*. This can also be written as `s1 &= s2`.

```
>>> ph1="now is the time for all good men to come to the aid of their party"
>>> words=set( ph1.split() )
>>> words
set(['party', 'all', 'good', 'for', 'their', 'of', 'is', 'men', 'to', 'time', 'aid', 'the', 'now', 'come'])
>>> ph2="the quick brown fox jumped over the lazy dog"
>>> words2=set( ph2.split() )
>>> words2
set(['brown', 'lazy', 'jumped', 'over', 'fox', 'dog', 'quick', 'the'])
>>> words.intersection_update(words2)
>>> words
set(['the'])
```

set.difference_update(*s2*)

Updates *s1* by removing all elements which are found in *s2*. This can also be written as `s1 -= s2`.

```
>>> ph1="now is the time for all good men to come to the aid of their party"
>>> words=set( ph1.split() )
>>> words
set(['party', 'all', 'for', 'their', 'of', 'time', 'aid', 'now', 'come'])
```

```
>>> ph2="to do good to men unthankful is to cast water into the sea"
>>> words2=set( ph2.split() )
>>> words2
set(['do', 'good', 'cast', 'is', 'men', 'the', 'water', 'to', 'sea', 'unthankful', 'into'])
>>> words.difference_update(words2)
>>> words
set(['party', 'all', 'for', 'their', 'of', 'time', 'aid', 'now', 'come'])
```

11.1.6 Statements and Sets

There are two statements that are associated with sets: the various kinds of **assignment** statements, and – because a set has an iterator – the **for** statement.

The Assignment Statements. We've seen basic **assignment** statement, and how it applies to `set`s. In the method functions section, we saw four augmented assignment statements, `|=`, `&=`, `-=` and `^=`. These parallel the augmented assignment statements we saw in *Assignment Combo Package*. These augmented assignment statements are used to modify a `set` by adding or removing elements.

Note that the augmented assignments statements only apply to a `set`. A `frozenset` can't be updated after it's created.

frozenset

A `frozenset` is a special kind of `set` that can't be updated. Generally, these are created once from a sequence or from another set. For some applications, there is no practical difference between a `frozenset` and a `set`. Your program can create a set, and never modify it again. However, in the next chapter, when we look at mappings, we'll see a situation when we will need to make a `frozenset` out of a `set` so that we can use it with a dictionary.

The for Statement. As with other collections, the **for** statement will step through each element of a `set`.

```
>>> fib=set( [1,1,2,3,5,8,13] )
>>> prime=set( [2,3,5,7,11,13] )
>>> for n in fib & prime:
...     print(n)
...
2
3
5
13
```

In this example, we've created a set, *fib*, of the first seven Fibonacci numbers. We also created a set, *prime*, of the first six prime numbers. Our **for** statement first computes the intersection of these two sets, then sets *n* to each value in that intersection.

11.1.7 Built-in Functions For Sets

A number of built-in functions create or deal with sets. The following functions apply to all collections, including sets.

`len`(*iterable*) → integer
  Return the number of items of a set, sequence or mapping.

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> len(craps)
4
```

max(*iterable*) → value
Returns the largest value in *sequence*.

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> max(craps)
(6, 6)
```

Recall that tuples are compared element-by-element. The tuple `(6, 6)` has a first element that is greater than all others.

min(*sequence*) → value
Returns the smallest value in *sequence*.

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> min(craps)
(1, 1)
```

Recall that tuples are compared element-by-element. The tuple `(1, 1)` has a first element that is less than all but one other tuple, `(1, 2)`. If the first elements are the same, then the second element is compared.

Iteration Functions. These functions are most commonly used with a **for** statement to process set items.

enumerate(*iterable*) → iterator
Enumerate the elements of a set, sequence or mapping. This yields a sequence of tuples based on the original set. Each of the result tuples has two elements: a sequence number and the item from the original set.

Note that sets do not have a defined ordering, so this can, in principle, yield the elements of the set in different orders. As a practical matter, the ordering doesn't spontaneously change. However, insertion or removal of an element may appear to change the enumerated set.

This is generally used with a **for** statement. Here's an example:

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> for position, roll in enumerate( craps ):
...     print( position, roll, sum(roll) )
...
0 (1, 2) 3
1 (1, 1) 2
2 (2, 1) 3
3 (6, 6) 12
```

sorted(*iterable [,key] [,reverse]*) → iterator
This iterates through an iterable object like a set in ascending or descending sorted order. Unlike the `sort()` method function of a list, this does not update the list, but leaves it alone.

This is often used with a **for** statement. It can also be used with the `list()` function to create an ordered list from a set.

Here's an example:

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> descending= list( sorted( craps, reverse=True ) )
>>> descending
[(6, 6), (2, 1), (1, 2), (1, 1)]
```

```
>>> craps
set([(1, 2), (1, 1), (2, 1), (6, 6)])
```

We've created an ordered list from the original set: *craps* is a set; *descending* is a list in descending order. Sets have no defined ordering, so creating a list from a set is the only way to impose a specific order on the elements.

Aggregation Functions. The following functions create an aggregate value from a set.

`sum(`*iterable*`)` → number
Sum the values in the iterable (set, sequence, mapping). All of the values must be numeric.

```
>>> odd_8 = set( range(1,8*2,2) )
>>> sum(odd_8)
64
>>> odd_8
set([1, 3, 5, 7, 9, 11, 13, 15])
```

`all(`*iterable*`)` → boolean
Return `True` if all values in the iterable (set, sequence, mapping) are equivalent to `True`.

The `all()` function is often used with Generator Expression, which is covered in *List Construction Shortcuts*.

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> hardways = set( (d1,d1) for d1 in range(1,7) )
>>> horn = hardways - craps
>>> horn
set([(3, 3), (4, 4), (5, 5), (2, 2)])
>>> all( 4 <= (d1+d2) <= 10 for d1,d2 in horn )
True
```

1. We created the set of craps rolls
2. We created the set of "hardways" rolls with a generator expression. The hard way is to roll a number with both dice equal.
3. The "horn" bets include the hardways which are not craps.
4. We evaluate :4 `<= (d1+d2)<= 10` for each roll using a generator expression. All the horn bets are between 4 and 10. [This isn't surprising, really. It's hard to find *simple* examples of `all()` or `any()`.]

`any(`*iterable*`)` → boolean
Return `True` if any value in the iterable (set, sequence, mapping) is equivalent to `True`.

The `any()` function is often used with Generator Expression, which is covered in *List Construction Shortcuts*.

```
>>> craps= set([(1, 2), (1, 1), (2, 1), (6, 6)])
>>> hardways = [ d1==d2 for d1,d2 in craps ]
>>> any(hardways)
True
>>> all(hardways)
False
```

1. We created the set of craps rolls.
2. We evaluated `d1==d2` in a generator expressions too see if the rolls are made "the hard way", that is, have both dice equal.

3.The `any()` function tells us that at least one element is `True`.

4.The `all()` function tells us that not all elements are `True`.

11.1.8 Example of Using Sets

Sets are all about membership and deciding if some value is in the set or out of the set. This, it turns out, is the essence of many of the basic rules for casino games. The random device (dice, wheel or cards) picks a value. Some set of bets are winners. If your bet is in that set, you'll get paid.

We'll break this example into two parts. The first part will show how to build some sets. Then we'll move on to use those sets.

set_example.py, Part 1

```
1  from __future__ import print_function
2  dice=set()
3  r1_win=set()
4  r1_lose=set()
5  point=set()
6  hardways=set()
7
8  r1_win.add( (5,6) )
9  r1_win.add( (6,5) )
10 r1_lose= set( [(1,1),(6,6),(2,1),(1,2)] )
11 for d1 in range(1,7):
12     r1_win.add( (d1,7-d1) )
13     for d2 in range(1,7):
14         dice.add( (d1,d2) )
15 hardways= set( [(2,2),(3,3),(4,4),(5,5)] )
16 point= dice-r1_win-r1_lose
17
18 print("winners", r1_win)
19 print("losers ", r1_lose)
20 print("points ", point)
21
22 assert hardways <= point
23 assert r1_win | r1_lose | point == dice
```

2. First, we create a number of empty sets that we'll use to examine throws of the dice in a Craps game. The *dice* set will contain the complete set of all 36 possible outcomes. The *r1_win* set will contain the different ways we can win on the first throw; it will have the various ways we can throw 7 or 11. The *r1_lose* set will contain the different ways we can lose on the first throw; it will have the various ways we can throw 2, 3 or 12. The *point* set is all of the remaining throws, which establish a point. Finally, the *hardways* set contains the various points on which the two dice are equal, rolling a value "the hard way".

7. We insert the two ways of rolling 11 into the *r1_win* set.

9. We insert the ways of rolling 2, 12, and 3 into the *r1_lose* set.

10. We've set *d1* to all values from 1 to-one-before 7. Therefore, the value of `(d1,7-d1)` will be one of the six ways to roll a 7. We add this to the *r1_win* set.

12. We've set *d1* to all the values from 1 to-one-before 7; independently, we've set *d2* to all values from 1 to 6. We put every combination of dice rolls into *dice*.

14. We create a set containing the four point rolls where the two dice are equal and assign this set to the variable *hardways*.
15. Finally, we take the complete set of dice, remove the roll 1 wins, remove the roll 1 losers, and assign this set to the variable *point*.

Note the two assertions that we make as part of our initialization:

- We assert that the dice rolls in *hardways* are a subset of the dice rolls in *points*. This is a matter of definition in Craps, and we need to be sure that the preceding statements actually accomplish this.
- We assert that the union of *r1_win*, *r1_lost* and *point* is the entire set of possible dice rolls. This, also, is a matter of definition, and we need be sure that our initialization procedure has established the proper conditions.

Once we've built some sets, we can now use the sets to evaluate some dice rolls. We can use this kind of dice-rolling experiment to evaluate a betting strategy.

set_example.py, Part 2

```
1  import random
2  for i in range(10):
3      d1=random.randrange(1,7)
4      d2=random.randrange(1,7)
5      roll= (d1,d2)
6      if roll in r1_win:
7          print(roll, "winner")
8      elif roll in r1_lose:
9          print(roll, "loser")
10     else:
11         if roll in hardways:
12             print(roll, "hard point")
13         else:
14             print(roll, "point")
```

1. We import the `random` module so that we can use the `randrange()` function to generate random die rolls.
5. After picking two numbers in the range of 1 to-one-before 7, we assemble the variable *roll* as the dice roll.
6. If *roll* is in the *r1_win* set, we have a winner on the first roll.
8. If *roll* is in the *r1_lose* set, we have a loser on the first roll.
11. Otherwise, we have a roll that has established a point. We can check for membership in the *hardways* set to see if it was one of the special ways to roll a 4, 6, 8 or 10.

11.1.9 Set Exercises

1. **Unique Words**.

   You can use Python's triple-quoted string to create a larger passage of text. You can split this into words, make the words lower-case, and then accumulate a set of distinct words in the text.

   Perhaps the hardest part of this is removing the punctuation. However, the list of punctuation marks is rather short, and you can generally replace all punctuation marks with spaces when doing simple kinds of analysis of English text.

Your program can start with something like the following:

```
text="""The next day being Sunday, the hands were turned up to divisions, and
the weather not being favourable, instead of the service the articles
of war were read with all due respect shown to the same, the captain,
officers, and crew, with their hats off in a mizzling rain. Jack, who
had been told by the captain that these articles of war were the rules
and regulations of the service, by which the captain, officers, and
men, were equally bound, listened to them as they were read by the
clerk with the greatest attention.  He little thought that there were
about five hundred orders from the Admiralty tacked on to them, which,
like the numerous codicils of some wills, contained the most important
matter, and to a certain degree make the will nugatory."""

clean=text.replace('.',' ').replace(',',' ').lower()
```

The rest of the program can split the text into individual words, create a `set` from those words and then display the unique words which occur in the paragraph.

Once you have that working, you can create a set of common English words, including “the”, a”, “to”, “of”, “in”, “on”, “by”, “as”, “and”, “or”, “not”, “be”, “make”, “do”, etc. The difference between your complete set of words and this set of common English words will be the unique or unusual words in the paragraph.

2. **Dice Rolls.**

   The game of Craps is defined around a large number of sets. The game has two parts: the first roll (usually called the “come out” roll, or “point off” roll), and the remaining rolls (or “point on” rolls) of the game.

   - On the point-off roll. There are first-roll winners (all the ways of rolling 7 or 11), first-roll losers (all the ways of rolling 2, 3 or 12). All remaining first-roll dice establish a point.
   - On the point-on rolls. There are losers (all the ways of rolling 7), winners (all the ways of rolling the point). All remaining rolls do not resolve the game.

   It’s very handy to have a list of sets. Each set in the list contains all the ways of rolling that number. We can create the empty list of sets as follows. This will give you a list, named *rolls*, that has empty sets in positions 2 through 12. It also has two empty sets in positions 0 and 1, but these won’t be used for anything.

```
rolls= []
for n in range(13):
    rolls.append( set() )
```

   Once you have the list named *rolls*, you can then enumerate all 36 dice combinations with a pair of nest loops like the following:

```
for d1 in range(1,7):
    for d2 in range(1,7):
        make a two-tuple (d1,d2)
        compute the sum, d1+d2
        add to the appropriate set in the rolls list
```

   Once you have the list of sets, you can compute sets which contains all the rolls for a win on the first roll and all the rolls which would lose on the first roll. These are simple union operations, using elements in the *rolls* list. Specifically, you’ll have to union `rolls[2]`, `rolls[3]` and `rolls[12]` for the first roll losers.

11.1.10 Set FAQ's

Sets are too mathematical and abstract; why are they in here? That's more of a complaint than a question. However, the point is that sets are useful and can simplify certain types of programs.

Also, and more importantly, it's important to see all of the various kinds of collections that Python offers. Most programming is about a collection of data. The more collections you've seen, the more you can exploit the various kinds of collections to build the program you need to write.

Many introductory books on programming will focus on a particular collection (often the list). This can leave the newbie to founder when it comes to doing things that don't fit well with the strengths of the list collection.

When would you ever need a `frozenset`? It isn't obvious at this point, but in the next chapter (*Mappings : The dict*), we'll uncover some reasons why Python has to have a frozenset. Looking forward a bit, the problem centers around mutability. A mutable object (like a list or a set) can't be used as a key for a dictionary.

Consider this: the dictionary key has to be a fixed label or tag for the element in the dictionary. Think of a word in the big old dictionary sitting on the corner of your desk. Words don't change their spelling. If we change the value of a set, it's now a new value; that's like changing the misspelling a word. Where is the new word in the old dictionary?

A frozenset can't be changed, and can be used as the key to a dictionary.

I can do all the set operations using just lists; why have the complexity of a set? Agreed, you can implement each set operation on a list. You'll note, however, that they're wordier than using the basic set operations.

11.2 Mappings : The `dict`

A mapping is an association (or "link") from one object to another. A real dictionary, for example, maps a defintion to a word. Note that this is a one-way association; you can't *directly* find the word from the definition; you can easily find the definition from the word.

We have to make a subtle distinction between the abstraction (a mapping) and the implementation (a dictionary). Most of the time, you'll use the only available mapping, the Python dictionary, `dict`. There are other mappings, however.

We'll look at what Python means by a Dictionary in *What Does Python Mean by "Dictionary"?*. We'll show how to create a Dictionary in *How We Create A Dictionary*. We'll look at the various operations we can perform on a Dictionary in *Operations We Can Perform On A Dictionary*.

We can't meaningfully compare two dictionaries; we'll look at this in *Comparing Dictionaries – Not A Good Idea*. There are a large number of method functions, which we'll look at in *Method Functions That Dictionaries Offer*. Some of the statements we've already seen interact with dictionaries; we'll look at this in *Statements and Dictionaries*. We'll look at some built-in functions in *Built-in Functions for Dictionaries*.

11.2.1 What Does Python Mean by "Dictionary"?

A `dict` maps a *key* to a *value*. In the big red-covered paper dictionary on my bookshelf, the key is a word, and the value is complex object, including a pronunciation guide, a definition, and an etymology.

Mappings are unidirectional, unordered collections. Each element is identified by a key. A dictionary represents a kind of space-time tradeoff.

Here's a depiction of a dictionary of 3 items. The keys are string color names, and the values are numeric color levels. This dictionary is one way to describe a nice midnight-blue. In his case, the keys are all strings and the values are all numbers.

```
>>> color= {"RED":51, "GREEN":0, "BLUE":153}
```

key	value
RED	51
GREEN	0
BLUE	153

We'll look at the features of a dictionary in some detail.

Unidirectional. A mapping is unidirectional, from key to value; you can't easily locate the key given the value. Also, note that each key occurs exactly once in the mapping. Values can occur more than once.

The mapping associates a value with a key. Locating a key from a value is not supported.

```
>>> color['RED']
51
>>> color[153]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 153
```

Unordered. A simple `dict` cannot preserve order. This is because it uses a *hashing* algorithm to identify a place in the `dict` for a given key. Every Python object must have a hash value: a simple distinct number. Objects, like strings or tuples, have hash values which summarize the string or tuple as a unique numeric value. The built-in function, `hash()` is used to do this calculation.

```
>>> color
{'BLUE': 153, 'GREEN': 0, 'RED': 51}
```

We created the dictionary in one order, Python shows us the dictionary in a different order.

Collection. A dictionary, like the sequences we looked at in *Basic Sequential Collections of Data*, is a kind of *collection* of objects. Since the dictionary as a whole is mutable, items can be inserted into the `dict`, found in the `dict` and removed from the `dict`.

```
>>> color['GREEN']= 16
>>> color
{'BLUE': 153, 'GREEN': 16, 'RED': 51}
```

A `dict` object has member methods that return a sequence of keys, a sequence of values, or a sequence of (*key* , *value*) tuples suitable for use in a **for** statement.

```
>>> color.keys()
['BLUE', 'GREEN', 'RED']
>>> color.items()
[('BLUE', 153), ('GREEN', 16), ('RED', 51)]
```

Immutable Keys. Above, we noted the mutability restriction on the key. The key object must compute a consistent *hash value*. This issue of consistency is important. If the key changes, how can we identify it in the dictionary?

For the immutable built-in types, the hash value is perfectly consistent: numbers, strings, tuples and frozensets are all good kinds of keys for a dictionary.

The mutable types – like lists, sets or dictionaries – present an obvious difficulty if their value should change.

However, you can "freeze" a `list` by making a `tuple` copy of it; similarly, you can freeze a `set` by making a `frozenset` copy of it.

Space vs. Time. We used the mathematical term "map" to define what a function does, back in *Adding New Verbs : The def Statement*. When we define a function, we write an algorithm which is, in effect, the mapping from the domain values to the range values. In a dictionary, Python explicitly stores a specific set of domain values and their associated range values.

A function like square root, for example, can map any positive floating-point number to that number's square root. We don't store all of the billions of possible floating-point numbers, instead the `math.sqrt()` function computes a mapping for each specific argument value using an algorithm. A function uses less storage, but is rather slow.

In the case of a dictionary, we can associate some specific floating-point numbers with their square roots. We don't have a completely general algorithm, just a list of keys and their associated values. This will use a considerable amount of storage, but will be very, very fast.

Other Mappings? We have to emphasize a 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 term mapping and dictionary are almost interchangeable.

In *Another Mapping : The defaultdict*, we'll look at another variety of mapping, called `collections.defaultdict`. This is a slightly different mapping. It's a dictionary, but with some extra features.

Python 3 will add the `collections.OrderedDict` class.

11.2.2 How We Create A Dictionary

A dictionary literal is created by surrounding a sequence of *key* : *value* pairs with *{ }*', and separating each *key* : *value* pair with `,`s. An empty dictionary is `{ }`.

Here are some examples. We'll describe each dictionary in detail, below.

```
wheel = { 0:"green", "00":"green",
 1:"red", 2:"black", 3:"red",
 4:"black", 5:"red", 6:"black" }
myBoat = { "NAME":"KaDiMa", "LOA":18,
 "SAILS":["main","jib","spinnaker"] }
theBets = { }
diceRoll = { (1,1): "snake eyes", (6,6): "box cars" }
```

wheel This dictionary has eight elements. Most of the elements have a number as their key; one of the elements has a string as the key. All the elements have a string as the value.

myBoat The *myBoat* dictionary has 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"]`.

theBets The *theBets* is an empty dictionary.

diceRoll The *diceRoll* variable is a dictionary 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"`.

Dictionary items and keys do not have to be the same type. Keys must be a type that can produce a hash value. Since lists, sets and dictionary objects are mutable, they are not permitted as keys. All other non-mutable types (especially strings and tuples) are legal keys.

Dictionary Factory Function. In addition to literal values, the following function also creates a dictionary object.

`dict`(*mapping*) → dictionary
: Creates a dictionary from the items in *mapping*. If the mapping is omitted, an empty dictionary is created.

`dict`(*sequence*) → dictionary
: Creates a dictionary from the items in *sequence*. Each item in the sequence must be a two-tuple with keys and values. For example,

```
dict( [ ('akey','the value'), ('key2','a value') ] )
```

`dict`(*param=value, ...*) → dictionary
: Creates a dictionary from the named parameters. Each parameter name becomes a key, and each parameter value becomes a value. For example

```
dict( akey='the value', key2='a value')
```

This only works when the keys are strings which satisfy the rules for Python variable names.

11.2.3 Operations We Can Perform On A Dictionary

Dictionaries have two operations: `[]` and `%`. The `[]` operator is similar to the other collection types, it is used to add, change or retrieve individual items from the dictionary.

The `[]` operator. The `[]` operator can identify a single value in the dictionary based on the key value. This operator does the key-to-value mapping for the dictionary. When we have a statement like `dictionary[key] = value`, we are updating the dictionary. When we use :`dictionary[key]` in an expression, we are looking something up in the dictionary.

Examples of dictionary operations.

```
>>> boat1={}
>>> boat1["NAME"]= "KaDiMa"
>>> boat1["SAILS"]= ["main","jib","spinnaker"]
>>> boat1["LOA"]= 15
>>> boat1["LOA"]= 18
>>> boat1
{'SAILS': ['main', 'jib', 'spinnaker'], 'LOA': 18, 'NAME': 'KaDiMa'}
>>> boat1["NAME"]
'KaDiMa'
>>> boat1["BEAM"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'BEAM'
```

This example starts by creating an empty dictionary, *boat1*. We provide a key of `"NAME"` and a value of `"KaDiMa"` , which updates the dictionary. We provide a key of `"SAILS"` and a value which is a list, `["main","jib","spinnaker"]`. We also set the value `15` for the key `"LOA"`; this turns out to be incorrect, so we replaced the `15` with an `18`.

When we ask for the value of *boat1*, the dictionary is displayed, showing the key:value pairs. Notice that the order does not correlate to the order in which we entered keys and values.

When we evaluate `boat1["NAME"]`, we see the value that is associated with this key.

When we evaluate `boat1["BEAM"]`, we see that any attempt to access a missing key gives us a `KeyError` exception.

Here are some other examples of picking elements out of a dictionary. In this case, we get the list value and use it in a **for** statement.

```
>>> for s in boat1["SAILS"]:
...     print(s)
...
main
jib
spinnaker
```

The % Operator. The string format operator works between string and dictionary. We prefer to use `str.format()`, however.

The string formatting method, `str.format()` can be applied to a dictionary as well as a sequence.

When this operator was introduced in *Sequences of Characters : str and Unicode*, the format specifications were applied to values from a sequence. We named the values by providing a simple position number. The format specification of `{0:d}` used the first value, the one at position zero. The format `{1:5.2f}` used the value at position one.

When we apply the format specifications to values that include dictionary, each format specification can include a dictionary key to pick a specific item from within the dictionary. We can use the position number along with the dictionary key in a syntax like `{0[LOA]:d}`. In this case item zero in the format arguments is expected to be a dictionary and the key value `"LOA"` will be the item that gets formatted.

For example:

```
>>> myBoat= { "NAME": "Red Ranger", "LOA": 42 }
>>> "{0[NAME]}, {0[LOA]:d} feet".format( myBoat )
'Red Ranger, 42 feet'
```

This will find `myBoat[NAME]` and use default formatting; it will find `myBoat[LOA]` and use `d` number formatting.

We can also provide a dictionary as the **only** argument to the format method. There are two ways to do this. We'll show more of this in *A Dictionary of Extra Keyword Values*.

The `dict()` function can build a dictionary from arguments where specific parameter names are provided. We can use '`key=value`' to build a dictionary when the key's are strings that follow Python variable name rules.

```
>>> dict( NAME="Red Ranger", LOA=42 )
{'LOA': 42, 'NAME': 'Red Ranger'}
```

The arguments to all functions follow this rule. Because the arguments to the `str.format()` method follow these standard rules, we can provide values in the form of a dictionary.

Consider this example:

```
>>> "{NAME}, {LOA:d} feet".format( NAME="Red Ranger", LOA=42 )
'Red Ranger, 42 feet'
```

The arguments to the `str.format()` method become a dictionary. The keys are used directly in the format strings.

11.2.4 Comparing Dictionaries – Not A Good Idea

Some of the standard comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`) don't have a lot of meaning between two dictionaries. There may be no common keys, nor even a common data type for keys or values. Since there is no real basis for comparison, dictionaries are simply compared by length. The dictionary with fewer elements is considered to be less than a dictionary with more elements.

The membership comparisons (**in**, **not in**) apply to the keys of a dictionary.

```
>>> diceRoll = { (1,1): "snake eyes", (6,6): "box cars" }
>>> (1,1) in diceRoll
True
>>> diceRoll[(1,1)]
'snake eyes'
>>> (2,1) in diceRoll
False
>>> diceRoll[(2,1)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: (2, 1)
>>> (2,1) not in diceRoll
True
```

If you want to work with the values, you have to use the `values()` method of the dictionary.

```
>>> wheel = { 0:"green", "00": "green",
 1:"red", 2:"black", 3:"red",
 4:"black", 5:"red", 6:"black" }
>>> "red" in wheel.values()
True
>>> "blue" in wheel.values()
False
```

11.2.5 Method Functions That Dictionaries Offer

A dictionary object has a number of method functions. These can be grouped arbitrarily into transformations, which change the dictionary, and accessors, which returns a fact about a dictionary.

Manipulators. The following manipulators make changes to a dictionary. With the exception of `setdefault()`, these methods do not return a value.

class dict

`dict.clear()`

Remove all items from the dictionary.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.clear()
>>> freq
{}
```

`dict.setdefault(`*key*, *value*`)` → object

Similar to `get()` and `d[key]`; get the item with the given key. However, this sets the supplied default in the dictionary, if the key did not exist. If no value is given for *default*, the value `None` is used.

```
>>> wheel= 18*['black']+18*['red']+2*['green']
>>> freq= { }
>>> color= random.choice(wheel)
>>> freq.setdefault( color, 0 )
```

```
0
>>> freq[color] += 1
>>> freq
{'red': 1}
```

`dict.update(new)`
Merge values from the new dictionary into the original dictionary, adding or replacing as needed. It is equivalent to the following Python statement. `for k in new.keys(): d[k]= new[k]`.

Accessors. The following accessors determine a fact about a dictionary and return that as a value.

class dict

`dict.copy()` → dictionary
Copy the dictionary to make a new dictionary. This is a *shallow copy*. All objects in the new dictionary are references to the objects in the original dictionary.

`dict.get(key[, default])` → object
Get the item with the given *key*, similar to `d[key]`. If the key is not present, supply *default* instead. If no value is given for *default*, the value `None` is used.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.get('red','N/A')
470
>>> freq.get('white','N/A')
'N/A'
>>> freq['red']
470
>>> freq['white']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'white'
```

`dict.has_key(key)`
If there is an entry in the dictionary with the given *key*, return `True`, otherwise return `False`.

This is usually written `key in dictionary`. The `dictionary.has_key( key )` form isn't used very often.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.has_key('red')
True
>>> freq.has_key('white')
False
>>> 'red' in freq
True
```

`dict.items()` → sequence
Return all of the items in the dictionary as a sequence of (*key* , *value*) tuples. Note that these are returned in no particular order.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.items()
[('black', 478), ('green', 52), ('red', 470)]
```

`dict.keys()` → sequence
Return all of the keys in the dictionary as a sequence of keys. Note that these are returned in no particular order.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.keys()
['black', 'green', 'red']
```

`dict.values()` → sequence
Return all the values from the dictionary as a sequence. Note that these are returned in no particular order.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.values()
[478, 52, 470]
```

11.2.6 Statements and Dictionaries

We can look at three statements and how they make use of dictionaries: the **for** statement and the **del** statement.

The for statement. The **for** statement uses an iterator to step through each value in a given sequence. A dictionary responds to the iterator protocol by iterating through the keys of a dictionary.

```
>>> monDict = { "JAN":1, "FEB":2, "MAR":3, "APR":4, "MAY":5, "JUN":6 }
>>> for d in monDict:
...     print(d, monDict[d])
...
MAR 3
FEB 2
APR 4
JUN 6
JAN 1
MAY 5
```

Notice that the keys are provided in no particular order. The dictionary is optimized for raw speed, and this means that they keys can be scrambled.

We can use the `sorted()` function to handle this.

```
>>> monDict = { "JAN":1, "FEB":2, "MAR":3, "APR":4, "MAY":5, "JUN":6 }
>>> for k in sorted(monDict):
...     print(k, monDict[k])
...
APR 4
FEB 2
JAN 1
JUN 6
MAR 3
MAY 5
```

Additional for techniques. Additionally, we can use several dictionary method functions to extract a sequence of values from the dictionary. We'll look at `items()`, `keys()` and `values()`.

Because of **for** statement works with multiple assignment, and the `items()` method function returns a sequence of tuples, we have a powerful technique for iterating through a dictionary. For example

```
>>> monDict = { "JAN":1, "FEB":2, "MAR":3, "APR":4, "MAY":5, "JUN":6 }
>>> for name, number in monDict.items():
...     print(number, name)
...
3 MAR
```

```
2 FEB
4 APR
6 JUN
1 JAN
5 MAY
```

The `items()` method of the dictionary named *monDict* returns a sequence with each entry transformed to a `(` *key* `,` *value* `)` tuple. The multiple assignment in the **for** statement assigns the keys to *name* and the values to *number* as it iterates through each element of the sequence. Note that the values returned bear little relationship to the order in which the dictionary was created.

The del statement. The **del** statement removes items from a dictionary. For example

```
>>> i = { "two":2, "three":3, "quatro":4 }
>>> del i["quatro"]
>>> i
{'two': 2, 'three': 3}
```

In this example, we removed a key (and it's associated value) from a dictionary by specifying which key we wanted removed.

11.2.7 Built-in Functions for Dictionaries

A number of built-in functions create or deal with dictionaries. The following functions apply to all collections, including dictionaries.

`len(`*iterable*`)` → integer

Return the number of items in the iterable (set, sequence or mapping).

```
>>> wheel = { 0:"green", "00": "green",
 1:"red", 2:"black", 3:"red",
 4:"black", 5:"red", 6:"black" }
>>> len(wheel)
8
```

`max(`*dictionary*`)` → value

Returns the greatest key in the dictionary.

```
>>> diceRoll = { (1,1): "snake eyes", (6,6): "box cars" }
>>> max(diceRoll)
(6, 6)
```

Since our keys are a variety of types (strings and ints), the `max()` comparison is somewhat unexpected.

`min(`*dictionary*`)` → value

Returns the least key in the dictionary.

```
>>> diceRoll = { (1,1): "snake eyes", (6,6): "box cars" }
>>> max(diceRoll)
(1, 1)
```

If you want to apply `max()` or `min()` to the values instead of the keys, you'll use the `values()` method. It would look like this.

```
>>> a = { 23:'skidoo', 7:'eleven', 4:'ever' }
>>> max(a)
23
>>> max(a.keys())
23
```

```
>>> max(a.values())
'skidoo'
```

Generally, functions like `sum()`, `any()` and `all()` don't make a lot of sense when applied to the keys of a dictionary. You often apply these to the values, however.

Iteration Functions. These functions are most commonly used with a **for** statement to process dictionary keys.

`enumerate`(*iterable*) → iterator

Enumerate the elements of a set, sequence or mapping. This yields a sequence of tuples based on the original tuple. Each of the result tuples has two elements: a sequence number and the key from the original dictionary.

Since dictionaries have no guaranteed ordering, this isn't completely sensible.

This is generally used with a **for** statement. Here's an example:

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> for position, color in enumerate(freq):
...     print(position, color, freq[color])
...
0 black 478
1 green 52
2 red 470
```

Note that the order as enumerated is not the order originally entered.

`sorted`(*iterable [,key] [,reverse]*) → iterator

This iterates through an iterable object like the keys of a mapping in ascending or descending sorted order. Unlike a list's `sort()` method function, this does not update the map, but leaves it alone.

This is generally used with a **for** statement. Here's an example:

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> for color in sorted( freq ):
...     print(color, freq[color])
...
black 478
green 52
red 470
```

Producing output sorted by value is a bit trickier. The keys must be unique, but the values don't have to be unique. That makes it impossible to determine which key belongs to a value.

What we do to report on a dictionary in value order is to use the list of tuples representation produced by the `items()` method.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> freq.items()
[('black', 478), ('green', 52), ('red', 470)]
>>> def by_freq( freq_pair ): return freq_pair[1]
...
>>> sorted( freq.items(), key=by_freq )
[('green', 52), ('red', 470), ('black', 478)]
```

`reversed`(*iterable*) → iterator

This iterates through an iterable (set, sequence, mapping) in reverse order.

Since dictionaries have no guaranteed ordering, this isn't completely sensible.

This is generally used with a **for** statement.

Aggregation Functions. These functions reduce a dictionary to a single aggregate value.

`sum(`*iterable*`)` → number

Sum the values in the iterable (set, sequence, mapping). All of the values must be numeric.

When applied to a mapping, this will sum the keys. More commonly, we want to sum the values.

```
>>> freq= { 'red':470, 'green':52, 'black':478 }
>>> sum( freq.values() )
1000
```

`all(`*iterable*`)` → boolean

Return `True` if all values in the iterable (set, sequence, mapping) are equivalent to `True`.

When applied to a mapping, this will test the keys. More often we will use a Generator Expression, which allows us to apply the *all* test to the values.

```
>>> myBoat = { "NAME":"KaDiMa", "LOA":18, "HULL":"mono",
...      "SAILS":["main","jib","spinnaker"] }
>>> all( v is not None for v in myBoat.values() )
True
```

`any(`*iterable*`)` → boolean

Return `True` if any value in the iterable (set, sequence, mapping) is equivalent to `True`.

When applied to a mapping, this will test the keys. More often we will use a Generator Expression, which allows us to apply the *any* test to the values.

```
>>> fireSail = { "NAME":None, "LOA":16, "HULL":"catamaran",
...      "SAILS":["main","jib"] }
>>> any( v is None for v in fireSail.values() )
True
```

11.2.8 Dictionary Exercises

1. **Word Frequencies**.

   A string can be split into individual words using the string's `split()` method. A dictionary can be used to accumulate the list of words and their frequency.

   By default, a string's `split()` method will break up the string on the spaces, giving us a sequence of individual words. Each word will have attached punctuation marks, something that is difficult to process without more powerful tools. For now, we'll tolerate the punctuation at the end of some words.

   ```
   import string

   myText= """Call me Ishmael.  Some years ago -- never mind how long
   precisely -- having little or no money in my purse, and nothing
   particular to interest me on shore, I thought I would sail about a
   little and see the watery part of the world."""
   words= myText.split()
   ```

   Iterate through this sequence, placing each word into a dictionary. The first time a word is seen, the frequency should be set to 1. Each time the word is seen again, increment the frequency. The final dictionary will be a frequency table.

   To alphabetize the frequency table, extract just the keys. A sequence can be sorted (see *Flexible Sequences : The list*). This sorted sequence of keys can be used to extract the counts from the dictionary.

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 dictionary 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. Use the full company names in *stockDict* to look up the full company name. This is the basic relational database join algorithm between two tables.

   The outline of processing looks like this: $shares \times price$

   ```
   for s in purchases:

       look up s[0] in  stockDict

       compute

       print a nice-looking line
   ```

   Create a second purchase summary that which accumulates total investment by ticker symbol. In the above sample data, there are two blocks of GE stock. These can be combined by creating a dictionary where the key is the ticker and the value is a list of blocks that have a common ticker symbol.

   The outline of the processing looks like this: `blocks[symbol]blocks[symbol]`

   ```
   blocks = {}
   for s in purchases:
       symbol= s[0]
       if  symbol in blocks:
           Append this block to the list
       else:
           Create a 1-element list in
   ```

   A pass through the resulting dictionary can then create a report showing each ticker symbol and all blocks of stock. The outline of the processing looks like this: $shares \times price$

   ```
   for symbol,blockList in blocks.items():
       totalValue= 0
       totalShares= 0
       for s in blockList:
           compute value as
           accumulate value in totalValue
           accumulate shares in totalShares
       print a nice-looking line showing totals
   ```

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 dictionary 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 simple 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 dictionary of all combinations, using the sum of the two dice as the key.

   Each value in the dictionary 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 = d[ d1+d2 ]
        newList = oldList + newTuple
        d[ d1+d2 ] = newList

Loop over all values in the dictionary
    print the key and the length of the list
```

11.2.9 Mapping FAQ's

If there's only one kind of mapping, the dictionary, why make a distinction between the two terms? Currently, there's only one *built-in* mapping, which is the dictionary. However, proposals are regularly floated around to add the "ordered dictionary" as a second kind of mapping. This other kind of mapping would use the Red-Black Tree algorithms to create a kind of mapping which would be somewhat slower than the hashed dictionary we have now, but would guarantee that they keys were always kept in order.

Also, in *Another Mapping : The defaultdict*, we'll look at another kind of dictionary, the default dictionary. It's not built-in, but it's widely used.

The word "map" seems to have a lot of meanings. If you're used to a map being a piece of paper that depicts land-masses, then I suppose this new use of map may be unusual. However, we're using map in the sense of route or path. If you think of someone mapping out their future, they are creating a route from where they are to where they want to be.

Mathematicians use the word map in this sense of association between two objects. They often use these term when defining a function which maps values between the domain and the range. Python folks borrowed this definition of map to talk about how a dictionary maps a key to a value. Further, as a kind of loop design, the mapping is very common

Indeed, much of programming involves associations between values and transformations from one representation of the value (a string, for example) to another representation (like a number). We lump all of this under "mapping" because we may implement it in the style of functions or in the style of a dictionary. It could be an algorithm which computes the range value from the domain value(s). It could, on the other hand, be a data structure that simply provides the domain value associated with the range value.

What if I don't have a single key for my dictionary? Let's say I want a phone book with last names and phone numbers as keys? What do I do then?

Great question. The database designers call this a *secondary index*. We want to have two different sets of keys for the same individual phone book objects. We can do this by creating a second dictionary with our alternate key.

Let's assume we have a list of tuples that looks like the following.

```
names= [ ( "last name", "first name", "phone number" ),
  ( "Howard", "Moe", "555-1111" ),
  ( "Howard", "Shemp", "555-2222" ),
  ( "Fine", "Larry", "555-3333" ), ]
```

We might want to turn this into two dictionaries doing something like the following. This will decompose the list into the individual name tuples, and assign each tuple to *nameTuple*. We can associate this tuple object in two different dictionaries. In this example, we'll assign the tuple to *byName* and *byPhone*.

```
byName = dict()
byPhone= dict()
for nameTuple in names:
    ln, fn, ph = nameTuple
    byName[ln]= nameTuple
    byPhone[ph]= nameTuple
```

11.3 Defining More Flexible Functions with Mappings

We are now in a position to show how Python uses both dictionaries and sequences to be very flexible in handling argument values to functions.

In *A List of Extra Positional Values* we'll look at some additional features of positional parameters. In *A Dictionary of Extra Keyword Values* we'll look at keyword parameters. Finally, in *Dictionary Use Under the Hood* we can provide some hints at additional internal uses which Python makes of dictionaries.

11.3.1 A List of Extra Positional Values

In *Flexible Definitions with Optional Parameters*, we hinted that Python functions can handle a variable number of parameters. At the time, we talked about optional vs. required parameters. This allowed *some* variability, but the complete set of parameters was still rigidly specified by the function definition.

We're now ready to look at functions that can work with an indefinite number of argument values. These functions do not have a rigid specification that provides a parameter name for every argument value. Examples of built-in functions that can process an indefinite number of argument values are `max()` and `min()`.

Recall that Python matches the actual argument values with the parameter names, using the following rules.

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`.

A Quick Example. Here's an example of a function with two parameters that permits some variability.

```
def roll( dice=2, sides=6 ):
    return [ random.randrange(1,sides+1) for i in xrange(dice) ]
```

We have four ways that we can use this function using positional parameters: `roll()`, `roll(5)`, `roll(1,8)`, `roll(4,12)`. These calls will roll two standard dice, five standard dice, one eight-sided dice and four 12-sided dice.

When confronted with additional positional argument values to a function, Python must raise a `TypeError` exception. While this makes sense, we can see that there are alternatives. For example, Python could silently ignore the extra values. This alternative is unacceptable, because it would give us no warning of making common mistake. We'll look at another, acceptable alternative below.

Here's an example of misusing our `roll()` function. We provided too many values and got a `TypeError` exception.

```
>>> roll(4,8,12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: roll() takes at most 2 arguments (3 given)
```

Excess Positional Argument Values. Python gives us a way to define a function that will collect all the "extra" argument values into a tuple instead of raising an exception. This allows a function to work with an indefinite number of argument values, the way `max()` and `min()` do.

If you want a collection of positional argument values in a tuple, you provide a parameter of the form `*`*extras*. Your variable, here called *extras*, will receive a sequence with all of the extra positional arguments. The `*` is part of the syntax and tells Python that this parameter gets all of the unexpected positional argument values.

The myMax Function. The following function accepts all of the positional arguments in a single parameter, *args*. This parameter will be a tuple with all of the argument values.

```
def myMax( *args ):
    max= args[0]
    for a in args[1:]:
        if a > max: max= a
    return max
```

The `*args` parameter specifies that a tuple of all arguments is assigned to the parameter variable *args*. We take the first of these values (`args[0]`) as our current guess at the maximum value, *max*. We use a **for** loop that will set *a* to each of the other arguments, computed as a slice of *args* starting with the second element.

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.

We can use `myMax()` the same way we use the built-in `max()`.

```
>>> myMax(4,8,12)
12
```

Bonus Questions. What happens when we ask for `myMax()`? Is this sensible? What does the built-in `max()` do?

11.3.2 Setting Parameters From A List

Above, we saw how we can have Python collect argument values into a single variable. This will have a tuple of all the arguments that didn't match a parameter variable.

This can be used in the other direction, also. We can provide a sequence which assigns values to a number of parameters.

Consider this function:

```
def add3( a, b, c ):
    return a + b + c
```

Python lets us prefix a sequence object with `*` to indicate that this object should supply the rest of the positional parameters with values.

We can evaluate this `add3()` function using the `*` notation to assign parameters from a sequence.

```
>>> add3( 9, 3, 2 )
14
>>> some_sequence = [ 9, 3, 2 ]
>>> add3( *some_sequence )
14
```

Of course, we can combine the last two lines into something like this.

```
>>> add3( *[ 9, 3, 2 ] )
14
```

A printf Function. Here's another example of a function with one fixed parameter and an unlimited number of additional positionl parameters.

We must have the fixed parameter first. It must be followed by all the extra arguments.

```
def printf( format_string, *vals ):
    print( format_string.format( *vals ) )
```

Here, we've defined the first parameter as to be *format_string*. Any other argument values will be collected into a parameter called *vals*.

Notice that this is another example of the "head-tail" pattern that we noted when talking about iterators. In this case, we have one positional parameter at the head and the remaining positional parameters are the tail.

We provided the *vals* sequence of argument values to the `str.format()` method function.

```
>>> printf( "spin: {0:d} {1}", 1, "red" )
spin: 1 red

>>> printf( "{0:d} rolls, ending roll: {1:d} {2:d}", rolls, dice[0], dice[1] )
23 rolls, ending roll: 3 4
```

11.3.3 A Dictionary of Extra Keyword Values

To add clarity to a function evaluation, we can provide argument values to a function using the parameter names. These are called keyword arguments. We can mix keyword and positional arguments when we evaluate a function. The only restriction is that we have to provide all values for the positional parameters first. This rule makes it easy to match up argument values with parameters based on their position in the function definition.

Here's a function with three parameters.

```
def diceRolls( rolls, dice=2, sides=6 ):
    r= []
    for i in xrange(rolls):
            r.append( [random.randrange(1,sides+1) for d in xrange(dice)] )
    return r
```

We can evaluate this function a number of ways. Two optional parameters gives us four combinations of forms. We can use keywords for each argument, also, giving us a total of 14 different forms for using this function. We won't enumerate them; we'll only show a few.

The first example uses positional arguments and default values.

The next examples uses a mixture of positional and keyword arguments.

The final example shows all keyword arguments.

```
>>> diceRolls(5)
[[5, 2], [1, 4], [6, 6], [4, 5], [6, 5]]
>>> diceRolls(5,sides=8)
[[3, 7], [3, 4], [4, 3], [6, 6], [1, 7]]
>>> diceRolls(dice=5,rolls=3)
[[1, 5, 6, 5, 5], [1, 1, 4, 3, 2], [3, 2, 6, 3, 6]]
```

Here's what happens if we try to break the rules. The first example shows what happens when we don't provide a value for a required parameter. The last example show what Python does with an unexpected keyword.

```
>>> diceRolls(dice=2,sides=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: diceRolls() takes at least 1 non-keyword argument (0 given)

>>> diceRolls(3,dice=5,label="yacht game")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: diceRolls() got an unexpected keyword argument 'label'
```

When confronted with additional keyword arguments, Python must raise a `TypeError` exception. While this makes sense, we can see that there are alternatives. For example, Python *could* silently ignore the extra keywords. This is unacceptable, because it would give us no warning of making common mistake. We'll look at another alternative below.

Excess Keyword Argument Values. Python gives us a way to define a function that will collect all the "extra" keyword argument values into a dictionary instead of raising an exception. This allows a function to work with an indefinite number of argument values.

If you want the extra keyword arguments collected into a dictionary, you provide a parameter of the form ****extras**. Your variable, here called *extras*, will receive a dictionary with all of the extra keywords and their argument values. The ****** is part of the syntax and tells Python that this parameter gets all of the unexpected keyword arguments.

Rate – Time – Distance. The following function accepts an arbitrary number of keyword arguments in a single parameter, named *args*.

```
def rtd( **args ):
    if "rate" in args and "time" in args:
        args["distance"]= args["rate"]*args["time"]
    if "rate" in args and "distance" in args:
        args["time"]= args["distance"]/args["rate"]
    if "time" in args and "distance" in args:
        args["rate"]= args["distance"]/args["time"]
    return args
```

We defined this function to accept an arbitrary number of keyword arguments. These are collected into a dictionary, named *args*. We identify the combination of "rate", "time" and "distance" by checking for a given key in the dictionary. For each combination, we can solve for the remaining value and update the dictionary by insert the additional key and value into the dictionary.

This function returns a small dictionary with the missing value computed from the other two values. If for some reason it cannot compute a new value from the input keyword arguments, it returns the original arguments dictionary. Another possibility for this situation is to raise an exception indicating that the problem "does not compute."

Here's two examples of using this `rtd()` function.

```
>>> print(rtd(rate=60, time=45/60))
{'distance': 45.0, 'rate': 60.0, 'time': 0.75}
>>> print(rtd(distance=173, time=2+50/60))
{'distance': 173, 'rate': 61.058823529411761, 'time': 2.8333333333333335}
```

The first one computes the distance when traveling 60 MPH for 45 minutes.

The second shows the average speed when going 173 miles in 2 hours and 50 minutes.

11.3.4 Setting Parameters From a Dictionary

Above, we saw how we can have Python collect keyword argument values into a single variable. This will have a dictionary of all the keyword arguments that didn't match a parameter variable.

This can be used in the other direction, also. We can provide a dictionary which assigns values to a number of parameters.

Consider this function:

```
def almost( a, b, eps ):
    return abs(a-b)/a <= eps
```

Python lets us prefix a dictionary object with `**` to indicate that this object should supply the rest of the positional parameters with values.

We can evaluate this `almost()` function using the `**` notation to assign parameters from a sequence.

```
>>> almost( 355/113, math.pi, 0.0000001 )
True
>>> keywords = {'a':355/113, 'b':math.pi, 'eps':0.0000001}
>>> almost( **keywords )
True
```

This flexibility allows us to create the arguments for a function in a variety of ways.

11.3.5 The Default Value Restriction

This is a topic left over from *Flexibility and Clarity : Optional Parameters, Keyword Arguments*. We introduced the concept briefly, but had to defer the details until we covered mutable types of objects, specifically list, set and map.

When defining a function, we can provide default values for the parameters. You can look at the rules in *Flexible Definitions with Optional Parameters*.

It's very important to note that you should not provide a mutable object as a default value. What will happen is that each time the function is evaluated and a default value is used, the same mutable object will be reused.

What not to do. Here's an example of a poorly-done function definition. Unwisely, it uses a mutable object as a default value.

```
import random
def createRolls( aList=[] ):
    for i in range(1000):
        roll = random.randint(1,6), random.randint(1,6)
        aList.append(roll)
    return aList
```

We've defined function that ca be used two ways. Here's one use case: we're providing a list that we want to have updated with a sequence of dice rolls.

```
>>> myRolls= []
>>> c= createRolls(myRolls)
>>> len(myRolls)
1000
>>> len(c)
1000
>>> c is myRolls
True
```

The above looks about right. The `createRolls()` created 1000 dice rolls and appended them to the given list, *myRolls*.

Here's what happens when we use the mutable default value.

```
>>> a= createRolls()
>>> len(a)
1000
>>> b= createRolls()
>>> len(b)
2000
>>> len(a)
2000
>>> a is b
True
```

What happened?

The first time we evaluated `createRolls()`, we used a default value for *aList*. This list was updated with 1000 dice rolls.

The second time, we also used a default value. Since it was the same default value, we updated the same list object with another 1000 dice rolls.

It turns out that there's only one mutable list object that's part of the definition of `createRolls()`.

Mutable Default Values. If you must have a mutable object as a default value, you'll need to do something like this.

```
import random
def createRolls( aList=None ):
    if aList is None: aList= []
    for i in range(1000):
        roll = random.randint(1,6), random.randint(1,6)
        aList.append(roll)
    return aList
```

The `if aList is None: aList= []` will create a fresh, new empty list when no argument value is provided. Each time the function is evaluated, it won't be reusing the single default value.

11.3.6 Dictionary Use Under the Hood

Python uses dictionaries internally for a variety of purposes. All variables, functions, modules and classes are actually kept in an internal dictionaries. The keys used are the variable name, function name, module name or class name. A statement like `a=2` creates an entry for *a* in an internal dictionary of local variables. It has the same effect as `locals()["a"]= 2`.

As each function executes, the function uses a `locals()` dictionary that is private to that function. The overall execution environment has the `globals()` dictionary.

When we import a module, for example, import processing will create a local dictionary in which the module's functions and variables are declared. That's why we have to say *module* . *function*. This tells Python which local dictionary will contain the expected name.

Looking forward to *Defining New Objects*, we note that a class is a dictionary of class variables and methods. Class attribute references are translated to lookups in this dictionary, e.g. *C.x* is translated to `C.__dict__["x"]`. An object contains a dictionary of instance variables created during object initialization, as well as a reference to the class that defines the method functions.

11.3.7 Flexible Function Exercises

1. **Sum**.

   The sum function is an example of a function that takes an indefinite number of arguments. You can refer to the definition for sigma in *Translating From Math To Python: Conjugating The Verb "To Sigma"* for more information.

   Write a sum function which uses the definition `def sum( *values ):`. This will return the sum of the tuple of values.

2. **Mean**.

   The mean function is an example of a function that takes an indefinite number of arguments. You can refer to the definition for mean in *Translating From Math To Python: Conjugating The Verb "To Sigma"* for more information.

   Write a mean function which uses the definition `def mean( *values ):`. This will return the mean of the tuple of values. This can rely on the `sum()` and the built-in `len()`.

3. **Temperature**.

   We can merge the various Centigrade and Fahrenheit conversion functions into a single function that depends on keyword arguments. You can refer to the definition these conversions in *Expression Exercises* for more information.

   Write a temperature function which uses the definition `def temperature( **values ):`. When called with `temperature( f=65 )`, convert to Celsius. This can be seen when the *values* dictionary has a key of `"F"` or `"f"`.

   When called with `temperature( c=65 )`, convert to Fahrenheit. This can be seen when the *values* dictionary has a key of `"C"` or `"c"`.

   If the dictionary also has `"wind"`, you can fold in a wind-chill calculation. Note that the wind-chill formula in *Expression Exercises* is for Fahrenheit only. If someone asks for `temperature( f=12, wind=15 )`, then you'll do wind-chill first in Fahrenheit, then convert to Celsius. If someone asks for `temperature( c=-5, wind=15 )`, then you'll convert to Fahrenheit first, then compute wind-chill in Fahrenheit.

11.3.8 Advanced Feature FAQ's

Why are there so many ways to pass parameters to functions? There are really two forces at work here: flexibility and clarity. Let's examine some tiers of complexity and show how we improve flexibility and clarity as our needs become more complex.

1. A function with no parameters (like `random.random()`) doesn't present any great challenge to make it either flexible or clear.
2. A function with one or two parameters (like most of the functions in `math`) are often easy to explain and document. The math functions like square root are simple enough that we don't need more explanation or help.
3. A function with a more than two parameters can be hard to understand. To clarify a function's arguments, using keyword parameters is a big help. For example, the `datetime` module's `date` might be used like this: `datetime.date( month=11, day=27, year=2005 )`.
4. We often have functions that differ in a single assumption. Rather than create two versions of a function to reflect these different assumptions, we might want to create a single function with an optional parameter. For example, the `int()` assumes base 10 when converting a string to an integer. `int('21')` is the number 21. However, when the string of digits is from another base, we can say `int('21',16)`. This is a handy short-cut: we only have to remember one function plus an additional option.

I'm just a newbie; will I ever make use of this? Not at first. However, you will read other people's Python programs. They'll use this technique. Rather than leave you guessing, we've provided some background so that you can work through someone else's Python program and really understand what it does and how it works.

11.4 Another Mapping : The `defaultdict`

We looked at dictionaries in *Mappings : The dict*. The built-in `dict` maps a *key* to a *value*.

The built-in `dict` type has a relatively simple response to a request for a non-existent key. When confronted with a key that doesn't exist, it raises a `KeyError` exception.

```
>>> freq= { 'red':1, 'black':2 }
>>> freq['green']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'green'
```

The Problem. Raising a `KeyError` exception isn't always the most desirable response. Let's look at something like the following.

```
import random

def even_odd( spin ):
    if spin == 0 or 38:
        even_odd = "0"
    elif spin % 2 == 0:
        even_odd = "even"
    else
        even_odd = "odd"

freq = { }
for i in range(1000):
    spin = randomc.randrange(0,38)
    result = even_odd( spin )
    freq[result] += 1
```

We're trying to accumulate a simple frequency distribution of even vs. odd vs. zero spins on a roulette wheel. While this **for** loop is short and sweet, it can't work.

Run it and see what happens.

When we get to `freq[result] += 1` (line 15), the required key value may not be in the dictionary.

We can change our **for** loop to this.

```
freq = { }
for i in range(1000):
    spin = randomc.randrange(0,38)
    result = even_odd( spin )
    if result not in freq:
        freq[result] = 0
    freq[result] += 1
```

This is unappetizing because we're executing the `if result not in freq` statement each time through the loop even though it's needed rarely.

A faster alternative is the following.

```
freq = { }
for i in range(1000):
    spin = randomc.randrange(0,38)
    result = even_odd( spin )
    try:
        freq[result] += 1
    except KeyError:
        freq[result] = 1
```

This is unappetizing: the **try** block is kind of big and ugly for a pretty standard problem.

Solution. There's something simpler, however, the `defaultdict` in the `collections` module.

The `collections` module gives us a very cool variation on the `dict` theme. The `defaultdict` doesn't raise `KeyError`; instead, it creates a new entry in the dictionary for us.

11.4.1 How We Create A Default Dictionary

There are several parts to creating a default dictionary.

1. We have to import the `collections` module. We'll talk more about modules and imports in *Modules : The unit of software packaging and assembly*.
2. We can then create our dictionary using the `collections.defaultdict()` function. This is like the built-in `dict()` function, but it has an extra parameter.

   The extra parameter is a factory function which will be called (with no arguments) to create any missing entries. This factory function is called instead of raising a `KeyError` exception.

Let's look at some examples.

```
import collections

freq = collections.defaultdict( int )
```

```
index = collections.defaultdict( list )
labels = collections.defaultdict( lambda: "N/A" )
```

freq This is the usual way to define a frequency table. We can then use `freq[someKey] += 1` without a second thought.

If *someKey* is not in the dictionary, the `defaultdict` will call the supplied factory function, `int()`, which returns a zero; which is added to the dictionary. Then it can add one to the value associated with the key.

index This is the usual way to define an *index*. An index has a key value and a list of values associated with that key. We can then use `index[someKey].append( anotherValue )` to put items into our index.

If *someKey* is not in the dictionary, the `defaultdict` will call the supplied factory function, `list()` which returns an empty list; which is added to the dictionary. Then it can append *anotherValue* to the list in the dictionary.

labels This is the usual way to have a dictionary of string labels. We can say `labels[someKey]` and get a string value, either the proper associated label or a special `"N/A"` string.

The **lambda:** is a way to define an anonymous function. Why do we need this? The `defaultdict` can't work with a simple literal, it must have a factory function. If we want a simple literal, we can "wrap" it in a *lambda expression*.

Lambda Expressions

Lambda expressions can be confusing. Whenever you see a lambda you can always rewrite it as something like this.

```
def return_na(): return "N/A"

labels= collections.defaultdict( return_na )
```

Here's the formal definition for the `collections.defaultdict()` function.

`collections.defaultdict`(*default_factory*) → dict with default factory

Creates a dictionary that will use the given *default_factory* to create a value instead of raising a `KeyError` exception.

11.4.2 Other Features of Default Dictionaries

A `defaultdict` is – in almost every respect – a dictionary. It does all the things we saw in *Operations We Can Perform On A Dictionary*.

Even though a `defaultdict` will fill in missing values, the `has_key()` method and the **in** operator still work properly.

Here are some examples.

```
>>> labels = collections.defaultdict( lambda: "N/A" )
>>> labels["hi mom"]
'N/A'
>>> labels[23]= "skidoo"
>>> labels[23]
'skidoo'
>>> labels[24]
'N/A'
```

```
9  >>> 'ralph' in labels
10 False
11 >>> labels.has_key('ralph')
12 False
13 >>> labels['ralph']
14 'N/A'
15 >>> 'ralph' in labels
16 True
```

1. We created a `defaultdict` named *labels*.
2. There is no entry with a key of `"hi mom"`. We got the default value.
4. We set a value for `23`.
5. When we get the value for `23`; it works as expected.
7. There is no entry with a key of `24`. We got the default value.
11. There's no entry with a key of `ralph`. Both the **in** operator and the `has_key()` method function return `False`.
13. When we get the value associated with the key of `"ralph"`, however, the `defaultdict` calls the initialization function (`lambda: "N/A"`) and supplies the default value.
15. After evaluating `labels['ralph']`, which forced creation of a default value, the key (`"ralph"`) is now in the dictionary.

Comparing. As we mentioned in *Comparing Dictionaries – Not A Good Idea* comparing dictionaries – as a whole – is meaningless and confusing.

The membership comparisons (**in**, **not in**) do apply to the keys of a dictionary.

Method Functions. Again, you can look at *Method Functions That Dictionaries Offer* for the list of method functions of a dictionary or a default dictionary.

The only important distinction is that `get()` never returns an `KeyError`. Instead it calls the initialization function to create a new entry in the dictionary.

Statements. You can see the various statement interactions in *Statements and Dictionaries*. This includes the **for** and **del** statements.

Built-in Functions. All of the built-in functions shown in *Built-in Functions for Dictionaries* apply to default dictionaries as well as standard dictionaries.

11.4.3 Default Dictionary Exercises

1. The `defaultdict` must have a function to create the default values. It can't work with a literal.

   What would go wrong if it did work with a single literal value? For immutable types of data (numbers, strings, tuples) this might not be a problem.

   For a mutable type of data (`list` or `set`) what would happen if multiple dictionary keys were associated with the same mutable object?

2. Look at *Word Frequenies*. Do this exercise using a `defaultdict` instead of a standard `dict`.

3. There are times when we want to display a dictionary sorted by value instead of sorted by key. We can't simply use `sorted( someDict.values() )` because we can't reliably deduce the key that went with each value. Consider this dictionary.

```
{(1, 2): 2, (1, 1): 1, (6, 6): 1}
```

If we sort the values into order, we don't know which key (`(1, 1)` or `(6, 6)`) the value 1 is associated with.

We can handle this, however, by creating an *inverted index* for the dictionary.

Part 1. First, create a dictionary of dice rolls. The key will be a dice tuple, the value will be a integer.

Create a `defaultdict` using `int()` as the factory for default values. Use a **for** loop to generate 1000 random rolls of two dice. Increment the frequency counts in your dictionary.

The result should be a dictionary with no more than 36 dice combinations and their frequencies.

Part 2. Invert this dictionary. The keys will be frequency. The value will be a list of dice rolls with that given frequency.

Create a `defaultdict` using `list()` as the factory for default values. Iterate through the frequency dictionary getting the roll and the count. Build the new dictionary by using the count as a key and appending the roll to the list.

Part 3. Print the new dictionary sorted in ascending order by the key. This key for the new dictionary is a count. The value will be a list of rolls that occured with the given frequency.

CHAPTER

TWELVE

WORKING WITH FILES

The Permanent Record

Files are one of the most important features of our operating system. For background, see *Hardware Terminology* for various kinds of hardware on which files reside.

In *Software Terminology* there is some background on the operating system structure and protocol that defines the "files" or "documents" we work with. The operating system insulates us from the complexities of various devices and provides us some handy abstractions that make it much easier to save, find and manage our documents.

Up until now, we've used very few files in just three ways. We've used a file named `python` (Windows `python.exe`) heavily. This file contains the binary program that *is* Python; we've run this program by typing a command in a terminal window or double-clicking an icon. We've also used a file named `idle.py`; this is a Python script that contains the **IDLE** program. Finally, we've also saved our various scripts in files and asked **Python** to run those files.

One of the most important things that our programs can do is read or write files. Files are a mixture of two unrelated concepts: they are a collection of data items, and they involve our OS notion of a file system, file names, directories, and devices. We'll introduce files in *External Data and Files*. We'll add the OS processing in *Files, Contexts and Patterns of Processing*. We'll wrap up with an overview of all the things files are used for in *File-Related Library Modules*.

12.1 External Data and Files

All programs must deal with *external data*. They will either accept data from sources outside the text of the program, or they will produce some kind of output, or they will do both. Think about it: if the program produces no output, how do you know it did anything?

By external data, we mean data outside of volatile, high-speed, primary memory; we mean data on peripheral devices. This may be persistent data on a disk, or transient data on a network interface. For now, it may mean transient data displayed on our terminal.

Most operating systems provide simple, uniform access to external data via the abstraction called a *file*. We'll look at the operating system implementation, as well as the Python class that gives us access to the operating system file in our programs.

In *File Objects – Our Connection To The File System*, we provide definitions of how Python works with files. We cover the built-in functions for working with files in *The File and Open Functions*. In *Methods We Use on File Objects*, we describe some method functions of file objects. We'll look at file-processing statements in *File Statements: Reading and Writing (but no Arithmetic)*.

12.1.1 File Objects – Our Connection To The File System

Abstractions Built on Top of Abstractions. Files do a huge number of things for us. To support this broad spectrum of capabilities, there are two layers of abstraction involved: the OS and Python. Unfortunately, both layers use the same words, so we have to be careful about casually misusing the word "file".

The operating system has devices of various kinds. All of the various devices are unified using a common abstraction that we call the *file system*. All of a computer's devices appear as OS files of one kind or another. Some things which aren't physical devices also appear as files. Files are the plumbing that move data around our information infrastructure.

Additionally, Python defines `file` objects. These file objects are the fixtures that give our Python program access to OS files.

The following figure shows this technology stack. Your program makes use of Python File objects. Python, in turn, makes use of OS file objects. Yes, it can be confusing that "file" is used for both things. However, you only have to focus on the Python file; the rest is just infrastructure to support your needs.

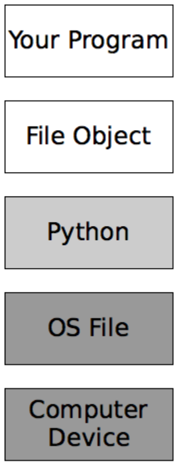

Figure 12.1: **Python File and OS File**

How Files Work. When your program evaluates a method function of a Python `file` object, Python transforms this into an operation on the underlying OS file. An OS file operation becomes an operation on one of the various kinds of devices attached to our computer. Or, a OS file operation can become a network operation that reaches through the Internet to access data from remote computers. The two layers of abstraction mean that one Python program can do a wide variety of things on a wide variety of devices.

12.1.2 Python File Objects

In Python, we create a `file` object to work with files in the file system. In addition to files in the OS's file system, Python recognizes a spectrum of *file-like objects*, including abstractions for network interfaces called pipes and sockets and even some kind of in-memory buffers.

Unlike sequences, sets and mappings, there are no Python literals for file objects. Lacking literals, we create a file object using the `file()` or `open()` factory function. We provide two pieces of information to this function. We can provide a third, optional, piece of information that may improve the performance of our program.

- The name of the file. The operating system will interpret this name using its "working directory" rules. If the name starts with `/` (or '`device:\`') it's an absolute name. Otherwise, it's a relative name; the current working directory plus this name identifies the file.

  Python can translate standard paths (using `/`) to Windows-specific paths. This saves us from having to really understand the differences. We can name all of our files using `/`, and avoid the messy details.

  We can, if we want, use raw strings to specify Windows path names using the `\` character.

- The access mode for the file. This is some combination of read, write and append. The mode can also include instructions for interpreting the bytes as characters.

- Optionally, we can include the buffering for the file. Generally, we omit this. If the buffering argument is given, `0` means each byte is transferred as it is read or written. A value of `1` means the data is buffered a line at a time, suitable for reading from a console, or writing to an error log. Larger numbers specify the buffer size: numbers over 4,096 may speed up your program.

Once we create the file object, we can do operations to read characters from the file or write characters to the file. We can read individual characters or whole lines. Similarly, we can write individual characters or whole lines.

When Python reads a file as a sequence of lines, each line will become a separate string. The `'\n'` character is preserved at the end of the string. This extra character can be removed from the string using the `rstrip()` method function.

A file object (like a sequence) can create an iterator which will yield the individual lines of the file. You can, consequently, use the file object in a **for** statement. This makes reading text files very simple.

When the work is finished, we also need to use the file's `close()` method. This empties the in-memory buffers and releases the connection with the operating system file. In the case of a socket connection, this will release all of the resources used to assure that data travels through the Internet successfully.

12.1.3 The File and Open Functions

Here's the formal definition of the `file()` and `open()` factory functions. These functions create Python file objects and connect them to the appropriate operating system resources.

`open`(*filename*, *mode*[, *buffering*]) → file

The *filename* is the name of the file. This is simply given to the operating system. The OS expects eitther absolute or relative paths; the operating system folds in the current working directory to relative paths.

The *mode* is covered in detail below. In can be `'r'`, `'w'` or `'a'` for reading (default), writing or appending. If the file doesn't exist when opened for writing or appending, it will be created. If a file existed when opened for writing, it will be truncated and overwritten. 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.

`file`(*filename*, *mode*[, *buffering*]) → file

This is another name for the `open()` function. It parallels other factory functions like `int()` and `dict()`.

Python expects the POSIX standard punctuation of `/` to separate elements of the filename path for all operating systems. If necessary, Python will translate these standard name strings to the Windows punctuation of `\`. Using standardized punctuation makes your program portable to all operating systems. The `os.path` module has functions for creating valid names in a way that works on all operating systems.

Tip: Constructing File Names

When using Windows-specific punctuation for filenames, you'll have problems because Python interprets the `\` as an escape character. To create a string with a Windows filename, you'll either need to use `\` in the string, or use an `r"  "` string literal. For example, you can use any of the following: `r"E:\writing\technical\pythonbook\python.html"` or `"E:\\writing\\technical\\pythonbook\\python.html"`.

Note that you can often use `"E:/writing/technical/pythonbook/python.html"`. This uses the POSIX standard punctuation for files paths, `/`, and is the most portable. Python generally translates standard file names to Windows file names for you.

Generally, you should either use standard names (using `/`) or use the `os.path` module to construct filenames. This module eliminates the need to use any specific punctuation. The `os.path.join()` function makes properly punctuated filenames from sequences of strings

---

The Mode String. The *mode* string specifies how the OS file will be accessed by your program. There are four separate issues addressed by the mode string: opening, bytes, newlines 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 OS file. If the OS file does not exist, raise an `IOError` exception. This is the default.

  w Open for writing. Start at he beginning of the OS file. If the OS file does not exist, create the OS file.

  a Open for appending. Start at the end of the OS file. If the OS file does not exist, create the OS file.

- **Bytes or Characters**. For the byte handling part of the mode string, there are two alternatives:

  b The OS file is a sequence of bytes; do not interpret the file as a sequence of characters. This is suitable for `.csv` files as well as images, movies, sound samples, etc.

  The default, if `b` is not included, is to interpret the file is a sequence of ordinary characters. The Python `file` object will be an iterator that yields each individual line from the OS file as a separate string. Translations from various encoding schemes like UTF-8 and UTF-16 will be handled automatically.

- **Universal Newlines**. The newline part of the mode string has two alternatives:

  U Universal newline interpretation. The first instance of `\n`, `\r\n` (or `\r`) will define the newline character(s). Any of these three newline sequences will be silently translated to the standard `'\n'` character. The `\r\n` is a Windows feature.

  The default, if `U` is not included, is to only handle this operating system's standard newline character(s).

- **Mixed Operations**. For the additional operations part of the mode string, there are two alternatives:

  + Allow both read and write operations to the OS file.

  The default, if `+` is not included, is to allow only limited operations: only reads for files opened with "r"; only writes for OS files opened with "w" or "a".

Typical combinations include the following:

- `"r"` to read text files.
- `"rb"` to read binary files. A `.csv` file, for example, is often processed in binary mode.
- `"w+"` to create new text file for reading and writing.

The following examples create Python `file` objects for further processing:

```
dataSource= open( "name_addr.csv", "rb" )
newPage= open( "addressbook.html", "w" )
theErrors= open( "/usr/local/log/error.log", "a" )
```

dataSource This example opens the existing file `name_addr.csv` in the current working directory for reading. The variable *dataSource* identifies this file object, and we can use this variable for reading strings from this file.

This file is opened in binary mode.

newPage This example creates a new file `addressbook.html` (or it will truncate this file if it exists). The file will be in the current working directory. The variable *newPage* identifies the file object. We can then use this variable to write strings to the file.

theErrors This example appends to the file `error.log` (or creates a new file, if the file doesn't exist). The file has the directory path `/usr/local/log/`. Since this is an absolute name, it doesn't depend on the current working directory.

Buffering files is typically left as a default, specifying nothing. However, for some situations, adjusting the buffering can improve performance. Error logs, for instance, are often unbuffered, so the data is available immediately. Large input files may be opened with large buffer numbers to encourage the operating system to optimize input operations by reading a few large chunks of data from the device instead of a large number of smaller chunks.

Tip: Debugging Files

There are a number of things that can go wrong in attempting to create a file object.

If the file name is invalid, you will get operating system errors. Usually they will look like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'wakawaka'
```

It is very important to get the file's path completely correct. You'll notice that each time you start IDLE, it thinks the current working directory is something like `C:\Python26`. You're probably doing your work in a different default directory.

When you open a module file in IDLE, you'll notice that IDLE changes the current working directory is the directory that contains your module. If you have your `.py` files and your data files all in one directory, you'll find that things work out well.

The next most common error is to have the wrong permissions. This usually means trying to writing to a file you don't own, or attempting to create a file in a directory where you don't have write permission. If you are using a server, or a computer owned by a corporation, this may require some work with your system administrators to sort out what you want to do and how you can accomplish it without compromising security.

The `[Errno 2]` note in the error message is a reference to the internal operating system error numbers. There are over 100 of these error numbers, all collected into the module named `errno`. There are a lot of different things that can go wrong, many of which are very, very obscure situations.

12.1.4 Methods We Use on File Objects

The Python `file` object is our view of the underlying operating system file. The OS file, in turn, gives us access to a specific device.

The Python `file` object has a number of operations that transform the `file` object, read from or write to the OS file, or access information about the `file` object.

Reading. The following read methods get data from the OS file. These operations may also change the Python `file` object's internal status and buffers. For example, at end-of-file, the internal status of the `file` object will be changed. Most importantly, these methods have the very visible effect of consuming data from the OS file.

`file.read(`*size*`)` → string

Read as many as *size* characters from file *f* as a single, large string. If *size* is negative or omitted, the rest of the file is read into a single string.

```
from __future__ import print_function
dataSource= open( "name_addr.csv", "r" )
theData= dataSource.read()
for n in theData.splitlines():
    print(n)
dataSource.close()
```

`file.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. 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, just the newline character.

```
from __future__ import print_function
dataSource= file( "name_addr.csv", "r" )
n= dataSource.readline()
while len(n) > 0:
    print(n.rstrip())
    n= dataSource.readline()
dataSource.close()
```

`file.readlines(`*hint*`)`

Read the next lines or as many lines from the next *hint* characters from file *f*. The *hint* size 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. If the file is at the end, *f.* `readlines()` returns a zero length list.

When we simply reference a `file` object in a **for** statement, this is the function that's used for iteration over the file.

```
dataSource= file( "name_addr.csv", "r" )
for n in dataSource:
    print(n.rstrip())
dataSource.close()
```

Writing. The following methods send data to the OS file. These operations may also change the Python `file` object's internal status and buffers. Most importantly, these methods have the very visible effect of producing data to the OS file.

`file.flush()`

Flush all accumulated data from the internal buffers of file *f* to the device or interface. If a file is buffered, this can help to force writing of a buffer that is less than completely full. This is appropriate for log files, prompts written to *sys.stdout* and error messages.

`file.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 or before *size*. This function is not available on all platforms.

`file.write(string)`

Write the given string to file *f*. Buffering may mean that the string does not appear on a console until a `close()` or `flush()` operation is used.

```
newPage= file( "addressbook.html", "w" )
newPage.write( "<html>\n<head><title>Hello World</title></head>\n<body>\n" )
newPage.write( "<p>Hello World</p>\n" )
newPage.write( "<\body>\n</html>\n" )
newPage.close()
```

`file.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.

```
newPage= file( "addressbook.html", "w" )
newPage.writelines( [ "<html>\n", "<head><title>Hello World</title></head>\n", "<body>\n" ] )
newPage.writelines( ["<p>Hello World</p>\n" ] )
newPage.writelines( [ "<\body>\n", "</html>\n" ] )
newPage.close()
```

Accessors. The following file accessors provide information about the `file` object.

`file.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.

`file.fileno()` → integer

Return the internal file descriptor (fd) number used by the OS library when working with file *f*. A number of modules provide access to these low-level libraries for advanced operations on devices and files.

`file.isatty()` → boolean

Return `True` if file *f* is connected to an OS file that is a console or keyboard.

`file.closed()` → boolean

This attribute of file *f* is `True` if the file is closed.

`file.mode()` → string

This attribute is the mode argument to the `file()` function that was used to create the file object.

`file.name`

This attribute of file *f* is the filename argument to the `file()` function that was used to create the file object.

Transfomers. The following file transforms change the `file` object itself. This includes closing it (and releasing all OS resources) or change the position at which reading or writing happens.

`file.close()`

Close file *f*. The closed flag is set. Any further operations (except a redundant close) raise an `IOError` exception.

`file.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 `0` (the default), move to the absolute position given by *offset*. `f.seek(0)` will rewind file *f*.

If *whence* is `1`, move relative to the current position by *offset* bytes. If offset is negative, move backwards; otherwise move forward.

If *whence* is 2, move relative to the end of file. `f.seek(0,2)` will advance file *f* to the end.

12.1.5 File Statements: Reading and Writing (but no Arithmetic)

A file object (like a sequence) can create an iterator which will yield the individual lines of the file. We looked at how sequences work with the **for** statement in *Looping Back : Iterators, the for statement and Generators*. Here, we'll use the file object in a **for** statement to read all of the lines.

Additionally, the **print** statement can make use of a file other than standard output as a destination for the printed characters. This will change with Python 3.0, so we won't emphasize this.

Opening and Reading From a File. Let's say we have the following file. If you use an email service like HotMail, Yahoo! or Google, you can download an address book in *Comma-Separated Values* (CSV) format that will look similar to this file. Yahoo!'s format will have many more columns than this example.

name_addr.csv

```
"First","Middle","Last","Nickname","Email","Category"
"Moe","","Howard","Moe","moe@3stooges.com","actor"
"Jerome","Lester","Howard","Curly","curly@3stooges.com","actor"
"Larry","","Fine","Larry","larry@3stooges.com","musician"
"Jerome","","Besser","Joe","joe@3stooges.com","actor"
"Joe","","DeRita","CurlyJoe","curlyjoe@3stooges.com","actor"
"Shemp","","Howard","Shemp","shemp@3stooges.com","actor"
```

Here's a quick example that shows one way to read this file using the file's iterator. This isn't the *best* way, that will have to wait for *The csv Module*.

```
1 dataSource = file( "name_addr.csv", "r" )
2 for addr in dataSource:
3     print(addr)
4 dataSource.close()
```

1. We create a Python `file` object for the `name_addr.csv` in the current working directory in read mode. We call this object *dataSource*.
2. The **for** statement creates an iterator for this file; the iterator will yield each individual line from the file.
3. We can print each line.
4. We close the file when we're done. This releases any operating system resources that our program tied up while it was running.

A More Complete Reader. Here's a program that reads this file and reformats the individual records. It prints the results to standard output. This approach to reading CSV files isn't very good. In the next chapter, we'll look at the `csv` module that handles some of the additional details required for a really reliable program.

nameaddr.py

```
1 #!/usr/bin/env python
2 """Read the name_addr.csv file."""
3 dataSource = file( "name_addr.csv", "r" )
4 for addr in dataSource:
5     # split the string on the ,'s
```

```
6      quotes= addr.split(",")
7      # strip the '"'s from each field
8      fields= [ f.strip('"') for f in quotes ]
9      print( fields[0], fields[1], fields[2], fields[4] )
10 dataSource.close()
```

3. We open the file `name_addr.csv` in our current working directory. The variable *dataSource* is our Python file object.
4. The **for** statement gets an iterator from the file. It can then use the iterator, which yields the individual lines of the file. Each line is a long string. The fields are surrounded by `"`s and are separated by `,`s.
7. We use the `split()` function to break the string up using the `,`s. This particular process won't work if there are `,`s inside the quoted fields. We'll look at the `csv` module to see how to do this better.
9. We use the `strip()` function to remove the `"`s from each field. Notice that we used a list comprehension to map from a list of fields wrapped in `"`s to a list of fields that are not wrapped in `"`s.

Seeing Output with print. The `print()` function does two things. When we introduced `print()` back in *Seeing Results : The print Statement*, we hustled past both of these things because they were really quite advanced concepts.

We covered strings in *Sequences of Characters : str and Unicode*. We're covering files in this chapter. Now we can open up the hood and look closely at the `print()` function.

1. The `print()` function evaluates all of its expressions and converts them to strings. In effect, it calls the `str()` built-in function for each argument value.
2. The `print()` function writes these strings, separated by a separator character, *sep*. The default separator is a space, `' '`.
3. The `print()` function also writes an end character, *end*. The default end is the newline character, `'\n'`.

The `print()` function has one more feature which can be very helpful to us. We can provide a *file* parameter to redirect the output to a particular file.

We can use this to write lines to *sys.stderr*.

```
1 from __future__ import print_function
2 import sys
3 print("normal output")
4 print("Red Alert!", file=sys.stderr)
5 print("still normal output", file=sys.stdout)
```

1. We enable the print function.
2. We import the `sys` module.
3. We write a message to standard output using the undecorated **print** statement.
4. We use the *file* parameter to write to *sys.stderr*.
5. We also use the:varname:*file* parameter to write to *sys.stdout*.

When you run this in **IDLE**, you'll notice that the error messages display in red, while the standard output displays in blue.

Print Command. Here is the syntax for an extension to the **print** statement.

```
print  >> file [ ,  expression , ... ]
```

The >> is an essential part of this peculiar syntax. This is an odd special case punctuation that doesn't appear elsewhere in the Python language. It's called the "chevron print".

Important: Python 3

This chevron print syntax will go away in Python 3. Instead of a **print** statement with a bunch of special cases, we'll use the `print()` function.

Opening A File and Printing. This example shows how we open a file in the local directory and write data to that file. In this example, we'll create an HTML file named `addressbook.html`. We'll write some content to this file. We can then open this file with FireFox or Internet Explorer and see the resulting web page.

addrpage.py

```python
#!/usr/bin/env python
"""Write the addressbook.html page."""
from __future__ import print_function
new_page = open( "addressbook.html", "w" )
print('<html>', new_page)
print(' <head>'
    '<meta http-equiv="content-type" content="text/html; charset=us-ascii">'
    '<title>addressbook</title></head>', file=new_page)
print(' <body><p>Hello world</p></body>', file=new_page )
print('</html>', file=new_page)
new_page.close()
```

12.1.6 Basic File Exercises

1. **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. What (if any) relevance does this have to file processing?

2. **Skip The Header Record.**

   Our `name_addr.csv` file has a header record. We can skip this record by getting the iterator and advancing to the next item.

   Write a variation on `nameaddr.py` which uses the `iter()` to get the iterator for the *dataSource* file. Assign this iterator object to *dataSrcIter*. If you replace the file, *dataSource*, with the iterator, *dataSrcIter*, how does the processing change? What is the value returned by `dataSrcIter.next()` before the **for** statement? How does adding this change the processing of the **for** statement?

3. **Combine The Two Examples.**

Our two examples, *addrpage.py* and *name_addr.py* are really two halves of a single program. One program reads the names and address, the other program writes an HTML file. We can combine these two programs to reformat a CSV source file into a resulting HTML page.

The name and addresses could be formatted in a web page that looks like the following:

```
<html>
<head><title>Address Book</title></head>
<body>
<table>
<tr><td>last name</td><td>first name</td><td>email address</td></tr>
<tr><td>last name</td><td>first name</td><td>email address</td></tr>
<tr><td>last name</td><td>first name</td><td>email address</td></tr>
...
</table>
</body>
</html>
```

Each of our input fields becomes an output field sandwiched in between `<td>` and `</td>`. In this case, we uses phrases like *last name*, *first name* and *email address* to show where real data would be inserted. The other HTML elements like `<table>` have to be printed as they're shown in this example.

Your final program should open two files: `name_addr.csv` and `addressbook.html`. Your program should write the initial HTML material (up to the first `<tr>`) to the output file. It should then read the CSV records, writing a complete address line between `<tr>` to `</tr>`. After it finishes reading and writing names and addresses, it has to write the last of the HTML file, from `</table>` to `</html>`.

12.1.7 File FAQ's

Why are there two meanings for "file"? It's a matter of running out of suitable synonyms. The operating system maintains files as collections of bytes on a disk or USB drive. Python gives us access to those OS files using a Python object called a file. It might have been nicer to call it a file handle, file channel, file socket or file descriptor. But the extra word would eventually get dropped, and we'd be back to Python files giving us access to OS files.

When we look under the hood, it actually gets more complex. Python's file object is built around the C language FILE, defined in the stdio library, which uses the OS file descriptors which give access to the data on the disk (known as a file). Whew!

Yes, it's rather complex. But it's also very, very important because all of your data will be in OS files, and you'll want to access those OS files using Python file objects.

Why do they have files? What's wrong with accessing devices directly? Wouldn't it be simpler? Actually, direct access to devices is a pretty ugly and complex problem. Without the unifying abstraction of "file", it would be nearly impossible to get useful data processing accomplished.

The differences between IDE, SATA, SCSI and USB drives is enough to make someone crazy, and they're all – basically – disks. Each one has unique subtleties to how the device is identified, how requests are sent to it, and the data comes back from the device. Thrown in CD's and DVD's and you've got even more complex rules for handling the various kinds of "mass storage" media that are connected to your computer.

When you start to look at network interfaces (wireless WiFi, Bluetooth, and Ethernet) you'll see more differences than similarities. Worse, your program would have to be customized to handle WiFi cards made by different manufacturers. For example, NetGear and LinkSys differences would be part of your program, not part of the operating system.

This is so important, we'll return to it in *File-Related Library Modules*.

12.2 Files, Contexts and Patterns of Processing

When we're working with files, there's a concept of a "context" – the place in our Python program where the file is open. In this context, our program is tying up numerous operating system resources that need to be released. This is done with the **with** statement. We'll look at this in *The with Statement*.

We'll look at some examples of processing with the `csv` module in *The csv Module*. This will include some typical file-processing design patterns:

- *Reading Files*
- *Reading and Sorting A File*
- *Reading Files With Header Lines*

In some of these examples, we'll be dealing with stock and mutual fund information that we downloaded from the internet. You can find this kind of information on http://finance.yahoo.com.

Also, we are going to use simple floating-point numbers to represent dollar amounts. This will give us answers which are good enough for now. Later (in *Fixed-Point Numbers : Doing High Finance with decimal*), we'll introduce the `decimal` module, which we can use to produce correct results.

12.2.1 The with Statement

When we open a file with the `open()` function, we tie up some operating system resources. It's polite to release those resources when we're done by using the `file.close()` method.

This is particularly important when we're writing a file. Our Python object depends on other objects in the supporting libraries and the operating system. These dependencies may be buffering data, attempting to optimize our use of (expensive, slow) phsyical media.

Consider a Python program which is supposed to be writing a file, and which ends abruptly. One (ore more) of those buffers may not have completely flushed to the device. To assure that data is properly sent to our devices, we **must** close the file.

The easiest way to do be sure that a file is closed properly is using the **with** statement.

The basic syntax of the **with** statement looks like this:

```
with expression as variable :
    suite
```

The words **with**, **as** and command:*:* are essential syntax.

The *suite* is an indented block of one or more statements. Any statement is allowed in the block, including indented **with** statements.

Semantics. The **with** statement evaluates the expression and assigns it to the given variable. The result of the expression **must** be an object which correctly supports the context manager protocol. The context manager will then be notified of entry and exit into the processing context.

The *suite* is the processing context which will be correctly managed. A `file` is a context manager which offers is the guarantee that the file will be closed when the context exits.

Example. Here's a quick example of the **with** statement.

```
from __future__ import print_function

with open('test.html','w') as web_page:
    print( "<html><head><title>test</title></head>", file=web_page )
    print( "<body><p>Hello world</p></body", file=web_page )
    print( "</html>", file=web_page )
```

Notice that we didn't need to do `web_page.close()`. That close was handled for us by the **with** statement.

12.2.2 The `csv` Module

One common and useful file format is the *Comma-Separated Values* (CSV) format. A standard CSV file uses a `,` to delimit values. If a value has a `,` in it, the value is quoted, usually with `"`. If a value has `"` in it, the `"` characters are doubled.

Other delimiters and quoting rules are possible. Beacuse of this, a wide variety of files can be processed as if they were "CSV". A file with tab separators, for example, can also be handled gracefully with this module.

Here's an example file.

name_addr.csv

```
"First","Middle","Last","Nickname","Email","Category"
"Moe","","Howard","Moe","moe@3stooges.com","actor"
"Jerome","Lester","Howard","Curly","curly@3stooges.com","actor"
"Larry","","Fine","Larry","larry@3stooges.com","musician"
"Jerome","","Besser","Joe","joe@3stooges.com","actor"
"Joe","","DeRita","CurlyJoe","curlyjoe@3stooges.com","actor"
"Shemp","","Howard","Shemp","shemp@3stooges.com","actor"
```

The `csv` module gives us a handy definition called a `reader` which will extract individual records from the file, properly match up the `"`s, and correctly split fields on the `,`s.

The `csv.reader()` function is an iterator object that both gets individual lines from the file and does all of the necessary decoding for us. We can use this CSV iterator with the **for** statement to correctly parse every line from the file.

To use the `csv` module, we must use **import csv**. This introduces the module's functions. We're interested in `csv.reader()` for this first example.

Some spreadsheet software writes unusual line-ending sequences to CSV files. In order to handle the unusual line-ending characters, we have to open the file with a mode of `"rb"`.

```
from __future__ import print_function
import csv
with open( "name_addr.csv", "rb" ) as naFile:
    rdr= csv.reader( naFile )
    for person in rdr:
        print( person[0], person[2], person[4] )
```

When you run this program, you'll notice that the header line in the file is being processed as if it were data. We'd like to skip past this gracefully. Since *rdr* is an iterator, we can use `rdr.next()` to get the first line from the file.

```
from __future__ import print_function
import csv
with open( "name_addr.csv", "rb" ) as naFile:
    rdr= csv.reader( naFile )
    header= rdr.next()
    for person in rdr:
        print( person[0], person[2], person[4] )
```

This version quietly saves the header in *header*, and processes the data rows separately.

12.2.3 Reading Files

Here's another example that reads a CSV (Comma-Separated Values) file format. A popular stock quoting service on the Internet will provide CSV files with current stock quotes. Here's an example of the file that we downloaded.

```
"^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 stock, date and time are quoted strings. The other fields are generally numbers, typically in dollars or percents with two digits of precision. There are a few exceptions to this format for indexes and mutual funds.

This is a very old example of the file. The prices of these stocks may have changed, but the file format hasn't changed one bit.

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`, without quotes. 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.

stockquote.py

```
1 #!/usr/bin/env python
2 from __future__ import print_function
3 import csv
4 with open( "quotes.csv", "r" ) as quote_file:
5     qRdr= csv.reader( quote_file )
6     for quote in qRdr:
7         stock, price, dt, tm, chg, opn, dHi, dLo, vol = quote
8         print(stock, price, dt, tm, chg, vol)
```

4. We open our quotes file for reading, creating an object named qFile. This file object in our Python program will read from the quotes.csv file on our disk.
5. By using the csv.reader function, we create an iterator which will parse each line of the CSV file, returning a list of data values with the quotes and commas removed.
6. We use a for statement to iterate through the sequence of lines in the file. Each line of the file is a list of values that comprise a single stock quote, quote.
7. We use multiple assignment to assign each field of the quote to a relevant variable.

When finished processing the file, it's role as context manager in the **with** statement will assure that it's closed. This will release any resources like file descriptors or buffers that were associated with this file.

12.2.4 Reading and Sorting A File

This example shows a short way to read, sort and write a file.

We can easily sort data in a list, using the `sort()` method function. So, our solution must first read the data, creating a list. We can sort the list, then write the list in sorted order for processing by another program.

In this case, we'll sort our stock quotes by company, the first field in each quote record. For simplicity we'll write the sorted CSV file to *sys.stdout*. We'll look at some extensions to this program to sort by different fields and write to a different output file.

stocksort.py

```
1  #!/usr/bin/env python
2  from __future__ import print_function
3  import csv
4  with open( "quotes.csv", "r" ) as quote_file:
5      qRdr= csv.reader( file )
6      data= [ tuple(quote) for quote in qRdr ]
7
8  def name(quote):
9      return quote[0]
10
11 data.sort( key=name )
12 for q in data:
13     print( q[0], q[1], q[2], q[3] )
```

4. We create file object referencing our `quotes.csv` file. We use `csv.reader()` to create an iterator which will parse each line of the CSV file, returning a list of data values with the quotes and commas removed.

6. We use a list comprehension to create a "list-of-tuples" data structure from the contents of the file. This comprehension creates a list as follows.

   - Iterate over each quote of the file, setting variable *quote* to each line produced by the CSV reader. This line will be a list of values with nine elements, representing the stock, price, date, time, change, opening Price, daily high, daily low and volume traded.
   - We transform each individual quote from a 9-item list into a 9-tuple.

8. We define a key function named `name()`. This function returns the key for sorting. In this case, the key is item zero of each quote, which is the name of the stock.

11. We sort the data sequence. We use our function definition to find the key for each quote. This kind of sort is covered in depth *Sorting a List: Expanding on the Rules*.

12. Once the sequence of data elements is sorted, we can then write the company, price, date and time in company name order.

Tip: Debugging CSV Input

One problem with file processing is that our Python data structure isn't a giant string of characters. However, the file is simply a giant string. Essentially, reading a file is a way of translating the characters into a useful Python structure.

The most common thing that can go wrong is not creating the expected structure in our Python program. In the *Reading and Sorting* example, we might not create our list of tuples correctly.

It is helpful to print the value of the *data* variable to get a good look at the data structure which is produced. Here we show the beginning of our "list of tuples". We've adjusted the Python output to make it a little more readable.

```
[('"^DJI"', '10452.15', '"9/26/2005"', '"1:50pm"', '+32.56',
  '10420.22', '10509.23', '10420.22', '137206720'),
 ('"^IXIC"', '2121.07', '"9/26/2005"', '"1:50pm"', '+4.23',
  '2127.90', '2132.60', '2119.17', '0'), ...
```

Looking at the intermediate results helps us be sure that we are reading the file properly.

A more interesting modification is to add various function definitions for different sorts. For instance, if we wanted to sort by price (field 1), we could make the following change. We can define any number of functions and use one of them in the `sort()` method function.

```
def name(quote):
    return quote[0]
def price(quote):
    return quote[1]

data.sort( key=price )
```

Bonus Question. Why did we add the calls to the built-in function `float()`? What happens if we take those function calls out? What is the difference between comparing strings of digits and comparing numeric values? For review, see *Sorting a List: Expanding on the Rules*.

12.2.5 Reading Files With Header Lines

This example uses data that we downloaded from a web-based portfolio manager. This portfolio manager's stock information comes in a file format that includes an extra header line with column titles in it. This file is called `dwnld_portinfo.csv`. Here is an example.

```
"",TICKER,"PRICE","PRICE CHANGE","# SHARES","P/E","PURCHASE PRICE","PURCHASE PRICE"
"","CAT","58.34","-0.58","50","17.26","43.5","43.5"
"","DD","38.63","-0.15","50","14.97","42.8","42.8"
"","EK","25.81","0.3","50","0","42.1","42.1"
"","GM","31.15","0.08","50","0","53.9","53.9"
"","USD",--,--,--,--,--,--
"Totals:","","","","","","",""
```

This file contains a header line that names the data columns, making processing much more reliable. If the web site adds a field or changes the order of the fields, we can use this column title information to assure that our program doesn't need to be changed.

We can use the column titles to create a dictionary for each line of data. By making a dictionary of each line, we can identify each piece of data by the column name, not by the position. Identifying data by column name is generally more clear. It's also immune the column order.

This file has two lines of junk that we want to gracefully ignore. First, it has a trailing "USD" line, which shows the cash position of the portfolio. Second, it has a "Totals:" line which doesn't seem to have anything. We'll need to discard these two lines.

portfolio.py

```
#!/usr/bin/env python
from __future__ import print_function, division
import csv
with open( "dwnld_portinfo-3.csv", "r" ) as posn_file:
    pDictRdr= csv.DictReader( posn_file )
```

```
 6      invest= 0
 7      current= 0
 8      for posn in pDictRdr:
 9          if posn[""] == "Totals:":
10              continue
11          if posn["TICKER"] == "USD":
12              continue
13          print(posn)
14          invest += float(posn["PURCHASE PRICE"])*float(posn["# SHARES"])
15          current += float(posn["PRICE"])*float(posn["# SHARES"])
16  print(invest, current, (current-invest)/invest)
```

4. We open our portfolio position file for reading, creating an object named *posn_file*.
5. We use our input file, *posn_file* to create a `csv.DictReader`. This reader will do three things: it will match up `"` characters, split fields on `,` characters, and use the first line of the file as keys to create a dictionary.

   Each row will be a dictionary. The key will be the column header, and the value will be this row's data value.
6. We also initialize two counters, *invest* and *current* to zero. These will accumulate our initial investment and the current value of this portfolio.
8. We use a **for** statement to iterate through the positions in the file. Each position will be a dictionary, assigned to the variable *posn*.

   We can get each field's value using the column title. For example, we get the ticker symbol using `posn["TICKER"]`.
9. Our first piece of processing is a filter. The totals line has the value `Totals:` in the unnamed column. We'll ignore the totals line at the end (`posn[""] == "Totals:"`) by continuing the loop. The cash position has a ticker symbol of `USD`. We'll ignore the cash position ('`posn["TICKER"] == "USD"`') by continuing the loop.
16. Our second piece of processing is some simple calculations. 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*.
18. When the loop has terminated, we can close the file, write out the two numbers, and compute the percent change.

The conversion to a dictionary makes our "business rules" relatively easy to read.

If we wanted to be really precise, we could say things like the following to separate the issue of identifying the cash position line from the processing for the cash position line. The boolean variable *cashPosition* is set to `True` when we identify the cash position line in the file.

```
cashPosition= data["TICKER"] == "USD"
if cashPosition:
    continue
```

Additionally, we could make the processing more clear by expanding it into the following. We would separate the conversion from string to number from the calculation using that number.

```
shares= float(data["# SHARES"])
purch= float(data["PURCHASE PRICE"])
invest += shares * purch
```

12.2.6 Advanced File 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 and count the lines of code. Once we can process a single file, we can then use the `glob` module to locate all of the `*.py` files in a given directory and process each file.

   Develop a `fileLineCount(name)` function 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\tlines`'

   For a sample application to analyze, 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\tlines`'. Read this 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.

   You should make liberal use of the string `%` operator for formatting the output.

3. **File Processing Pipeline**.

   The previous two exercises produced programs which can be part of a processing pipeline. The first exercise 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  path/to/data | python lineSummary.py
   ```

   This is an important "fit and finish" issue for GNU/Linux programs. A well-behaved program can use `sys` to get argument values so that an names of files or directories are not "hard-coded" into the program. Additionally we should always use *sys.stdout* and *sys.stdin* to make it easy to reuse programs.

12.3 File-Related Library Modules

When we first look at the problem we're trying to solve, it's often difficult to see how we apply Python. This chapter is really about the question "Now that I know the language, how do I get started on my real problem?"

The answer is – almost always – "What information do you have and what processing do you want to do?" This chapter will help you apply the file abstraction to your problem.

Sources and Sinks. When we look at the information we have, it can flows in one of the following directions. places.

- **From someone's head to a file**. Our program's job is some kind of knowledge capture. Even if we're writing a program to help artists paint or musicians compose, we're capturing knowledge (or ideas or art or relationships) that started in someone's head. We'll then be encoding the knowledge (or idea or artwork) and saving it on a device attached to a computer. In short, we'll be creating files.
- **From a file to someone's head**. Our program will be reading and processing data that reside computer. If we're reading a web page, looking at a stock portfolio or reviewing results of a simulation, data starts in computer files and we read them. If we're playing a game, we're reading the game information and player actions from files and displaying the state of the game.
- **From file to file**. Our program will be reading and writing data On the computer. For example, if we're applying an audio filter to an MP3 file, we're starting with a file, processing that data in that file, and creating a new file.

All the processing we want to do will involve files in one way or another. Files are either input or output or both. We'll focus on disk files because they're the easiest to work with as a beginner.

We'll talk about how data is organized on files in *File Organization and Structure*.

There are a number of library modules that are relevant to file processing.

- *File and Directory Access*
- *Generic OS Services*
- *Data Persistentence*
- *Data Compression and Archiving*
- *Internet Data Handling*
- *Python Runtime Services*

We can then look at some common variations on file processing in *Files are the Plumbing of a Software Architecture*.

12.3.1 File Organization and Structure

In order to successfully read data from a file, the bytes, characters and lines must have some kind of organization.

We've already seen on file organization up close: CSV format. We looked at this in *The csv Module*.

From the Ground Up. At the foundation, a file is a sequence of bytes. Any other interpretation of those bytes is the responsibility of the application program. In our case, the program is Python; it includes numerous library modules to handle various kinds of encodings and data organization.

The bytes in a file can have a variety of interpretations. It might be images or financial records or GPS coordinates. Anything is possible. Anything.

The files that are easiest to work with will contain text. By text we mean characters in some well-defined encodding. A very popular encoding is US-ASCII. Other encodings include UTF-8 and UTF-16. The encoding is required to properly interpret the bytes as characters.

A file of text can have higher-level structures. The most basic text file formats interpret the characters as a sequence of variable length lines. Each line terminated with a *newline* character. The newline character is coded as `'\n'` in Python.

We could interpret a file of characters as an XML or HTML document. In these cases, the XML or HTML rules tell us how to interpret the characters as tags, coded data, elements, attributes, processing instructions and other elements of these languages.

A file of characters could be understood as a Python program. Usually, we emphasize this by making sure the file name ends in `.py`.

A `.csv` file is a sequence of lines. Each line has one or more fields, wrapped in `""`, and separated by `,`s.

Hiding the Details. We use the notion of "layers" to understand the structure of files. At the foundation layer (bytes) to ASCII or Unicode characters to yet higher layers built on this foundation. On top of the character foundation, our choices fan out in many, many directions. We'll stick to the most common file types: HTML, XML, Python, .CSV.

This idea of layers of meaning – from bytes to characters to lines to meaningful records – is yet another application of the abstraction principle. Python allows us to imagine that a file consists of lines of characters. It provides us an abstraction that conceals the details of all those bytes and how they are encoded. We can do the same thing in our software by writing a function that reads a line and transforms it into a meaningful tuple or object that we can process.

Non-Character Files. There are many common file formats which do not have obvious character encodings. Image files contain encodings of the picture elements, pixels. A photo of your family on vacation in Stockholm might be a 4.6 megapixel image. This image has 2560 × 1920 individual dots, each of which can be any of 16 million different colors. A raw image file could be 14 million individual bytes of data.

When I look at my computer, I see that the `.jpeg` files are much smaller. It turns out that these files are compressed. Some clever experts defined ways to reduce the number of bytes required to capture most of the image with enough accuracy that some of us barely notice the difference between a JPEG image and a RAW image.

An audio file might have 48,000 samples per second, spread over three minutes leads to 8.2 million individual samples. Each sample could be one of 4000 amplitude levels, leading us to 12.4 million individual bytes of data. An MP3 file uses a clever algorithm to compress all of these bytes down to 3.5 million bytes that sound pretty much the same as the original AIFF audio file.

Database Files. A *database* is one or more very highly organized files. A database may contain text, audio and images. That means that the content of a database may contain bytes that must be interpreted as characters, bytes that can be interpreted as MP3-encoded audio, and bytes that can be understood as JPEG-encoded images.

The top-most layer is the "meaning" of all that data and all those bytes. In this case, the database may be a summary of decades of custom quilt-making, with pictures, stories, and descriptions of dozens of quilts.

Tip: Debugging File Formats

When we talk about how data appears in files, we are talking about "data representation." This is a difficult and sometimes subtle design decision. A common question is "How do I know what the data is?" . There are two important points of view.

- The program you are designing will save data in a file for processing later. Since you are designing the file, you get to choose the representation. You can pick something that is easy for your Python program to write. Or, you can look at other programs and pick something that is easy for the other programs to read. This can be a difficult balancing act.
- The program you are designing must read data prepared by another program. Since someone else designed the file, you will interpret the data they provide. If their format is something that Python can easily interpret, your program will be very simple. However, the more common situation is that their format is not something Python can interpret, and you must write this interpretation yourself.

12.3.2 File and Directory Access

The "File and Directory Access" section of the Python Library has a number of very useful modules for working with files and directories.

`os.path` The `os.path` module contains operating-system agnostic functions for managing path and directory Names. Since these functions are tailored for each operating system, this is the best way to assure portability of your program.

The `os.path` module helps us parse and create correct file names. This module addresses the most obvious differences among operating systems: the way that files are named. In particular, the *path separator* can be either the POSIX standard `/`, or the windows `\`. Additionally, there's a MacOS Classic mode that 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.

A serious mistake is to use ordinary string functions with literals for the path separators. For example, a program using \ as the separator will only work on Windows, and won't work anywhere else. A less serious mistake is to use *os.pathsep*. The best approach is to use the functions 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 )`'

```
>>> import os
>>> fn='/Users/slott/Documents/Writing/NonProg2.5/notes/portfolio.py'
>>> os.path.basename( fn )
'portfolio.py'
```

`os.path.dirname`(*path*) → directory

Return the directory name, the first half of the result created by '`os.path.split( path )`'

```
>>> import os
>>> fn='/Users/slott/Documents/Writing/NonProg2.5/notes/portfolio.py'
>>> os.path.dirname(fn)
'/Users/slott/Documents/Writing/NonProg2.5/notes'
```

`os.path.exist`(*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.

```
>>> import os
>>> import time
>>> fn='/Users/slott/Documents/Writing/NonProg2.5/notes/portfolio.py'
>>> os.path.getatime( fn )
1246637163.0
>>> time.ctime(_)
'Fri Jul  3 12:06:03 2009'
```

`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*) → long integer

Return the size of a file, in bytes, reported by `os.stat()`.

```
>>> import os
>>> fn='/Users/slott/Documents/Writing/NonProg2.5/notes/portfolio.py'
>>> os.path.getsize( fn )
175L
```

`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[, ...])` → string
Join path components using the appropriate path separator. This is the best way to assemble long path names from component pieces. It is operating-system independent, and understands all of the operating system's punctuation rules.

```
>>> import os
>>> os.path.join( '/Users', 'slott', 'Documents', 'Writing' )
'/Users/slott/Documents/Writing'
```

`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.

```
>>> import os
>>> fn='/Users/slott/Documents/Writing/NonProg2.5/notes/portfolio.py'
>>> os.path.split( fn )
('/Users/slott/Documents/Writing/NonProg2.5/notes', 'portfolio.py')
```

`os.path.splitdrive(path)` → tuple
Split a pathname into a drive specification and the rest of the path. Useful on DOS/Windows/NT. Useless for Linux or Mac OS.

`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 tuple (*root* , *ext*) is such that *root* + *ext* yields the original path.

```
>>> import os
>>> fn='/Users/slott/Documents/Writing/NonProg2.5/notes/portfolio.py'
>>> dir, file = os.path.split(fn)
>>> os.path.splitext( file )
('portfolio', '.py')
```

The following example is typical of the manipulations done with `os.path`:

```
from __future__ import print_function
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() == '.HTML':
        newFile= os.path.join( dir, file ) + '.BAK'
        print(oldFile, newFile)
        process( oldFile, newFile )
```

This program imports the `sys` and `os.path` modules. The variable *oldFile* is set to each file name that is listed in the sequence *sys.argv* by the **for** statement.

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.

The extension is tested to be `.HTML`. The processing only applies to these files. A new file name is joined from the path, base name and a new extension (`.BAK`). The old and new file names are printed and some processing, defined in the `process()`, uses the *oldFile* and *newFile* names.

Path Processing

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.

`shutil` The `shutil` module automates copying entire files or directories. This saves the steps of opening, reading, writing and closing files when there is no actual processing, simply moving files.

When we have complex programs that need to preserve a backup copy of a file or rename a file, we have two choices for our design.

- **Use Shell Commands**. We can exploit the shell commands of **cp** or **mv** (Windows: **copy** and **rename**). To do this, we have to break our processing down into tiny pieces, some of which are Python programs, and others are shell commands. We can use a shell script (or .BAT file) to jump back and forth between the Python steps and the shell command steps.
- **Use the shutil Module**. On the other hand, we can use `shutil` and do everything in Python, improving performance and simplifying the processing down to a single Python program.

`shutil.copy(`*source*, *destination*`)`
Copy data and mode bits, basically the GNU/Linux command **cp source destination**. If *destination* is a directory, a file with the same base name as *source* is created. If *destination* is a full file name, this is the destination file.

`shutil.copyfile(`*source*, *destination*`)`
Copy data from *source* to *destination*. Both names must be files.

`shutil.copytree(`*source*, *destination*`)`
Recursively copy the entire directory tree rooted at *source* to *destination*. *destination* must not already exist. Errors are reported to standard output.

`shutil.rmtree(`*path*`)`
Recursively delete a directory tree rooted at *path*.

`glob` The GNU/Linux shell expands wild-cards to complete lists of file names; the verb is *to glob* (really). The `glob` module makes the name globbing capability available to Windows programmers. The `glob` module includes the following function that locates all names which match a given pattern.

`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.

```
import glob, sys
for wildcard in sys.argv[1:]:
    for f in glob.glob(wildcard):
        process( f )
```

This can make 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.

`fnmatch` The `fnmatch` module has the essential algorithm for matching a wild-card pattern against file names. This module implements the Unix shell wild-card rules. These rules are used by `glob` to locate all files that match a given pattern. The module contains the following function:

`fnmatch.fnmatch(`*filename*, *pattern*`)` → boolean

Return `True` if the filename string matches the pattern string.

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
```

`fileinput` The `fileinput` module helps you read complex collections of text files in a relatively simple way. This is particularly helpful for creating grep-like processing, where your application reads all of the files in a large directory tree.

`filecmp` The `filecmp` contains a number of functions that help you build file comparison programs. This is handy for expanding on the basic **diff** program. It is also helpful for moving beyond simple file comparison into comparing two complete directory structures or comparing sections of complex documents.

12.3.3 Generic OS Services

This chapter describes a number of modules that are specifically designed to be the same in Linux, Mac OS and Windows. By using this module, you can be assured that your Python program will work the same everywhere.

`os` The `os` module contains an interface to many operating system-specific functions that manipulate processes, files, file descriptors, directories and other operating system resources. This module is specific to the operating system. 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 leave all pathname manipulation to `os.path`.

The `os` module exports a number of things. These constants are like variables, but changing their value will not have any beneficial effects on your program. The following definitions in this module provide useful information about the operating system.

`os.name`
: One of `POSIX`, `nt`, `dos`, `os2`, `mac`, or `ce`.

`os.curdir`
: String representing the current directory (`.`, generally)

`os.pardir`
: String representing the parent directory (`..`, generally)

`os.sep`
The (or the most common) pathname separator character (`/` generally, `\` on Windows). Most of the Python library routines will translate the standard `/` for use on Windows.

It is better to use the `os.path` module to construct or parse path names.

`os.altsep`
The alternate pathname separator (`None` generally, or `/` on Windows).

`os.pathsep`
The component separator used in `$PATH` (`:` generally, `;` on Windows).

`os.linesep`
The line separator in text files (the standard newline character, `\n`, or the Windows variant, `\r\n`). This is already part of the `readlines()` function and the `file` iterator.

`os.defpath`
The default search path that the operating system uses to find an executable file.

`os.chdir(`*path*`)`
Change the current working directory to *path*.

```
import os
os.chdir( "/Volumes/Slott02/Writing/Tech/PFNP/Notes" )
```

`os.getcwd()` → string
Return the current working directory path.

```
import os
print(os.getcwd())
```

`os.remove(`*filename*`)`
Delete (“remove”, “unlink” or “erase”) the file.

`os.unlink(`*filename*`)`
Delete (“remove”, “unlink” or “erase”) the file.

12.3.4 Data Persistentence

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 between streams of bytes (on a file) and Python objects. This is very nice for saving a Python object to a disk file.

`cPickle` Faster version of pickle, but you cannot subclass any of the classes, since it’s written in C, not Python.

`copy_reg` Register pickle support functions.

`shelve` Python object persistence.

`marshal` Convert Python objects to streams of bytes and back (with different constraints).

`sqlite3` This is a SQL-compatible relational database. It does a great deal and is very sophisticated.

Additionally, modules to access the widely-used DBM database manager is available.

12.3.5 Data Compression and Archiving

These modules can help work with populate file compression and archiving formats. These formats include `.zip` files as well as `.tar` files and `.gzip` and `.gz` files.

`zlib` A module for reading and writing data that has been compressed with the ZIP standard compression algorithms.

`gzip` A module for reading and writing data that has been compressed with the GNU ZIP compression algorithms.

`bz2` A module for reading and writing data that has been compressed with the GNU BZ2 compression algorithms.

`zipfile` A module for reading or creating a zip-format archive file.

`tarfile` A module for reading or creating a TAR-format archive file.

12.3.6 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 Comments (RFC's). The following modules are for handling Internet data structures. These modules and the related standards are beyond the scope of this book. We provide them as signposts so that you can research available modules and not reinvent each of these various wheels.

`email` An email and MIME handling package.

`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.

12.3.7 Python Runtime Services

There are a number of modules described in the Runtime Services section of the Library Reference. We want to emphasize just one, `sys`.

The `sys` module provides access to some objects used or maintained by the interpreter and to functions that interact with the interpreter.

Most importantly, the `sys` module provides access the three standard OS files used by Python.

`sys.stdin`
: Standard input file object; used by the `raw_input()` function. Also available via `sys.stdin.read()` and related methods of the file object.

`sys.stdout`
: Standard output file object; used by the `print()` function. 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.

This can be used as follows:

```
from __future__ import print_function
import sys

print("some error message", file=sys.stderr)
```

12.3.8 Files are the Plumbing of a Software Architecture

There are as many software architectures as there are architects. All of these architectures are collections of software components that are connected by files. As newbies, we can look at four common architectural variations. You're reading this book because you have a particular problem you want to solve. At this point, it may not be obvious how to get from the broadly-defined concept of *file* in the previous section down to the brass tacks of a working application program. This section will look at the ways in which applications are assembled from the available components and held together with files.

We'll look at the following architectural patterns to show you what kinds of file processing you'll need to do.

- **Command-Line Interface (CLI) Applications**. Sometimes these are called utilities, commands, filters, batch programs, or text-interface programs. These programs run from the command prompt or terminal tool with little fanfare.
- **Graphic User Interface (GUI) Applications**. Sometimes these are called fat-client programs or desktop programs. This includes word processors, spreadsheets, graphic programs, audio processing, video processing. Almost all desktop computer games fit into this category. These programs include rich user interaction.
- **Web Applications**. Sometimes these are called web sites. These programs work through a web browser; the application software is located on a web server, not on the user's desktop computer.
- **Embedded Controls**. Sometimes these are called real-time or programmed logic control applications. These programs control a device or system like a dishwasher, microwave oven, heat pump, robot or radar system.

Command-Line Interface Applications. The GNU/Linux world has hundreds, perhaps thousands of CLI applications. Windows also has a large number. Everything from the common **ls** command to more complex commands like **java** and **python** all work by reading and writing files. All of these command-line applications have some common features. These features are so important, that we'll devote all of *Fit and Finish: Complete Programs* to this subject.

There are a few central fittings to making a useful command-line application. An excellent example is the GNU/Linux **grep** program (or the Windows **find** program).

- The input comes from the standard input file. Additionally, the names of input files are provided as command-line parameters. Operating system redirection can make a disk file available on standard input.
- The results go to the standard output file and the standard error file. A program option can provide an output file name. Operating system redirection can direct standard output to a disk file.
- The application's behavior is tailored through options provided on the command line.

File operations you will use.

- Read characters from *sys.stdin*.
- Write characters to *sys.stdout* and *sys.stderr*.
- Create a `file` object, given the name of a disk file. Read or write characters using that file object.

Graphic User Interface Applications. GUI applications include **IDLE**, your favorite word processor, spread sheet and web browser. Most of what we use computers for are the GUI applications. In a few cases, the GUI application is a wrapper or veneer that surrounds and underlying command-line application.

There are a few central fittings to making a useful GUI application. An excellent example is **IDLE**.

- The input comes from files as well as the human user. The human user's input is handled by a sophisticated graphics library, like `pyGTK` or `Tkinter`. This library unifies mouse and keyboard events, and shares these devices politely with all other applications.
- The results go to files as well as the human user. Display to the user is handled by a graphics library. This library supports the broad variety of display devices, and shares this device politely with all the other applications.
- The application's behavior is controlled through interactive point-and-click. This is called an event-driven interface. The user's commands are events to which the application responds.

You will often create a `file` object, given the name of a disk file. After all, that's usually the point of using an application. The Python programming will read or write characters using that file object.

Since you're using the graphics library to interact with the mouse, keyboard and display devices, you won't use files for these user interaction devices directly.

Web Applications. You use a web application when you run a web browser like **FireFox**, **Safari**, **Chrome**, **Opera** or **Internet Explorer**. Your browser is a GUI application: it reads from the mouse and keyboard and displays back to the user. Browsers use sophisticated graphics libraries, some of which are highly tailored toward doing browsing.

More important, however, is the role the browser plays in the overall application. A browser application connects you with a web server. When you request a web page (by typing the URL or clicking on a link), your browser makes a request from a web sever. When you fill in a form and click a submit (or search or buy now) button, you are making a request of a web server.

Writing a web application means putting the right programming on a web server. Web programming happens in a variety of forms, and uses a number of different languages. The reason for the complexity of web applications is to spread out the workload and allow a large number of people to make requests and efficiently share the web server.

The core of web applications is the HTML language. When you make a web request, the reply is almost always a page of HTML. Your web browser opens a kind of file called a socket. The browser writes the request, and then reads the reply. The reply will be HTML which is *rendered* and presented as "page" of content.

Serving Web Content. On the other side of the web transaction, the web server is waiting for requests from browsers. The server reads the request, locates the content, and sends the HTML page to the browser. The browser will also request the various pieces of "media" (graphics, sounds, etc.), which are sent separately.

Some HTML pages are *static*, which means that the web server takes an HTML file from the disk and sends it through the internet to your browser. This job is very simple and easily standardized. A program named **Apache httpd** handles this job very nicely.

Some HTML pages are *dynamic*, which means that some program created customized HTML, and sent this through the internet to your browser. Often, this program will be a partner with **Apache httpd**. Generally, you'll simplify your life by using a web framework for this kind of programming.

File operations you might use in a web program.

- Create a `file` object, given the name of a disk file that exists on the web server.
- Read files that are located on a web server.

- Open a "file" that connects to yet another webserver and get data from another server to prepare data for presentation on a web page.

You don't have access to the user's computer or anything on the user's computer; only the browser can do that. All of your file operations are confined to your web server. You can, through HTML, make it easy for someone to download files to their desktop computer, but you have no direct access.

The general approach is to use any of the Python web-frameworks. You can research Django, TurboGears, Quixote and Zope to see a spectrum of just a few alternatives. There are dozens of frameworks to help you manage these popular kinds of applications.

Embedded Control Applications. Let's imagine that we are inventing a new kind of heat pump controlled by a computer. We've bought our heating and refrigeration coils, we've got a reversing valve and a variable-speed motor. We've rigged up a working set of hardware in our garage, but we need a computer and software to control all of this hardware.

We'll need to create interfaces that transform information from the outside world like temperature, pressure, valve position, motor speed into electronic signals the computer can read. We'll also need to transform electronic signals into actions like starting a motor or changing a value position. We need to purchase and configure the necessary computer parts. We also need to write *device drivers*.

Our device drivers are the glue that connects our file system to our temperature probes, coolant pressure sensors, valve position sensor and motor speed indicator. Each of these devices can appear as a file. When we read from the temperature file, for example, our driver uses this request to gather information from the thermistor, encode that as a number, and provide this number to our program.

While there's a large amount of computer engineering involved, you will still use some standard file operations. You will create a `file` object, given the name of a device which appears as a file. You will read or write data using that file object.

CHAPTER

THIRTEEN

DATA + PROCESSING = OBJECTS

One of the most powerful and useful features of Python is its ability to define new classes of data. The next chapters will introduce the *class* definition and the basics of object-oriented programming.

Objects: A Retrospective reviews the key features of the objects we've used so far. Having looked at what we've already learned, we can then introduce the basics of how we define the class of an object in *Defining New Objects*.

13.1 Objects: A Retrospective

To make sense of class definitions, we'll talk about objects in *The Ubiquitous Object*, and review the built-in object classes in *The Built-in Classes – A Review*. In *Data, Processing and Philosophy – What Does It All Mean?* we define the basic semantics of objects and the classes which define their structure and behavior.

13.1.1 The Ubiquitous Object

Our programs create and manipulate data objects. It turns out that all of Python programming boils down to this one theme: a program creates and manipulates objects. After the previous chapters in which we worked with objects we can meaningfully define what an object is. This chapter will show how we can define our own new, unique classes of objects.

Each piece of data has properties that we use to typify or classify the data object. Each object has data and processing, which we can call the object's *attributes* and *operations*.

In the Python language, we write some operations using operators, like `+` and `*`. We write other operations as method functions, like a String's `someString.lower()` method. And some operations are functions in prefix notation like `len(someString)`.

Under the hood, all operations are implemented by *method functions*. These functions have generic names but implementations which are specific to each type of data. The method that performs the `*` operation for a number is different from the method that performs the `*` for a list. `2*3` and `2*["red",21,2.7]` have very different results, which depend on the type of data involved in the operation.

Each type of data, from a simple boolean (like `True`) to a complex file (created with the `file()` factory function), has attributes and operations.

- For the simple, unstructured types, there is only one attribute and that is the value. Booleans and numbers are these kinds of simple, unstructured types. How many attributes can the value `True` or `3.1415926` have?
- A more complex type, like a list, will have a collection of items, plus attributes like the length, and the list's unique hash code. A collection will have method functions to access the collection. Additionally,

a mutable collection may have method functions to change the collection by adding and removing elements.

- A really complex type, like a file, has many attributes, some of which come from outside the Python environment. Attributes include a name, a modification time, a size, permissions. A file is associated with operating system resources, and a file's operations will move data to or from external devices.

Each object is an instance of a *class*. A class defines the attributes and operations of each object that is a member of the class. We'll use the word type and class interchangeably.

A typical program will written as a number of class definitions and a final main function. The main function's job is to create the objects required to perform the job of the program. The program's behavior is the result of interactions among these objects. This parallels the way that a business enterprise is the net effect the interactions among the people who purchase materials, create products, sell the products, receive payment and manage the finances.

13.1.2 The Built-in Classes – A Review

In *Getting Our Bearings*, we looked around at where we'd been and where we were going. In that section, we reviewed the basic statements and data types of Python. Since we're rounding another mark, it's time to get our bearings again, and see what the next leg of our course looks like.

Because it's easiest to learn by doing, we've been using a number of built-in object classes. Here are the types of data we've seen so far.

- **None**. A unique constant, handy as a placeholder when no other value is appropriate. A number of built-in functions return values of `None`. The `None` literal is the only instance of a special class, `NoneType`, that has no attributes and a very limited number of operations.

  Since there's only a single instance of `None`, we compare a variable against the `None` object with the **is** operator.

- **NotImplemented**. A unique constant, returned by special methods to indicate that a method is not implemented. This allows Python to try alternative methods if they're available. The `NotImplemented` literal is the only instance of a special class, `NotImplementedType`.

- **Numeric**. The various numeric types have relatively simple, unstructured values. For obvious reasons, these are all immutable.

  – **Boolean (bool)**. This type has a tiny domain with just two literal values: `False` and `True`. A number of other values are equivalent to these two values. There is also a tiny domain of operations, including **and**, **or** and **not**. Some other operators (like the comparisons) produce boolean result values.

  – **Integer or Whole Numbers (int)**. The literal values are written as strings of digits. These values have a number of operations, including arithmetic operations, special bit-fiddling operations and comparison operations.

  – **Long Integers (long)**. These are integers of arbitrary length. They grow as needed to precisely represent numeric results. The literal values are written as strings of digits ending with `L`. These values have a number of operations, including arithmetic operations and comparison operations.

  – **Floating-Point or Scientific Notation (float)**. These are numbers coded as a fractional "mantissa" and an exponent. Scientists and engineers use powers of 10, as in 6.022×10^23. The Python language abbreviates the "$\times$ 10" with the letter `E` or `e`. The literal values are strings of digits (with a decimal point) and an optional E or e exponent, for example. `6.022e23`.

    Most computer processors use a notation based on powers of 2, so ranges and precisions vary. Typically these are called "double precision" in other languages, and are often 64 bits long. These values have a number of operations, including arithmetic operations and comparison operations.

- **Complex (complex)**. These are a pair of floating-point numbers of the form '`(a + bj)`', where *a* is the real part and *b* is the "imaginary" part. These values have a number of operations, including arithmetic operations and comparison operations.
- **Sequence**. The sequence types are collections of objects identified by their order or position, instead of a key. All sequences have a few operations to concatenate and repeat the sequence. Sequences have `in` and `not in` operations to determine if an item is part of the sequence. Additionally sequences have the `[]` operation which selects an item or a slice of items.
  - Immutable sequences are created as need and can be used but never changed.
    * **String (str)**. A string is a sequence of individual ASCII characters. Strings have a number of operations that return facts about the string or transform the string and create a new string.
    * **Unicode (unicode)**. A Unicode string is a sequence of individual Unicode characters. Unicode strings have a number of operations that return facts about the string or transform the string and create a new string.
    * **Tuple (tuple)**. A tuple is a sequence of Python items. It has a few operations for accessing individual items in the tuple.
  - Mutable sequences can be created, appended to, changed, and have elements deleted.
    * **List (list)**. A list is a sequence of Python items. Operations like `append()` and `pop()` can be used to add or remove from a lists. Operations like `sort()` can change the order of the list.
- **Set**. A set is a simple collection of objects. There is no ordering or key information. This makes them very efficient. Sets have `add()` and `remove()` operations, as well as `in` and `not in` operations.
- **Mapping**. A mapping is a collection of objects identified by keys instead of order.
  - **Dictionary (dict)**. A dictionary is a collection of objects (values) which are indexed by other objects (keys). 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. Dictionaries have the `[]` operation to select an element from the dictionary. Dictionaries have methods like `has_key()` to determine if a key is present in the dictionary. Dictionaries also have methods like `items()`, `keys()` and `values()` to produce sequences from the contents of the dictionary.
  - **Default Dictionary**. We had to import this from the `collections` package. The `defaultdict` behaved just like a dictionary in every respect but one. When we attempt to get a value that's not in the dictionary, it evaluates a default function.
- **Callable**. When we create a function with the **def** statement, we create a `callable` object. There are a number of attributes; for example, the *___name___*, and *func_name* attributes both have the function's name. There is one important operation, "calling" the function. That is, performing the eval-apply cycle (see *The Evaluate-Aply Rule* for a review) to the function's argument values.
- **File (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.

13.1.3 Data, Processing and Philosophy – What Does It All Mean?

Beginning in *Instant Gratification : The Simplest Possible Conversation* we've been creating, manipulating and accessing Python objects without asking the deep, philosophical question "What is an object?"

As with other real-world things, it's easier to provide a lot of examples than it is to work up an elaborate, legalistic definition. Objects are like art: I can't define it, but I know what I like. As hard as it is, we'll give the definition a whirl, because it does help some people write better software.

Each object encapsulates both data and processing into a single definition. We'll sometimes use synonyms and call these two facets *structure* and *behavior*, *attributes* and *operations* or *instance variables* and *method functions*. The choice of terms depends on how philosophical or technical we're feeling. The structure and behavior terms are the most philosophical; the attribute and operation terms are generic object-oriented design terms. Instances variables and method functions are the specific ways that Python creates attributes and operations to reflect structure and behavior.

In Python, we can understand objects by looking at a number of features, adapted from [Rumbaugh91].

- **Identity**. An 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 can be identified by their address. Unless we do something special, the built-in `id()` function gives us a hint about the memory location of an object, revealing the distinction between two objects. We can see this by doing `id("abc"), id("defg")`, which shows that two distinct objects were being examined.
- **State**. Many objects have a state, and that state is often changeable. The object's current state is described by its *attributes*, implemented as instance variables in Python.

  Our two nearly identical coffee cups have distinguishing attributes. The locations (back-left corner of desk, on the mouse pad) and the ages (yesterday's, today's) are attributes of each cup of coffee. I can change the location attribute by moving a cup around. Even if both cups are on the back-left corner, the cups have unique identity and remain distinct. I can't easily change the age; today's coffee remains today's coffee until enough time has passed that it becomes yesterday's coffee.

  In software world, my two strings ('`"abc"`' and '`"defg"`') have different attribute values. Their lengths are different, they respond differently to various method functions like `upper()` and `lower()`.

  As a special case, some objects can be stateless. While most objects have a current state, it is possible for an object to have no attributes, making it like a function. Such objects have no *hysteresis* – no memory of any previous actions.
- **Behavior**. Objects have behavior. The object's behavior is defined by its *operations*, or, in Python terminology, its method functions. Some objects can be termed "passive" because they are used by other objects, and don't do much processing. Some objects can be termed "active" because they do considerable processing. These distinctions are arbitrary, some objects have passive and active methods.

  A coffee cup really only has a few behaviors: it admits additional coffee (to a limit), it stores a finite amount of coffee, and coffee can be removed. Coffee cups are passive and don't initiate these behaviors. The coffee machine, however, is an active object. The coffee machine has a timer, and can perform its behavior of making coffee autonomously.

  String objects have a large number of behaviors, defined by the method functions, many of which we looked at in *Sequences of Characters : str and Unicode*. All of our collection classes can be considered as passive objects.
- **Classification**. Objects with the same attributes and behavior belong to a common *class*. Both of our string objects (`"abc"` and `"defg"`) belong to a common class because they have the same attributes (a string of characters) and the same behavior.
- **Inheritance**. A class can inherit operations and attributes from a parent class, reusing common features. A *superclass* is a generalization. A *subclass* overrides superclass features or adds new features, and is a specialization.

Both of our coffee cups are instances of cup, which is a subclass of a more general class, "drinking vessel". This more general class includes other subclasses like glassware and stemware.

When we described the string data type, we put it into a broader context called `sequence` and emphasized the common features that all sequence types had. We also emphasized the unique features that defined the various subclasses of sequence. All of the sequence types have the `[]` operator to select an individual item. Only strings, however, had an `upper()` method function. Only lists had the `append()` method function.

- **Polymorphism**. A general operation, named in a superclass, can have different implementations in the various subclasses. 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. Because objects of these distinct classes respond to a common operator, they are polymorphic.

Program Design. Up to this point in our programming career, we've been looking at our information needs and the available Python structures. If it was a temperature, we used a number; for the color of a space on the Roulette wheel, we used a string. In the case of something more complex, like a pair of dice, we used a function which created a tuple.

As we become more sophisticated, we begin to see that the various types of data that are built-in to Python aren't exactly what we need. It isn't possible to foresee all possible problems. Similarly, it isn't possible to predict all possible kinds of data and processing that will be required to solve the unforeseeable problems. That's why Python lets us define our own, brand-new types of data.

Class Definition. Python permits us to define our own classes of objects. This allows us to design an object that is an exact description of some part of our problem. We can design objects that reflect a pair of dice, a Roulette wheel, or the procedure for playing the game of Craps. A class definition involves a number of things.

- The name of the new class.
- An optional list of any classes that are the basis for this class definition. If there are any, we call these other classes the *superclasses* for our new class. Generally, we'll use the class `object` as the superclass for our class definitions.
- All of the method functions for this new class. Each method is, in effect, another function of this class. Defining a method function, is just like defining a function, and involves three things.
  - The name of the method function.
  - A list of zero or more parameters to this function. In order to identify the specific object instance, all method functions have one mandatory parameter.
  - A suite of statements for this method function.

The object's attributes (also called instance variables) are not formally defined as part of the class. They are generally created by a special method function that is executed each time an object is created. This initialization method function is allocated responsibility for creating the object's instance variables and assigning their initial values.

Object Creation. After we define the class, we can create instances of the class. Every object is in instance of one of more classes. Each object will have unique identity; it will have a distinct set of instance variables; it will be identified by a unique object identifier. Objects have an internal state, defined by the values assigned to the object's instance variables. Additionally, each object has behavior based on the definitions of the method functions. An object is said to *encapsulate* a current state and a set of operations.

Because every object belongs to one or more defined classes, objects share a common definition of their attributes and methods. The class definition can also specify superclasses, which helps provide method functions. We can build a family tree of classes and share superclass definitions among a variety of closely-related subclasses.

It helps to treat each class definition as if the internal implementation details where completely opaque. A class should be considered as if it were a *contract* that specifies *what* the class does, but keeps private all of the details of *how* the class does it. All other objects within an application should use only the defined methods for interacting with an object. When we use a list's `append()` method, we know what will happen, but we don't know precisely how the list object adds the new item to the end of the list. Unlike Java and C++, Python has a relatively limited mechanism for formalizing this distinction between the defined interface and the private implementation of a class.

Life Cycle of an Object. Each object in our program has a lifecycle. The following is typical of most objects.

- **Definition**. The class definition is read by the Python interpreter or it is built-in to the language. Class definitions are created by the **class** statement. Examples of built-in classes include files, strings, sequences, sets and mappings. We often collect our class statements into a file and **import** the class definitions to a program that will use them.
- **Construction**. An object is constructed as an instance of a class: Python allocates memory that it will use for tracking the unique ID of the object, storing the instance variables, and associating the object with the class definition. An `__init__()` method function is executed to initialize the attributes of the newly created instance.
- **Access and Manipulation**. The object's methods are called (similar to function calls we covered in *Better Arithmetic Through Functions*) by client objects, functions or scripts. There is a considerable amount of collaboration among objects in most programs. Methods that report on the state of the object are sometimes called *accessors*; methods that change the state of the object are sometimes called *manipulators*.
- **Garbage Collection**. Eventually, there are no more references to this instance. For example, consider a variable with an object reference which is part of the body of a function. When the function finishes, the variable no longer exists. 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.

Important: Class and Instance

Once we've defined the class, we only use the class to make individual objects. Objects – instances of a class – do the real work of our program.

When we ask a string to create an upper case version of itself (`"hi mom".upper()`), we are asking a specific object (`"hi mom"`) to do the work. We don't ask the general class definition of string to do this. The meaning of `str.upper()` isn't very clear.

This can be a little mystifying when we start to define our own classes. The problem usually stems from confusing class definitions with function definitions. We don't use *instances* of a function for anything, we use the function itself. Functions, consequently, are a bad model of how class definition works. Classes are a kind of factory for creating objects. Objects do the real work.

The most important examples to keep in mind are string objects, file objects and list objects. These are the most typical examples of the kinds of objects we'll create. Each string (or file or list) object is an instance of the respective class definition.

Under the hood, the definition of a class creates a new class object. This class object is used to create the instance objects that do the work of our program. The class object is mostly just a container for the suites of statements that define of the method functions of a class. Additionally, a class object can also own class-level variables; these are, in effect, shared by each individual object of that class. They become a kind of semi-global variable, shared by objects of a given class.

Garbage Collection

It is important to note that Python counts references to objects. When object is no longer referenced, 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. 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. When we look back at the processing, we see that the string "456" was never referenced by any variable. These objects will be destroyed as part of garbage collection.

13.1.4 Object and Class Exercises

1. **Object Identification**.

   When we evaluate an expression as simple as 3+5, Python creates an integer object with a value of 3, an integer object with a value of 5, then applies a method function to add these values, and create a new object which is the sum.

   Look at some of your earlier exercises in *Arithmetic and Expressions* and identify all of the objects in a given expression. Pay particular attention to each operator (like `+`, `-`, `*` or `/`) which will create a new object.

   Since `()` merely group expressions, do they create new objects?

2. **Iterator Objects**.

   In *While We Have More To Do : The for Statement* we looked at the **for** statement. In *Basic Sequential Collections of Data* we looked at how the **for** statement iterates through a sequence. In *Looping Back : Iterators, the for statement and Generators* we looked at the iterator and how the **for** statement makes use of this iterator. A sequence has a method function (iter) which creates the iterator which yields each item of the sequence so the **for** statement can assign them to a variable.

   When you evaluate `iter( [ 1,2,3] )`, for example, you can see the iterator object being created. This iterator object has a cryptic-looking name, for example, `<listiterator object at 0x107afd0>`.

   Each time you evaluate something like `iter( [ 1,2,3] )` you get a slightly different response. Does this indicate that a new object being created? Does this make sense? If each object can capture a unique state, does this mean each iterator is independent?

   If we create a single list, for example `a=range(100)`, can we have multiple iterators which provide different views of the same list object, *a*?

3. **Temporary Objects in Functions**.

   When a function is being evaluated, objects will be created by each operation in each expression. What happens to those objects? Does any object persist after the function's evaluation is complete? What happens to the object created by (or named in) the **return** statement?

Look at some of your earlier exercises in *Organizing Programs with Function Definitions*. Identify the life-span of all objects created by a specific function.

13.1.5 Class FAQ's

Why define classes? Isn't it simpler to have a group of related functions? In a sense, a class is a group of related functions. However, the formal class definition allows you do some additional things that would be inconvenient with a group of related functions.

First, the class acts like a common name for the functions, saving you from have to write elaborate prefixes on your functions. For example, if you had a group of functions to work with a block of stocks, you might prefix each name with `sb_` to assure that each function name was unique. This is error-prone and tedious.

Second, the class allows easy sharing of instance variables. All of the instance variables are collected under `self`, assuring that they are available to each function.

Third, and more important than these technical considerations, is the mental tools of abstraction and encapsulation. When we define classes, we encapsulate some functions so we can set aside the details. This allows us to reduce the complexity of a class to a somewhat simpler abstraction, namely, the functions that interface to the class, not the details of how the class works internally. Thinking at this more abstract level lets us wrap our finite brains around slightly larger problems.

Can I use something shorter than `self` ? Yes and no. Yes, the language allows any variable name for the *self variable*. No, if you do, you will find it hard to share your Python programs with other people. The name *self* is very well established. A number of tools expect it. Most importantly, the other Python programmers from whom you may get software will expect it, also.

What is the advantage of having objects collaborate? Our first programs tended to have a single, long procedure that made use of many objects. Once we've got the hang of programming, we can break that long procedure down into separate pieces, assign each piece to a method function of an object. Breaking long procedures down and allocating them to separate objects is part of our divide and conquer life-style.

As we get more proficient with object definition, we can examine a programming problem by writing a description and looking at the nouns and verbs in that description. The nouns will become objects, defined by classes. The verbs will become method functions. This collaboration among the nouns will fit naturally with the original description of the problem, giving us confidence that our program will work.

13.2 Defining New Objects

In *Class Definition: The class and def Statements* we show the syntax for creating class definitions; we cover the use of objects in *Class Use: Making New Objects*. We'll discuss the notion of attribute or instance variable in *The State of Being – Instance Variables*. The initial state of an object is set by a special method that we'll look at in *At The Starting Line – Setting The Initial Values*. We provide some exercises in *Class Definition Exercises*.

13.2.1 Class Definition: The class and def Statements

We define our own class of objects with **class** statement. Since the class encapsulates instance variables as well as method functions, a class definition can get lengthy.

Here's the simplest form for a class definition.

```
class className :
    suite of method defs
```

In Python 3, this will be the norm.

While we're using Python 2, we're going to use this slightly more complex version.

```
class className ( [ superclass ] , ... ) :
    suite of method defs
```

The *className* is the name of the class. This name will be 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.

We'll generally provide a superclass of `object` in Python 2. This provides some benefits that – while important – are also beyond the scope of the book.

The *suite of defs* is a series of definitions for the method functions of the class. This is indented within the **class** definition.

The suite of defs can contain any Python programming. Generally, we try to limit our class definition to the following things:

- 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.
- Method function definitions.
- Sometimes, we may provide class-wide "constants" – variable definitions that provide a handy short hand name for a value that doesn't change.

The heart of the class definition is the suite of method function definitions.

```
def  methodName  ( self, [ parameter [ = initializer ] ]  , ...  ):
    suite of statements
```

This definition looks just like a function definition, with two exceptions.

First, it's indented within the **class** statement suite.

Second, each of the method functions must have a first positional argument, *self*, which Python uses to manage the unique object instances. When referring to any method function or instance variable of the class, the instance qualifier *self.* must be used.

Example Definition. Here's an example of a class with a single method definition. This class models a real-world object, a die. Note the indentation of the class definition suite. Each method function **def** begins at one level of indentation; each method function's suite is at a second level of indentation.

die.py

```
1 #!/usr/bin/env python
2 import random
3 class Die( object ):
4     """Simulate a 6-sided die."""
5     def roll( self ):
6         """Return a random roll of a die."""
7         u= random.randrange(6)
8         self.value= u+1
9         return self.value
```

2. We imported the `random` module to provide the random number generator.

3. We defined the simple class named `Die`. The indented suite contains the docstring and the single method function of this class.
4. The docstring has a pithy summary of the class. As with function docstrings, the class docstring is retrieved with the `help()` function.
5. We defined a single method function: `roll()`. The instance variable, *self*, provides access to any variables that are created in an instance of this class. All functions that are part of a class are provided with the instance reference as the first positional parameter.
7. When this method is executed, is sets a local variable, u, to a random value between 0 and 5. Since this variable has no instance qualifier, it is local to the function and will vanish when the function finishes.
8. The next statements sets a instance variable, *self.value*, to the random value plus 1. Since *self.value*, is qualified by the instance, the variable is part of the state of an object and lives as long as the object does.

The processing steps are silly, but they shows the difference between a local variable, u, that doesn't live with the object, and an instance variable, *self.value*, that defines the state of the object.

Tip: Debugging a Class Definition

When we get syntax errors on a class definition, it can be in the class line or one of the internal method function definitions.

If we get a simple `SyntaxError` on the first line, we have misspelled **class**, left off a `(` or `)`, or omitted the `:` that begins the suite of statements that defines the class.

If we get a syntax error further in the class definition, then our method functions aren't defined correctly. Be sure to indent the **def** once (so it nests inside the class). Be sure to indent the suite of statements inside the **def** twice.

13.2.2 Class Use: Making New Objects

When we use the class name as if it were a function evaluation (for example, `d= Die()`), three things will happen.

- A new object is created. The object is given a reference (named ___*class*___) to to its class definition. This `d.__class__` instance variable makes `d` an instance of a class. In this case, the class is `Die`.
- If the class defines the `__init__()` method function, this is evaluated. Typically, this will initialize other instance variables.
- The resulting object will be saved in variable *d* for later use.

Note the similarity between using our class definition as an object factory and the built-in factory functions like `int()`, `float()`, `bool()`, `str()`, `list()`, `tuple()`, `set()` and `dict()`. A class name is also the name of an object factory.

Let's create and interact with two objects of our `Die` class. First, we'll execute the class definition module using **IDLE**s `F5` or **Run** menu, item **Run Module**.

If we are working from the command line, or using a different tool, we have to import our definitions, using `from die import *`.

```
>>> from __future__ import print_function
>>> from die import *
>>> d1= Die()
```

```
 4  >>> d2= Die()
 5  >>> print(d1.roll(), d2.roll())
 6  1 3
 7  >>> print(d1.value, d2.value)
 8  1 3
 9  >>> print(d1, d2)
10  <die.Die object at 0x7fd30> <die.Die object at 0x7fd50>
```

3. We use our `Die` class to create two variables, *d1*, and *d2*; both are new objects, instances of `Die`.

5. We evaluate the `roll()` method of *d1*; we also evaluate the `roll()` method of *d2*. Each of these calls sets an 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 evaluate `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 evaluate the `roll()` method.

9. We also ask for a representation of each object. Unless we provide a method named `__str__()` in our class, this is what Python reports. Note that the numbers are different, indicating that these are distinct objects, each with private instance variables.

Note that we used the class definition to make two objects, *d1*, and *d2*. The objects are the focus of our program. We have manipulators (like the `roll()` method) and accessors (the *value* attribute) for these objects.

Tip: Debugging Object Construction

Assuming we've defined a class correctly, there are a three of things that can go wrong when attempting to construct an object of that class.

- The class name is spelled incorrectly.
- You've omitted the `()` after the class name. If we say `d= Die`, we've assigned the class object, `Die`, to the variable *d*. We have to say `d= Die()` to use the class name as a factory and create an instance of a class.
- You've got incorrect argument values for the parameters of the `__init__()`.

If we get a `NameError: name 'Hack' is not defined`, then the class (`Hack`, in this example) is not actually defined. This could mean one of three things: our class definition had errors in the first place, our definition class name isn't spelled the same as our object creation (either we spelled it wrong when defining the class, or spelled it wrong when using the class to create an object.) The third possible error is that we have defined the class in a module, imported it, but forgot to quality the class name with the module name.

If our class wasn't defined, it means we either forgot to define the class, or overlooked the `SyntaxError` when defining it. If our class has one name and our object constructor has another name, that's just carelessness; pick a name and stick to it. If we are trying to import our definitions, we can either qualify the names properly, or use `from module import *` as the import statement.

Another common problem is using the class name without `()`s. If we say `d= Die`, we've assigned the class object (`Die`) to the variable *d*. We have to say `d= Die()` to create an instance of a class.

If we've defined our class properly, we can get a message like `TypeError: __init__() takes exactly 2 arguments (1 given)` when we attempt to construct an object. This means that our `__init__()` method function doesn't match the object construction call that we made.

The `__init__()` function must have a *self* parameter name, and it must be first. When we construct an object, we don't provide an argument value for the *self* parameter, but we must provide values for all of the other parameters after *self*.

If your initialization function, `__init__()`, doesn't seem to work, the most likely cause is that you have misspelled the name. There are two _ before and two _ after the `init`.

13.2.3 The State of Being – Instance Variables

Each ordinary method function definition must have the instance qualifier – traditionally the variable *self* – as the first positional parameter. This name qualifies the instance variables and the method functions of this object.

Note: Yes, there are exceptions

The exceptions to using the `self` instance variable are beyond the scope of this book.

We'll see two kinds of references to variables and functions in the suites of statements in a class.

- **Names qualified by `self`**. When we say `self.name`, the variable *name* is *bound* to this object. These variables are part of the object, and have the same life as the object. The variable exists after the end of any method function evaluation.

  Similarly, the name of a function that is qualified with *self* refers to a method function that is part of this class definition.

- **Unqualified names**. Names not qualified by `self` are called *free*. These variables are ordinary local variables that has a scope that is tied to this execution of the method function. When the function finishes, the variable will be removed.

  Similarly, the name of a function that is not qualified refers to a free function that is defined outside this class.

  A free variable may also be a reference to a global variable or function, or a builtin function.

In the following example, the method function `rollMany()` evaluates `self.roll()`. The `self` qualifier shows that the `roll()` function is part of the `Dice` class.

The method function `roll()` evaluates `random.randrange()`. Since this does not use the `self` qualifier, it is defined outside this class definition.

```
class Die( object ):
    """Simulate a 6-sided die."""
    def roll( self ):
        """Return a random roll of a die."""
        u= random.randrange(6)
        self.value= u+1
        return self.value
    def rollMany( self, n ):
        all= [ self.roll() for i in range(n) ]
        return tuple(all)
```

13.2.4 At The Starting Line – Setting The Initial Values

We've emphasized that the behavior of each object is declared through the method functions of the object's class. The method functions are a formal contract between the object and its client objects, specifying what the object does.

The attributes, however, do not have formal definitions. Each object's attributes are implemented through instance variables, which – like all Python variables – are created as needed by an assignment statement. In order to guarantee that all of the instance variables exist during the entire life of the object, it is best to initialize them by providing a method with the special name of `__init__()`. The `__init__()` method is always called automatically by Python when the object is created; we can exploit this to assure a correct initialization.

In this example, we updated our `Die` to add an `__init__()` function. This function will provide a default value for the *self.value* attribute.

die.py, version 2

```
import random
class Die( object ):
    """Simulate a 6-sided die."""
    def __init__( self ):
        """Initialize the die."""
        self.value= None
    def roll( self ):
        """Return a random roll of a die."""
        self.value = random.randrange(6) + 1
        return self.value
```

Bonus Questions. In the first version of `Die`, what would happen if we did the following?

```
dx = Die()
print(dx.value)
dx.roll()
print(dx.value)
```

Compare this with what happens when we do this with the new version of `Die`. Which class has better behavior?

Arguments to Control Initialization. Method functions can have parameters. All of the techniques we've seen for ordinary function definitions apply to method functions. We can have additional positional parameters after *self*, keyword parameters, default values, as well as the `*` and `**` collections of additional parameters.

As with all method functions, the `__init__()` method function can accept parameters. This allows us to correctly initialize an object at the same time we are creating it. The object can begin its life in a specific state. Since we don't call the `__init__()` function directly, this raises a question. How are argument values assigned to the parameter variables?

The class name becomes a factory function that makes new instances of the class. When we evaluate the class, using `()`s, we can pass argument values to the class factory. The argument values we give to the class factory are given to the `__init__()` method function.

For any class, `C`, if we say `a= C( some values )`, Python acts as though we said

```
a= C()
a.__init__(  some values )
```

Example Class Definition. This next example is a class that defines a geometric point. The class provides some operations that manipulate that point. When we create a Point instance, we'll provide an x and y coordinate. To define the point (x,y)=(3,2), we could say `Point(3,2)`. This would, in effect, do the following for us `p= Point(); p.__init__( 3, 2 )`.

```
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 )
```

Here's an example of creating a Point at coordinates (2,3) via `Point(2,3)` and then manipulating that point. First we move it -1 unit on the x axis and 2 units on the y axis. Then we move it -2 on both axis.

```
>>> from point import Point
>>> p = Point(2,3)
>>> print(p)
<point.Point instance at 0x98d148>
>>> print(p.x, p.y)
2 3
>>> p.offset( -1, 2 )
>>> print(p.x, p.y)
1 5
>>> p.offset2( -2 )
>>> print(p.x, p.y)
-1 3
```

After using the `offset()` and `offset2()` manipulations, the point is now at (-1,3).

Other Special Names. In addition to the specially-named `__init__()` method, there are many other specially-named methods that are automatically used by Python; these special methods can simplify our programming. After `__init__()`, the next most important special method function name may be `__str__()`.

The `__str__()` method is used to return the string representation of the object. For example, we can add this method to our `Point` class to return an easy-to-read string for a `Point`.

```
class Point( object ):
    # ...other methods...
    def __str__( self ):
        return "({0:d},{1:d})".format(self.x, self.y)
```

Don't Forget `self`. 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 the supplied value for both x and y offsets.

The `self` variable

Programmers experienced in Java or C++ sometimes object to seeing the explicit *self.* in from of variable names or function names. In Java and C++, there is a *this.* qualifier which is assumed by the compiler. In Java, it must sometimes be used explicitly for disambiguation of names. In Python, the late binding and dynamic creation of attributes makes it too difficult to *assume* a qualifier on any particular name. Rather than make assumptions, Python asks you to identify the instance variables with *self.*.
When we talked about function definitions in *Keeping Track of Variable Names – The Namespace*, we talked about the local namespace that contained the function's variables. Each method function follows this rule and has a local namespace. This allows us to use variables in a method functions, confident that they won't conflict with any other method function, free function or global variable. An object's instance variables are tucked away in a namespace called *self*.

13.2.5 Operations – Access and Manipulation

The method functions allow us to access and manipulate the instance variables of an object. The method functions create a formal interface for using the object. We can think of the method functions the way we think of the buttons on the front panel of a microwave oven. We don't know what goes on inside the oven, but we do know that pushing certain buttons in a certain order will reheat our left-over General Tso's Chicken.

Let's look at an example of using our `Die` class.

```
>>> d1= Die()
>>> d1.roll()
1
>>> d1.roll()
2
```

In this case, we created an object, *d1*, which is defined by the `Die` class. When we say `Die()`, we are creating a new object, and implicitly evaluating `Die.__init__()` to initialize that object.

After creating an instance of `Die`, we then evaluated the `roll()` method of that instance. This method updates the instance variables, *self.value*, with a new random number. It also returns the value of the instance variable.

A method which returns information without changing any of the instance variables is sometimes called an *accessor*. A method which changes an instance variable is sometimes called a *manipulator*.

Tip: Debugging Class vs. Object Issues

Perhaps the biggest mistake newbies make is attempting to exercise the method functions of a class instead of a specific object. You can't easily say `Die.roll()`, you'll get the cryptic `TypeError: unbound method` error message. The phrase "unbound method" means that no instance was being used.

When you say `d1= Die()`, you are creating an instance. When you see `d1.roll()`, then you are asking that specific object to do its `roll()` operation.

13.2.6 Politics: Collaboration and Responsibility

The real work of a program occurs when objects collaborate. We define classes that depend on other classes; we create multiple instances of objects; objects evaluate method functions of other objects.

What we do to build up an application is to allocate specific areas of responsibility to different classes. We implement each class so that it handles just its narrow area of specialization. Each class has a formalized contract with other classes, allowing us to develop and debug each class as a separate, smaller and more manageable exercise.

We'll start out by creating a `tuple` object that contains five instances of the `Die` class from the `die.py` module. We'll use that tuple object to generate a dozen rolls of these five dice. This is the kind of thing that might form part of a simulation for multi-dice games like Yacht, Kismet, Yatzee, Zilch, Zork, Greed or Ten Thousand.

```
1  from __future__ import print_function
2  import die
3  my_dice= ( die.Die(), die.Die(), die.Die(), die.Die(), die.Die() )
4  for i in range(12):
5      for d in my_dice:
6          d.roll()
7      print([d.value for d in my_dice])
```

2. We imported our `die.py` module to get the `Die` class definition.
3. We created a `tuple`, *my_dice* with five distinct objects, each an instance of `die.Die`.
4. Within the outer **for** loop, we used an explicit loop to iterate through the `Die` instances in the *my_dice* variable.

   We evaluated the the `roll()` method of each individual `die.Die` instance.
7. To print the results, we used a list comprehension to collect the values of the individual `Die` instances.

`Dice` and `Die` Collaboration. Here's a new class, `Dice`, which uses instances of our `Die` class. We'll put this into the `die.py` file, right behind our `Die` class definition. This leads to a file that is

die.py, version 3

```
import random

class Die( object ):
    ... already given ...

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.value
        return t
    def getTuple( self ):
        "Return a tuple of the dice values."
        return tuple( [d.value for d in self.myDice] )
    def hardways( self ):
        "Return True if this is a hardways roll."
        return self.myDice[0].value == self.myDice[1].value
```

1. We import the `random` module. The `Die` class will collaborate with the `random` module to simulate a single die.

3. We define `Die`. See *die.py, version 2*.

6. We define `Dice`, which will collaborate with `Die` to simulate a pair of dice.

8. The `__init__()` method creates an instance variable, *myDice*, which has a tuple of two instances of the `Die` class. The `__init__()` method is often called a *constructor*.

11. The `roll()` method changes the overall state of a given `Dice` object by changing the two individual `Die` objects it contains. This kind of method is often called a manipulator. It uses a **for** loop to assign each of the internal `Die` objects to variable *d*. It then calls the `roll()` method of each `Die` object. This technique is called *delegation*: a `Dice` object delegates the work to two individual `Die` objects.

16. The `getTotal()` and `getTuple()` methods return information about the state of the object. These kinds of methods are often called accessors. Sometimes they are called *getters* because their names often start with "get".

    - 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 access the value instance variable of each instance of `Die`.
    - The `getTuple()` method returns the values showing on 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 a tuple.

A Function Which Uses Die and Dice. The following function exercises an instance of this `Dice` class to roll two dice a dozen times and print the results.

```
import die
def test2():
    x= die.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.

Alternatives. The `roll()` method could also be written as

```
def roll( self ):
    [ x.roll() for x in self.myDice ]
```

This will apply the `roll()` method to each `Die` in *myDice*. Interestingly it also creates a list object. Since the `roll()` function doesn't return a value, this list object will actually be a sequence of `None` values. Since it isn't assigned to a variable, it quietly blinks out of existence and is lost forever. So, each time `Dice.roll()` is called a little list of `None`s is created and removed.

The `getTotal()` method could also be written as

```
def getTotal( self ):
    "Return the total of two dice."
    return sum( d.value for d in self.myDice )
```

13.2.7 Keeping Organized

Here are two additional topics: how we'll organize our class definitions, and how to create an empty class definition.

Class Definitions. In the long run, we'll put our class definitions in files and **import** them into our application programs. When doing this in **IDLE**, we should do the following.

1. Define your class in a file. In the following example, we'll put the `Die` in a file named `die.py`. Don't be fussy and put every single class into a separate file. Often, several classes will work together; it helps if they are also in a file together.
2. Save the file.
3. Hit the `F5` or use the **Run** menu, item **Run Module** to execute the class definition statement(s).
4. In the *Python Shell* window, create instances of your objects and exercise them.
5. When you need to make changes, go back to the window with the class definitions and make the changes. Re-import the module by cycling back to step 2, above.

We'll expand on this technique in *Modules : The unit of software packaging and assembly*.

Empty Class Definitions, the pass Statement. Note that the *suite of defs* in a class definition is required. Sometimes, however, we don't need any definitions; in this case we have to use the **pass** statement.

When we introduced creation of a specialized exception class in *Raising The White Flag in Exceptional Situations*, we showed how to use **pass** as a place-filler for the *suite of defs*. The **pass** statement is the "do nothing" place-filler; we use it when the syntax rules required a *suite of defs*, but we really don't have anything to add.

Here's an example exception definition that uses the **pass** statement. We want our own class of exceptions, but we don't have any new or different processing, just a new name.

```
class MyError( Exception ):
    pass
```

13.2.8 Class Definition Exercises

1. **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.

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

   Typically, you will average the SACR over a number of similar dives. You will want to create a `Dive` class with start pressure, finish pressure, time and depth. Typical values are a starting pressure of 3000, ending pressure of 700 to 1500, 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 it a little simpler to put 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 integers, 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, the difference is number of minutes of duration for the dive. You'll want to create a function `getDuration()` to do just this computation for each dive.

class Dive(*object*)

`__init__`(*start*, *finish*, *in*, *out*, *depth*)
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 into time tuples of hours and minutes. The `parseTime()` can be used to do this for both the in time and the out time.

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__`()
Return a nice string representation of the dive information.

getSACR()
Compute the SACR value from the starting pressure, final pressure, time and depth information. The duration can be computed using the `getDuration()` function.

parseTime(*hhmm_string*)
Pick apart a `HH:MM` time and convert the strings to integers to produce a 2-tuple of hours and minutes after midnight.

getDuration(*in_time*, *out_time*)
Accepts two 2-tuples of hours and minutes, normalizes these to minutes past midnight, and returns the difference. This is the dive's duration in minutes.

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 ),
]
```

Your application can then process a sequence of `Dive`s, get the SACR for each dive, and compute the average SACR over all the dives in the dive log. Here's a start on the final program.

```
total= 0
for d in log:
    print(d, d.getSACR())
    total += d.getSACR()
print(total, len(log))
```

2. **Stock Valuation**.

A block of shares in a stock has a number of attributes, including a purchase price, purchase date, and number of shares in the block. Commonly, methods are needed to compute the total spent to buy the stock, and the current value of the stock. An investor may have multiple blocks of stock in a company; this collection is called a Position.

Beyond a simple collection of shares are larger groupings. A Portfolio, for example, is a collection of Positions; it has methods to compute the total value of all positions of stock. We'll look at Position and Portfolio in a subsequent exercise. For now, we'll just lock at a block of shares.

When we purchase stocks a little at a time, each block of shares has a different price. We want the total value of the entire set of shares, plus the average purchase price for the set of shares as a whole.

First, define a `ShareBlock` class which has the purchase date, price per share and number of shares.

class ShareBlock(*object*)

`__init__`(*self*, *purchDate*, *purchPrice*, *shares*)
: Populate the individual instance variables with date, price and shares. We'll define another class with the ticker symbol that can act as a container for the several of these blocks for a particular company.

`__str__`(*self*)
: Return a nice string that shows the date, price and shares.

getPurchValue(*self*)
: Computer the purchase value as the $price \times shares$.

getSaleValue(*self*, *salePrice*)
: Given a *salePrice*, compute the sale value using the sale price in $price \times shares$.

getROI(*self*, *salePrice*)
: Given a *salePrice*, compute the return on investment as $\frac{(value_sale - value_purchase)}{value_purchase}$.

We can load our 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.

```
blockGM = [
ShareBlock( purchDate='25-Jan-2001', purchPrice=44.89, shares=17 ),
ShareBlock( purchDate='25-Apr-2001', purchPrice=46.12, shares=17 ),
ShareBlock( purchDate='25-Jul-2001', purchPrice=52.79, shares=15 ),
ShareBlock( purchDate='25-Oct-2001', purchPrice=37.73, shares=21 ),
]
blockEK = [
ShareBlock( purchDate='25-Jan-2001', purchPrice=35.86, shares=22 ),
ShareBlock( purchDate='25-Apr-2001', purchPrice=37.66, shares=21 ),
ShareBlock( purchDate='25-Jul-2001', purchPrice=38.57, shares=20 ),
ShareBlock( purchDate='25-Oct-2001', purchPrice=27.61, shares=28 ),
]
```

We can tally the purchase price of a block, for example, as follows:

```
totalGM= 0
for s in blockGM:
    totalGM += s.getPurchValue()
print(totalGM)
```

Once we have the `ShareBlock` class working, we can move on to processing the entire position.

3. **Stock Position**.

   In *Stock Valuation*, we looked at a block of stock shares. A collection of these blocks represents a position on that stock. We can define an additional class, `Position`, which will have an the name, symbol and a sequence of `ShareBlocks` for a given company.

   class Position(*object*)

   `__init__`(*self*, *name*, *symbol*, *block_list*)
   : Accept the company name, ticker symbol and a collection of `ShareBlock` instances.

`__str__`(*self*)
Return a string that contains the symbol, the total number of shares in all blocks and the total purchase price for all blocks.

`getPurchValue`(*self*)
Sum the purchase value for each block.

`getSaleValue`(*self*, *salePrice*)
Given a *salePrice*, sum the sale value for each block.

`getROI`(*self*, *salePrice*)
Given a *salePrice*, compute the return on investment as $\frac{(value_sale - value_purchase)}{value_purchase}$. 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",
        [ ShareBlock( purchDate='25-Oct-2001',
            purchPrice=42.84, shares=18 ) ] )
]
```

You can now write a main program that writes some simple reports on each `Position` object in the *portfolio*, and the overall portfolio. This report should display the individual blocks purchased. This should be followed with a total price paid, and then the overall average price paid (the total paid divided by the total number of shares).

4. **Statistics Library**.

We can create a class which holds a sequence of samples. This class can have functions for common statistics on the object's sequence of samples.

For additional details on these algorithms, see the exercises in *Doubles, Triples, Quadruples : The tuple* and the exercises in *Common List Design Patterns*.

class `Samples`(*object*)

`__init__`(*self*, *sequence*)
Save a sequence of samples in an instance variable. It could, at this time, also precompute a number of useful values, like the sum, count, min and max of this set of data.

`__str__`(*self*)
Return 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.

`mean`(*self*)
Return the sum divided by the count.

`min`(*self*)
Return the smallest value in the sequence of data values.

`max`(*self*)
Return largest value in the sequence of data values.

`variance`(*self*)
The `variance()` is a more complex calculation. 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.

`stdev`(*self*)
Return the square root of the variance.

`mode`(*self*)
The `mode()` returns the most popular of the sample values. The following algorithm can be used to locate the mode of a set of samples.

Computing the Mode

(a) **Initialize**. Create an empty dictionary, *freq*, for frequency distribution.

(b) **For Each Value**. For each value, *v*, in the sequence of sample values.

  Unknown? If *v* is not a key in *freq*, then add *v* to the dictionary with a value of `0`.

  Increment Frequency. Increment the frequency count associated with *v* in the dictionary.

(c) **Extract**. Extract the dictionary items as a sequence of tuples (*value*, *frequency*).

(d) **Sort**. Sort the sequence of tuples in ascending order by frequency.

(e) **Result**. The last value in the sorted sequence is a tuple with the most common sample value and the frequency count.

CHAPTER

FOURTEEN

MODULES : THE UNIT OF SOFTWARE PACKAGING AND ASSEMBLY

The basic Python language is rich with built-in features. These include several sophisticated built-in data types, numerous basic statements, a variety of common arithmetic operators and a library of built-in functions. As cool as Python is, the real power lies *outside* this kernel of built-in features. Python's real strength lies in the vast number of extension modules that add specialized features for problems of every kind.

There are a vast (and growing) set of powerful and sophisticated features of Python in the library of extension modules. There are several advantages to this arrangement. First, it allows modules to be added easily, further extending the power of Python. Second, it allows each program to load only the relevant modules, keeping your program as simple as possible. Third, it allows a module to be replaced, allowing the developer to choose among competing components to create the best solution to a problem.

Important: The *Batteries Included* Principle

In the Python community, the ease with which people add modules to Python leads to the *Batteries Included* principle. Many people use Python because there is almost always some group of modules which are directly applicable to their problems.

We've already made use of several modules. In *The math Module – Trig and Logs* we covered `math` and `random` modules. In *Files, Contexts and Patterns of Processing* we touched on several modules: `sys`, `glob`, `fnmatch`, `os`, `os.path` and `shutil`.

We'll do the formal introductions to modules in *Module Definitions – Adding New Concepts*.

Of the hundreds of available modules, we'll look at three of the most important modules in the Python library.

14.1 Module Definitions – Adding New Concepts

A *module* provides a convenient large-scale grouping of Python classes, functions and objects. Modules allow us to divide a large problem into smaller problems, each of which has a solution of manageable complexity. This also promotes reuse of components, further reducing the cost of solving a data processing problem.

In *Divide and Conquer with Modules* we describe the basic meaning of modules. In *Defining a Module: Creating Python Files* we describe how to define a module. In *Using A Module: The import Statement* we cover the **import** statement that lets us use a module. We'll look at variations on the **import** statement in *Some Variations On the import Statement*.

A module has some additional layers of meaning that we'll touch on in *Thinking In Modules, and the Declaration of Dependence*. We'll look at the technique of abstraction, again, and show how this applies to designing modules in *Dividing and Conquering – The Art Of Design*. This chapter ends with some style notes in *Style Notes* and some FAQ's in *Module FAQ's*.

14.1.1 Divide and Conquer with Modules

A module is essentially a file that contains Python programming. A module can contain new verbs in the form of function definitions. It can contain the definitions of new types of objects in the form of class definitions. A module can also create new objects.

A module introduces clusters of related verbs and types of data. By providing focused groups definitions, a module helps us summarize the associated details into a larger concept. Grouping details into larger concepts is central to successfully tackling more sophisticated problems. This conceptual tool can be called "abstraction" or "chunking": we summarize a chunk of details with an abstract summary. For example, I can say "Don MacLean's *American Pie*," which is shorter than me trying to singing a song that begins "A long, long time ago, ..." and proceeding through enough verses that you, too, know which song I'm talking about.

When we looked at `math` in *Better Arithmetic Through Functions* we saw a module that defined numerous mathematical functions, plus some objects like *math.pi*. When we write mathematical programs, we need this module, and can import it and use it. When we write other kinds of programs, we can ignore this module.

When we run the following program, Python is doing a great deal for us behind the scenes.

```
from __future__ import print_function
import random
print(random.randrange( 0, 37 ))
```

Here are the highlights of what happens when we `import random`.

- The `random` module defines the class `random.Random` for us. This saves us a lot of research into random numbers and how to create them on a digital computer. This class is very valuable by itself.
- The module defines an object, *random._inst*, of the `random.Random` class.
- The module defines convenience functions like `random.randrange()`, which use the *random._inst* object to generate random numbers using the algorithms in the class `random.Random`.
- Python keeps track of modules it has imported, so it doesn't import `random` a second time.

14.1.2 Defining a Module: Creating Python Files

A module is a file; the file name is the name of the module plus `.py`. For example, we create a file named `die.py`, this is the `die` module. This can include definitions of classes and functions related to the game of Craps.

1. **Shebang Line**. The first line of every Python file should be a "shebang" line. To Python, this line is just a comment. To a standard shell program (in GNU/Linux, MacOS, or any POSIX-compliant operating system), this line announces the interpreter than is expected to read and process the file. For Windows, this line is still just a Python comment. See *Comments and Scripts* for more on this technique.

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

2. **Docstring Lines**. The second line of a module file should be a triple-quoted string that defines the contents of the module file. As with other Python doc strings, the first line of the string is the pithy summary of the module. This is followed by a more complete definition that describes what the module does and how we should use it.

   ```
   """die.py - basic definitions for Die and Dice

   class Die defines a single 6-sided die.

   class Dice defines a pair of 6-sided dice.

   functions test1 and test2 perform simple sanity checks of the module.
   """
   ```

3. **From Future Lines**. If the module needs any future features, there might be a `from __future__` import statement to introduce any Python 3 features.

   ```
   from __future__ import division
   ```

   Often, these must be the first *executable* statements in the module.

4. **Imports**. Most modules will contain of some **import** commands. Typically, these are provided after the docstring comments. Also, the general policy is that each import names just one module.

5. **Body of the Module**. The body of the by a sequence of **class** and **def** statements to define classes and functions. Sometimes there will be additional assignment statements to create objects, also.

6. **Main Program Switch**. Some modules have a main program. This, generally, is implemented as a main program switch at the end of the module. We'll return to this in *Script or Library? The Main Program Switch*.

Example Module. For example, we can create the following module, called `die.py`. This module's primary purpose is to define two classes that we want to make use of in other programs. However, there are also two convenience functions. First we'll look at the module file; in the next section we'll look at ways we can import and use this module.

die.py

```
#!/usr/bin/env python
"""die.py - basic definitions for Die and 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.dice = ( Die(), Die() )
    def roll( self ):
```

```
        [ d.roll() for d in self.dice ]
    def total( self ):
        return sum(d.value for d in self.dice)

def roll( times=4 ):
    d=Dice()
    for i in range(times):
        d.roll()
        print(d.dice[0].value, d.dice[1].value)
```

In addition to two class definitions, which is typical, this module includes two test functions, `test1()` and `test2()`. These can be used to assure that the various elements of the module work correctly. They also serve as a kind of documentation for how the module should be used.

14.1.3 Using A Module: The import Statement

When python imports a module, the Python statements in the file are executed, and a module object is created.

The simplest and most commonly-used **import** statement simply lists modules to import. It has the form

,

```
import module [  ... ]
```

In this form, Python locates the module file, opens it, reads and evaluates the Python statements. The result of any **def**, **class** or **assignment** is to build objects within the module.

After this statement is executed, the named module object is fully populated and ready for use.

For example.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math']
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
```

1. Initially, we have just a few items in the global namespace.
2. Then we do `import math` to build the `math` module.
3. We can see the `math` module added to the global namespace.
4. Inside the `math` module we can see many names of math functions and objects. Also, we can see that a module contains a ___*doc*___ string, a ___name___ string, and the ___file___ string.

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. The names of these objects are collected into a *namespace*. This is the "space" in which the names are interpreted.

To make use of the elements of this new module, we qualify each name with the module name; for example, `math.pi`.

Important: Qualified Names – The Friends and Family Plan

Our script or program that uses a simple **import** must use qualified names.

For example, if a script uses `import die`, it must create objects of `die.Die` class or `die.Dice` class.

The module itself, internally, can't use qualified names. Within our `die` module, the definition of the `Dice` class is in the same module as the definition of the `Die` class, so the names are not qualified.

Here's an example of using the die module.

```
>>> import die
>>> d= die.Dice()
>>> d.roll()
>>> print(d.total())
4
```

More About Modules and Names. In *Keeping Track of Variable Names – The Namespace* we talked about a function having a local namespace. A namespace keeps the variables defined within a function separate from all other function's variables and any global variables. This namespace is created when the function is evaluated, and disposed of when the evaluation is finished. All of the variables created inside the function's suite of statements are silently disposed of.

When we looked at class definitions, we used a number of namespaces. A method function has a local namespace, just like an ordinary function. Further, the *self* parameter defines the namespace for the object's instance variables. To use an instance variable, we use a qualified name: *self.variable* provides the namespace, a dot, and the variable from within that namespace. Each object is effectively a namespace for the object's attributes; the namespace is created when the object is created.

When a module is imported, the module – as a single object of class `module` – is imported into the global namespace. Each individual class, function or variable defined within the module's file becomes part of the module's local namespace. This assures us that each module can define classes, functions and variables without worrying about conflicts with other modules. When we use a two-part name (`die.Dice`, or `math.sqrt`), we provide the namespace first, then the name within the namespace.

Important: Bad Behavior

Importing a module means that the module file is executed similar to a script file. This means that all of the statements in the module are executed.

The standard expectation is that a library module will contain only definitions. Any executable statements should be inside function 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 additional work that a module does is unexpected and makes the module hard to use.

Python doesn't enforce this distinction between a script, which does something useful, and a library module, which defines things that a script will use. It is purely a matter of best practice in designing a Python modules and programs.

14.1.4 Some Variations On the import Statement

There are several variations on the **import** statement. We looked at these briefly in *The math Module – Trig and Logs*. In this section, we'll cover some more details of the variations available on the **import** statement.

Basic Import. The simplest and most commonly-used **import** statement simply lists modules to import. We looked at this above. **,**

```
import module [  ... ]
```

When we want to use a function from the `math` module, we must tell Python which namespace contains the function by putting the module name and a dot (`.`): `math.sin(0.7854)`, for example. This explicit qualification is important to everyone else who'll be reading your program. Using the module name with the function or class name makes the origin of the object clear.

Exposing Certain Names. Another variation on import introduces selected names from the module into the local namespace. This import has the form: **,**

```
from module  import name [  ... ]
```

This performs the basic module import but moves the given names into the local namespace. The selected names can be used without the overall module namespace qualifier.

For example:

```
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None}
>>> from math import sin, cos, tan
>>> locals()
{'cos': <built-in function cos>, '__builtins__': <module '__builtin__' (built-in)>, 'sin': <built-in function sin>,
>>> sin(0.7071)
0.64963178370469132
```

1. The `locals()` value shows the standard built-in objects.
2. We execute `from math import sin, cos, tan`.
3. 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: `sin(0.7071)`, for example.

This must be used judiciously because it tends to conceal the origin of objects. The best use for this is in the rare situation when we have a package of several alternative versions of a particular module and want to import one of the alternatives .

Exposing All Of A Module's Names. The third 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 available in the local namespace. This is only appropriate for interactive debugging.

Since this makes the origin of the name obscure, we suggest avoiding it.

Renaming A Module. Another variation on import allows you to have multiple, competing implementations for a module.

```
import module as newName
```

This retains the explicit qualification of names so we can see which module the name belongs to. But the qualifying name is the new name.

We often use this when we have different variants that we want to use in a consistent manner. For example, we may have several random number generator modules. Each module can have the same set of class and funcion names, but different algorithms to implement those classes and functions.

```
option = "Standard" # or "Homebrew"
if option == "Standard":
    import random as rnd
else:
    import homebrewed_lcm as rnd
print(rnd.randrange(1,20))
```

In this case, we've imported `random` and named it *rnd*. We can easily change this to import `homebrewed_lcm` as *rnd*. This gives us a way to make a very easy switch between alternative implementations.

We often use this when we have different variants that we want to use in a consistent manner. This is often done for database modules. Often a number of databases will all use the `dbapi` interface. This allows a single application program to work with any one of a number of compatible database modules.

We also use this technique for XML or JSON file parsers; we might have several alterantive XML parsers. We can use the **import as** statement to select which variant we want.

Finally, we can also use this to create a handy short-hand for a long module name. This can happen when you have packages of modules, and the package path names get long. This lets you keep your programming short by using an abbreviation for a longer, more precise module name.

14.1.5 Thinking In Modules, and the Declaration of Dependence

Modules shouldn't just "happen". A well-designed module uses the abstraction principle to organize and simplify a concept. We'll talk about the design of a module, the import processing and the physical location of the module file. Import processing is really part of a bigger picture that we call the *Declaration of Dependence*.

Designing A Module. There are two purposes for modules; some module files may do both.

- **Library Modules**. A library module contains definitions of classes and functions. It is used via the **import** statement. Some modules can be used in a number of situations, like the examples we'll look at later in this part. Other modules are focused on a problem domain or specific to a particular application program. A well-designed library module doesn't do much more than provide definitions.

  At first, we used library modules that were shipped with Python. It's time to start writing our own library modules.

- **Main Modules**. A main module is a script that does the useful work of an application. It is used from the command line, via **python {module}.py**. Typically, it imports library modules and also has a main function that does the real work of the application. In addition to the main script, it may have its own function or class definitions.

  At first, we wrote main module scripts that we simply executed. Its time to start importing our own modules so that we can create more complex applications.

How can a module be both library and main program? Most main progams contain some useful class and function definitions. We may, for example, have a cool module which analyzes the stock prices downloaded from a web page. After using this for a few months to make some investment decisions, we might decide to automate the download of the stock prices. Rather than rewrite this module, we create a new main module that imports the analyzer, and also uses `urllib2` to download the prices.

In this case, the stock price analyzer is a main module, and analyzes a file that was downloaded manually. It is also a library module, used by another main module that automates the download and analysis.

Python encourages a very neat approach to integrating applications and libraries. We cover the *Main Program Switch* in *The Standard Command-Line Interface*; this distinguishes between a component acting as the top-level main program, and a component being used as a library by an even higher-level program.

Contents of a Module. The principle of abstraction tells us to elide the useless details and emphasize the useful details. In a software context, the useful details are called the interface, the outside of the box. Think of your home entertainment system: it has some knobs on the front and wires on the back. This is the interface. The internals of each component aren't relevant; what is relevant is the places where signal goes in and music comes out.

When designing a module, we have to be very clear on the *interface*: where information goes in, where information goes out, and what controls we might want to have over the processing. Behind the interface, there will be processing details that don't really matter to someone who uses the module. In some cases, the "details" may be just a function definition. In other cases, the details may be a number of complex class definitions.

A module reflects some knowledge about data and processing. Our module must reflect a tidy, easy-to-explain bundle of concepts. This is the "coherence" or "conceptual integrity" principle. A module isn't a jump of stuff in one file. It's reflects concepts we use to simplify our programming.

Knowledge Capture. A program in Python represents knowledge. We have a spectrum of Python programming concepts from the very fine-grained statements to the very inclusive packages of modules and application programs. This spectrum includes the following ways of composing and grouping knowledge.

- **Statement**. A statement makes a specific state change. The assignment statement will update variables in our Python environment. We use other statements (like **if** statements or **while** statements) to choose precisely which assignment statements get executed.
- **Function**. A function groups a suite of statements to compute a specific result or perform a specific task. Functions are designed to be an indivisible, atomic unit of work. Functions can be easily combined with other functions, but never taken apart. A a well-chosen function name clarifies and defines a single, useful concept.
- **Class**. A class groups a set of related functions and the private data elements they share. The class represents several closely related tasks, always with a narrowly defined responsibility. Classes may be simple, perhaps only a single function or attribute, or a complex collection of attributes and method functions. The intent is to clearly delineate responsibility for maintaining data or performing an algorithm.
- **Module**. A module is a group of any Python definitions, including variables, classes and functions. A module should provide a closely related set of one or more class definitions and related convenience functions and objects. The *conceptual integrity* of a module is it's central feature. We put things into a module because they are closely related.
- **Package**. A package is a group of modules: the directory structure of the package is the directory that contains all the packages. Additionally, packages contain some additional files that Python uses to locate all of the elements of the package.
- **Application**. The application is the top-most "executable" script that a user invokes when they want to do useful work. There is a relationship between the commands that a user sees and the packaging of components that implement those commands. This relationship reveals two different concerns: usability and maintainability. The application-level view – the command or GUI presented to the user – should be focused on usability. The design of modules and packages is focused on the technical concerns of maintenance and adaptability.

Import Processing – The Declaration of Dependence. When we write a script that imports a module, we are making a formal Declaration of Dependence. We are saying that our script depends on another module. Python does several things to honor this dependency.

1. If the module is known, this means that it has already been imported, and nothing more needs to be done. This has the pleasant consequence of allowing a complex program with many modules to be very casual about the order of the various **import** statements.

   An **import** statement is really only executed the first time it is seen. Every other time, it's just a Declaration of Dependence.

2. If the module is not known, it needs to be imported.

   (a) The *path* is searched to find the module file. If it cannot be found, an exception is raised. We'll return to the path concept later.

   (b) Once found, the module file is read and executed. All of the resulting variables, functions and classes are part of the module's namespace.

(c) The **import** statement can do some optional processing to assure that module elements are created in the global namespace. This is the `from module import ...` kind of import.

Primarily, an **import** statement is a declaration of the support a module requires. In a more complex application, there may be several libraries, each with it's own collection of imports. This can create a web of dependencies among the various components.

Additionally, a module import is a form of execution, just like executing a script. A module, therefore, can do processing as well as define functions and classes. It is best if any processing is carefully segregated from the definitions. That's why we make a firm distinction between library modules and main programs.

File System Locations. For modules to be available for use, the Python interpreter must find the module file on the *module search path*. When you provide an **import** statement, Python searches each of the directories on the path for the named module.

You can see Python's understanding of the path by importing the sys module and looking at the variable *sys.path*. This variable lists all the places Python will look for a module. There are several consequences to this use of *sys.path* to contain a list of directories.

Here's a Windows search path. Note that Windows uses \ as a file path separator, and Python strings use \ as an escape character, which leads to using \\ as the Python language representation of a single \. This *sys.path* lists more than a dozen locations where modules can be found.

```
1  >>> import sys
2  >>> import pprint
3  >>> pprint.pprint(sys.path)
4  ['',
5   '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/distribute-0.6.15-py2.6.egg',
6   '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/Django-1.3-py2.6.egg',
7   '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/django_piston-0.2.2-py2.6.egg',
8   '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/xlrd-0.7.1-py2.6.egg',
9   '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/docutils-0.7-py2.6.egg',
10  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/Sphinx-1.1.2-py2.6.egg',
11  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/Jinja2-2.6-py2.6.egg',
12  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/Pygments-1.4-py2.6.egg',
13  '/Users/slott/Documents/Projects/DiningPhilosophers',
14  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python26.zip',
15  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6',
16  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/plat-darwin',
17  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/plat-mac',
18  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/plat-mac/lib-scriptpackages',
19  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/lib-tk',
20  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/lib-old',
21  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/lib-dynload',
22  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages',
23  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/PIL',
24  '/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/setuptools-0.6c11-py2.6.egg-info']
```

This list shows the built-in parts of Python, the add-ons I've downloaded, and all of the various projects that I'm using this computer for. Let's examine the search path to see where things will be found.

4. `''` is the first location. The empty string stands for your current working directory.
5. The next locations (lines 5 to 12) are Python packages provided as `.egg` files. These are all tools that I downloaded and added to my Python environment.
13. This is a local project. It was included via the `PYTHONPATH` environment variable.
14. The next eight locations are the essential ingredients of Python itself. This group starts with `...python26.zip` and ends with `...python2.6/site-packages`.

23. The next two locations are Python packages that are not provided as `.egg` files. These are also tools that I downloaded and added.

Setting Your Path. To be sure a module can be imported, you have to be sure that the file is found on Python's *sys.path*. To do this, you have to do any one of the following things.

- Put your module into the directory that Python sees as the current working directory. This works great for learning, but in the long run, you don't want to have your working files and your program all piled together in the same directory. Eventually, you'll want to install your programs in someplace more permanent, and separate from your working data files.

  When you are working in **IDLE**, **IDLE** will put a module's working directory in the path when you load or run the module.

- Put your module into an existing library of modules. Python has a directory called `Lib/site-packages` in which you can put your own modules. This is usually associated with the Python installation directory. See *Let There Be Python: Downloading and Installing* for more information. In the example above, many modules were located in my file:*site-packages* directory.

  Most Python modules and packages include a *setup.py* file which will properly install the module into file:*site-packages* directory.

- Extend the list of directory paths by putting a `.pth` file in the `site-packages` directory. A `.pth` file is a one-line file that provides the directory location for a given module.

- Extend the list of directory paths to include the directory for your module by changing the Python environment. In GNU/Linux, the

  `PYTHONPATH` environment variable can be used to define the directories expected to contain modules.

  In the Windows environment, the `Python_Path` symbol in the Windows registry is used to locate modules as well as the

  `PYTHONPATH` environment variable.

  Windows. We use a command like **SET PYTHONPATH pathtomodule** to name directories that Python should also search for modules. Separate each directory name with `;`.

  All Other OS. We use a command like **export PYTHONPATH=path/to/module** to name directories that Python should also search for modules. Separate each directory name with `:`.

  See the sidebar, *Debugging Imports*, for more information on determining why your module won't import.

Since the *sys.path* object is a list, there are two important consequences.

- Lists are mutable. Your program can add directories to this list. This can be confusing and hard to maintain. Some applications do this, however, to provide a very high degree of flexibility. You may read someone else's program which updates *sys.path*. It isn't the best policy.
- Lists are ordered. If a module occurs in two directories, Python will locate the one that's first in the search path. You can use this to create a test version in your local directory which is loaded before the "released" version in your `site-packages` directory.

For now, we can put our modules into the same directory as our main script. When we open a file in **IDLE**, that file open will also changes what **IDLE** sees as the current working directory. Keeping a script and the related modules in one directory is the minimum we need to do to assure that our script can import our modules.

Tip: Debugging Imports There are four things that can go wrong with an **import**: (1) the module can't be found; (2) the module isn't valid Python; (3) the module doesn't define what you thought it should define; (4) the module name isn't unique and some other module with the same name is being found.

Be sure the module's `.py` file name is correct, and it's located on the *sys.path*. Module filenames are traditionally all lowercase, with minimal punctuation. Some operating systems (like GNU/Linux) are case-sensitive and a seemingly insignificant difference between `Random.py` and `random.py` can make your module impossible to find.

The two most visible places to put module files are the current working directory and the Python `Lib/site-packages` directory. For Windows, this directory is usually under `C:\python26\`. For GNU/Linux, this is often under the `/usr/lib/python2.6/` directory. For MacOS users, this will be in the `/System/Library/Frameworks/Python.framework/Versions/Current/` directory tree.

If your module isn't valid Python, you'll get syntax errors when you try to import it. You can discover the exact errors by trying to execute the module file using the `F5` key in **IDLE**.

If the module doesn't define what you thought, there are two likely causes: the Python definitions are incorrect, or you've omitted a necessary module-name qualifier. For example, when we do `import math` everything in that module requires the `math` qualifier. Within a module, however, we don't need to qualify names of other things defined in the same module file.

If your Python class or function definitions aren't correct, it has nothing to do with the modularization. The problem is more fundamental. Starting from something simple and adding features is generally the best way to learn.

The *sys.path* is a list, which is searched in order. Your working directory is searched first. When your module has the same name as some extension module, your module will conceal that extension module. I've spent hours discovering that my module named `Image` was concealing PIL's `Image` module.

14.1.6 Module Exercises

To create a definitional module, you will group together some related class definitions into a single file. Be sure to include a docstring at the beginning of the file.

The test functions should be put into a separate script that imports the module and uses the material from the module file. This means that the test functions will have to use qualified names for the classes in the module.

You may have some existing tests that have an implicit assumption that they are in the same namespace as the class definitions. Once you put this into separate files, the tests will be in a separate namespace from the class definitions. While you can finesse this with `from myNewModule import *`, this is not the best programming style.

Let's assume you have the following kind of script as the result of a previous exercise.

Original File

```
#!/usr/bin/env python
"""Definition of class X and Y."""
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`

New Library Module

```
#!/usr/bin/env python
"""Definition of class X and Y."""
class X( object ):
    does something
class Y( X ):
    does something a little different
```

Your new application will look like the following because you will have to qualify the class and function names that are created by the module.

New Application Script

```
#!/usr/bin/env python
"""Program which uses X and Y."""
import myModule
x1= myModule.X()
x1.someMethod()
y2= myModule.Y()
y2.someOtherMethod()
```

1. **die.py module**.

   Finalize the `die` module with the classes `Dice` and `Die`. Write a demonstration script that rolls dice and gathers simple statistics to show that the distribution of dice rolls is what we expect: 2.77% of the rolls are 2's up to 16.6% which are 7's and then back down to about 2.77% which are 12's.

2. **roulette.py module**.

   Define a `roulette` module which includes the `Wheel` class definition. An instance of `Wheel` should have a `spin()` method that returns a spin of the wheel, showing the number and the color. A separate script should exercise the wheel to gather a number of spins showing how many red, how many black and how many green (for zero or double zero).

3. **divelog.py module**.

   The Dive Log exercise in *Class Definition Exercises* contains a definition of the `Dive` class. This should be separated into the `divelog` module. A separate file can import this and use it to define a collection of dives and compute SACR or other statistics on the collections of dives.

4. **stock.py module**.

   The `ShareBlock` and `Position` exercises in *Class Definition Exercises* contains a definition of a number of related classes. These classes should be separated into the `stock` module. A separate file can import this module and use it to define a collection of stock positions and compute purchase value, current value, annualized ROI or other statistics on the stocks.

14.1.7 Dividing and Conquering – The Art Of Design

The "divide and conquer" strategy broadly characterizes a number of problem-solving techniques. We'll look at a few suggestions for ways to divide and conquer a problem. There are a large number of excellent books on the subject of Design Patterns that can help us structure a solution to data processing problems.

When we have a large problem, we have to break the problem down into sensible pieces so we can tackle each piece successfully. We then knit the pieces together to create our desired solution. Slicing up a big problem is a matter of isolating parts of the problem by looking for closely-related data elements or easy-to-define processing.

The Input-Process-Output Pattern. One common technique is to look for the "Input-Process-Output" pattern. In this case, we slice the problem into three separate parts: reading the input, doing the necessary processing, and writing the output. This pattern applies particularly well when we have input data in a different format from our output data. We may, for example, have tab-delimited input, but be producing HTML output. Or, we may have CSV input and we are sending a batch of emails.

When we apply the "Input-Process-Output" pattern we often work with four modules.

- The first module we create will do the processing. It will depend on other modules to read input and produce output. This module contains classes that define the real-world objects our program works with. If we are building a web site that shows an architect's portfolio, the object might be "building projects" with a picture, a description, a date, a cost or other descriptive information. Perhaps each project may be associated with a CSV file of project details which must be added up to compute the total value of the project. This module is often challenging to create because it embodies the core knowledge about the problem.
- The next module parses the input file. If our input is CSV, this module is already written. If our file is tab-delimited, we can use `string.split("\t")`. This module will create objects that feed the processing module. Since we already have parts of the solution completed, that makes it easy to test this module interactively.
- Another module prepares the output. Since the input and the processing are complete, this module is even easier to test. We have the results of the processing, we are left with preparing the final output file.
- The final module is the overall script. It creates the input-reading object, the output-writing objects. It uses the input-reading objects to create the objects for processing. It then writes the final results.

The Model-View-Control Pattern. Another common technique is to look for layers of interaction. In this case, we may be applying a variation on the "Model-View-Control" pattern. At its simplest, the "Model-View-Control" pattern has a data model, a GUI view of that data, and control objects that allow the user to manipulate the model using the view.

This gives some guidance on what our modules will contain.

- The first module we have to create is the Model, which will do the processing. This module contains classes that define the real-world objects our program works with.
- The view components are part of `Tkinter` or `pyGTK`. We don't write these, we study the documentation and tutorials to learn how these classes work.
- The next modules we write will handle the application control. These modules will create GTK or Tk widgets; the events created by these widgets will be directed to the model objects.
- Often, we will have to provide additional modules to handle input and output of our objects.
- The overall application program will create the GUI control objects. These objects will create the view widgets. When the user selects the right menu item, the control object will locate a file. It will create a reader object and use this to read the model objects from disk files or from the internet.

The “Model-View-Control” pattern is often combined with the “Input-Process-Output” pattern. The Input and Output are lumped into a component called “Persistence”, View and Control are lumped into “Presentation” and the Process and Model are generally two names for the same thing. This gives us the “Three Tier Architecture” or “Presentation-Processing-Persistence”. Often the presentation is handled by a web server using Apache, the processing is handled by an application server running Python and the persistence is handled by a database like MySQL. When run on GNU/Linux, they call this the LAMP architecture: GNU/Linux, Apache, MySQL and Python.

While a web applications can be complex, at the core each individual web page manages a simple transaction which is an input-process-output pattern. A request defines the processing, input comes from the user or files or a database, and the output is almost always a page of HTML.

14.1.8 Style Notes

Modules are a critical organizational tool for final delivery of Python programming. Python software is always 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.

Most modules have names that are typically `lower_case_with_underscores()` or `mixedCase()`. Module names are generally all lowercase letters. This conforms to the usage on most file systems. This promotes portability to operating systems where file names are more typically single words.

Some Python modules are an object-oriented façade over a C-language module. In this case the C/C++ module is named in all lowercase and has a leading underscore (e.g. `_socket`).

Module Contents. 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** or **Subversion** 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 $"
```

Choosing Names. The bulk of most modules are class, exception and function definitions. Since the module name implicitly qualifies everything created in the module, it is never necessary to put a prefix in front of each name within a module to show its origin. This is common in other programming languages, but never done in Python.

For example, a module that contains classes and functions related to statistical analysis might be calls `stats.py`. The `stats` module might contain a class for tracking individual samples. This class does not need to be called `statsSample` or `stats_sample`. A client application that contains an `import stats` statement, would refer to the class as `stats.Sample`. Additional qualification is redundant.

The qualification of names sometimes devolves to silliness, 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 the parameter names are quite obvious.

Private Elements of a Module – None of Your Business. It is common to have parts of a module that are intentionally "private" to the way the module is currently written. These are things that we might change in the future when we think of a better way to handle them. Or, they could be parts of the module that shouldn't be tampered with.

Any element of a module with a name that begins with *_single_leading_underscore* is never 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. They're considered to be a private part of the implementation, not a public part of the interface.

Exceptions. A common feature of modules is to create a module-wide exception class. The usual 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:

```
from __future__ import print_function
import aModule

try:
    aModule.aFunction()
except: aModule.Error, ex:
    print("problem", ex)
    raise
```

With this style, the origin of the error is shown clearly.

14.1.9 Module FAQ's

Why does Python have so many library modules? One alternative to having separate modules is to have a large Python interpreter, with all modules built-in. Removing, adding or replacing a module would involve a complex (and risky) procedure for rearranging the **python** program. To reduce risk, we would have to include lots of complex checking and verification steps. If we wanted to avoid this complexity, we'd have to endure the risk of something going wrong because of an unforeseen special case. What the Python community does instead is create separate modules with clearly-defined dependencies. But it does allow everybody – and their brother – to define a new module and post it to the Internet.

How do separate modules make my programs simpler? Isn't it more complex to break a program into piece
When you look at the various modules available, most of them are rather large. A module that you use is full of programming you didn't do. Clearly, it saves you time.

More importantly, a module allows your program to be *conceptually* simpler. Rather than a big pile of details, you can make use of the concepts in the module. You can then work with larger, more complex and more abstract data structures. It's much easier to work with a long number than to work with a big list of individual digits.

What possible benefit is there in replacing a module? No module is "perfect". A module which uses the least memory may also be rather slow. A module which is fastest may use too much memory to make your program work reliably. A really fast module may be difficult to understand and improve on. A module that's easy to understand and improve may be too slow for your program.

Since no module can ever meet all possible performance needs, you need to be able to choose an implementation. The Python library is full of alternative solutions to a common problem. Being able to choose an optimal solution gives you the power to create the best possible solution to your unique problem.

How do you draw the line on the contents of a module? You could put each function in a separate module, or you could put everything in one module. How do you find appropriate middle ground?

This is can be a difficult problem. The fundamental principle is the following: **A Python Module is the Unit of Reuse**. When you design a module, you should be thinking of the module as a discrete component of your overall solution.

A module will often have multiple, closely related, classes and function definitions. A module will rarely have a single class or a single function; unless that single object can be used – by itself – in several applications.

There are a number of guidelines for what makes a good module. The seminal article is D. L. Parnas *On the Criteria to Be Used in Decomposing Systems into Modules* [Parnas72]. Subsequent to this, some additional rules have been applied to this problem. First, a module should be coherent; that is, it should be easy to summarize in a short description. Second modules should be loosely coupled; that is, there is a well-defined interface and other modules only make use of the interface. The simpler an interface is, the looser the coupling and the better the coherence.

14.2 Fixed-Point Numbers : Doing High Finance with `decimal`

In *Floating-Point Numbers, Also Known As Scientific Notation*, we saw that the built-in floating decimal point numbers are great for scientific and engineering use but don't work very well for financial purposes. US currency calculations, for example, are often done in dollars and cents, with exactly two digits after the decimal point.

We'll look at the problems that arise when we try to do currency calculations in floating-point numbers in *Representing Numbers*. We'll look at the basics of decimal numbers in *Using decimal Numbers*, then the various ways we can control rounding in decimal arithmetic in *Rounding, known as "Quantization"*. We'll look at more sophisticated rounding control in *Controlling Rounding*.

14.2.1 Representing Numbers

In *Floating-Point Numbers, Also Known As Scientific Notation*, we saw that floating-point numbers are for scientific and engineering use and don't work well for financial purposes. US dollar calculations, for example, are often done in dollars and cents, with exactly two digits after the decimal point.

If we try to use floating-point numbers for dollar values, we have problems. Specifically, the slight discrepancy between binary-coded floating-point numbers and decimal-oriented dollars and cents become a serious problem. Just try this simple experiment.

```
>>> 2.35
2.3500000000000001
```

Here's the sequence of events that leads to this.

1. Python parsed the numeric literal `2.35`. Which you have provided in decimal notation.
2. Python creates a `float` number that approximates this decimal value. There isn't an exact representation in the computer's internal notation. The fraction $\frac{39,426,457}{2^{24}}$ hints at this representation error.

3. Python displays the floating-point number in decimal notation. This means converting the binary internal value to base ten.

You can try to use the round() to lop off this extra little bit of error. For example, you might try to use the following. What happens?

```
round( 2.35, 2 )
```

The rounded result is still a binary number and are we sill trying to express a binary fraction using decimal digits.

Internal Bits and Bytes.

The Python literal 2.35 becomes a floating-point value of the form $m \times 2^e$. The *mantissa*, $m \leq 1$, is a fractiona of the form, $\frac{n}{2^{53}}$.
Internally, the value is effectively this.

$$\frac{5,291,729,562,160,333}{2^{53}} \times 2^2$$

Which is very precise, but still not 2.35.

Handling Math. While rounding seems like a good idea, there is some sophistication required to handle interest rates which are often in small fractions of a percent. For example, if an interest rate is 8.25%, 0.0825, we have 4 decimal places of precision that we have to preserve. If we apply this rate to a large amount of money, say 123,456,789.10, the precise answer has 15 digits, six of which are to the right of the decimal point. On some computers, floating-point numbers can't represent this many digits correctly.

What to do?

It's time to look at the decimal module.

14.2.2 Using decimal Numbers

Here's an example of creating and using some decimal numbers.

```
>>> from decimal import *
>>> Decimal("2.35")
Decimal("2.35")
>>> Decimal("135.99") * Decimal("0.075")
Decimal("10.19925")
```

There are several important things to note about creating and using decimal numbers.

- The source is always a string. This has to be done because a Python float value, like 135.99, is converted from Python's language (in base 10) to the computer's hardware representation (in base 2) and some precision is lost in the process.

  To avoid the Python language conversion, string literals are used.

- A decimal object retains all the digits of a precise answer to any mathematical operation. In the case of repeating fractions, there is a default upper limit of 28 digits.

- A decimal object can return a new decimal object with a different quantization by rounding. There are a number of rounding rules, and we'll look at them in detail, below.

- decimal objects are considerably slower than float objects. For the most part, "slow" is relative and decimal is fast enough for everything except processing JPEG images or MP3 sound samples.

Here's another example, which is closely related to the stock price examples we looked at in *Files, Contexts and Patterns of Processing*. Let's say that I bought 135 shares of Apple back when it was trading at $20.44. What would I have made if I sold it at $80.25?

```
>>> purchase = Decimal("20.44")
>>> current = Decimal("80.25")
>>> shares = 135
>>> shares * purchase
Decimal("2759.40")
>>> shares * current
Decimal("10833.75")
>>> shares*(current-purchase)
Decimal("8074.35")
```

Tip: Debugging `decimal`

There are a number of things that can go wrong with using `decimal` numbers. If we forget to import `decimal` (note that the module has a lower-case name), then we'll get `NameError` messages.

If we use an `import decimal`, we must say `decimal.Decimal` to make a new number. If we use `from decimal import *`, we can say `Decimal` to make a new number.

We can convert integers or strings to Decimal numbers. If we accidentally provide a floating-point value instead of a String, we get `TypeError: Cannot convert float to Decimal. First convert the float to a string`.

14.2.3 Rounding, known as "Quantization"

When doing currency and financial calculations, we often have a need to round numbers to a nearest number. When doing percentage calculations, we may have to round to the nearest penny. For financial reporting, we may want to round to the nearest thousand (or million) dollars.

Accountants have a wide variety of rounding schemes, all of which have their various uses. The `decimal` module handles rounding through a number of techniques. Before we can look at quantization and rounding, we need to look at how decimal numbers are handled internally.

Three Pieces of Information. A `decimal` number is really three closely-related pieces of information: the sign, s, which is +1 or -1, the digits, d, and an exponent, e. You can think of a number, n, as $n = s \times d \times 10^e$.

For example, the number 6.25% is 0.0625, which is a positive sign, a string of digits '625' and an exponent of -4. $625 \times 10^{-4} = 0.0625$.

Internally, the sign is coded a little strangely. Positive numbers have an internal code of 0, negative numbers have an internal code of 1. You can see this when you use the `as_tuple()` method of a `decimal` number.

```
>>> import decimal
>>> i=decimal.Decimal('0.0625')
>>> i
Decimal("0.0625")
>>> i.as_tuple()
(0, (6, 2, 5), -4)
```

Technically, a `decimal` number is an immutable object. The arithmetic operations will create new numbers, but the don't change an existing number. This means that `decimal` objects can be used as keys in a mapping.

Quantizing a Number. You can *quantize* any Decimal number to a specific number of digits before or after the decimal place. We don't call this "rounding" because we're not always rounding, we may be truncating. The general term for rounding or truncating is quantizing.

When you quantize, you can specify a rounding rule directly. Additionally, we can provide a general rounding rule in the *decimal context*.

When quantizing, you provide a `decimal` number which has the desired number of decimal places to the `quantize()` method of a number. Here's an example.

```
>>> from decimal import Decimal
>>> total= Decimal( "135.99" ) * Decimal( ".075" )
>>> total
Decimal("10.19925")
>>> total.quantize( Decimal('0.01') )
Decimal("10.20")
>>> total.quantize(Decimal("0.01"),decimal.ROUND_DOWN)
Decimal("10.19")
```

You can see that the default context specifies that values are rounded. However, we can specify a specific rounding rule as part of the quantize operation. There are a number of rounding rules.

- `ROUND_CEILING`. This rounds all fractions to the next higher positive number. They call this rounding towards Infinity. Positive numbers will get be rounded away from zero, getting larger. Negative numbers will get smaller in magnitude, being moved closer to zero.
- `ROUND_DOWN`. This chops off all fractions, rounding towards zero. Positive numbers will get smaller. Negative numbers will get smaller in magnitude, being moved closer to zero.
- `ROUND_FLOOR`. This rounds all fractions toward the next lower negative number. This call this rounding towards -Infinity. Positive numbers will get rounded toward zero, getting smaller. Negative numbers will get larger in magnitude, being moved away from zero.
- `ROUND_HALF_DOWN`. This rounds off to the nearest number. A value in the middle is rounded toward zero. When rounding a value to `Decimal('1')`, a value of 0.5 becomes zero.
- `ROUND_HALF_EVEN`. This rounds off to the nearest number. A value in the middle is rounded to the nearest even number. When rounding a value to `Decimal('1')`, a value of 1.5 becomes 2, where a value of 0.5 becomes zero.
- `ROUND_HALF_UP`. This rounds off to the nearest number. A value in the middle is rounded away from zero. When rounding a value to `Decimal('1')`, a value of 0.5 becomes 1.
- `ROUND_UP`. This chops off all fractions, rounding away from zero. Positive numbers will get larger. Negative numbers will get larger in magnitude, being moved away from zero.

The default context uses `ROUND_HALF_EVEN` as the rounding rule. You can see this by executing `decimal.getcontext()`.

Pennies and Dollars. The quantize method needs a Decimal object that is really only used to provide an exponent. Some people find that creating that Decimal object is a bit too wordy.

Specifically, the expression `someNumber.quantize( Decimal('0.01') )` seems to be a lot of typing for a simple concept. Here's another approach that may be a little more clear.

We can define a pair of useful constants for rounding to pennies or dollars.

```
pennies = Decimal( "0.01" )
dollars = Decimal( "1.00" )

total= Decimal("123.45") * Decimal("0.075")
final= total.quantize( pennies, ROUND_UP )
```

Tip: Debugging `decimal` Rounding

The quantization method appears strange at first because we provide a Decimal object rather than the number of decimal positions. The built-in `round()` function rounds to a number of positions. The `quantize()` method of a `decimal` number, however, uses a `decimal` number that defines the position to which to round.

If you get `AttributeError: 'int' object has no attribute '_is_special'`, from the `quantize()` function, this means you tried something like `aDecimal.quantize(3)`. You should use something like `aDecimal.quantize(Decimal('0.001'))`.

14.2.4 Controlling Rounding

The Context. The decimal module defines a default context for our program. A context provides a number of pieces of information that control how numbers are processed. The context controls precision used, sets the rules for rounding, defines which signals are treated as exceptions, and limits the range for exponents.

The most important value in the context is the general rounding rule used by `quantize()`. Initially, it is `ROUND_HALF_EVEN`. For some applications, it is appropriate to update the context with a different rounding rule to make all `quantize()` operations consistent.

The context's precision provides an upper limit on how many digits are carried for repeating decimal fractions. For example, the decimal value of 22/7 repeats '142857' an infinite number of times. The initial `decimal` context has 28 digits of precision. That means that `Decimal('22')/Decimal('7')` is `Decimal("3.142857142857142857142857143")`. You can see that there are 28 digits; the '142857' repeats four times, preceded by a '3' and followed by the first three digits of the repeating decimal.

The context also has the largest and smallest exponents which can be handled. Generally, the built-in context allows numbers as large as 10 to the 10 millionth power. Generally, this will allow processing US Federal Budget calculations which only run to the trillions of dollars, for which 10 to the 15th power would be adequate.

For more advanced use, the context specifies flags and traps. These are used to locate a number of numeric processing situations. These are signalled internally and can be trapped and treated as errors. The following list shows the kinds of signals that occur.

- `Clamped`. The exponent upper limit was exceeded.
- `DivisionByZero`, which is also a `DecimalException` and an `exceptions.ZeroDivisionError`.
- `Inexact`, which includes `Overflow` and `Underflow`. These indicate that the precision was exceeded and digits had to be dropped. In the case of repeating fractions, this isn't an error. In other cases, it may mean that the numbers are faulty or there are design problems.
- `InvalidOperation`. This often happens when working with the "special numbers" or doing invalid operations like taking square root of negative numbers.
- `Rounded`. Digits were discarded due to rounding. This is often a consequence of quantizing.
- `Subnormal`. The exponent lower limit was exceeded.

The default context traps `Overflow`, `InvalidOperation`, and `DivisionByZero` as errors.

Changing the Context. Generally, we modify the context that is automatically present at the beginning of our program. We get the context object and change the values we need to change. For example, the following will set the rounding rule to `ROUND_HALF_UP`. This rule applies to all `quantize()` operations that happen after this statement changes the context.

```
getcontext().rounding=ROUND_HALF_UP
```

Once in a while, we may have a function which uses a slightly different context from the rest of the program. In this case, we want to save the old context, make a change, and then put the old context back into place. We might do something like the following.

```
def someFunction():
    with localcontext() as ctx:
        ctx.rounding= ROUND_HALF_UP
        ctx.precision += 2
        # Do the real work
```

14.2.5 Decimal Exercises

1. **File Processing**.

   In the exercises at the end of *Files, Contexts and Patterns of Processing*, we saw some exercises that involved financial calculations. We converted all of the currency values to floating-point numbers. You can revisit those exercises, replacing the `float()` factory function with the `decimal` factory function. How cool is that?

2. **Stock Class Definitions**.

   In the exercises at the end of *Defining New Objects*, a few of the exercises worked with blocks of stock and stock positions. We didn't carefully specify how currency should be handled. You can revisit those exercises and include specific `decimal` factory functions in the `__init__()` methods to assure that proper `decimal` currency values are used.

   You'll need to make one other change. Your various constructors will need to use strings for prices instead of floating-point numbers.

   ```
   blockGM = [
   ShareBlock( purchDate='25-Jan-2001', purchPrice='44.89', shares='17' ),
   ShareBlock( purchDate='25-Apr-2001', purchPrice='46.12', shares='17' ),
   ShareBlock( purchDate='25-Jul-2001', purchPrice='52.79', shares='15' ),
   ShareBlock( purchDate='25-Oct-2001', purchPrice='37.73', shares='21' ),
   ]
   blockEK = [
   ShareBlock( purchDate='25-Jan-2001', purchPrice='35.86', shares='22' ),
   ShareBlock( purchDate='25-Apr-2001', purchPrice='37.66', shares='21' ),
   ShareBlock( purchDate='25-Jul-2001', purchPrice='38.57', shares='20' ),
   ShareBlock( purchDate='25-Oct-2001', purchPrice='27.61', shares='28' ),
   ]
   ```

   You may notice that the file-reading exercises involve reading strings from files. The class creation exercises create stock `ShareBlock` or `Position` objects using strings. You're now in a position to combine file reading and object creation, along with the `decimal` package to do real work in Python.

14.3 Time and Date Processing : The `time` and `datetime` Modules

Too often, programmers attempt to write their own date, time or calendar manipulations. Calendar programming is very complex, and there are a number of shoals that are not clearly charted. The use of the `time` and `datetime` modules makes our applications simpler, and much more likely to be correct. The clock and calendar are hopelessly complex data objects, best described by these modules.

In *Concepts: Point in Time and Duration*, we'll look at the concepts of a *point in time* and a *duration*. Then we'll look at the `datetime` module in *The datetime Module*. The formal definitions for some of the module is in *Formal Definitions in datetime*.

We'll look at the `time` module in *The time module*, and the formal definitions in *Formal Definitions in time*.

The Y2K Lessons Learned

In the late '90's, programmers like me scrambled to fix the cruft that had accumulated from decades of sloppy date calculations. The whole date thousand of programs shared a number of assumptions about dates that – in retrospect – were really bone-headed.
Invalid assumption one. A two-digit year was sufficient. When we write 3/18/85, we only write two digits, and we have to use context clues to figure out what the date means. In this case, if I say that it's the birthdate of my great-grandmother, you'd know that the date meant 1885. A two-digit short-hand year is fine for people, but unsuitable for computing. The century information is essential even it isn't shown to the human user of the software.
Invalid assumption two. Ordinary professional programmers can actually understand the complex and nuanced Gregorian calendar. One example of the complexity is the leap year rule: years divisible by 4, except years divisible by 100, including years divisible by 400.
Invalid assumption three. The service life of software is short enough that we'll write new software before January 1st, 2000. Software doesn't wear out; a well-written piece of software can be used for decades. The author's personal best is 17 years of continuous service before the software was replaced during a complete reworking of the business' computer systems.
A huge technical mistake that compounded the problem was to embed date calculations in application programs. The Python idea is the date calculations must be in a separate module, and any change or fix can be made in one place only and will fix every program that uses the module. The real core Y2K problem stemmed from a failure to isolate different concepts – like date – into separate modules.
The consequence of the assumptions and mistakes was side-tracking thousands of programmers. Instead of creating useful solutions to data processing problems, they were remediating program after program that had a date calculation somewhere inside it.

14.3.1 Concepts: Point in Time and Duration

To make best use of the `datetime` module, we must separate the closely-related concepts of point, distance and units of measure. We need to apply these concepts to time. The problem we have is that "day" is a duration (24 hours) as well as a unit (the 24th day of September.)

Point In Time. A *point in time* is also known as a *date*, *timestamp* or *datetime*. A single point in time can be measured to any degree of precision. When measured to the nearest whole day, we call it a date. When measured to the nearest second, we sometimes call it a datetime. Our operating system may measure to the nearest millisecond, and our database may measure to the nearest nanosecond. It is important to note that date, datetime and timestamp all refer to a point in time.

A datetime can be shown in a wide variety of formats, including or omitting any of a number of details. For example, the date 12/7/2005 is in the fourth quarter of the year, is in the 49th week, and is a Wednesday, but we don't often show these additional details. The Python `datetime.datetime` object has these additional attributes, but doesn't show them by default.

Sometimes confusion arises because a datetime can be mistaken for two things jammed together: a date and a time of day. A datetime is not two separate things. A datetime is a very precise point in time measured to the nearest second or minute, which includes a number of units of measurement (years, months, days, hours, minutes and seconds). A date is just a datetime that is only measured to the nearest day instead of the nearest minute.

Time of Day. A time of day (for example, 10:08 AM) can be one of two things. It can be part of a datetime, with the date information assumed; 10:08 AM could be the time portion of "10:08 AM 4/20/2007". Or, it can be a generic time of day used for scheduling purposes; in this case, there's no specific date. The generic time of day is a rare situation. A datetime which omits the details of the date is much more common.

Sometimes we display just a time of day to a person because the person can work out the date from the context in which information is displayed.

A point in time, unless it's a generic time used for scheduling, is part of a complete datetime object. We may *show* only the time of that more complete datetime object to the person using the software.

Duration. A *duration* is sometimes called a *delta time* or *offset*. Durations can be measured in various units like years, quarters, months, weeks, days, hours, minutes or seconds.

Since both a point in time and a duration can be measured in similar units, this can be confusing. Durations, for example, aren't a specific time of day (10:08 AM), but a number of hours (10 hours, 8 minutes).

Irregularity. Note that we measure time in units which have gaps and overlaps, mostly because the irregular concepts of month and year. If we only used days and weeks, life would be simpler. Months and years really throw a monkey wrench into the works.

For example, the durations of "90 days" and "3 months" are similar, but not exactly the same. For the simple durations like weeks, days, hours, minutes and seconds, the conversions are simple; we can normalize to days or seconds without any trouble or confusion.

Among the cultural times like months, quarters and years, the conversions are pretty clear. However, there are a lot of special cases. This makes converting back and forth between simple times and cultural times is very difficult.

If we think of a duration being measured in days, then one hour is 1/24 = 0.04166 and one minute is 1/60th of that = 0.0006944. On the other hand, we can think of a duration in seconds, then one day is 86,400 seconds. Both views are equivalent. Our operating systems, generally, like to work in seconds. Other software, like databases, prefer to work in days because that meets people's needs a little better than working in seconds.

Point and Duration Calculations. You can combine a timedelta and a datetime to compute new point in time. You can also extract a timedelta as the difference between two datetimes. Doing this correctly, however, requires considerable care. That's why this operation is done best by the `datetime` package.

There are two overall approaches to date and time processing in Python.

- The OS-friendly `time` module. This module has two different numeric representations for a point in time: a `time.struct_time` object or a floating-point number. The module works with durations as a floating-point number of seconds. It requires conversions between seconds and a `time.struct_time` object. While low-level, this module maps directly to the portable C libraries.
- The person-friendly `datetime` module. This module defines several classes for a point in time, plus a class for a duration. The `datetime.date`, `datetime.time`, `datetime.datetime` and `datetime.timedelta` classes embody considerable knowledge about the calendar, and remove the need to do conversions among the various representations.

Unless you have a need for C-language compatibility, you use the `datetime` module for all of your date and time-related computations. We'll present `datetime` first.

In addition to the `datetime` module, the `calendar` module also contains useful classes and methods to handle our Gregorian calendar. We won't look at this module, however, since it's relatively simple.

14.3.2 The `datetime` Module

The Gregorian calendar is extremely complex; some of that complexity reflects the irregularities of our planet's orbit around the Sun. One of the many complexities is the leap year, which has rules that are intended to create calendar years with integer numbers of days that approximate the astronomical year of about 365.2425 days.

The `datetime` module contains the objects and methods required to correctly handle the sometimes obscure rules for the Gregorian calendar. It is possible to use date information in a `datetime` object to usefully convert among the world's calendars. For details on conversions between calendar systems, see *Calendrical Calculations* [Dershowitz97]. Additionally, this package also provides for a time delta, which captures the duration between two datetimes.

One of the ingenious tricks to working with the Gregorian calendar is to assign an ordinal number to each day. We start these numbers from an *epochal date*, and use algorithms to derive the year, month and day information for that ordinal day number. Similarly, this module provides algorithms to convert a calendar date to an ordinal day number. Following the design in [Dershowitz97], this class assigns day numbers starting with January 1, in the (hypothetical) year 1. Since the Gregorian calendar was not defined until 1582, all dates before the official adoption are termed *proleptic*. This epoch date is a hypothetical date that never really existed on any calendar, but which is used by this class.

There are four classes in this module that help us handle dates and times in a uniform and correct manner.

`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 and `time.struct_time` objects for use with the `time` module, and ordinal day numbers.

`datetime.datetime` An instance of `datetime.datetime` has all the attribute for a complete date with the time information. There are a number of methods for creating `datetime.datetimes`s, and converting `datetime.datetimes`s to various other forms, like floating-point timestamps and `time.struct_time` objects for use with the `time` module, and ordinal day numbers.

`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.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.

There are a number of interesting date calculation problems that we can solve with this module. We'll look at the following recipes:

- *Getting Days Between Two Dates*
- *Getting Months Between Two Dates*
- *Computing A Date From An Offset In Days*
- *Computing A Date From An Offset In Months*
- *Input of Dates and Times*

Getting Days Between Two Dates. Because `datetime.datetime` objects have the numeric operators defined, we can create `datetime.datetime` objects and subtract them to get the difference in days and seconds between the two times. The difference between two `date` or `datetime` objects is a `timedelta` object.

```
>>> import datetime
d>>> d1 = datetime.datetime.now()
>>> d2 = datetime.datetime.today()
>>> d2 - d1
datetime.timedelta(0, 14, 439322)
>>> d1
datetime.datetime(2009, 7, 9, 6, 44, 19, 45987)
>>> d2
datetime.datetime(2009, 7, 9, 6, 44, 33, 485309)

Surf http://stackoverflow.com for about a minute.

>>> d3 = datetime.datetime.now()
>>> d3 - d1
datetime.timedelta(0, 95, 848826)
```

The difference between *d2* and *d1* was the object `datetime.timedelta(0, 14, 439322)`, which means zero days, 14 seconds and 439,322 microseconds.

The difference between *d3* and *d1* was the object `datetime.timedelta(0, 95, 848826)`, which means zero days, 95 seconds and 848,826 microseconds.

If we said `td= d3-d1`, then `td.days` is the number of days between two dates or datetimes. `td.seconds` is the number of seconds within the day, from 0 to 86400. The *seconds* attribute is zero if you get the difference between two `dates`, since they have to time information. `td.seconds/60/60` is the number of hours between the two datetimes.

If we do `td.days/7`, we compute the number of weeks between two dates. **Getting Months Between Two Dates**. The number of months, quarters or years between two dates uses the instance variables of the `datetime.datetime` object. If we have two variables, *begin* and *end*, we have to compute *month numbers* from the dates. A month number includes the year and month information.

We compute a month number for a date as follows:

```
endMonth= end.year*12+end.month
startMonth = begin.year*12+begin.month
endMonth - beginMonth
```

The result is the months between the two dates. This correctly handles all issues with months in the same or different years. **Computing A Date From An Offset In Days**. To computing a date in the future using an offset in days, we can add a `timedelta` object to a `datetime` object. The `timedelta` object can be constructed with a day offset or a seconds offset or both. In the following example, we'll compute the date which is 5 days in the future.

```
now= datetime.datetime.now()
now + datetime.timedelta(days=5)
```

The result of this calculation is a new `datetime` object.

We can do a similar calculation for weeks, we just use something like `6*7` to compute a date six weeks in the future. **Computing A Date From An Offset In Months**. When we want to compute a date that is offset by a number of months, quarters or years, we can use the `datetime.replace()` do to things like the following example. This will add three months to the current date to compute a future date. We use both year and month in the future date calculation to assure that the date wraps around the end of the year correctly.

```
>>> import datetime
>>> due = datetime.datetime.today()
>>> thisYM= due.year*12+due.month-1
>>> nextYM= thisYM + 3
>>> due.replace( year=nextYM//12, month=nextYM%12 + 1 )
datetime.datetime(2009, 10, 9, 6, 50, 10, 831933)
```

This will raise an `ValueError` if you try to create an invalid date like February 30th.

Note that this parallels our *Computing A Date From An Offset In Months* example. In both cases, we work with a month number that combines month and year into a single serial number. **Input of Dates and Times**. We get date and time information from three soures. We may ask the operating system what the current date or time is. We may ask the person who's running our program for a date or a time. Most commonly, we often process a file which has date or time information in it. For example, we may be reading a file of stocks with dates on which a trade occurred.

- **Getting a Date or Time From The OS**. We get time from the OS when we want the current time, or the timestamp associated with a system resource like a file or directory. The current time is created by the `datetime.datetime.now()` object constructor. See *Advanced File Exercises* for examples of getting file timestamps. When we get a file-related time, we get a floating-point number of seconds past the epoch date. We can convert this to a proper `datetime` with `datetime.datetime.fromtimestamp()`.

  ```
  >>> import os
  >>> import datetime
  >>> mtime= os.path.getmtime( "Makefile" )
  >>> datetime.datetime.fromtimestamp( mtime )
  datetime.datetime(2009, 6, 9, 21, 10, 26)
  ```

  The file named `Makefile` was modified 6/9/2009 at 9:10:26 PM.

- **Getting A Date or Time From A User**. Human-readable time information generally has to be parsed from a string. People write times and dates with an endless variety of formats that have some combination of years, days, months, hours, minutes and seconds.

  In this case, we have to first parse the time, using `datetime.datetime.strptime()`.

  ```
  >>> someInput= "3/18/85" # stand-in for raw_input()
  >>> inDT = datetime.datetime.strptime( someInput, "%m/%d/%y" )
  >>> inDT
  datetime.datetime(1985, 3, 18, 0, 0)
  ```

- **Getting a Date or Time From A File**. Files often have human-readable date and time information. However, some files will have dates or times as strings of digits. For example, it might be `20070410` for April 10, 2007. This is still a time parsing problem, and we can use `datetime.datetime.strptime()`.

  ```
  >>> someInput= "20070410" # stand-in for someFile.read()
  >>> aDate = datetime.datetime.strptime( someInput, "%Y%m%d" )
  >>> aDate
  datetime.datetime(2007, 4, 10, 0, 0)
  ```

14.3.3 Formal Definitions in `datetime`

We'll present some of the formal definitions of the `datetime.datetime` class, since it offers the most features. The `datetime.date` and `datetime.time` classes are simplifications of the `datetime.datetime` class.

We'll also look at formal definitions of the `datetime.timedelta` class.

In the definitions below, you'll see a distinction between UTC (Coordinated Universal Time, also known as Zulu Time and Greenwich Mean Time) and local time. Your local time is an offset from UTC time, and that offset varies if you have standard time and daylight time. The rules vary by county around the United States, making the time zone boundaries a rather complex problem.

However, your computer already has the localtime offset. It works internally in UTC, and converts the universal time into local time as needed. You can borrow this design pattern in your programs, also. If you need to share information widely, consider keeping track of dates and times in UTC inside your programs, and converting to local time for display and human input purposes.

class `datetime.datetime`

`datetime.today()` → datetime.datetime

   Current local date or datetime: same as `datetime.datetime.fromtimestamp( time.time() )`. See `now()` and `utcnow()` for variations that may produce more precise times.

`datetime.now()` → datetime.datetime
: Current local date or datetime. If possible, supplies more precision than using a `time.time()` floating-point time. See `utcnow()`.

`datetime.utcnow()` → datetime.datetime
: Current UTC date or datetime. If possible, supplies more precision than using a `time.time()` floating-point time. See `now()`.

`datetime.fromtimestamp(`*timestamp*`)` → datetime.datetime
: Current local date or datetime from the given floating-point time, like those created by `time.time()`.

`datetime.utcfromtimestamp(`*timestamp*`)` → datetime.datetime
: Current UTC date or datetime from the given floating-point time, like those created by `time.time()`.

`datetime.fromordinal(`*ordinal*`)` → datetime.datetime
: Current local date or datetime from the given ordinal day number. The time fields of the `datetime` will be zero.

`datetime.fromordinal(`*date*, *time*`)` → datetime.datetime
: Combine date fields from *date* with time fields from *time* to create a new `datetime` object.

The following methods return information about a given `datetime` object. In the following definitions, *dt* is a `datetime` object.

The `datetime.timedelta` object holds a duration, measured in days, seconds and microseconds. There are a number of ways of creating a `datetime.timedelta`. Once created, ordinary arithmetic operators like `+` and `-` can be used between `datetime.dateime` and `datetime.timedelta` objects.

14.3.4 The `time` module

The `time` module contains a number of portable functions needed to format times and dates. The `time` module can represent a moment in time in any of three forms.

- A tuple-like `time.struct_time` object which contains the year, month, day, hour, minute, second, weekday, day of the year and a daylight savings time flag. This is a handy form for calculations that involve years, months or quarters. This isn't the best way to handle weeks, days, hours, minutes or seconds.
- A floating-point number that measures time in seconds past an epoch. The epoch is typically January 1, 1970. This form is useful for calculations that involve weeks, days, hours, minutes or seconds. This isn't the best way to handle months, quarters or years.
- A string, in a variety of formats that you can specify. This is for presentation to users or for accepting input from users. However, it's hard to do any processing or math on this form.

It is an unfortunate consequence of our calendar and clock that we need to have three representations for a given date and time. There isn't a lot we can do about simplifying the calendar. All we can do is cope with it through a comprehensive set of Python libraries.

For the most part, the "seconds past epoch" representation of dates and times works well for a broad number of uses. It has the downside of being opaque when you try to look at it the number. What day of the week does "1247137510.6811409" fall on?

Seconds past an epoch time has the advantage of being a standard floating-point number. If you round it off to the nearest second, it is a 10-digit number. If you round it off to the nearest day (there are 86,400 seconds in a day), it's only a five digit number, for example: `14434`. We won't need a 6th digit until 2282.

The other common formats for date information are strings like `'2005-10-10 22:10:07'`. These must be converted from a string to one of the two numeric forms (seconds or `time.struct_time` object) before any useful processing can be done.

Here's a step-by-step example for displaying the current time (`time.time()`) using the GNU/Linux standard format for day and time. This shows a standardized and portable way to produce a time stamp.

```
>>> import time
>>> now= time.time()
>>> lt = time.localtime( now )
>>> time.strftime( "%x %X", lt )
'07/09/09 07:08:14'
>>> time.strftime( "%x %X", time.localtime( time.time() ) )
'07/09/09 07:08:47'
```

1. 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. We save this in the variable *now*.
2. The `time.localtime()` function uses the operating system's local timezone information to convert from a floating-point timestamp in UTC to `time.struct_time` object with the details of the current local date and time. We save this in the variable *lt*.
3. The `time.strftime()` function formats a `time.struct_time` object. We use the formatting codes that will do locale-specific time (`"%x"`) and date (`"%X"`) formatting. This allows the operating system's localization features to specify the format for date and time, assuring that the user's preferences are honored.

There are a number of interesting date calculation recipes that apply to the `time` module.

- *Getting Days Between Two Dates*
- *Getting Months Between Two Dates*
- *Computing A Date From An Offset In Days*
- *Computing A Date From An Offset In Months*
- *Input of Dates and Times*

These are some common recipes for date arithmetic. **Getting Days Between Two Dates**. To get the number of days between two dates, we calculate the difference between two floating-point timestamp representation of points in time. When we subtract these values we get seconds between two dates. Since there are 86,400 seconds in a day, we can convert this number of seconds to a number of days, weeks, hours or even minutes.

```
>>> import time
>>> now= time.time()

Surf http://stackoverflow.com for about 5 minutes.

>>> d2 = time.time()
>>> d2 - now
275.53438711166382
>>> (d2 - now) / 60
4.5922397851943968
```

The difference is in seconds. When we divide by 60, that's the difference in minutes. When we divide by 86400, that's the difference in days. **Getting Months Between Two Dates**. To get the number of months, quarters or years between two dates, we use the `time.struct_time` objects.

```
>>> start = time.localtime( prior_time )
>>> end = time.localtime( time.time() )
>>> start
(2009, 7, 9, 7, 8, 14, 3, 190, 1)
>>> end
(2009, 7, 9, 7, 17, 14, 3, 190, 1)
```

```
>>> endMonth= end.tm_year*12+end.tm_mon
>>> startMonth = start.tm_year*12+start.tm_mon
>>> endMonth - startMonth
```

In this case, we've created *start* and *end* using `time.localtime()` conversions. We could also create the `time.struct_time` objects from parsing user input.

Given two `time.struct_time` objects, *start* and *end*, we must compute *month numbers* that combine year and month into a single integer value that we an process correctly. **Computing A Date From An Offset In Days**. To compute a date in the future using weeks or days, we can add an appropriate offset to a floating-point timestamp value. Since the floating-point timestamp is in seconds, a number of days must be multiplied by 86,400 to convert it to seconds. A week is $7 \times 86400 = 604,800$ seconds long.

```
>>> next_week = 7*86400 + time.time()
>>> time.localtime( next_week )
(2009, 7, 16, 7, 23, 21, 3, 197, 1)
```

Computing A Date From An Offset In Months. To compute a date in the future using a number of months or years, we have to create the `time.struct_time` object for the base date, and then update selected elements of the tuple. Once we've updated the structure, we can then converting it back to a floating-point timestamp value using `time.mktime()`.

Note that we have to be careful to handle the year correctly. The easiest way to be sure this is done correctly is to do the following:

1. Create a "month number" from the starting year and month, y*12+m.
2. Add a number of months (or 12 times the number of years) to this month number.
3. Extract the year and month from the resulting value by dividing by 12 (to get the new year) and using the remainder as the new month.

```
>>> import time
>>> now= time.localtime( time.time() )
>>> thisYM= now.tm_year*12+now.tm_mon-1
>>> nextYM= thisYM+3
>>> dueYear, dueMonth =nextYM//12, nextYM%12+1
>>> nextSec= time.mktime( (dueYear,dueMonth,now.tm_mday,0,0,0,0,0,0) )
>>> time.localtime( nextSec )
(2009, 10, 9, 1, 0, 0, 4, 282, 1)
```

Input of Dates and Times. When we get a time value, it's generally in one of two forms. Sometimes a time value is represented as a number, other times it's represented as a string.

- **Getting a Date or Time From The OS**. We often get the OS time when we want the current time, or the timestamp associated with a system resource like a file or directory. The current time is available as the `time.time()` function.

  See *Advanced File Exercises* for examples of getting file timestamps. When we get a file-related time, the OS gives us a floating-point number of seconds past the epoch date. There are two kinds of processing: simple display and time calculation.

  To display an OS time, we need to convert the floating-point timestamp to a `time.struct_time`. We use `time.localtime()` or `time.gmtime()` to make this conversion. Once we have a `time.struct_time`, we use `time.strftime()` or `time.asctime()` to format and display the time.

  ```
  >>> import os
  >>> import time
  >>> mtime= os.path.getmtime( "Makefile" )
  >>> time.localtime( mtime )
  (2009, 6, 9, 21, 10, 26, 1, 160, 1)
  ```

```
>>> time.strftime( "%x %X", time.localtime(mtime) )
'06/09/09 21:10:26'
```

- **Getting A Date or Time From A User**. Human-readable time information generally has to be parsed from a string. Human-readable time can include any of the endless variety of formats with some combination of years, days, months, hours, minutes and seconds. In this case, we have to first parse the time, creating `time.struct_time`. The simplest parsing is done with `time.strptime()`.

  Here's an example of parsing an input string from a person. This will create `time.struct_time` called *theDate*.

  ```
  >>> import time
  >>> someInput= "3/18/85" # stand-in for raw_input()
  >>> theDate= time.strptime( someInput, "%m/%d/%y" )
  >>> theDate
  (1985, 3, 18, 0, 0, 0, 0, 77, -1)
  ```

- **Getting a Date or Time From A File**. Files often have human-readable date and time information. However, some files will have dates or times as strings of digits. For example, it might be 20070410 for April 10, 2007. This is still a time parsing problem, and we can use `time.strptime()` to pick apart the various fields. We can parse the 8-character string using

  ```
  >>> import time
  >>> someInput= "20070410" # stand-in for someFile.read()
  >>> time.strptime( someInput, "%Y%m%d" )
  (2007, 4, 10, 0, 0, 0, 1, 100, -1)
  ```

14.3.5 Formal Definitions in `time`

The three representations for time in this module (floating-point seconds after the epoch, a `time.struct_time` object, and a string) leads to a number of conversations back and forth between these three forms.

We'll present some of the formal definitions of the `time` module. We'll look at the `struct_time` object, then at functions which work with the `struct_time` view of a point in time, and the floating-point seconds view of a point in time. We'll finish with functions that convert back and forth to strings.

The `time.struct_time` Object. A `time.struct_time` object is a little bit like a nine-element tuple, and a little bit like a class definition. You create a proper `time.struct_time` object via a little 2-step dance which we'll show below. First, we'll look at the object itself.

A `time.struct_time` object has the following instance variables. Each of these is a read-only attribute; you can't update a `time.struct_time` object.

tm_year Year (four digits, e.g. 1998)

tm_mon Month (1-12)

tm_mday Day of the month (1-31)

tm_hour Hours (0-23)

tm_min Minutes (0-59)

tm_sec Seconds (0-61), this can include leap seconds on some platforms

tm_wday Weekday (0-6, Monday is 0). This can be hard to figure out when you're creating a new time. Fortunately, you can supply -1, and the `time.mktime()` function will determine the weekday correctly.

tm_yday Day of the year (1-366). This can be hard to figure out when you're creating a new time. Fortunately, you can supply -1, and the `time.mktime()` function will determine the day of the year correctly.

tm_isdst Daylight savings time flag: 0 is the regular time zone; 1 is the DST time zone. -1 is a value you can set when you create a time for the `mktime()`, indicating that `mktime()` should determine DST based on the date and time.

Working With `time_struct` Objects. The `time` module includes the following functions that create a `time.struct_time` object. The source timestamp can be a floating-point "seconds-past-the-epoch" value or a formatted string.

`time.gmtime`(*seconds*) → time.struct_time
Convert a timestamp with seconds since the epoch to a `time.struct_time` object expressing UTC (a.k.a. GMT).

`time.localtime`(*seconds*) → time.struct_time
Convert a timestamp with seconds since the epoch to a `time.struct_time` expressing local time.

`time.strptime`(*string*, *format*) → time.struct_time
Parse the *string* using the given *format* to create a `time.struct_time` object expressing the given time string. The format parameter uses the same directives as those used by `time.strftime()`; it defaults to `"%a %b %d %H:%M:%S %Y"` which matches the formatting returned by the `time.ctime()` function.

If the input can't be parsed according to the format, a `ValueError` is raised.

```
someInput= "3/18/85" # stand-in for raw_input()
theDate= time.strptime( someInput, "%m/%d/%y" )
print(theDate)
```

When you want to create a proper `time.time_struct` object, you'll find that there are a few fields for which you don't know the initial values. For example, it's common to know year, month, day of month, hour, minute and second. It's rare to know the day of the year or the day of the week. Consequently, you have to do the following little two-step dance to create and initialize a `time.struct_time`.

In this example, we'll create a proper structure for 4/21/2007 at 2:51 PM. We can fill in six of the nine values in a `time.struct_time` tuple. We just throw -1 in for the remaining values.

```
ts= time.localtime( time.mktime( (2007,4,21,14,51,00,-1,-1,-1) ) )
```

The value for *ts*, `(2007, 4, 21, 14, 51, 0, 5, 111, 1)`, has the day of week (0 is Monday, 5 is Saturday) and day of year (111) filled in correctly.

Working With Floating-Point Time. The `time` module includes the following functions that create a floating-point "seconds-past-the-epoch" value. This value can be generated from the operating system, or converted from a `time.struct_time` object.

Because a floating-point time value is a simple floating-point number, you can perform any mathematical operations you want on that number. Since it is in seconds, you can divide by 86,400 to convert it to days.

`time.time`() → float
Return the current timestamp in seconds since the Epoch. Fractions of a second may be present if the system clock provides them.

```
now= time.time()
```

`time.mktime`(*struct_time*) → float
Convert a `time.struct_time` object to seconds since the epoch. The weekday and day of the year fields can be set to -1 on input, since they aren't necessary. However, the DST field is used.

In this example, we convert a **`time.struct_time`** object into a list so we can update it. Then we can make a floating-point time from the updated structure.

```
lt= time.localtime( time.time() )
nt= list( lt )
nt[1] += 3 # add three to months attribute
future= time.mktime( nt )
```

Working with String Time. The following functions of the `time` module create time as formatted string, suitable for display or writing to a log file.

`time.strftime`(*format*, *struct_time*) → string
Convert the **`time.struct_time`** object, *structure* to a string according to the *format* specification. A format of "%x %X" produces a date and time.

```
lt= time.localtime( time.time() )
print(time.strftime( "%Y-%m-%dT%H:%M:%S", lt))
```

`time.asctime`(*struct_time*) → string
Convert a **`time.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 floating-point time in seconds since the Epoch to a string in local time. This is equivalent to '`time.asctime(time.localtime(seconds))`'.

If no time is given, use the current time. `time.ctime()` ` does the same thing as ``time.asctime(time.localtime(time.time()))`.

Additional Functions and Variables. These are some additional functions and variables in the `time` module.

This function appears in the time module. The `sched` module reflects a better approach to time-dependent processing.

`time.sleep`(*seconds*)
Delay execution for a given number of seconds. The argument may be a floating-point number for subsecond precision. Operating system scheduling vagaries and interrupt handling make this function imprecise.

`time.clock`() → float
Return the CPU time or real time since the start of the process or since the first call to `clock()`. This has as much precision as the system is capable of recording.

The following variables are part of the `time` module. They describe the current locale.

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. Your operating system has ways to define your locale.

time.timezone Difference in seconds between UTC and local Standard time. Often a multiple of 3600 (all US timezones are in whole hours). Your operating system has ways to define your locale.

time.tzname The name of the timezone.

Conversion Specifications. When we looked at Strings, in *Sequences of Characters : str and Unicode*, we looked at the % operator which formats a message using a template and specific values. The `strftime()` and `strptime()` functions also use a number of conversion specifications to convert between `time.struct_time` and strings.

The following examples show a particular date (Satuday, August 4th) formatted with each of the formatting strings.

Table 14.1: Overall Formatting

%c	Locale's appropriate full date and time representation	'Saturday August 04 17:11:20 2001'
%x	Locale's appropriate date representation	'Saturday August 04 2001'
%X	Locale's appropriate time representation	'17:11:20'
%%	A literal '%' character	'%'

Table 14.2: Date Formatting

%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'
%d	Day of the month as a 2-digit decimal number	'04'
%j	Day of the year as a 3-digit decimal number	'216'
%m	Month as a 2-digit decimal number	'08'
%U	Week number of the year (Sunday as the first day of the week)	'30'
%w	Weekday as a decimal number, 0 = Sunday	'6'
%W	Week number of the year (Monday as the first day of the week)	'31'
%y	Year without century as a 2-digit decimal number	'01'
%Y	Year with century as a decimal number	'2001'

Table 14.3: Time Formatting

%H	Hour (24-hour clock) as a 2-digit decimal number	'17'
%I	Hour (12-hour clock) as a 2-digit decimal number	'05'
%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'
%Z	Time zone name (or '' if no time zone exists)	''

Tip: Debugging `time`

Because of the various conversions, it's easy to get confused by having a floating-point time and a `time_struct` time. When you get `TypeError` exceptions, you are missing a conversion between the two representations. You can use the `help()` function and the Python Library Reference (chapter 6.10) to sort this out.

14.3.6 Date and Time Exercises

1. **Annualized ROI**.

   In order to compare portfolios, we might want to compute an annualized ROI. This is ROI as if the stock were held for exactly one year. In this case, since each block has different ownership period, the ROI must be adjusted to be a full year's time period. We then have a basis for comparing ROI's among various positions. We can use this to return an average of each annual ROI weighted by the current value of the position.

See *Defining New Objects* exercises. This calculation is a collaboration between each block of `ShareBlock` and `Position`. As with the value calculations, a block-by-block calculation is added to `ShareBlock` and a higher-level reduction algorithm is used in `Position`.

The annualization requires computing the duration of stock ownership. The essential feature here is to parse the date string to create a time object and then get the number of days between two time objects.

Given the sale date, purchase date, sale price, *sp*, and purchase price, *pp*.

Compute the period the asset was held. There are two choices:

- 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 )
  ```

- Use `datetime.datetime()` to create `datetime.datetime` objects for the sale date, *s*, and purchase date, *p*. The weighting, *w* is computed as the following, which truncates the difference to whole days.

  ```
  w= ( s - p ).days/365.2425
  ```

Here's another code snippet that does some 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
>>> _ / 365.2425
0.41342395805526466
```

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.

If we held the stock for 151 days, that is .413 years. If the return for the 151 days was 3.25%, the return for the whole year could have been 7.86%. In order to provide a rational basis for comparison, we use this annualized ROI instead of ROI's over different durations.

All of this processing must be encapsulated into a method that computes the ownership duration.

`time.`**`ownedFor`**`(`*saleDate*`)` → float
    This method computes the days the stock was owned.

`time.`**`annualizedROI`**`(`*salePrice*, *saleDate*`)` → float
    We would need to add an `annualizedROI()` method to the `ShareBlock` 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.

2. **Date of Easter.**

The following algorithm is based on one by Gauss, and published in Dershowitz and Reingold, *Calendrical Calculations* [Dershowitz97].

Date of Easter after 1582

Let Y be the year for which the date of Easter is desired.

(a) **Century**. Set $C \leftarrow \lfloor \frac{Y}{100} \rfloor + 1$. (When Y is not a multiple of 100, C is the century number; i.e., 1984 is in the twentieth century.)

(b) **Shifted Epact**. Set $E \leftarrow (14 + 11 \times (Y \bmod 19) - \lfloor \frac{3 \times C}{4} \rfloor + \lfloor \frac{(5+8 \times C)}{25} \rfloor) \bmod 30$. (Note that the $\lfloor a \div b \rfloor$ mathematical operation is usually implemented by the `a//b` operation in Python. Also, `int(a/b)` will work.)

(c) **Adjust Epact**. If $E = 0$ **or** $[E = 1$ **and** $10 < (Y \bmod 19)]$: add 1 to E.

(d) **Paschal Moon**. Set $R \leftarrow$ datetime.date(Y, April, 19). April is month number 4. (You'll need to use a `datetime.timedelta( days=e )` object.)

(e) **Easter**. Locate the Sunday after the Paschal Moon.

Set $Q \leftarrow P + 7 - (P \bmod 7)$.

(For $P \bmod 7$, you'll need to use `p.toordinal()`. You'll need to make a `datetime.timedelta` out of the offset, $7 - (P \bmod 7)$.)

(f) **Return the Date**. Q is the date of Easter.

Recently, Easter fell on 4/11/2004 and 3/27/2005.

14.3.7 Date and Time FAQ

Why are there two modules, `datetime` and `time` ? This is a lesson on history, really. The `time` module came first, and is a Python wrapper around the underlying C standard library. The `datetime` module is a better reflection of the things we actually do with dates and times.

The historical evolution shows something of how software captures knowledge. The `time` module shows a very technical understanding of time: it is a number of seconds, which is easy to gather from the hardware clock. This can be resolved to a date, but any processing has to be done in seconds.

The `datetime` module, on the other hand, is organized around days, which is the way we look at our calendar. Working in days is more typical of a large number of real-world problems.

As people did more and more useful work with computers, they realized that the `time` module was too simplistic; it didn't define a useful abstraction for dates and times. Based on real-world problems, people wrote the `datetime` module to solve those problems.

Why does datetime have to be so complicated? It's true, our calendar is complicated. If we only had 12 months of 30 days each, our lives would be simpler. If someone could just speed up our orbit around the sun by 1.438%, this wouldn't be so complex.

Sadly, there's just no simple way to convert between days and months. A `datetime` object, thankfully, conceals all of the details, allowing us to work with an object that has both day information and month information.

Why does the `time` module have two representations for a point in time? Why is there no formal duration that's compatible with a `time.struct_time` object?

The time-as-seconds representation is a duration that's technically very simple. It turns out that a point in time can be viewed as a duration measured against an epochal date. Everything's a floating-point number. Not much can go wrong.

However, people like to see their calendar, not a number of seconds past an epochal date. So the `time.struct_time` was added just to make it easy to display time values or accept time-oriented inputs. Further, for business rules that involve months, the `time.struct_time` information is useful.

Both time-as-seconds, and time-as-structure are required. Some programs will use one representation more than the other.

14.4 Text Processing and Pattern Matching : The `re` Module

The `re` module is the core of text processing. The `re` module provides sophisticated ways to create and use *regular expressions*. A regular expression is a kind of formula that specifies patterns in text strings.

The name "regular expression" comes from an earlier mathematical treatment of "regular sets". For our purposes, the set theory will be boiled away. We're still stuck with the phrase.

Regular expressions give us a simple way to specify a set of related strings by describing the pattern they have in common. We write a pattern to summarize some set of matching strings. This pattern string can be compiled into an object that efficiently determines if and where a given string matches the pattern.

For example the pattern `ab.*` describes the set of words that includes "abacterial" "abandoners" and "abandoning" among many others. Looking at it from the other direction, the string "abashments" matches the pattern, where the phrase "academical" does not match the pattern.

In this chapter, we'll look at why this pattern matching even matters in *How Does Pattern Matching Help Us?* Then we'll look at how we write these regular expression patterns in *How To Create Patterns Using Regular Expressions*. Once we have the pattern, we'll look at how we use it in *Objects We Use For Pattern Matching and Parsing*.

We'll look at a big example in *Some Examples*.

14.4.1 How Does Pattern Matching Help Us?

There are a number of common problems that we have to solve when processing strings. When we get strings as input from files, as a response to `raw_input()`, or from a GUI, we often need to look at the string as meaningful groups of characters, not as individual characters.

Since a `str` is a sequence, we're limited to doing things with single characters or simple slices. Without a lot of fancy footwork, we're limited to simple contiguous substrings at fixed positions.

What if we need some flexibility? It is very helpful to allow people to type variable amounts of *whitespace* – spaces, tabs, etc. – in a file. Also, people like some flexibility in entering numbers. We don't want to force them to type "03/08/1987" with those silly-looking '0's in front. We'd like to accept "03/08/1987" as gracefully as "3/8/1987".

Processing text can take one of three common forms.

1. *Match* a string against a given pattern to see if it is valid or not. Matching operations are *anchored* at the beginning of the string, and compare the beginning of the string to the pattern.
2. *Search* a string for the presence of given pattern. Searching operations are not anchored, and locate the first place in the string that the pattern occurs.
3. *Parse* a string using a given pattern, breaking it into substrings based on the content, not fixed slices.

For example, a file may contain lines like `"Birth Date:  3/8/87"` or `"Birth Date:  12/02/87"`. When we're reading lines like these from the file, we may do any of the following.

- Matching to determine that the string has the right pattern of text. In this case, a matching pattern might be `"Birth Date:"` followed by *digits* `/` *digits* `/` *digits*. A match pattern must be found at the beginning of the target string.
- Searching for the date pattern. A searching pattern could be *digits* `/` *digits* `/` *digits*. A search pattern can be found anywhere within the string.
- We may further parse the date string to extract groups of digits for month, day and year. A parsing pattern can separate the various digit groups from the surrounding context.

We can accomplish these matching, searching and parsing operations with the `re` module in Python. A *regular expression* (RE) is a rule or pattern used for matching, searching and parsing strings.

The Filename Wildcard. The fairly simple "wild-card" filename matching rules are kinds of regular expressions, also. These rules are embodied in two packages that we looked at in *Advanced File Exercises*: `fnmatch` and `glob`.

The filename regular expressions in `fnmatch` and `glob` use special characters that don't have their usual literal meaning. When we write a glob pattern, characters simply match themselves. However, the `*` character matches any sequence of characters in a file name. The `?` character matches any single character in a file name.

The `re` module provides considerably more sophisticated pattern matching capabilities than these simple rules. It uses the same principle: some punctuation marks have special meanings as part of pattern specification.

File Searching. An example program which does this is called **grep**. This is a GNU/Linux application program; the name means Global Regular Expression Print. (Windows users may be familiar with the **findstr** DOS command, which does approximately the same thing.)

The **grep** (or **findstr**) program reads one or more files, searches for lines that match a given regular expression and prints the matching lines.

Using Regular Expressions. The general recipe for using regular expressions in your program is the following.

1. Be sure to include `import re`.
2. Define the pattern string. We write patterns as string constants in our program.
3. Evaluate the `re.compile()` function to create a `re.Pattern` object. This `re.compile()` function is a factory that creates usable pattern objects from our original pattern strings. The pattern object will do the real work of matching a target string against your regular expression.

   Usually we combine the pattern and the compile.

   ```
   >>> date_pattern = re.compile( "Birth Date: +(.*)" )
   ```

4. Use the `re.Pattern` object to match or search the candidate strings. The result of a successful match or search will be a `re.Match` object. In a sense, the `Pattern` object is a factory that creates `Match` objects from string input.

   ```
   >>> match = date_pattern.match( "Should Not Match" )
   >>> match
   >>> match = date_pattern.match( "Birth Date: 3/8/87" )
   >>> match
   <_sre.SRE_Match object at 0x82e60>
   ```

When a string doesn't match the pattern, the pattern object returns `None` (which is equivalent to `False`.)

A successful match creates a `re.Match` object; and any object is equivalent to `True`. We can use the match object in an **if** statement.

5. Use the information in the `re.Match` object to parse the string. The match object is only created for a successful match, and provides the details to help us work with the original string.

```
>>> match.group()
'Birth Date: 3/8/87'
>>> match.group(1)
'3/8/87'
>>> match.groups()
('3/8/87',)
```

Pattern as Production Rule. One way to look at a regular expression is as a production rule for constructing strings. You can think of the pattern as the rule producing a giant collection of all possible strings.

When you use the pattern to matching a target, you're looking for your target string in that giant set of possibilities.

As a practical matter, the Regular Expression module doesn't actually enumerate all of the strings that a pattern describes. The set of possible strings could be infinite.

Pragmatically, the match algorithm looks at each clause of your regular expression pattern and locates the matching characters in the candidate string. If the next character in the string matches the next clause in the regular expression rule, the algorithm goes forward. When the algorithm runs out of clauses in the pattern, it has found a match.

In many cases, a clause in the pattern will have alternatives. In this case, the algorithm places bookmarks in the target string and pattern at each alternative choice. If the next character in the target string doesn't match the next clause in the pattern, then the algorithm backtracks and tries a different choice in the pattern. In this way, the matching tries out the various alternatives in the pattern, looking for some way to match the entire pattern against the string.

For example, a Regular Expression pattern could be `"aba"`. This production rule describes a string created from `a`, followed by `b`, followed by `a`. This simple rule only builds one possible string; consequently, only candidate strings containing the exact sequence `"aba"` will be found by the pattern's `match()` method.

A more complex RE pattern could be `"ab*a"`. This production rule describes a string created from `a`, followed by any number of `b`, followed by `a`. This describes an infinite set of strings including `"aa"`, `"aba"`, `"abba"`, etc. Note that the phrase "any number of" includes zero. That's why `"aa"` matches: it has zero `b`.

Note that the `*` character means "repeat the previous RE". This is different from the way `fnmatch` works. We'll explore the special characters in the `re` module in detail in the next section.

14.4.2 How To Create Patterns Using Regular Expressions

We'll cover the basics of creating and using RE's in this section. The full set of rules is given in section 4.2.1 of the *Python Library Reference* document [PythonLib]. This is a deep subject, and you can find several books that will help you unlockl the secrets of regular expressions.

To understand the rules, we have to make a distinction between ordinary characters and special characters. Most characters like letters and numbers are ordinary, they match what they appear to mean. For example, an `x` in a pattern matches the letter `x`, nothing more.

Some characters, however, have special meanings. Mostly these are punctuation marks; they don't match a character, but they are a pattern or a modification to a previous pattern. For example, a `.` in a pattern doesn't match the period character, it matches *any* single character.

But what if we want to match a `.`? We must *escape* that special meaning by using a `\` in front of the character. For example, `\.` escapes the special meaning that `.` normally has; it creates a single-character RE that matches only the character `.`.

Additionally, some ordinary characters can be made special with the escape character, `\`. For instance `\d` does mot match `d`, it matches any digit; `\s` does not match `s` it matches any whitespace character.

Any ordinary character, by itself, is a RE. Example: `"a"` is a 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. A special character is an RE when it is escaped with `\`. For example, `.` and `*` are special characters, but `\.` and `\*` are simple one-character RE's.

The special character `.` is a RE that matches any single character. Example: `"x.z"` is a RE that matches the strings like `"xaz"` or `"x9z"`, but doesn't match strings like `"xabz"` or `"xz"`.

The special characters `[]` create a RE that matches any one of the characters in a set defined by the characters in the `[]`. Example: `"x[abc]z"` matches any of `"xaz"`, `"xbz"` or `"xcz"`.

> A range of characters can be specified using a `-`. The character before and after the `-` must be in proper order. For example `"x[1-9]z"`.
>
> 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-z0-9_]"`.
>
> To include a `-`, it must be the first or last character in the `[]`s. If `-` is not first or last, then it indicates a range or characters.
>
> A `^` must not be the first character in the `[]`s. If `^` is first, it modifies the meaning of the `[]`s.
>
> Some common sets of characters have shorter names. `[0-9]` can be abbreviated `\d` (*d* for digit). `[ \t\n\r\f\v]` can be abbreviated `\s` (*s* for space). `[a-zA-Z0-9_]` can be abbreviated `\w` (*w* for word).

The special character `^` modifies the brackets `[^...]`. This creates an RE 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.

> To include a `-`, it must be the first or last character in the `[]`s.
>
> Some common sets of characters have shorter names. `\D` (*D* for non-digits) is the same as `[^0-9]`, the opposite of `\d`. `\S` (*S* for non-space) is the same as `[^ \t\n\r\f\v]`, the opposite of `\s`. `\W` (*W* for non-word) is the same as `[^a-zA-Z0-9_]`, the opposite of `\w`.

An RE can be formed from concatenating RE's. Example: `"a.b"` is three regular expressions, the first matches `a`, the second matches any character, the third matches `b`. While this may seem obvious, it's a necessary rule that helps us figure out which RE's are modified by the `*` or `|` operators.

> This is perhaps the most important rule for defining regular expressions. This rule tells us that we can put any number of one-part regular expressions together in a sequence to make a new, longer RE.
>
> Note that there's no special character that puts RE's together; the sequence of RE's is implied. This is similar to the way mathematicians imply multiplication by writing symbols next to each other. For example, $2\pi r$ means $2 \times \pi \times r$.

An RE can be a group of RE's with `()`. This creates a single RE that is composed of multiple parts. Also, this defines parts of the string that will be captured by the `Match` object.

Example: `"(ab)c"` is a regular expression composed of two regular expressions: `"(ab)"` (which, in turn, is composed of two RE's) and `"c"`. This matches the string `"abc"`. This grouping is used with the repetition operators (`*`, `+`, `?`) shown below, and the alternative operator, `|`.

`()` also identify RE's for parsing purposes. The elements matched within `()` are remembered by the regular expression processor and set aside in the resulting `Match` object. By saving matched characters, we can decompose a string into useful groups.

An RE can be repeated using `*` , `+` or `?` Several repeat constructs are available: `"x*"` repeats `x` zero or more times; `"y+"` repeats `y` 1 or more times; `"z?"` repeats `z` zero or once, it makes the previous RE optional.

Example: `"1(abc)*2"` matches `"12"` (zero copies of `abc`) or `"1abc2"` or `"1abcabc2"`, etc. Since the `(abc)` part of this pattern uses `()`s, the sequence of expressions is repeated as a whole. The first match, against `"12"`, is often surprising; but there are zero copies of `abc` between `1` and `2`.

Example: `"1[abc]*2"` matches `"12"` or `"1a2"` or `"1b2"` or `"1abacab2"`, etc. Since the `[abc]` part of this pattern uses `[]`s, any one of the characters in the `[]` will match. The first match, against `"12"`, is often surprising; but there are zero instances of any of the `abc` character set between `1` and `2`.

Two RE's are alternatives, using `|`. The alternative construct allows you to combine a number of different rules into a single pattern. For example, you might have two allowed forms for dates: `mm/dd/yyyy` or `dd-mon-yyyy`. You might write the following pattern: `r"(\d+/\d+/\d+)|(\d+-\w+-\d+)"` to match either alternative.

The character `^` is an RE that only matches the beginning of the line.

The chacaters `$` is an RE that only matches the end of the line.

Example: `"^$"` matches a completely empty line.

14.4.3 Some Examples

Match Some Dates. Here's an example of a pattern to match two different kinds of dates. We'll use the `re.compile()` function to build a pattern from our original pattern specification string. Once we have this pattern, we'll use it to match a number of candidate strings. We'll examine the groups which matched successfully to see what happened.

```
>>> import re
>>> p = "(\d+/\d+/\d+)|(\d+-[a-zA-Z]+-\d+)"
>>> pat = re.compile( p )
>>> pat.match( "9/10/5" )
<_sre.SRE_Match object at 0x68d58>
>>> _.groups()
('9/10/5', None)
>>> pat.match( "10-sep-56" )
<_sre.SRE_Match object at 0x68d58>
>>> _.groups()
(None, '10-sep-56')
>>> pat.match( "hi mom" )
```

1. We import the `re` module.
2. This is the pattern specification string, it has two RE's with the alternative operator, `|`. One alternative has a sequence of five RE's: `\d+`, `/`, `\d+` `/` and `\d+`. The `\d+` RE is the set of digits, repeated one or more times. The other alternative is also five RE's: `\d+`, `-`, `[a-zA-Z]+`, `-` and `\d+`. The `[a-zA-Z]+` is a set of letters repeated one or more times. The set of letters uses two range specifications to cover the ASCII alphabet.
3. We compile the pattern string to make the `Pattern` object named *pat* that will do the real work.

4. We apply our pattern against the candidate string `"9/10/56"`. This creates a `Match` object, which means that the string matched the pattern. When we evaluate the `groups()` method of a `Match` object, we get a tuple of the `()` groups in the pattern. The first set of `()`s matched `9/10/56`. The second set of `()`s didn't match anything.
5. We apply our pattern against the candidate string `"10-sep-56"`. This creates a `Match` object, which means that the string matched the pattern. When we evaluate the `groups()` method of a `Match` object, we get a tuple of the `()` groups in the pattern. The first set of `()`s didn't match anything. The second set of `()`s matched `10-sep-56`.
6. We apply our pattern against the candidate string `"hi mom"`. The response is `None`, which isn't shown. Because this expression did not create a `Match` object, it means that the string did not match the pattern.

Match A Property File Line. This pattern matches the kind of line that is often found in a properties file or a configuration file.

```
"\s*(\w+)\s*[:=]\s*(.*)"
```

There are six regular expressions:

- `\s*` is zero or more whitespace characters. This allows any amount of indentation.
- `(\w+)` is any character in the set A-Z, a-z, 0-9, _ repeated one or more times. This is a common definition of a "word". It is a broad definition and will also capture a sequence of digits as a "word". The `()`s will capture this identifier for parsing purposes.
- `\s*` is zero or more whitespace characters. This allows any amount of space after the identifier.
- `[:=]` matches either `:` or `=` as a separator between the identifier and the value.
- `\s*` is zero or more whitespace characters. This allows any amount of space after the separator.
- `(.*)` matches any number of characters, this will match the rest of the string. The `()`s will capture this value for parsing purposes.

Match a Time. Here'a pattern to match various kinds of times with `hh:mm:ss` and `hh:mm:ss.sss` formats.

```
"(\d+):(\d+):(\d+\.?\d*)"
```

This pattern matches a one or more digits with `(\d+)`, 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. Again, the punctuation marks are quietly dropped, since we only want to process the numbers.

A Python Identifier. This is a pattern which defines a Python identifier.

```
"[_A-Za-z][_A-Za-z1-9]*"
```

This embodies the rule of starting with a letter or `_`, and containing letters, digits or `_`.

```
"^\s*import\s"
```

The pattern above matches a Python **import** statement. It matches the beginning of the line with `^`; it matches zero or more whitespace characters with `\s*`; it matches the sequence of letters `import`; it matches one more whitespace character, and ignores the rest of the line.

14.4.4 Objects We Use For Pattern Matching and Parsing

There are several processing steps that we use with regular expressions. As we showed in the processing recipe above, the most common first step is to compile the RE definition string to make a `Pattern` object.

This object can then be used to match or search candidate strings. A successful match returns a `Match` object with details of the matching substring.

He's the formal definition for the `re.compile()` function of the `re` package. This translates an RE string into a `Pattern` object that can be used to search a string or match a string.

`re.compile(`*string*`)` → Pattern
: Create a `Pattern` object from an RE string. The object that results is for use in searching or matching; it has several methods, including `match()` and `search()`.

  The following example shows the pattern `r"(dd):(dd)"` which should match strings which have two digits, a `:`, and two digits. We'll match the candidate string `"23:59"`, which produces a `Match` object. When we try to match the string `"hi mom"`, we get result of `None`.

```
>>> import re
>>> hhmm_pat= re.compile( r"(\d\d):(\d\d)" )
>>> hhmm_pat.match( "23:59" )
<_sre.SRE_Match object at 0x68d58>
>>> _.groups()
('23', '59')
>>> hhmm_pat.match( "hi mom" )
```

There are some other options available for `re.compile()`, see the *Python Library Reference*, [PythonLib] section 4.2, for more information.

The raw string notation (`r"pattern"`) is generally used to simplify the \s required. Without the raw notation, each \ in the string would have to be escaped by a \, making it \\. This rapidly gets cumbersome.

Important: Confusing Class Names

As you work though the various examples, you'll see that the `type()` claims the object class names are `SRE_Pattern` and `SRE_Match`. We've fudged the class names in the book to make the explanation simpler. Also, in the future, there may be other, alternative RE packages, and the class names may be slightly different.

When we say `import re`, clearly something in the `re` module is then importing and using a module name `_sre`.

We don't need to know much more than this. That's why the names don't precisely match what we think they should say based on other, simpler, Python modules.

The following methods are part of a compiled `Pattern`. Assume that we assigned the pattern to the variable *pattern*, via a statement like '`pat = re.compile...`'.

class `re.Pattern`

`pattern.match(`*string*`)` → Match
: Match a 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.

`pattern.search(`*string*`)` → Match
: Search a candidate string for the compiled regular expression, *pat.*. Search 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()` find the pattern in the candidate string, a `Match` object is created to describe the match. The following methods are part of a `Match` object; we'll use the variable name *match*.

`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.

If you ask for more than one group, a tuple is returned with the matching sting from all of the requested groups.

```
>>> import re
>>> hhmm_pat= re.compile( r"(\d\d):(\d\d)" )
>>> match = hhmm_pat.match( "23:59" )
>>> match.group(1,2)
('23', '59')
```

Tip: Debugging Regular Expressions If you forget to import the module, then you get `NameError` on every class, function or variable reference.

If you spell the name wrong on your **import** statement, or the module isn't on your Python Path, you'll get an `ImportError`. First, be sure you've spelled the module name correctly. If you `import sys` and then look at `sys.path`, you can see all the places Python look for the module. You can look in each of those directories to see that the files are named.

There are two large problems that can cause problems with regular expressions: getting the regular expression wrong and getting the processing wrong.

The regular expression language, with it's special characters, escapes, and heavy use of \ is rather difficult to learn. If you get `error` exceptions from `re.compile()`, then your RE pattern is improper. For example `error: multiple repeat` means that your RE is misusing `"*"` characters. There are a number of these errors which indicate that you are likely missing a \ to escape the special meaning of one or more characters in your pattern.

If you get `TypeError` errors from `match()` or `search()`, then you have not used a candidate string with your pattern. Once you've compiled a pattern with `pat= re.compile("some pattern")`, you use that pattern object with candidate strings: `matching= pat.match("candidate")`. If you try '`pat.match(23)`', 23 isn't a string and you get a `TypeError`.

Beyond these very visible problems are the more subtle problem with a pattern that doesn't match what you think it should match. We'll look at this separately, in *More Debugging Hints*.

14.4.5 More Debugging Hints

In *Debugging Regular Expressions* we talked a bit about debugging. Beyond these very visible problems are the more subtle problem with a pattern that doesn't match what you think it should match. It helps to have example strings that are supposed to match, and example strings that are not supposed to match. You can then construct simple test scripts like the following.

```
import re
pat= re.compile( r"\d+" )
assert pat.match( "2" )
assert pat.match( "1234565.789" )
assert not pat.match( "a" )
```

If your parsing isn't working, then a test script like the following helps to debug the patterns so you can see what is matching and being parsed and what is being ignored.

```
import re
pat= re.compile( r"(\d+):(\d+)" )
m= pat.match( "23:59" )
```

```
assert m.groups() == ('23','59')
m= pat.match( "1234565:78.9" )
assert m.groups() == ('1234565','78.9')
assert not pat.match( "a" )
```

In this last example, you'll note that our pattern matched digits, but our test data included a `.`. Either our test is wrong, or our pattern is wrong. This is the art of debugging: what was *really* supposed to happen? Did it happen?

In this case, we'll have to rewrite the pattern to get the test to pass.

Unit Test Framework. This way of testing our patterns is so important, we sometimes create separate modules just for proving that our patterns work. The example shown above with **assert** statements is just the tip of the iceberg.

The Python `unittest` module provides a way to create special test modules that exist simply to prove that our software really works intended.

This is beyond the scope of this book, so we'll stick with simple scripts that use the **assert** statement.

14.4.6 Geeky Text Processing Example: Web Server Logs

In *Putting Generators To Use* we looked at a fairly complex set of string manipulations, done the hard way. These can be redone as regular expressions, leading to a dramatic improvement of this example. In the example, we looked for log entries based on the first four characters being the year, `"2003"` in that example. We can now improve that example to use a regular expression to examine each line.

Here's a snippet of a log file that we want to analyze. Note that it has some line with dates, and other lines with junk that we want to skip.

```
log= """
2003-07-28 12:46:42,843 INFO  [main] [] -
-------------------------------------------------------------------
XYZ Management Console initialized at: Mon Jul 28 12:46:42 EDT 2003
Package Build: 452
-------------------------------------------------------------------

2003-07-28 12:46:50,109 INFO  [main] [] - Export directory does not exist
2003-07-28 12:46:50,109 INFO  [main] [] - Export directory created successfully
2003-07-28 12:46:50,125 INFO  [main] [] - Starting Coyote HTTP/1.1 on port 9842
2003-07-28 12:57:14,046 INFO  [Thread-11] [] - request.getRequestURI =...
2003-07-28 12:57:18,875 INFO  [Thread-11] [admin] - Logged in
2003-07-28 12:57:19,625 INFO  [Thread-11] [] - request.getRequestURI =...
"""
```

This sequence decodes a complex input value into individual fields and then computes a single result.

```
>>> import re
>>> datePat= re.compile("(\d\d\d\d)-(\d\d)-(\d\d)")
>>> logLine = "2003-07-28 12:46:50,109 INFO  [main] [] - Export directory does not exist"
>>> dateMatch= datePat.match( logLine )
>>> dateMatch.group( 0, 1, 2, 3 )
('2003-07-28', '2003', '07', '28')
>>> y,m,d= map( int, dateMatch.group(1,2,3) )
>>> import datetime
>>> lineDate= datetime.date( y, m, d )
>>> lineDate
datetime.date(2003, 7, 28)
```

1. The first **import** statement incorporates the `re` module.
2. The *datePat* variable is the compiled `Pattern` object which matches three numbers, using `(dddd)` or `(dd)`, separated by `-`s. This matches the log date stamp very precisely: a four-digit number, followed by two two-digit numbers.

   The digit-sequence RE's are surround by `()`s so that the material that matches is returned as a group. A `Match` object will have four groups: group 0 is everything that matched, groups 1, 2, and 3 are successive digit strings.
3. The *logLine* variable is sample input, read from our log file. Typically, this will be one line of input read inside a **for** loop.
4. The *dateMatch* variable is a `Match` object that indicates success or failure in matching. If *dateMatch* is `None`, no match occurred. Otherwise, the `dateMatch.group()` method will reveal the individually matched input items.
5. `dateMatch.group()` shows the various groups that are available in the `Match` object. Group 0 is the entire match. Groups 1, 2 and 3 are the various elements of the date.
6. Setting *y*, *m*, and *d* involves a number of steps. First we use `dateMatch.group()` to create a tuple of requested items. Each item in the tuple will be a string. Second, the `map()` function is used to apply the built-in `int()` 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.
7. Finally, *lineDate* is computed as the a `datetime.date` object with the given year, month and day values.

14.4.7 Text Processing Exercises

1. **Extend the Log Processing**.

   Extend the example pattern for analyzing log records. In he example above, it matches just the date; expand it to match date and time. Change the result to be a `datetime.datetime` object.

   You can revisit the example in *Putting Generators To Use* and do more sophisticated date and time processing on the log entries. This is because you can now compare the log entry to a start or stop time. You can also compute the time between log entries.
2. **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`).
3. **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 each of your patterns, looking for the first one that works. It will return the date as a tuple of (year, month, day).

   In some earlier exercises (*Class Definition Exercises*) we glossed over the date processing to evaluate our stock portfolio. You can use this do add the neccessary date parsing.

14.4.8 Patterns and Regular Expression FAQ's

Regular Expressions are hard! Is there any easier way to do text processing? Not really. The alternative is a lot of character and slice operations. While superficially easier to understand, there is a lot more programming involved. And it is hard to build something as flexible as a regular expression. Once you get the hang of writing RE's, you can adapt to small changes in the input data with a small change to the RE.

Why are patterns called "regular expressions"? You can read about this in Wikipedia. The mathematics behind regular expressions are based on Kleene's theories of regular sets. The name regular expressions comes from the expressions that describe the regular sets. The sets contain all of the strings matched by the expressions.

While the name isn't descriptive, we're stuck with it. Worse, the RE package has features that are not part of Kleene's original mathematics, making it do more than the formal definition of regular expressions.

I can do all of this with the string methods of search, index, rindex, and slices. Do I really need RE's? Your perception is correct, that the RE module doesn't do anything new. While RE's can be hard to learn, the time invested pays handsome dividends. Once you get the hang of writing RE's, your programs are simpler than the equivalent program done with string methods.

In some cases, however, the string methods (`split()`, specifically) are simpler than regular expressions. The decision is based on what gives you a simpler, more reliable, more readable program.

CHAPTER

FIFTEEN

FIT AND FINISH: COMPLETE PROGRAMS

We've covered almost all of the statements of the Python language, and all of the useful built-in types. We've seen how to organize our programs using functions, classes and modules. We've seen how to define our own types. At this point, we have almost everything we need to write useful programs.

This part covers the remaining topics in producing a polished result. We'll look at the basic principles of an application script in *Wrapping and Packaging Our Solution*. We'll look at four different common patterns for finished programs in *Architectural Patterns – A Family Tree*. Finally, in *Professionalism : Additional Tips and Hints*, we'll look at some additional techniques that make your project into a professional, polished product.

15.1 Wrapping and Packaging Our Solution

There is considerable overlap between a library module and a main program script. The significant difference should be embodied in a small piece of programming we'll examine in *Script or Library? The Main Program Switch*. Once we've looked at that, we can talk about the remaining features of a complete command-line program in *The Standard Command-Line Interface*.

The material on the **exec** statement in *BTW – The exec Statement* is here for completeness. Logically, there's a nice symmetry between **import** and **exec**. As a practical matter, however, we rarely need to use **exec**.

15.1.1 Script or Library? The Main Program Switch

Back in *Thinking In Modules, and the Declaration of Dependence*, we identified two general species of Python modules: "main program" scripts and library modules. Some Python files do the main work of a program, while other files provide the definitions of classes and functions.

The library vs. script distinction is part of our intent when designing a module; there's no formal way to state this in Python. Library modules can do some processing while being imported; a main module can provide some definitions as well as the main script. While this distinction is informal, the overall intent should be clear: it either either provides definitions or knits definitions together to do useful work.

The biggest and most obvious distinction is that the main program is the file run by the user. This can be an icon the user double-clicked, or a command the user typed at a command prompt. In either case, a single Python file initiates the processing. This is what makes a given Python file the "main" application.

If you look at Python application programs, you'll see that the name of the application almost always matches one of the file names. For example, the **IDLE** application is launched by a file named `idle.py`. This file contains the main part of the application. IDLE has numerous other files, which contain class and function definitions.

Program Varieties. There are several subspecies of programs. We touched on this concept in *Files are the Plumbing of a Software Architecture*.

In this book, we've focused exclusively on command-line interface (CLI) programs because they are simpler to create. A richly interactive Graphic User Interface (GUI) program is generally more complex to build. Further, the core functionality for a GUI is often easiest to develop and debug as a CLI program. Once you have the CLI program working, you can wrap it up with a GUI.

To some programmers it seems more logical to design the user experience of a GUI first, and get the windows, menus, and buttons to work first. "After all,", they argue, "the user's interaction is the most important part of the software." As a practical matter, however, this doesn't work out well. It turns out to be far better to get the essential data and processing defined and working first. Once this works reliably and correctly, it's easy to add a GUI to an already working program.

What this usually means is that we have the following structure.

- One more more modules that defines the essential work of the program. This is a "model" of the real world defined with Python objects.
- We often write a command-line application script that imports the model.
- We can also write a GUI application script that imports the model. This includes the graphical "view" and the "control" logic.

This clean separation between the modules that do the work and the modules that provide the user experience makes our life simpler in the long run because each side of the application can be focused on a particular part of the task.

We'll return these "varieties" of main programs in *Architectural Patterns – A Family Tree*.

Evolution. Programs are built up from modules. In some cases, a program evolves as a series of modules. First, we start with something really basic. Then we write a module that imports our first module, and implements better input and output. Then we figure out how the `optparse` module works and we write a module which imports the second and adds a better CLI. Then we write a GUI in GTK, which imports all of our previous modules. At each step, we are building additional features around the original small core of data or processing.

Sometimes, we create a program using someone else's complete program. We might expand on someone else's program or we might be knitting two programs together to make something new.

In all of these cases, we will have modules which can be used as main programs, but are also absorbed into a larger and more complex program. Python gives us a very elegant mechanism for turning a main program into a module that can be imported into a larger program.

The `__name__` variable. The global `__name__` variable is the name of the currently executing module. It helps us determine if a module is the main module – the module being run by Python – or a library module being imported.

When the `__name__` variable is equal to `'__main__'` this is the initial (or top-level or outermost) file is being processed. When a module is being imported, the `__name__` variable is the name of the module being imported.

If a module is the main program, it must do the useful work. If it is a being imported, on the other hand, it is merely providing definitions to some other main program, and should do no work except provide class and function definitions.

You can type the following at the command line prompt in **IDLE**. If you want to experiment, create a file with just one line: `print(__name__)` and import this to see what it does.

```
>>> __name__
'__main__'
```

This `__name__` variable allows a module to be used as both a main program and as a library for another program. This can be called the "main-import switch", as it helps a module determine if it is the main program or it is an import into another main program. It gives us the ultimate flexibility to expand, refine and reuse our modules for a variety of purposes.

A main program script generally looks like the following.

```
#!/usr/bin/env python
"""Module docstring"""

import someModule

def main():
    *the real work*

if __name__ == "__main__":
    main()
```

Tip: Debugging the Main Program Switch

There are two sides to the main program switch. When a module is executed from the command line, you want it to do useful things. When a module is imported by another module, you want it to provide definitions, but not actually do anything.

Command-Line Behavior. If you get a `NameError`, you misspelled ___*name*___. If, on the other hand, nothing seems to happen, then you may have misspelled `"__main__"`.

Another common problem is providing all of the class and function definitions, but omitting the main script entirely. The **class** and **def** statements all execute silently. If there's no main script to create the objects and call the functions, then nothing will happen.

Import Behavior. If things happen when you import a module, it's missing the main program switch. When a module is evolving from main program to library that is used by a new main program, we sometimes leave the old main program in place.

The best way to handle the change from main program to library is to put the old main program into a function with a name like `main()`, and then put it the simple main program switch that calls this `main()` function when the module name is `"__main__"`.

15.1.2 The Standard Command-Line Interface

The glitzy desktop applications from big-name companies like Apple and Microsoft are the most visible parts of our computer system. Many programs, however, have minimal user interaction. They are run from a command-line prompt, perform their function, and exit gracefully.

Almost all of the core GNU/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 some explicitly interactive programs like editors (**ex**, **vi**, **emacs**, etc.), the core elements of GNU/Linux are command-line programs that lack a glitzy GUI.

In a way, we do interact with programs like **ls** (Windows **dir**). When we run the commands from the command prompt, we provide options and operands (or "arguments"). The options begin with `-` (Windows uses `/`).

The operands are not decorated with punctuation; usually they are file names, but could be permissions or user names.

For example, we might do an **ls -s /usr**, which provides an option of `-s` and an argument of `/usr`. (For Windows, an example is **dir /o:s "C:\Documents and Settings"**, which has an option of `/o:s` and an argument of `"C:\Documents and Settings"`.)

When the program runs, we see two kinds of output, usually intermixed into one stream. We see the output plus any error messages. We can use some redirection operators like `>` to capture the output and send it to a file. We can use `2>` to capture the errors and send them to a file.

This redirection is beyond the scope of this book, but is covered in all of the books on GNU/Linux programming.

Command-Line Interface (CLI) programs. There are two critical features that make a CLI program well-behaved. First, the program should accept parameters (options and arguments) in a standard manner. Second, the program should generally limit output to the standard output and standard error files created by the operating system. When any other files are written it must be by user request and possibly require interactive confirmation. It's bad behavior to silently overwrite a file.

The standard handling of command-line parameters is given as 13 rules for UNIX commands, as shown in the intro section of UNIX man pages. These rules describe the program name (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 is typically the program name plus an extension of `.py`. Example: `python`, `idle.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.
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. Example: **ls -a**, **rm -i *.pyc**.
4. Single-character options are preceded by `-`. Multiple-character options are preceded 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. Example **ls -ldai** 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. Example: **cut -ds** is an argument value of `s` for the `-d` option.
7. The argument value to an option cannot be optional. If an option requires an argument value, presence of the option means that an argument value will follow. The option is already optional; having an optional argument doesn't make much sense.
8. Groups of option-arguments following an option must be a single word; either separated by commas or quoted. A space would mean another option or the beginning of the operands. Example: `-d "9,10,56"`: three numbers separated by commas form the argument value for the `-d` option.
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. The operands are often file names, and the order in which the files are processed may be significant. Example: **ls -l -a** is the same as **ls -a -l** and **ls -la**.
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. Example, **cat file1 - file2** will process `file1`, standard input and `file2` in that order.

Parsing Command-Line Options. These rules are handled by the `getopt` module, the `optparse` module and the *sys.argv* variable in the `sys` module.

Important: But Wait! This is fine GNU/Linux, but what about Windows?

Windows programmers have several choices. The most common solution is to use the UNIX rules. They are compatible with Windows, simple and – most important – standardized by POSIX. This means that your program will use the `-` character for options, where the Microsoft-supplied programs will use `/`. How often do you use the Microsoft-supplied programs?

Another choice is to extend the `getopt` or `optparse` modules to handle Windows punctuation rules. This would allow you to seamlessly fit with the Microsoft command-line programs.

And, of course, you can always write your own option parser that looks for arguments which begin with `/`.

The command line arguments used to start Python are put into the *sys.argv* variable of the `sys` module as a sequence of strings.

For example, when we run something like

```
python casinosim.py -g craps
```

The operating system (Linux or Windows) sees the **python** command and runs the Python interpreter, passing the remaining arguments to the Python interpreter as a list of strings: `["casinosim.py", "-g", "craps"]`.

The first operand to the Python interpreter is always the top-level script to run. Python sets `__name__` to `"__main__"` and executes the file, `casinosim.py`. The other argument values are placed into *sys.argv*.

Overview of `optparse`. First, of course, we have to think about our main program and how we want to use it. Once we've figured out the arguments and options, we can then use `optparse` to transform the arguments in *sys.argv* into options and arguments our program can use.

The `optparse` module parses the command-line options in a three-step process.

1. Create an empty parser.
2. Define the options that this parser will handle.
3. Parse the arguments. This gives you a tuple with two objects. One object has the options as attributes. The other object is a list of the arguments that followed the options.

Once we have the options and arguments, we can then do the real work of our program.

Parameter Parsing. Let's say we polished up some of our exercises to create a complete program with the following synopsis. `-v-h-d mm/dd/yy-s symbolfile`

```
portfolio.py
```

This program has the following options

`-v`
: Verbosity. This can be repeated to increase the detail of the logging.

`-h`
: Help. Provides a summary of `portfolio.py`.

`-d mm/dd/yy`
: A particular sale date at which to evaluate the portfolio.

`-s symbol`
: A particular symbol to select from the portfolio.

`file`
: The name of a file with the portfolio data in CSV format.

These options can be processed as follows:

```
import optparse

parser= optparse.parser()
# -h automtically added by default
parser.add_option( "-v", action="count", dest="verbosity" )
parser.add_option( "-d", action="store", dest="date" )
parser.add_option( "-s", action="store", dest="symbol" )
options, filenames = parser.parse()

# options.verbosity is the count of -v options
# options.date is a string that must be further parsed
# options.symbol is a symbol string
# filenames is a list of files to process
```

Often, this option processing is packaged into a function called `main()`.

Formal Definitions. Here are some formal definitions for parts of `optparse`.

`optparse.parser`(*...*) → Parser
: Create a parser with the default option of `-h` and `--help` that provides help on the command.

  You can also override the program name, version number, usage text and description that `optparse` will deduce from the context in which it's run.

  You can provide the argument value of `add_help_option=False` to suppress creating the `-h` and `--help` options.

  If you provide `version="someString"`, this will automatically add a `--version` option that displays the version number.

class `optparse.Parser`

`parser.add_option`(*option_string*, *action*, *...*)
: Add an option to the parser. You can provide any combination of short or long option strings of the form `"-o"` or `"--option"`. You must provide at least one, you can provide both.

  The keyword parameter, *action* is essential for determining what is to be done with that option.

  Most parameters have a *dest* which is the destination attribute of the *options* object that gets created.

  You'll define the option with a collection of keyword arguments. There are a number of common cases.

  - Positive Flags. `add_option( "-f", "--flag", action="store_true", dest="flag", default=False )` In this case *option.flag* will be created and set to `True`.
  - Negative Flags. `add_option( "-f", "--flag", action="store_false", dest="flag", default=True )` In this case *option.flag* will be created and set to `False`.

- Options with String Values. `add_option( "-s", "--string", action="store", dest="option", type="string")` In this case *option.string* will be created and set to the value of the `-s` option.
- Options with Numeric Values. `add_option( "-i", "--int", action="store", dest="option", type="int")` In this case *option.int* will be created and set to the value of the `-i` option.
- Options that are Objects. `add_option( "-c", "--command", action="store_const", const=SomeObject, dest="command")` In this case *option.command* will be created and set to the value `SomeObject`.
- Options that are Counted. `add_option( "-v", "--verbose", action="count", dest="verbosity")` In this case *option.verbosity* will be created and set to the number of `-v` options present.

`parser.parse()` → options, arguments

Parse the *sys.argv* options and arguments, creating an *options* object with all of the options and an *arguments* list with all of the argument strings.

15.1.3 An Example Program

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: `-v-s samples`

`dicesim.py`

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.

dicesim.py

```python
#!/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)
"""

from __future__ import print_function, division
import dice
import optparse

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("{0:d} samples, average is {1:f}".format( samples, t/samples ))

```

```
23  def main():
24      parser= optparse.parser()
25      parser.add_option( "-v", "--verbose", action="count", dest="verbosity" )
26      parser.add_option( "-s", "--samples", action="store", type="int", dest="samples" )
27      parser.set_defaults( verbosity=0, samples=100 )
28      options, args = parser.parse()
29      dicesim( options.samples, options.verbosity )
30
31  if __name__ == "__main__":
32      main()
```

2. **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.

10. **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.

14. **Actual Processing**. 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.

23. **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.

31. **Main 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, then `__name__` is `"__main__"`; the arguments are parsed by the function `main()`, which calls `dicesym()` to do the real work.

This is a typical layout for a complete Python main program. We strive for two objectives. First, keep the `main()` program focused; second, provide as many opportunities for reuse as possible.

15.1.4 Main Program Exercises

1. **Create Programs**.

   Refer back to exercises in *Arithmetic and Expressions*. See sections *Expression Exercises*, *Condition Exercises*, *For Statement Exercises*, *While Statement Exercises*, *Function Exercises*. Modify these scripts to be stand-alone programs. In particular, they should get their input via `optparse()` from the command line instead of `raw_input()` or other mechanism.

2. **Larger Programs**.

   Refer back to exercises in *Basic Sequential Collections of Data*. See sections *String Exercises*, *Tuple Exercises*, *List Exercises*, *Dictionary Exercises*, *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 `optparse()`.

3. **Object-Oriented Programs**.

   Refer back to exercises in *Class Definition Exercises*. Modify these scripts to be stand-alone programs.

15.1.5 BTW – The exec Statement

The **import** statement, in effect, executes the module file. Typically, a library-oriented module is a simple sequences of definitions. The **import** statement executes all of those definitions. It also creates a `module` object. Different variations on **import** add to this by introducing different names into the global namespace.

Further, Python also optimizes the modules brought in by the **import** statement so that they are only imported once.

The **exec** statement is similar to **import**, except it does not create a `module` object. Consequently, it doesn't do any optimization to execute a module file just once.

The **exec** statement executes a suite of Python statements.

```
exec  expression
```

The *expression* can be an open file (created with the `open()` function), a string value which contains Python language statements, as well as a `code` object created by the `compile()` function.

Additionally, this form of the **exec** statement executes in a given namespace.

```
exec  expression in namespace
```

The *namespace* is a dictionary what will be used for any global variables created by the statements executed.

```
>>> code="""a= 3
... b= 5
... c= a*b
... """
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> exec code in results
>>> results['a']
3
>>> results['c']
15
```

The functions `eval()` and `execfile()` do almost similar things.

Warning: warning
These are potentially dangerous tools. These break something we call the *Fundamental Assumption*: the source you are reading is the source that is being executed. A program that uses the **exec** statement or `eval()` function is incorporating other source statements into the program dynamically. This can be hard to follow, maintain or enhance.
Generally, the **exec** statement is something that should be avoided. There are almost always more suitable solutions that involve extensible class design patterns.

15.2 Architectural Patterns – A Family Tree

This is the tail end of the discussion started in *Files are the Plumbing of a Software Architecture*. The idea is to give you a framework for classifying the problems you have, and choosing an architecture which will solve that problem effectively. Here is a family tree that shows the four most common patterns of application program architecture.

Each of these patterns has pros and cons. In many cases, you can write a core module that does the essential work, and package that core module as a complete program that fits one of these patterns.

- **Programs That Run Exclusively On Our Computer**. These programs are easier to build, since you don't have to coordinate a lot of computer and network resources. If the number of users grows, these programs can be difficult to support; at some point, one instance of web-based program may be easier to deal with than lots copies of a desktop program.

  - **Command-Line Interface (CLI) Programs**. These programs are run from the command-line prompt or put into shell scripts. We've looked at these in detail, since they form the basis for all other kinds of programs.
  - **Graphic User Interface (GUI) Programs**. These programs are generally started by double-clicking an icon. These programs are interactive, allowing a user to create and manipulate data objects. All games are GUI programs; our *office suite*, including word processors, spread sheets, graphics programs, schedule managers and contact managers are interactive GUI programs.

    Almost universally, a good GUI program is a graphical veneer wrapped around a core of essential processing. That core often has a CLI as well as a GUI. For this reason, we focus on CLI programming.
  - **Emebedded Control Programs**. This is the software the controls a device or system like a dishwasher, microwave oven, heat pump, robot or radar system. This is beyond the scope of this book. It's also not the best application for Python.
- **Programs That Share Resources Though The Internet**. We also call these client-server programs: a client application communications with a server application using the Internetworking protocols. Sometimes, additional *middleware* is used to facilitate cooperation between client and server programs.
  - **Web Applications**. Web applications are one species of client-server programs. They use the HTTP (Hypertext Transfer Protocol). In this case a browser is the client of a web server.
  - **File Transfers**. The File Transfer Protocol (FTP) can be used to copy files from machine to machine on the internet. In this case an FTP client connects with an FTP server.
  - **Email**. The SMTP, POP and IMAP protocols can be used for various parts of email processing. SMTP is the Simple Mail Transfer Protocol and handles routing of email. POP (Post-Office Protocol) and IMAP (Internet Message Access Protocol) are ways to handle individual mail messages.
  - **Database**. Many database applications are client-server architectures. A client application will access a database server. There are some standard protocols for this (ODBC and JDBC). There are many more non-standard protocols.

    Python applications try to make the non-standard database protocols at least conform to a standard interface specification called DB-API.

  There are many, many Internetworking Protocols that form the basis for client-server programming. These include DHCP (Dynamic Host Configuration Protocol), DNS (the Domain Name System), NTP (Network Time Protocol), SSH (Secure Shell), SNMP (Simple Network Management Protocol. All of these protocols define a client and server relationship.

15.2.1 Command-Line Interface Variations

There are a number of design patterns for CLI programs. This is not an exhaustive list, but some patterns you can use for thinking about your problem.

- **Filter**. A filter reads an input file, perform an extract or a calculation and produces a result file that is derived from the input.

  In traditional data processing, we often write filter programs which check a file for errors or discrepancies, sometimes called an "edit". Or, we might write a program which takes in a master file and some transactions and produces an updated master file. This, too, is a kind of filter. Finally, programs that produce easy-to-read reports or summaries are also filters.

- **Compiler**. A compiler usually performs extremely complex transformations from one or more input files to create an output file. Often, the input is a language of some sort, similar to the Python language.
- **Interpreter**. In an interpreter, statements in a language are read and processed. Some Unix utilities (like **awk**) combine filtering and interpreting. A database server (like MySQL or Oracle) is actually a kind of interpreter: it accepts statements in the SQL language, and uses those statements to create, modify or extract information from a database.

A tremendous amount of data processing can be accomplished with these basic flavors of programs. When we have complex problems, we can often use these patterns to decompose the problem into smaller problems, each of which is easier to solve in isolation. Then we can knit these smaller solutions together to tackle our real data processing problem.

We often write CLI programs that use Internet-based resources. Just as a hint, you'll want to make use of `urllib` or `urllib2`. These modules allow your CLI program to read an Internet resource as if it were a local file. With these modules, you don't need a browser or other complex graphical program to do useful work on the Internet.

15.2.2 Client-Server Architecture

Client-Server applications are more complex. They have at least two intersting parts. More sophisticated applications will have more parts.

- **Client Programs**. These programs reside on our computer, but connect our computer to other computers through the Internet. The things that run on our desktop, like a browser or a mail reader, are client programs. An on-line game like World of Warcraft™ or Second Life™ has a sophisticated client-side program.

  We often characterize these programs by the protocol they use. When we use a browser, most of the URL's that we enter begin with HTTP, which is the Hypertext Transfer Protocol. An FTP client uses the File Transfer Protocol; it 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.

- **Server Programs**. Servers are the backbone of the Internet. This includes web servers, FTP servers, mail servers, game servers, and all of the things that we want to use on the Internet. Often, we want to provide programs that will be offered as part of a web server. This is challenging programming, since a web server is shared by many concurrent users.

  In the GNU/Linux world, servers are often called *daemons*. Consequently, many programs will have names that end in "d" to indicate that they are a daemon or server. In the Windows world, there is a special kind of program called a "service" that is similar to a daemon.

  We also characterize these programs by the protocols they use. An HTTP server, for instance, responds to browser requests for web pages. An HTTP deamon, would be called `httpd`. The Apache organization, for example, offers a program commonly called Apache; the proper name is **httpd**.

  An FTP server responds to FTP client requests for file transfers. This would be called **ftpd**.

  Generally, servers are launched as a command-line program, but then runs – hopefully – forever. Sometimes a server will have an separate manager program that help you start and stop the server itself.

15.2.3 GUI and Web Variations

When we look at GUI programs and Web Browser programs, some of the distinctions between these two categories may be subtle. The programming can be very different, even though the programs will look similar when you run them.

Here are some defining features that can help you sort this out and choose a pattern that can help you solve your programming problem.

- **Thin-Client Programs**. Many programs exist on a web server, and you interact with them through a browser (Internet Explorer™, FireFox™, Safari™).

  When the bulk of the work is done by the server, we call these *thin client* programs, since the client does very little of the work. (We'll look at *fat client* programs below.)

  When you interact with any web site, no matter how sophisticated, you are only using a browser (plus some plug-ins) on your desktop. Your browser interacts with the web server via the HTTP protocol.

  The programming for the web site is almost entirely part of the web server. A generic browser is all you use on your desktop.

  There are a number of widely-used toolkits for building web sites: Django, TurboGears, CherryPy, Quixote, Pylons, web.py are some places to start learning about this high-powered approach to programming. Each of these allows you to write programs and content that fit together seamlessly to form a web site.

- **Fat-Client Programs**. Many programs exist on a web server, and you interact with them through a browser.

  When the bulk of the work is done by javascript inside your browser, we call these *fat client* programs, since the client does a great deal of the work. (We looked at *thin client* programs above.)

  The programming for the web site is split between the web server and the web pages. The bulk of the application is scripts written in Javascript and downloaded with the web page.

- **Desktop Programs**. Some programs, like FileZilla™ also run on your desktop computer, but use Internet-based files and resources. These programs look like Desktop Programs, but they need an Internet connection to work fully; we may have to provide a computer name, address or URL.

  These programs are often the most sophisticated. They combine Internetwork protocols with GUI processing.

15.2.4 Internetworking Protocols

All of our client-server programs involve a protocol to govern the relationship between the client application and the server. There are a vast number of internetworking protocols defined by documents called RFC's (Requests for Comments).

An RFC will specify (in considerable technical detail) the client and server responsibilities, the requests from client to server and the responses from the server. Many of the RFC's describe protocols that have a similar pattern.

1. The server will create a *socket* and listen for connection requests. Generally, a specific port number is used so that a host computer can have several servers working concurrently.

   The world-wide web, for example, depends on HTTP servers which usually listen for requests on port 80.

2. A client will create a socket, connect it to the server, and make a request. The Python libraries (chapter 18, Internet Protocols and Support) has components for working with many of the standard procotols.

An application could, for example, use `urllib2` to make a request of a web server.

3. The server will receive and process the request. It will respond in some way to the client.

   An HTTP server, for example, might receive a `GET` request for a particular URL. It would locate the relevant file and send the requested page back to the client.

4. The client will take action based on the server's response.

   In the case of a web browser, it receives the HTML page, makes requests for any of the additional images mentioned, and then renders the page by drawing it in the browser window.

15.3 Professionalism : Additional Tips and Hints

A common question – that's really more of a complaint – is "I get the language, I get the data structures, I get the library, I just don't know how to get started on the program I want to write."

This chapter addresses some of those "how do I get started?" and "how do I finish what I've started?" questions.

In *The Software Life-Cycle* we'll talk about the whole life-cycle of a piece of software, from concept through use to replacement. We'll look at the initial efforts in *Inception – Getting the Characters Right* and *Elaboration – Overcoming Obstacles*. Most of this book has been about programming.

Finally, we'll talk about additional ways to stay organized. *Quality Assurance – Does it Work?* includes some notes on *quality assurance*: how do you know your program works? *Configuration Management – Pieces and Parts* inclues some notes on a topic called *configuration management*: what do you have? We'll wrap up with *Transition – Installing the Final Product*.

15.3.1 The Software Life-Cycle

In *What is Programming?* we noted that the life of software has four acts. In that section, we glossed over the first two acts in order to get to the third, which was the programming part.

Let's go back and look at all four acts in a little bit more depth.

Inception. Software begins as a concept. You start with an idea for solving a problem that involves data and processing. The solution seems to involve information resources that are on a computer, or can be put on a computer.

This idea is the inception for a software development project.

One of the harder parts of getting started is defining the problem. It is not easy to clearly articulate problems because we often jump right past defining the problem into solving it. The most significant distraction to careful problem definition is having focus on the technology failing to recongize who are the actors and what their goals are.

The inception of a project is both the definition of the problem and defining what constitutes a "successful solution". You have a bunch of questions to answer: "Who uses it?" "What will they do?" "How will it work?" "When and where will they use it?" and most important "Why will they use my software?"

We'll look at this in some more depth in *Inception – Getting the Characters Right*.

Elaboration. As we proceed from concept to implementation, we have to elaborate a number of details. We look at details of the problem, our proposed solution, the selected technology and of our software design.s

One very complex activity is applying technology to solve the problem. When you look at *Architectural Patterns – A Family Tree* and all the choices available, it's very hard to pick one that fits the problem, locate appropriate libraries and frameworks, and determine what you need to do to apply all that technology to your problem. We can't really look at this in too much depth becuase there are simply so many choices.

We can, however, talk about the initial steps of transforming an idea into software. Some folks call this "turning the corner" from analyzing the problem to creating a solution. Things work out well when this is a gradual shift in focus and not an abrupt change in technology and terminology.

We'll look at this in *Elaboration – Overcoming Obstacles*.

Construction. The bulk of this book was about construction of software. There are some more things we can – and will – say about construction. There are two specific areas that are important aspects of the work, but aren't specifically related to Python or writing programs in the Python language.

- *Quality Assurance – Does it Work?*
- *Configuration Management – Pieces and Parts*

Transition. The final step in software development is the transition from the developer's hands to the user's hands. The reason we raised the curtain on this four-act play was to create software that someone can use to be happier and more productive.

Software that you've built for your own use can still benefit from a formal move into a "finished" area. It's good to close the curtain on development and call a project complete.

We'll look at some ways of doing this in *Transition – Installing the Final Product*.

15.3.2 Inception – Getting the Characters Right

In *Why Read This Book?* and *What is a Program?* we emphasized that programs were about data and processing. Also, in *Getting Our Bearings* we stated that this book covered processing first and data second.

That overview really only tells part of the story.

Our overview ignores what is – perhaps – the most important part of the concept underlying any piece of software: Who are the actors that use the application?

Actors and Goals. Really, the core part of inception is asking yourself the following questions.

- Who are the actors? Who interacts with my application? How can we classify the actors?

  It's important that we classify all the individuals into roles based on what they're permitted or required to do. Some pieces of software have a single class of actor (often called "the user"). Other pieces of software may have several roles that different people fulfill. Sometimes a single person will act in different roles.

- For each class of actor, what is their goal? How does my software help them meet that goal and be happier or more productive?

  Actors have goals. They don't do some task randomly; they have a purpose. It's very important to recognize that purpose and assure that the software we're writing fits that purpose.

Use Cases. When an actor interacts with your software, the sequence of interactions forms a *use case*. Most use cases are pretty simple, and have a goal and a sequence of interactions that the actor engages in to reach their goal.

Some use cases are more complicated and may have a number of variant scenarios to reach the same goal.

It's important to note that every program, not matter how trivially small, has at least one use case. Someone interacts with your program to achieve a goal.

Even if your program is a simple script, run from the command-line, there's still a use case. The interaction is generally really simple: the actor runs the program, the system responds with the results. Even though it's simple, it's still an interaction, and it must be considered as part of your program's use cases.

During the opening acts of software development, we merely want to identify the use cases in some general way. We want to give them titles, perhaps define which actors will engage in them. We might want to state the goals for each use case.

As we move into Elaboration, we'll provide more extensive information.

15.3.3 Elaboration – Overcoming Obstacles

We elaborate several things as our software moves from concept to implementation.

The first things we usually do is expand on our use cases. The most important part of this is to clearly understand the actor's goals and the problem they have to overcome to achieve their goals. The use cases are the interactions that the actors need to have in order to meet their goals.

Writing use cases well is beyond the scope of this book. We can only suggest that well-written use cases clearly define the actors, their goals, and how they interact with a system to achieve those goals. Once you have the use cases in hand, it's much easier to design and write your Python programs.

Once we've organized the use cases around the actors goals and their interactions with our software, we can move on to picking some technologies and designing our software.

Core Skills. We've mentioned the *abstraction* principle many times. The elaboration task is really about providing the proper level of abstraction. A program is a complex thing. The user cases, the computer, the chosen technologies, the modules and classes, even the context in which the program is used are all important at one time or another.

In order to manage all these details, we need to make careful use of abstraction.

Abstraction is a two-way street. Sometimes we add details to a general concept, sometimes we summarize a lot of details into a general concept.

The most important part of the abstraction skill is determining how best to summarize a lot of details so the *relevant* information is apparent, and the *irrelevant* information is concealed. Doing this well means that we often have to rearrange our summaries, concepts and abstractions so that they properly reflect the relevant features.

Technology Choice. We often have too many technology choices. When we look at the brief overview in *Architectural Patterns – A Family Tree* we can see a dismaying variety of ways to attack any given problem. All of them are good, and all of them can be made to work.

Without extensive experience it's hard to know which involve a lot of complex programming, and which are pretty easy. As tools and platforms change, the complexities move around, making it difficult to make any blanket statements about technology choices.

Often, we solve problems using the technologies we're most familiar with. We call this the "Hammer Syndrome". If the only tool you have is a hammer, every problem will look like a nail.

The only way to overvome the hammer syndrome is to learn how to use a wide variety of programming tools. You need to write programs with a variety of architectures: Command Line, GUI and Web in particular. Once you've written programs in these three important architectures, you are better able to choose an architecture that solves your problem effectively.

Design Details. Once we've got use cases and a technology, we can start to design a solution. As noted in *Script or Library? The Main Program Switch*, we're going to have modules that define the essential "model" of the real-world objects, and we're going to have a main program module.

Often, these two modules start out as a single module with everything. Later – as our program matures – we may *refactor* to split it into pieces.

Often, we start with class definitions. To write our class definitions, it sometimes helps to rough out a design as a simple file of definitions with document strings. It helps to put a crisp responsibility statement and a list of collaborators in the docstring to help focus on what a class does.

```
class Bin( object ):
    """A bin on a roulette wheel.

    Responsibilities:

    - Number of bin
    - Color
    - Even/Odd
    - High/Low
    - Red/Black
    - Column number
    - etc.

    Collaborates with:

    - Wheel
    """
    pass

class Wheel( list ):
    """A collection of 38 Bins.

    Responsibilities:

    - Holds all the bins
    - Picks a bin at random

    Collaborators:

    - Bin
    - Some simulation
    """
    pass

class Simulation( object ):
    """TODO: don't know quite what this does...

    Reponsibilities:

    Collaborators:

    - Wheel
    """
    pass
```

We can use a file like this for “thinking out loud” about our design. What classes do we think we’ll need? What will they do? What real-world thing do they parallel?

15.3.4 Quality Assurance – Does it Work?

We could talk a lot about how Python and the practices of quality assurance fit together.

Instead, we’ll leave you with a few hints and suggestions.

Simpler Is Better. First, we have to paraphrase Albert Einstein, and suggest that software must be "made as simple as possible, but no simpler."

We have to add E. W. Dijkstra's observation that "simplicity and elegance are unpopular because they require hard work and discipline to achieve".

To this, I'll add "If it's really hard, you're doing it wrong." Most Python libraries are simple and solve some prople really well. If you pick the wrong library, you may have to really struggle to get it to do what you want.

If you find that you're really struggling, it may mean that you've picked the wrong tool for the job. You might want to stop, take a step back, and look around for alternative tools, or an alternative approach.

Building the Right Thing. Quality Assurance starts during inception when we ask ourselves if we're really solving the right problem for the right people. This fundamental question – "What problem does this solve?" – must be carried through every step of building software.

It helps to expand this question slightly:

- What is the problem?
- Who has this problem?
- Why do they have this problem?
- When and Where do they have this problem?

Often, this fundamental question is ignored. New technology is an attractive nuisance, and too many programmers are seduced by technology that isn't really helping them solve any problems.

This aspect of quality assurance requires some reflection and consideration. Sometimes it requires wisdom or insight. Other times it requires someone to ask the "dumb question" of "why are we building this?" or "why are we building it this way"?

Build the Thing Right. Quality Assurance contains a more technical consideration, also. This comes into play during Construction and Transition. This fundamental question – "Does this actually solve the problem?" – must also be asked.

We often expand on this slightly:

- Are we using using Python (and the various libraries) the right way?
- Does our program actually *work*?

  This question should always be followed by "What evidence do you have?" Which leads us to the final question.
- Does it produce correct answers for test cases?

One technique for determining if we're building something the right way is to write tests for all of the packages, modules, classes and functions we created. Besides providing evidence of correct behavior, writing tests is a very popular way to gain experience with the Python language, the libraries, and the modules and classes we're trying to design.

This sense of quality assurance requires technical tools. We'll look at two Python modules that help with using tests as part of quality assurance.

- `doctest`. This module examines the docstrings for classes and functions to locate test scenarios.
- `unittest`. This module executes unit test scripts. A unit test script is a special-purpose module designed to test other modules.

A Fixture That Needs Testing. Let's look at a really simple module that defines a single function. We'll show examples of how to write tests for this module.

Here's our initial version of this module, without giving miuch thought to how we would test this module for correct behavior.

wheel.py

```
#!/usr/bin/env python
"""A really simple module."""

def even( spin ):
    """even( spin ) -> true if the spin is even and not zero.
    """
    return spin % 2 == 0
```

Note carefully that our docstring for the `even()` function doesn't match the actual definition. We have a bug, and we'll show how testing reveals this bug.

Doctest Example. To use `doctest`, we need to write docstrings that includes actual examples of using the function. We make those examples look like they were copied and pasted from Python in interactive mode.

Let's put this function in a module called `wheel.py`.

wheel.py

```
#!/usr/bin/env python
"""A really simple module.

>>> import wheel
>>> wheel.even( 2 )
True
"""

def even( spin ):
    """even( spin ) -> true if the spin is even and not zero."

    >>> even( 1 )
    False
    >>> even( 2 )
    True
    >>> even( 0 )
    False
    """
    return spin % 2 == 0
```

1. In the module docstring, we show how the module as a whole should be used. We created an interactive Python log showing how we *expect* this module to behave in general.
2. In the `even()` docstring, we also show how the function should be used. In this case, we wrote the log we *expected* to see. This docstring shows what would happen if the function was written correctly.

Here's a separate script that will examine the docstrings, locate the test sequences, execute them, and compare actual results against the expected results in the docstrings.

```
import wheel
import doctest
doctest.testmod(wheel)
```

1. We import the module we're going to test.

2. We import doctest.
3. We evaulate the doctest.testmod() function against the given module.

Here's the output from running this.

```
MacBook-5:notes slott$ python test1_doctest.py
**********************************************************************
File "/Users/slott/Documents/demo/roulette/wheel.py", line 16, in wheel.even
Failed example:
    even( 0 )
Expected:
    False
Got:
    True
**********************************************************************
1 items had failures:
   1 of   3 in wheel.even
***Test Failed*** 1 failures.
```

Interestingly, our even() function has a bug. In Roulette, the numbers 0 and 00 are neither even nor odd. Our even() function doesn't handle 0 at all.

Python 3

Starting with Python 2.6, we don't need the little 3-line test driver module to run doctest. We will be able to run the entire test suite from the command line.

Unittest Example. To use unittest, we need to write a module that includes some tests cases and a main script. The test cases are formalized as subclasses of unittest.TestCase. Each of these class definitions embodies a series of test methods applied to a test fixture.

```
1  #!/usr/bin/env python
2  import unittest
3
4  import wheel
5
6  class TestEven( unittest.TestCase ):
7      def test_0( self ):
8          self.assertFalse( wheel.even(0) )
9      def test_1( self ):
10         self.assertFalse( wheel.even(1) )
11     def test_2( self ):
12         self.assertTrue( wheel.even(2) )
13
14 if __name__ == "__main__":
15     unittest.main()
```

2. We import unittest.
4. We import the module we're going to test.
6. We define a subclass of unittest.TestCase.
   - Each method function that begin with test... is a different test of our "fixture" – in this case, the function wheel.even().
   - Each method function is based on an "assert" or "fail" method function of unittest.TestCase. There are dozens of these method functions to help us specify the behavior of our fixture. In this case, we used the assertTrue() and assertFalse() functions.

15. We use the main program switch to execute `unittest.main()` only when this module is the main module.

    The `unittest.main()` function will locate all of the subclasses of `TestCase`. It locate all method functions with names that start with `test`. It will then create the object, execute the methods, and count the number of tests that pass, fail or have errors.

Here's the output.

```
MacBook-5:notes slott$ python test1_unittest.py
F..
======================================================================
FAIL: test_0 (__main__.TestEven)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "notes/test1_unittest.py", line 8, in test_0
    self.assertFalse( wheel.even( 0 ) )
AssertionError

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

The first part (`F..`) is a summary of the tests being run. It shows a test failure followed by two successes. This is followed by the details of each failure.

If everything works, you'll see a string like **...** and nothing more.

This shows the bug in our `even()`. In Roulette, the numbers 0 and 00 are neither even nor odd. Our `even()` function doesn't handle 0 correctly.

15.3.5 Configuration Management – Pieces and Parts

It's common to build applications from a number of separate cmoponents. We might, for example, have a simple application that has a library module with the essential classes and functions, plus a main application module, plus some unittest scripts.

Further, our application might depend on other libraries that we downloaded from the internet. For example, we might be using PIL or pyAudio to process images or audio samples.

How do we manage the configuration of our various components?

Accounting 101. Configuration management is really an accounting practice. The heart of configuration management is to tracking the various assets that are part of our overall application program.

Unlike real-world accounting, however, we don't have tangible physical objects that we can put an asset tag on. Instead we have files on our computer that we need to keep track of.

Further, software components have version numbers which are a very important thing. We have to be very careful to distinguish between two different versions of the same component or library that we downlaoded.

Naming. Half the battle in tracking our software assets is to have directory names that include version numbers. For example, when we install Python under Windows, we need to put it in a directory named `C:\python2.5`. That way, we can add Python 3.0 without breaking our existing installation.

Completeness. The other half of the battle in tracking our assets is having a complete list of everything we're using. It's important to keep a file with the following information.

- The web sites from which your downloaded components.

- The version number you downloaded.
- The kind of license under which you're using the component.

  Generally, the license will be one of the following: Academic Free License (AFL), Attribution Assurance License, BSD License, GNU General Public License (GPL), GNU Library or Lesser General Public License (LGPL), Mozilla Public License 1.1 (MPL 1.1), Python Software Foundation License.

15.3.6 Transition – Installing the Final Product

Software is infinitely changeable. You can always tweak and adjust your Python programs.

However, at some point, you want to declare the project as "done". Usually in the sense of "done for now." Or "done until I think of something to add."

We need to transition our software to some reasonably final form. When we're provising software to someone else, this packaging step is essential. When we're using software for our own purposes, this step is optional but important.

Organizing Your Workbench. We need to make a distinction between a "working copy" of some module and the official "library copy" of the module.

When we're learning, everything's a working copy. Further, everything starts out jumbled into a single directory.

As we move toward more professional level of craftsmanship, we start to organize our projects into separate directories. Further, we'll need to install our modules in the Python site-packages directory.

The Python Library. When we looked at the **import** statement in *Thinking In Modules, and the Declaration of Dependence*, we touched on the search path. The search path is the list of places that Python searches for a module. If you import the `sys` module, you'll see that *sys.path* is the list of locations that are searched.

What we in the section on modules was the first locations searched was `''` (an empty string). This is Python's short-hand for the current working directory. Because of this, we can simply pile our modules into a single directory, and everything works nicely.

As soon as we start moving things around, we need to do one of two things:

- Add the locations of our modules to the `PYTHONPATH` environment variable.
- Put our modules into Python's `site-packages` directory.

Installed Packages. Installing modules in the official `site-packages` directory is done with a set of Python tools called `distutils`. We can only touch on the essence of distutils here.

To make our package easily installed in `site-packages`, we need to write a `setup.py` file. This file contains the information required to package and install our module.

Here's a typical distutils `setup.py` module.

```python
#!/usr/bin/env python
from distutils.core import setup

from distutils.core import setup
setup(name='My Roulette Module',
      version='1.0',
      py_modules=['roulette'],
      )
```

This will allow you to do the following to install your module in the site-packages library.

```
python setup.py install
```

Once you've done this, you can work in any directory on your computer and have access to your new module. This allows you to create new applications based on modules you've already written.

CHAPTER

SIXTEEN

APPENDICES

16.1 Debugging Tips

16.1.1 Let There Be Python: Downloading and Installing

Tip: 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.

16.1.2 Instant Gratification : The Simplest Possible Conversation

Tip: Debugging Windows Command Prompt

In the unlikely event that you can't use Python from the **Command Prompt**, you have an issue with your Windows "path". Your path tells the **Command Prompt** where to find the various commands. The word **python** becomes a command when the `python.exe` file is on the system's path.

Generally, you should reinstall Python to give the Python installer a chance to set the path correctly for you. If, for some reason, that doesn't work, here's how you can set the system path in Windows.

Setting the Windows Path

1. Open the Control Panel.

   Use the **Start** menu, **Settings** sub menu to locate your **Control Panel**.

2. Open the System Control Panel

   Double-click the **System** Control Panel. This opens the *System Properties* panel.

3. Open the Advanced Tab of the System Control Panel

   Click the **Advanced** tab on the **System** Control Panel.

   There are three areas: Performance, Environment Variables and Startup and Recovery. We'll be setting the environment variables.

4. Open the Environment Variables of the Advanced Tab of the System Control Panel

   Click the **Environment Variables...** button.

   This dialog box has a title of *Environment Variables*. It shows two areas: user variables and System variables. We'll be updating one of the system variables.

5. Edit the Path variable

   This dialog box has a title of *Environment Variables*. Scroll through the list of System variables, looking for `Path`. Click on the `Path` to highlight it.

   Click the **Edit...** button.

   This dialog box has a title of *Edit System Variable*. It has two sections to show the variable name of `Path` and the variable value.

6. Add Python's location to the Path value

   This dialog box has a title of *Edit System Variable*. It has two sections to show the variable name of `Path` and the variable value.

   Click on the value and use the right arrow key to scroll through the value you find. At the end, add the following `;C:\python26`. Don't forget the `;` to separate this search location from other search locations on the path.

   Click **OK** to save this change. It is now a permanent part of your Windows setup on this computer. You'll never have to change this again.

7. Finish Changing Your System Properties

   The current dialog box has a title of *Environment Variables*. Click **OK** to save your changes.

   The current dialog box has a title of *System Properties*. Click **OK** to save your changes.

Tip: Debugging Typing a Python Statement

When you see the `...` prompt from Python, it means that your statement is incomplete. Are you missing a `)` to make the `()` pairings complete? Did you accidentally use the `\`? Hit `Return` twice and you'll get a nice syntax error and you'll be back at the `>>>` where you can try again.

Also, you'll get unexpected errors if you try to use `[]`, and `{}` the way mathematicians do. Python only uses `()` to group expressions. If you try to use `[]`, you'll get a `TypeError: unsupported operand type(s) for [] : 'list' and 'int'`. If you try to use `{}`, you get a `SyntaxError: invalid syntax`.

16.1.3 IDLE Time : Using Tools To Be More Productive

Tip: Debugging A Script

If your `idle` script file doesn't work, there are some common things to confirm:

- Your file is in the same directory that the **Terminal** starts in. If you are unsure, you can use the **pwd** to print the working directory. In my case it is `/home/slott`. That's where I put my `idle` startup file.
- Your file is plain text. A word processor won't save files as plain text automatically, so you should use something like **gedit** to assure that you're creating a plain text file.

Tip: Debugging An Alias

If your alias doesn't work, there are some common things to confirm:

- Your `.profile` works correctly. You can type **sh -v .profile** or **bash -v .bash_profile** to test it. If you see error messages, likely you missed an apostrophe or messed up the spaces.

16.1.4 Simple Arithmetic : Numbers and Operators

Tip: Debugging Octal Numbers (Leading Zero Alert)

A number that begins with a zero is supposed to be in base 8. If you are copying numbers from another source, and that other uses leading zeros, you may be surprised by what Python does. If the number has digits of 8 or 9, it's illegal. Otherwise, the number isn't decimal.

I spent a few hours debugging a program where I had done exactly this. I was converting a very ancient piece of software, and some of the numbers had zeroed slapped on the front to make them all line up nicely. I typed them into Python without thinking that the leading zero meant it was really base 8 not base 10.

16.1.5 Better Arithmetic Through Functions

Tip: Debugging Function Expressions

If you look back at *Syntax Rule 6*, you'll note that the `()`s need to be complete. If you accidentally type something like `round(2.45` with no closing `)`, you'll see the following kind of exchange.

```
>>> round(2.45
...
...
... )
2.0
```

The `...` is Python's hint that the statement is incomplete. You'll need to finish the `()`s so that the statement is complete.

16.1.6 Extra Functions: `math` and `random`

Tip: Debugging Math Functions

The most common problem when using math functions is leaving off the `math` to qualify the various functions imported from this module. If you get a "`NameError: name 'cos' is not defined`" error message, it means you haven't included the `math` qualifier.

The next most common problem is forgetting to '`import math`' each time you run **IDLE** or **Python**. You'll get a "`NameError: name 'math' is not defined`" error if you forgot to import math.

The other common problem is failing to convert angles from degrees to radians.

16.1.7 Special Ops : Binary Data and Operators

Tip: Debugging Special Operators

The most common problems with the bit-fiddling operators is confusion about the relative priority of the operations. For conventional arithmetic operators, `**` is the highest priority, `*` and `/` are lower priority and `+` and `-` are the lowest priority. However, among `&`, `^` and `|`, `<<` and `>>` it isn't obvious what the priorities are or should be.

When in doubt, add parenthesis to force the order you want.

16.1.8 Peeking Under the Hood

Tip: Debugging the **from __future__** statement

There are two common spelling mistakes: omitting the double underscore from before and after `__future__`, and misspelling `division`.

- If you get `ImportError: No module named _future_`, you misspelled `__future__`.
- If you get `SyntaxError: future feature :replaceable:`divinizing` is not defined‘, you misspelled `division`.

16.1.9 Seeing Results : The print Statement

Tip: Debugging the **print** Statement

One obvious mistake you will make is misspelling **print**. You'll see `NameError: name 'primpt' is not defined` as the error message. I've spelled it "primpt" so often, I've been tempted to rewrite the Python language to add this as an alternative.

The other common mistake that is less obvious is omitting a comma between the values you are printing. When you do this, you'll see a `SyntaxError: invalid syntax` message.

If the result of a print statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**s Python shell to examine the processing one step at a time.

16.1.10 Expressions, Constants and Variables

Tip: Debugging the Assignment Statement

There are two common mistakes in the assignment statement. The first is to choose an illegal variable name. If you get a `SyntaxError: can't assign to literal` or `SyntaxError: invalid syntax`, the most likely cause is an illegal variable name.

The other mistake is to have an invalid expression on the right side of the =. If the result of an assignment statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**‘s *Python Shell* window to examine the processing one step at a time.

Tip: Debugging the Augmented Assignment Statement

There are two common mistakes in the augmented assignment statement. The first is to choose an illegal variable name. If you get a `SyntaxError: can't assign to literal` or `SyntaxError: invalid syntax` the most likely cause is an illegal variable name.

The other mistake is to have an invalid expression on the right side of the assignment operator. If the result of an assignment statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**'s Python shell to examine the processing one step at a time.

16.1.11 Assignment Bonus Features

Tip: Debugging Multiple Assignment Statements

There are three common mistakes in the augmented assignment statement. The first is to choose an illegal variable name. If you get a `SyntaxError: can't assign to literal` or `SyntaxError: invalid syntax` the most likely cause is an illegal variable name.

One other mistake is to have an invalid expression on the right side of the assignment operator. If the result of an assignment statement doesn't look right, remember that you can always enter the various expressions directly into **IDLE**'s Python shell to examine the processing one step at a time.

The third mistake is to have a mismatch between the number of variables on the left side of the = and the number of expressions on the right side.

16.1.12 Can We Get Your Input?

Tip: Debugging the `raw_input()` Function

There are two kinds of mistakes that occur. The first kind of mistake are basic syntax errors in the `raw_input()` function call itself.

The other mistake, which is more difficult to prevent, is to provide invalid input to the script when it runs. Currently, we don't quite have all of the language facilities necessary to recover from improper input values. When we cover the **try** statement and exception processing in *The Unexpected : The try and except statements* we'll see how we can handle invalid input.

Tip: Debugging the **del** statement

If we misspell a variable name, or attempt to delete a variable that doesn't exist, we'll get an error like `NameError: name 'hack' is not defined`.

16.1.13 Truth and Logic : Boolean Data and Operators

Tip: Debugging Logic Operators

The most common problem people have with the logic operators is to mistake the priority rules. The lowest priority operator is **or**. **and** is higher priority and **not** is the highest priority. If there is any confusion, extra parentheses will help.

16.1.14 Making Decisions : The Comparison Operators

Tip: Debugging Comparison Operators

The most common problem people have with the comparison operators is to attempt to compare things which cannot meaningfully be compared. They ask, in essence, "which is larger, the Empire State Building or the color green?" An expression like `123 < 'a'` doesn't really make a lot of sense, even though it is legal Python.

Python copes with senseless comparisons using it's coercion rules. In this case, the number `123` is coerced to a string `'123'`, and the two strings are compared using the ordinary alphabetical order rules. In this case, digits come before letters and any number will be less than any word.

Sorting out the rules for coercion and comparison can be very confusing. Consequently, it should be avoided by exercising reasonable care in writing sensible programs. You should inspect your program to be sure you are comparing things sensibly. You can also put in explicit conversions using the various factory functions in *Functions are Factories (really!)*.

16.1.15 Processing Only When Necessary : The if Statement

Tip: Debugging the **if** statement.

If you are typing an **if** statement, and you get a `SyntaxError: invalid syntax`, you omitted the `:`.

A common problem with **if** statements is an improper condition. You can put any expression in the if or elif statement. If the expression doesn't have a boolean value, Python will use the `bool()` function to determine if the expression amounts to `True` or `False`. It's far better to have a clear boolean expression rather than trust the rules used by the `bool()` function.

One of the more subtle problems with the **if** statement is being absolutely sure of the implicit condition that controls the **else** clause. By relying on an implicit condition, it is easy to overlook gaps in your logic.

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's the subtle bug in this example. 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 "field pays even money" condition. 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 found that did not match by any of the **if** or **elif** clauses.

16.1.16 While We Have More To Do : The for and while Statements

---

Tip: Debugging the **for** Statement

If you are typing a **for** statement, and you get a `SyntaxError: invalid syntax`, you omitted the `:`.

The most common problem is setting up the sequence properly. Very often, this is because of the complex rules for the `range()` function, and we have one too many or one too few values.

A less common problem is to misspell the variable in the **for** statement or the *suite*. If the variable names don't match, the **for** statement will set a variable not used properly by the suite. An error like `NameError: name 'j' is not defined` means that your suite *suite* expected *j*, but that was not the variable on your **for** statement.

Another problem that we can't really address completely is writing a **for** statement where the *suite* doesn't do the right thing in the first place. In this case, it helps to be sure that the suite works in the first place. An execution trace (see *Where Exactly Did We Expect To Be?*) can help. Also, you can enter the statements from the suite separately to the Python shell to see what they do.

Tip: Debugging the **while** Statement

If you are typing a **while** statement, and you get a `SyntaxError: invalid syntax`, you omitted the `:`.

There are several problems that can be caused by an incorrectly designed **while** statement.

The while loop never stops! The first time you see this happen, you'll probably shut off your computer. There's no need to panic however, there are some better things to do when your computer appears "hung" and doesn't do anything useful.

When your loop doesn't terminate, you can use `Ctrl-C` to break out of the loop and regain control of your computer. Once you're back at the `>>>` you can determine what was wrong with your loop. In the case of a loop that doesn't terminate, the **while** expression is always `True`. There are two culprits.

- You didn't initialize the variables properly. The **while** expression must eventually become `False` for the loop to work. If your initialization isn't correct, you may have created a situation where it will never become `False`.
- You didn't change the variables properly during the loop. If the variables in the **while** expression don't change values, then the expression will never change, and the loop will either never iterate or it will never stop iterating.

If your loop never operates at all, then the **while** expression is always `False`. This means that your initialization isn't right. A few **print** statements can show the values of your variables so you can see precisely what is going wrong.

One rare situation is a loop that isn't supposed to operate. For example, if we are computing the average of 100 dice rolls, we'll iterate 100 times. Sometimes, however, we have the "degenerate case", where we are trying to average zero dice rolls. In this case, the **while** expression may start out `False` for a good reason. We can get into trouble with this if some of the other variables are not be set properly. This can happen when you've made the mistake of creating a new variable inside the loop body. To be sure that a loop is designed correctly, all variables should be initialized correctly, and no new variables should be created within the loop body; they should only be updated.

If your loop is inconsistent – it works for some input values, but doesn't work for others – then the body of the loop is the source of the inconsistency. Every **if** statement alternative in the suite of statements within the loop has establish a consistent state at the end of the suite of statements.

Loop construction can be a difficult design problem. It's easier to design the loop properly than to debug a loop which isn't working. We'll cover this in *A Digression On Design*.

16.1.17 Becoming More Controlling

Tip: Debugging the **break** Statement

The most common problem with the **break** statement is an incorrect condition on the surrounding **if** statement, or an incorrect condition on the **while** statement. Designing these repetitive statements can be a relatively difficult problem, so we have some additional material on just that subject in *A Digression On Design*.

The most important debugging tool is the `print()` function. You can print the values of relevant variables in various positions in your loop body to be sure that things work the way you expect.

Tip: Debugging the **continue** Statement

The most common problem with the **continue** statement is an incorrect condition on the surrounding **if** statement, or an incorrect condition on the **while** statement. Designing these repetitive statements can be a relatively difficult problem, so we have some additional material on just that subject in *A Digression On Design*.

Tip: Debugging the **assert** Statement

The **assert** statement is an important tool for debugging other problems in your program. It is rare to have a problem with the **assert** statement itself. The only thing you have to provide is the condition which must be true. If you can't formulate the condition in the first place, it means you may have a larger problem in describing what is supposed to be happening in the program in general. If so, it helps to take a step back from Python and try to write an English-language description of what the program does and how it works.

Clear **assert** statements show a tidy, complete, trustworthy, reliable, clean, honest, thrifty program. Seriously. If you can make a clear statement of what must be true, then you have a very tight grip on what should be happening and how to prove that it really is happening. This is the very heart of programming: translating the program's purpose into a condition, creating the statements that make the conditions true, and being able to back this design up with a proof and a formal assertion.

16.1.18 Turning Python Loose with More Sophisticated Scripts

Tip: Debugging Explicit Python Scripts

There are two things which can go awry with a script: the operating system can't find Python or the operating system can't find your script file. Here's a debugging procedure which may help.

1. Start with the simplest possible operating system command: `python`. If this doesn't work, Python isn't installed correctly: reinstall Python. If this *does* work, then the shell or command prompt can find Python, and that is not the problem.
2. Use your operating system's directory and file commands (Windows: **CD** and **DIR**; GNU/Linux or MacOS: **pwd** and **ls**) to find your script file. If your file is not in the current directory, then you can either change directories, or include the directory name in your file.

- Change your current working directory to the correct location of your files. For Windows: use **CD**; for GNU/Linux and MacOS: use **cd**. For example, if your files are in an `exercises` directory, you can do **cd exercises**.
- Include the directory name on your file. For example, if your files are in an `exercises` directory, you can run the `script1.py` script with **python exercises/script1.py**.

3. If you can find Python, and you appear to be in the correct directory, the remaining problem is misspelling the filename for your script. This is relatively common, actually. First time GNU/Linux and MacOS users will find that the shell is sensitive to the case of the letters, that some letters look alike, it is possible to embed non-printing characters in a file name, and it is unwise to use letters which confuse the shell. We have the following advice.
   - File names in GNU/Linux should be one word, all lower case letters and digits. These are the standard Python expectations for module names. While there are ways around this by using the shells quoting and escaping rules, Python programs avoid this.
   - File names should avoid punctuation marks. There are only a few safe punctuation marks: `-`, `.` and `_`. Even these safe characters should not be the first character of the file name.
   - Some Windows programs will tack an extra `.txt` on your file. You may have to manually rename the file to get rid of this.
   - In GNU/Linux, you can sometimes embed a space or non-printing character in a file name. To find this, use the **ls -s** to see the non-printing characters. You'll have to resort to fairly complex shell tricks to rename a badly named file to something more useful. The `%` character is a wild-card which matches any single character. If you have a file named `script^M1.py`, you can rename this with **mv script%1.py script1.py**. The `%` will match he unprintable `^M` in the file name.

Tip: Debugging Implicit Python Scripts

If an implicit script doesn't work, you have two problems to resolve. First, would it work as an explicit command? If not, then fix that before doing anything else. Say you have a script `implicit.py` that won't run. The first thing to test is **python implicit.py**. If this doesn't work, see *Let Python Run It*.

The most common problem with implicit scripts is the hidden hand-shake between the shell and the first line of the script file. Most GNU/Linux variants will use `#!/usr/bin/env python`.

You may not have the **env** program in your UNIX. Try typing just **env** at the shell prompt. If this returns an error, you will need to know where Python installed. Enter **which python**. The shell will search its path for Python and report the directory in which it was found. This might be `/usr/local/bin/python`. Use this in the `#!` line. For example, `#!/usr/local/bin/python`.

16.1.19 Adding New Verbs : The def Statement

Tip: Debugging Function Definition

There are three areas for mistakes in function definition: the **def** statement itself, the **return** statement and using the function in an expression.

The syntax of the **def** statement contains three parts. If you have syntax errors on the definition, you've got one of these three wrong, or you're misspelling "def".

- The name, which is a Python name, following the name rules in *Python Name Rules*.
- The parameters, which is list of names, separated by commas. The `()`s around the parameter list is required, even if there are no parameters. The `,` to separate parameters is required.

- The indented suite, a block of Python statements.

The **return** statement is how the return value is defined. If you omit this, your function always returns `None`. The **return** statement also ends execution of the function's body; if you have this statement out of place, your function may not fully execute.

When you use the function, you have to pass actual argument values for each parameter variable. The matching is done by position: the first argument value is assigned to the first positional parameter.

Tip: Direct Python and Help()

When executing `help()` while using Python directly (*not* using **IDLE**), you'll be interacting with a help viewer that allows you to scroll forward and back through the text.

For more information on the help viewer, see *Getting Help*.

On the Mac OS or GNU/Linux, you'll see an `(END)` prompt telling you that you've reached the the document; hit `q` to exit from viewing help.

Tip: Debugging Optional Parameters

There are three areas for the common mistakes in function definition: the **def** statement itself, the **return** statement and using the function in an expression.

The syntax of the **def** statement contains three parts. If you have syntax errors on the definition, you've got one of these three wrong, or you're misspelling "def".

- The name, which is a Python name, following the name rules in *Python Name Rules*.
- The parameters, which is list of names, separated by commas. The `()`s around the parameter list is required, even if there are no parameters. The `,` to separate parameters is required. If we use initializers, the `=` to separate the variable and the initial value is required punctuation. Also note that the initializer is evaluated when the **def** statement is executed, not when the function is evaluated.
- The indented suite, a block of Python statements.

In our syntax summaries, we use `...` to show things that can be repeated, we don't actually enter the `...`.

The required parameters (which have no initial values) must precede the optional parameters (which have initial values).

The **return** statement is how the return value is defined. If you omit this, your function always returns `None`. The **return** statement also ends execution of the function's body; if you have this statement out of place, your function may not fully execute.

When you use the function, you have to provide actual argument values for each parameter that doesn't have an initializer. Since positional parameters are simply matched up in order, the arguments you present when you use of the function must match parameters of the definition.

Tip: Debugging Keyword Parameters

When you use a function, you have to provide actual argument values for each parameter that doesn't have an initializer. Two things can go wrong here: the syntax of the function call is incorrect in the first place, or you haven't provided values to all parameters.

You may have fundamental syntax errors, including mis-matched `()`, or a misspelled function name.

You can provide argument values by position or by using the parameter name or a mixture of both techniques. Python will first extract all of the keyword arguments and set the parameter values. After that, it will match up positional parameters in order. Finally, default values will be applied. There are several circumstances where things can go wrong.

- A parameter is not set by keyword, position or default value
- There are too many positional values.
- A keyword is used that is not a parameter name in the function definition.

16.1.20 Common List Design Patterns

Tip: Debugging List Comprehensions

List comprehensions have rather complex syntax. There are a number of ways to get `SyntaxError` messages. The expression, the for-clause and the overall structure of the expression, including balancing the `[]` are all sources of problems. Debugging a list comprehension is most easily done by building up the list comprehension one clause at a time.

A simple comprehension has two clauses: the expression clause and the for-clause. You can try each part out in IDLE individually.

- If the for-clause is wrong, the original sequence will not be correct.
- If the expression clause is not correct, the resulting sequence will be incorrect.

The for-clause of a list comprehension can be seen by entering just the for-clause as a separate statement. The expression clause can be evaluated for specific values to be sure that it works correctly.

For example, `[ 2*i+1 for i in range(5) ]` can be debugged in two parts. First, assure that `range(5)` produces the source sequence you expected. Second, assure that `2*i+1` works for values of *i* from 0 to 4.

Tip: Debugging List Sorting

There are three kinds of problems that can prevent a customized sort operation from working correctly.

- Our key function doesn't have the right form. It must be a function that extracts the key from an item of the sequence being sorted.

  ```
  def key( item ):
      return something based on the item
  ```

- The data in your list isn't regular enough to be sorted. For example, if we have dates that are represented as strings like `'1/10/56'`, `'11/19/85'`, `'3/8/87'`, these strings are irregular and won't sort very nicely. As humans, we know that they should be sorted into year-month-date order, but the strings that Python sees begin with `'1/'`, `'11'` and `'3/'`, with an alphabetic order that may not be what you expected.

  To get this data into a usable form, we have to *normalize* it. Normalizing is a computer science term for getting data into a regular, consistent, usable form. In our example of sorting dates, we'll need to use the `time` or `datetime` modules to parse these strings into proper Python objects that can be compared.

16.1.21 The Unexpected : The try and except statements

Tip: Debugging Exception Design

The try statement has at least two suites: the **try** suite and at least one **except** suite. Each of these can have `:`s missing, or be indented incorrectly. Since these are large, composite statements, there are a lot of places where problems can occur.

One other problem is that we may have put the wrong statements in the **try** suite. If we evaluate a statement that raises an exception, but that statement is not in a **try** suite, the exception won't get handled. If our **try** statement doesn't seem to catch the exception, one possibility is that we didn't enclose correct statement in the **try** statement.

Since Python reports the line number where the exception was raised, we can see where the exception originated and adjust the location of the **try** or **except** clauses to include the proper statements.

Another problem is that the exception is raised and the exceptions on our **except** statements don't match. We'll address this in *Debugging Exception Handling*.

Including too many statements in the **try** suite is just as bad as having too few statements. Including statements which cannot raise an exception in the first place can lead to confusion when reading the program. When we look at a program we wrote two weeks ago, we don't want to struggle to understand what it means. We'd like to be reasonably clear. To this end, a **try** suite should be as small as possible to handle the exception.

Second, we may be raising an exception for the wrong reason. Since a **raise** statement is always associated with an **if**, **elif** or **else** suites, the conditions on the **if** statement define the exceptional condition. We should be able to clearly articulate the condition that leads to the **raise** statement. Problems in the **if** statement will surface as errors in exception processing.

Tip: Debugging Exception Handling First, we may have the wrong exceptions named in the **except** clauses. If we evaluate a statement that raises an exception, but that exception is not named in an **except** clause, the exception won't get handled.

Since Python reports the name of the exception, we can use this information to add another **except** clause, or add the exception name to an existing **except** clause. We have to be sure we understand why we're getting the exception and we have to be sure that our handler is doing something useful. Exceptions like `RuntimeError`, for example, shouldn't be handled: they indicate that something is corrupt in our Python installation.

You won't know you spelled an exception name wrong until an exception is actually raised and the **except** clauses are matched against the exception. The **except** clauses are merely potential statements. Once an exception is raised, they are actually evaluated, and any misspelled exception names will cause problems.

Second, we may be raising the wrong exception. If we attempt to raise an exception, but spelled the exception's name wrong, we'll get a strange-looking `NameError`, not the exception we expected.

As with the **except** clause, the exception name in a **raise** clause is not examined until the exceptional condition occurs and the **raise** statement is executed. Since **raise** statements almost always occur inside **if**, **elif** or **else** suites, the condition has to be met before the **raise** statement is executed.

16.1.22 Looping Back : Iterators, the for statement, and the yield statement

Tip: Debugging Iterators

There are several common problems with using an explicit iterator.

- Skipping items without processing them.

- Processing the same item twice
- Getting a `StopIteration` exception raised when trying to skip the first item.

Generally, the best way to debug a generator is to use it in a very simple iteration statement that prints the result of the iteration. Printing the items will show us precisely what is happening. We can always change the **print** statement into a comment, but putting a `#` in front of `print`.

Here's a good design pattern for skipping the first item in a sequence.

```
i = iter( someSequence )
next(i)         Skips an item on purpose
while True:
    a= next(i)
    some processing
    print(a)
```

Skipping items happens when we ask for the `next()` method of the iterator one too many times.

Processing an item twice happens when we forget to ask for the `next()` method of the iterator. We see it happen when a program picks off the header items, but fails to advance to the next item before processing the body.

Another common problem is getting a `StopIteration` exception raised when trying to skip the header item from a list or the header line from a file. In this case, the file or list was empty, and there was no header. Often, our programs need the following kind of **try** block to handle an empty file gracefully.

```
i = iter( someSequence )
try:
    next(i)  Skips an item on purpse
except StopIteration:
    No Items -- this is a valid situation, not an error
```

Tip: Debugging Generator Functions

One of the most common problems people have with writing a generator is omitting the **yield** statement. Without the **yield** statement, you have written a simple function: the **for** statement can't initialize it and get individual values from it.

If you get a `TypeError: iteration over non-sequence` error, you have omitted the **yield** statement from your generator function.

Additionally, all of the iterator problems are applicable when creating a generator function. We could have problems that cause us to skip items, process items twice or get an unexpected `StopIteration` exception.

16.1.23 Collecting Items : The `set`

Tip: Debugging `set()`

A common mistake is to do something like `set( 1, 2, 3 )`, which passes three values to the `set()` function. If you get a `TypeError: set expected at most 1 arguments, got n`, you didn't provide proper tuple to the set factory function.

Another interesting problem is the difference between `set( ("word",) )` and `set( "word" )`.

- The first example provides a 1-element sequence, `("word,")`, to `set()`, which becomes a 1-element set.

- The second example passes a 4-character string, "word", which becomes a 4-element set.

In the case of creating sets from strings, there's no error message. The question is really "what did you mean?" Did you intend to put the entire string into the set? Or did you intend to break the string down to individual characters, and put each character into the set?

16.1.24 External Data and Files

Tip: Constructing File Names

When using Windows-specific punctuation for filenames, you'll have problems because Python interprets the `\` as an escape character. To create a string with a Windows filename, you'll either need to use `\` in the string, or use an `r" "` string literal. For example, you can use any of the following: `r"E:\writing\technical\pythonbook\python.html"` or `"E:\\writing\\technical\\pythonbook\\python.html"`.

Note that you can often use `"E:/writing/technical/pythonbook/python.html"`. This uses the POSIX standard punctuation for files paths, `/`, and is the most portable. Python generally translates standard file names to Windows file names for you.

Generally, you should either use standard names (using `/`) or use the `os.path` module to construct filenames. This module eliminates the need to use any specific punctuation. The `os.path.join()` function makes properly punctuated filenames from sequences of strings

Tip: Debugging Files

There are a number of things that can go wrong in attempting to create a file object.

If the file name is invalid, you will get operating system errors. Usually they will look like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'wakawaka'
```

It is very important to get the file's path completely correct. You'll notice that each time you start IDLE, it thinks the current working directory is something like `C:\Python26`. You're probably doing your work in a different default directory.

When you open a module file in IDLE, you'll notice that IDLE changes the current working directory is the directory that contains your module. If you have your `.py` files and your data files all in one directory, you'll find that things work out well.

The next most common error is to have the wrong permissions. This usually means trying to writing to a file you don't own, or attempting to create a file in a directory where you don't have write permission. If you are using a server, or a computer owned by a corporation, this may require some work with your system administrators to sort out what you want to do and how you can accomplish it without compromising security.

The `[Errno 2]` note in the error message is a reference to the internal operating system error numbers. There are over 100 of these error numbers, all collected into the module named `errno`. There are a lot of different things that can go wrong, many of which are very, very obscure situations.

16.1.25 Files II : Some Examples and Some Modules

Tip: Debugging CSV Input

One problem with file processing is that our Python data structure isn't a giant string of characters. However, the file is simply a giant string. Essentially, reading a file is a way of translating the characters into a useful Python structure.

The most common thing that can go wrong is not creating the expected structure in our Python program. In the *Reading and Sorting* example, we might not create our list of tuples correctly.

It is helpful to print the value of the *data* variable to get a good look at the data structure which is produced. Here we show the beginning of our "list of tuples". We've adjusted the Python output to make it a little more readable.

```
[('"^DJI"', '10452.15', '"9/26/2005"', '"1:50pm"', '+32.56',
  '10420.22', '10509.23', '10420.22', '137206720'),
 ('"^IXIC"', '2121.07', '"9/26/2005"', '"1:50pm"', '+4.23',
  '2127.90', '2132.60', '2119.17', '0'), ...
```

Looking at the intermediate results helps us be sure that we are reading the file properly.

16.1.26 Files III : The Grand Unification

Tip: Debugging File Formats

When we talk about how data appears in files, we are talking about "data representation." This is a difficult and sometimes subtle design decision. A common question is "How do I know what the data is?" . There are two important points of view.

- The program you are designing will save data in a file for processing later. Since you are designing the file, you get to choose the representation. You can pick something that is easy for your Python program to write. Or, you can look at other programs and pick something that is easy for the other programs to read. This can be a difficult balancing act.
- The program you are designing must read data prepared by another program. Since someone else designed the file, you will interpret the data they provide. If their format is something that Python can easily interpret, your program will be very simple. However, the more common situation is that their format is not something Python can interpret, and you must write this interpretation yourself.

16.1.27 Defining New Objects

Tip: Debugging a Class Definition

When we get syntax errors on a class definition, it can be in the class line or one of the internal method function definitions.

If we get a simple `SyntaxError` on the first line, we have misspelled **class**, left off a `(` or `)`, or omitted the `:` that begins the suite of statements that defines the class.

If we get a syntax error further in the class definition, then our method functions aren't defined correctly. Be sure to indent the **def** once (so it nests inside the class). Be sure to indent the suite of statements inside the **def** twice.

Tip: Debugging Object Construction

Assuming we've defined a class correctly, there are a three of things that can go wrong when attempting to construct an object of that class.

- The class name is spelled incorrectly.
- You've omitted the `()` after the class name. If we say `d= Die`, we've assigned the class object, `Die`, to the variable *d*. We have to say `d= Die()` to use the class name as a factory and create an instance of a class.
- You've got incorrect argument values for the parameters of the `__init__()`.

If we get a `NameError: name 'Hack' is not defined`, then the class (`Hack`, in this example) is not actually defined. This could mean one of three things: our class definition had errors in the first place, our definition class name isn't spelled the same as our object creation (either we spelled it wrong when defining the class, or spelled it wrong when using the class to create an object.) The third possible error is that we have defined the class in a module, imported it, but forgot to quality the class name with the module name.

If our class wasn't defined, it means we either forgot to define the class, or overlooked the `SyntaxError` when defining it. If our class has one name and our object constructor has another name, that's just carelessness; pick a name and stick to it. If we are trying to import our definitions, we can either qualify the names properly, or use `from module import *` as the import statement.

Another common problem is using the class name without `()`s. If we say `d= Die`, we've assigned the class object (`Die`) to the variable *d*. We have to say `d= Die()` to create an instance of a class.

If we've defined our class properly, we can get a message like `TypeError: __init__() takes exactly 2 arguments (1 given)` when we attempt to construct an object. This means that our `__init__()` method function doesn't match the object construction call that we made.

The `__init__()` function must have a *self* parameter name, and it must be first. When we construct an object, we don't provide an argument value for the *self* parameter, but we must provide values for all of the other parameters after *self*.

If your initialization function, `__init__()`, doesn't seem to work, the most likely cause is that you have misspelled the name. There are two `_` before and two `_` after the `init`.

Tip: Debugging Class vs. Object Issues

Perhaps the biggest mistake newbies make is attempting to exercise the method functions of a class instead of a specific object. You can't easily say `Die.roll()`, you'll get the cryptic `TypeError: unbound method` error message. The phrase "unbound method" means that no instance was being used.

When you say `d1= Die()`, you are creating an instance. When you see `d1.roll()`, then you are asking that specific object to do its `roll()` operation.

16.1.28 Module Definitions – Adding New Concepts

Tip: Debugging Imports There are four things that can go wrong with an **import**: (1) the module can't be found; (2) the module isn't valid Python; (3) the module doesn't define what you thought it should define; (4) the module name isn't unique and some other module with the same name is being found.

Be sure the module's `.py` file name is correct, and it's located on the *sys.path*. Module filenames are traditionally all lowercase, with minimal punctuation. Some operating systems (like GNU/Linux) are case-sensitive and a seemingly insignificant difference between `Random.py` and `random.py` can make your module impossible to find.

The two most visible places to put module files are the current working directory and the Python `Lib/site-packages` directory. For Windows, this directory is usually under `C:\python26\`. For GNU/Linux, this is often under the `/usr/lib/python2.6/` directory. For MacOS users, this will be in the `/System/Library/Frameworks/Python.framework/Versions/Current/` directory tree.

If your module isn't valid Python, you'll get syntax errors when you try to import it. You can discover the exact errors by trying to execute the module file using the `F5` key in **IDLE**.

If the module doesn't define what you thought, there are two likely causes: the Python definitions are incorrect, or you've omitted a necessary module-name qualifier. For example, when we do `import math` everything in that module requires the `math` qualifier. Within a module, however, we don't need to qualify names of other things defined in the same module file.

If your Python class or function definitions aren't correct, it has nothing to do with the modularization. The problem is more fundamental. Starting from something simple and adding features is generally the best way to learn.

The *sys.path* is a list, which is searched in order. Your working directory is searched first. When your module has the same name as some extension module, your module will conceal that extension module. I've spent hours discovering that my module named `Image` was concealing PIL's `Image` module.

16.1.29 Fixed-Point Numbers : Doing High Finance with `decimal`

Tip: Debugging `decimal`

There are a number of things that can go wrong with using `decimal` numbers. If we forget to import `decimal` (note that the module has a lower-case name), then we'll get `NameError` messages.

If we use an `import decimal`, we must say `decimal.Decimal` to make a new number. If we use `from decimal import *`, we can say `Decimal` to make a new number.

We can convert integers or strings to Decimal numbers. If we accidentally provide a floating-point value instead of a String, we get `TypeError: Cannot convert float to Decimal. First convert the float to a string`.

Tip: Debugging `decimal` Rounding

The quantization method appears strange at first because we provide a Decimal object rather than the number of decimal positions. The built-in `round()` function rounds to a number of positions. The `quantize()` method of a `decimal` number, however, uses a `decimal` number that defines the position to which to round.

If you get `AttributeError: 'int' object has no attribute '_is_special'`, from the `quantize()` function, this means you tried something like `aDecimal.quantize(3)`. You should use something like `aDecimal.quantize(Decimal('0.001'))`.

16.1.30 Time and Date Processing : The `time` and `datetime` Modules

Tip: Debugging `time`

Because of the various conversions, it's easy to get confused by having a floating-point time and a `time_struct` time. When you get `TypeError` exceptions, you are missing a conversion between the two representations. You can use the `help()` function and the Python Library Reference (chapter 6.10) to sort this out.

16.1.31 Text Processing and Pattern Matching : The `re` Module

Tip: Debugging Regular Expressions If you forget to import the module, then you get `NameError` on every class, function or variable reference.

If you spell the name wrong on your **import** statement, or the module isn't on your Python Path, you'll get an `ImportError`. First, be sure you've spelled the module name correctly. If you `import sys` and then look at `sys.path`, you can see all the places Python look for the module. You can look in each of those directories to see that the files are named.

There are two large problems that can cause problems with regular expressions: getting the regular expression wrong and getting the processing wrong.

The regular expression language, with it's special characters, escapes, and heavy use of \ is rather difficult to learn. If you get `error` exceptions from `re.compile()`, then your RE pattern is improper. For example `error: multiple repeat` means that your RE is misusing `"*"` characters. There are a number of these errors which indicate that you are likely missing a \ to escape the special meaning of one or more characters in your pattern.

If you get `TypeError` errors from `match()` or `search()`, then you have not used a candidate string with your pattern. Once you've compiled a pattern with `pat= re.compile("some pattern")`, you use that pattern object with candidate strings: `matching= pat.match("candidate")`. If you try '`pat.match(23)`', 23 isn't a string and you get a `TypeError`.

Beyond these very visible problems are the more subtle problem with a pattern that doesn't match what you think it should match. We'll look at this separately, in *More Debugging Hints*.

16.1.32 Wrapping and Packaging Our Solution

Tip: Debugging the Main Program Switch

There are two sides to the main program switch. When a module is executed from the command line, you want it to do useful things. When a module is imported by another module, you want it to provide definitions, but not actually do anything.

Command-Line Behavior. If you get a `NameError`, you misspelled ___*name*___. If, on the other hand, nothing seems to happen, then you may have misspelled `"__main__"`.

Another common problem is providing all of the class and function definitions, but omitting the main script entirely. The **class** and **def** statements all execute silently. If there's no main script to create the objects and call the functions, then nothing will happen.

Import Behavior. If things happen when you import a module, it's missing the main program switch. When a module is evolving from main program to library that is used by a new main program, we sometimes leave the old main program in place.

The best way to handle the change from main program to library is to put the old main program into a function with a name like `main()`, and then put it the simple main program switch that calls this `main()` function when the module name is `"__main__"`.

16.2 Bibliography

16.2.1 Use Cases

16.2.2 Computer Science

16.2.3 Design Patterns

16.2.4 Languages

16.2.5 Project Management

16.2.6 Problem Domains

16.3 Glossary

Here are some –perhaps– useful definitions of concepts.

16.3.1 Hardware Terminology

We want to define some terms that we'll be using throughout the book. We're going to build up our Python understanding from this foundational terminology. In the computer world, many concepts are new, and we'll try to make them more familiar to you. Further, some of the concepts are abstract, forcing us to borrow existing words and extend or modify their meanings. We'll also define them by example as we go forward in exploring Python.

This section is a kind of big-picture road map of computers. We'll refer back to these definitions in the sections which follow.

The first set of definitions are things we lump togther under "hardware", since they're mostly tangible things that sit on desks and require dusting. The next section has definitions that will include "software": those intangible things that don't require dusting.

Computer, Computer System Okay, this is perhaps silly, but we want to be very clear. We're talking about the whole *system* of interconnected parts that make up a computer. We're including all the Devices, incluing displays and keyboards and mice. We're drawing a line between our computer and the network that interconnects it to other computers.

A computer is a very generalized appliance. Without software, it's just a lump of parts. Even with the general softare components we'll talk about in *Software Terminology*, it doesn't do anything specific. We reserve the term "application software" for that software that applies this very general system to our specific needs.

Inside a computer system there are numerous electronic components, one of which is the *processor*, which controls most of what a computer does. Other components include *memory*.

It helps to think of two species of computers: your personal computer – desktop or laptop – sometimes called a "client" and shared computers called "servers". When you are surfing a web site, you are using more than one computer: your personal computer is running the web browser, and one or more server computers are responding to your browser's requests. Most of the internet things you see involve your desktop and a server somewhere else.

We do need to note that we're using the principle of *abstraction*. A number of electronic devices are all computers on which we can do Python programming. Laptops, desktops, iMacs, PowerBooks, clients, servers, Dells and HP's are all examples of this abstraction we're calling a computer system.

Device, Peripheral Device We have a number of devices that are part of our computers. Most devices are plugged into the computer box and connected by wires, putting them on the periphery of the computer. A few devices are wireless; they connect using Bluetooth, WiFi (IEEE 802.11) or infrared (IR) signals. We call the connection an *interface*.

The most important devices are hidden within the box, physically adjacent to the central processor. These central items are memory (called random-access memory, RAM) and a disk. The disk, while inside the box, is still considered peripheral because once upon a time, disks were huge and expensive.

The other peripheral devices are the ones we can see: display, keyboard and mouse. After that are other storage devices, including CDs, DVDs, USB drives, cameras, scanners, printers, drawing tablets, etc. Finally we have network connections, which can be Ethernet, wireless or a modem. All devices are controlled by pieces of software called *drivers*.

Note that we've applied the abstraction principle again. We've lumped a variety of components into abstract categories.

Memory, RAM The computer's working memory (*Random-Access Memory*, or RAM) contains two things: our data and the processing instructions (or program) for manipulating that data. Most modern computers are called *stored program digital* computers. The program is stored in memory along with the data. The data is represented as digits, not mechanical analogies. In contrast, an analog computer uses mechanical analogs for numbers, like spinning gears that make an analog speedometer show the speed, or the strip of metal that changes shape to make an analog meat thermometer show the temperature.

The central processor fetches each instruction from the computer's memory and then executes that instruction. We like to call this the *fetch-execute* loop that the processor carries out. The processor chip itself is *hardware*; the instructions in memory are called *software*. Since the instructions are stored in memory, they can be changed. We take this for granted every time we double click an icon and a program is loaded into memory. The data on which the processor is working must also be in memory. When we open a document file, we see it read from the disk into memory so we can work on it.

Memory is *dynamic*: it changes as the software does its work. Memory which doesn't change is called *Read-Only Memory* (ROM).

Memory is *volatile*: when we turn the computer off, the contents vanish. When we turn the computer on, the contents of memory are random, and our programs and data must be loaded into memory from some persistent device. The tradeoff for volatility is that memory is blazingly fast.

Memory is accessed "randomly": any of the 512 million bytes of my computer's memory can be accessed with equal ease. Other kinds of memory have sequential access; for example, magnetic cassette tapes must be accessed sequentially.

For hair-splitters, we recognize that there are special-purpose computing devices which have fixed programs that aren't loaded into memory at the click of a mouse. These devices have their software in read-only memory, and keep only data in working memory. When our program is permanently stored in ROM, we call it *firmware* instead of software. Most household appliances that have computers with ROM.

Disk, Hard Disk, Hard Drive We call these *disk* drives because the memory medium is a spinning magnetizable disk with read-write heads that shuttle across the surface; you can sometimes hear the clicking as the heads move. Individual digits are encoded across the surface of the disk; grouped into blocks of data. Some people are in the habit of calling them “hard” to distinguish them from the obsolete “floppy” disks that were used in the early days of personal computing.

Our various files (or “documents”) inluding our programs and our data will – eventually – reside on some kind of disk or disk-like device. However, the operating system interposes some structure, discipline and protocol between our needs for saving files and the vagaries of the disk device. We’ll look at this in *Software Terminology* and again in *Working with Files*.

Disk memory is described as “random access”, even though it isn’t completely random: there are read-write heads which move across the surface and the surface is rotating. There are delays while the computer waits for the heads to arrive at the right position. There are also delays while the computer waits for the disk to spin to the proper location under the heads. At 7200 RPM’s, you’re waiting less than 1/7200th of a second, but you’re still waiting.

Your computer’s disk can be imagined as persistent, slow memory: when we turn off the computer, the data remains intact. The tradeoff is that it is agonizingly slow: it reads and writes in milliseconds, close to a million times slower than dynamic memory.

Disk memory is also cheaper than RAM by a factor of at almost 1000: we buy 500 gigabytes (500 billion bytes, or 500,000 megabytes) of disk for $100; the cost of 512 megabytes of memory.

Human Interface, Display, Keyboard, Mouse The human interface to the computer typically consists of three devices: a display, a keyboard and a mouse. Some people use additional devices: a second display, a microphone, speakers or a drawing tablet are common examples. Some people replace the mouse with a trackball. These are often wired to the computer, but wireless devices are also popular.

In the early days of computers – before the invention of the mouse – the displays and keyboards could only handle characters: letters, numbers and punctuation. When we used computers in the early days, we spelled out each command, one line at a time. Now, we have the addition of sophisticated graphical displays and the mouse. When we use computers now, we point and click, using graphical gestures as our commands. Consequently, we have two kinds of human interfaces: the *Command-Line Interface* (CLI), and the *Graphical User Interface* (GUI).

A keyboard and a mouse provide inputs to software. They work by interrupting what the computer is doing, providing the character you typed, or the mouse button you pushed. A piece of software called the Operating System has the job of collecting this stream of input and providing it to the application software. A stream of characters is pretty simple. The mouse clicks, however, are more complex events because they involve the screen location as well as the button information, plus any keyboard shift keys.

A display shows you the outputs from software. The display device has to be shared by a number of application programs. Each program has one or more windows where their output is sent. The Operating System has the job of mediating this sharing to assure that one program doesn’t disturb another program’s window. Generally, each program will use a series of drawing commands to paint the letters or pictures. There are many, many different approaches to assembling the output in a window. We won’t touch on this because of the bewildering number of choices.

Historically, display devices used paper; everything was printed. Then they switched to video technology. Currently, displays use liquid crystal technology. Because displays were once almost entirely video, we sometimes summarize the human interface as the Keyboard-Video-Mouse (KVM).

In order to keep things as simple as possible, we’re going to focus on the command-line interface. Our programs will read characters from the keyboard, and display characters in an output window. Even though the programs we write won’t respond to mouse events, we’ll still use the mouse to interact with the operating system and programs like **IDLE**.

Other Storage, CD, DVD, USB Drive, Camera These storage devices are slightly different from the internal disk drive or hard drive. The differences are the degree of volatility of the medium. Packaged CDs and DVDs are read-only; we call them CD Read-Only Memory (CD-ROM). When we burn our own CD or DVD, we used to call it creating a Write-Once-Read-Many (WORM) device. Now there are CD-RW devices which can be written (slowly) many times, and read (quickly) many times, making the old WORM acronym outdated.

Where does that leave Universal Serial Bus USB drives (known by a wide variety of trademarked names like Thumb Drive™ or Jump Drive™) and the memory stick in our camera? These are just like the internal disk drive, except they don't involve a spinning magnetized disk. They are slower, have less capacity and are slightly more expensive than a disk.

Our operating system provides a single abstraction that makes our various disk drives and "other storage" all appear to be very similar. When we look at these devices they all appear to have folders and documents. We'll return to this unification in *File-Related Library Modules*.

Scanner, Printer These are usually USB devices; they are unique in that they send data in one direction only. Scanners send data into our computer; our computer sends data to a printer. These are a kind of storage, but they are focused on human interaction: scanning or printing photos or documents.

The scanner provides a stream of data to an application program. Properly interpreted, this stream of data is a sequence of picture elements (called "pixels") that show the color of a small section of the document on the scanner. Getting input from the scanner is a complex sequence of operations to reset the apparatus and gather the sequence of pixels.

A printer, similarly, accepts a stream of data. Properly interpreted, this stream of data is a sequence of commands that will draw the appropriate letters and lines in the desired places on the page. Some printers require a sequence of pixels, and the printer uses this to put ink on paper. Other printers use a more sophisticated page description language, which the printer processes to determine the pixels, and then deposits ink on paper. One example of these sophisticated graphic languages is PostScript.

Network, Ethernet, Wireless, WiFi, Dial-up, Modem A network is built from a number of cooperating technologies. Somewhere, buried under streets and closeted in telecommunications facilities is the global Internet: a collection of computers, wires and software that cooperates to route data. When you have a cable-modem, or use a wireless connection in a coffee shop, or use the Local Area Network (LAN) at school or work, your computer is (indirectly) connected to the Internet. There is a physical link (a wire or an antenna), there are software protocols for organizing the data and sharing the link properly. There are software libraries used by the programs on our computer to surf web pages, exchange email or purchase MP3s.

While there are endless physical differences among network devices, the rules, protocols and software make these various devices almost interchangeable. There is stack of technology that uses the principle of abstraction very heavily to minimize the distinctions among wireless and wired connections. This kind of abstraction assures that a program like a web browser will work precisely the same no matter what the physical link really is. The people who designed the Internet had abstraction very firmly in mind as a way to allow the Internet to expand with new technology and still work consistently.

16.3.2 Software Terminology

Hardware terminology is pretty simple. You can see and touch the hardware. You're rarely confused by the difference between a scanner and a printer.

Software, on the other hand, is less tangible. Programming is the act of creating new software. This terminology is perhaps more important than the hardware terminology above.

Note that Software is essential for making our computer do anything. The varius components and devices – without software – are inert lumps of plastic and metal.

Operating System The Operating System (OS) ties all of the computer's devices together to create a usable, integrated computer system. The operating system includes the software called *device drivers* that make the various devices work consistently. It manages scarce resources like memory and time by assuring that all the programs share those resources. The operating system also manages the various disk drives by imposing some organizing rules on the data; we call the organizing rules and the related software the *file system*.

The operating system creates the desktop metaphor that we see. It manages the various windows; it directs mouse clicks and keyboard characters to the proper application program. It depicts the file system with a visual metaphor of folders (directories) and documents (files). The desktop is the often shown to you by a program called the "finder" or "explorer"; this program draws the various icons and the dock or task bar.

In addition to managing devices and resources, the OS starts programs. Starting a program means allocating memory, loading the instructions from the disk, allocating processor time to the program, and allocating any other resources in the processor chip.

Finally, we have to note that it is the OS that provides most of the abstractions that make modern computing possible. The idea that a variety of individual types of devices and components could be summarized by a single abstraction of "storage" allows disk drives, CD-ROMs, DVD-ROMs and thumb drives to peacefully co-exist. It allows us to run out and buy a thumb drive and plug it into our computer and have it immediately available to store the pictures of our trip to Sweden.

Program, Application, Software A program is started by the operating system to do something useful. We'll look at this in depth in *What is a Program?* and *Goal-Directed Activities*. Since we will be writing our own programs, we need to be crystal clear on what programs really are and how they make our computer behave.

There isn't a useful distinction between words like "program", "command", "application", "application program", and "application system". Some vendors even call their programs "solutions". We'll try to stick to the word program. A program is rarely a single thing, so we'll try to identify a program with the one file that contains the main part of the program.

File, Document, Data, Database, the "File System" The data you want to keep is saved to the disk in *files*. Sometimes these are called *documents*, to make a metaphorical parallel between a physical paper document and a disk file. Files are collected into *directories*, sometimes depicted as metaphorical *folders*. A paper document is placed in a folder the same way a file is placed in a directory. Computer folders, however, can have huge numbers of documents. Computer folders, also, can contain other folders without any practical limit. The document and folder point of view is a handy visual metaphor used to clarify the file and directory structure on our disk.

This is so important that *Working with Files* is devoted to how our programs can work with files.

Boot Not footwear. Not a synonym for kick, as in "booted out the door." No, boot is used to describe a particular disk as the "boot disk". We call one disk the boot disk because of the way the operating system starts running: it pulls itself up by it's own bootstraps. Consider this quote from James Joyce's Ulysses: "There were others who had forced their way to the top from the lowest rung by the aid of their bootstraps."

The operating system takes control of the computer system in phases. A disk has a *boot sector* (or *boot block*) set aside to contain a tiny program that simply loads other programs into memory. This program can either load the expected OS, or it can load a specialized boot selection program (examples include BootCamp, GRUB, or LiLo.) The boot program allows you to control which OS is loaded. Either the boot sector directly loads the OS, or it loads and runs a boot program which loads the OS.

The part of the OS that is loaded into memory is just the kernel. Once the kernel starts running, it loads a few handy programs and starts these programs running. These programs then load the rest of the OS into memory. The device drivers must be added to the kernel. Once all of the device drivers are loaded, and the devices configured, then the user interface components can be loaded and started. At this point, the "desktop" appears.

Note that part of the OS (the kernel) loads other parts of the operating system into memory and starts them running. It pulls itself up by its own bootstraps. They call this bootstrapping, or booting. The kernel will also load our software into memory and start it running. We'll depend heavily on this central feature of an OS.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

EIGHTEEN

OTHER BACK MATTER

The following toolset was used for production of this book.

- Python 2.7b2
- Sphinx 1.0.7
- Docutils 0.7
- Komodo Edit 6.1.3
- TexShop 2.43

BIBLIOGRAPHY

[Jacobson92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering*. A Use Case Driven Approach. 1992. Addison-Wesley. 0201544350.

[Jacobson95] Ivar Jacobson, Maria Ericsson, Agenta Jacobson. *The Object Advantage*. Business Process Reengineering with Object Technology. 1995. Addison-Wesley. 0201422891.

[Boehm81] Barry Boehm. *Software Engineering Economics*. 1981. Prentice-Hall PTR. 0138221227.

[Comer95] Douglas Comer. *Internetworking with TCP/IP*. Principles, Protocols, and Architecture. 3rd edition. 1995. Prentice-Hall. 0132169878.

[Cormen90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction To Algorithms*. 1990. MIT Press. 0262031418.

[Dijkstra76] Edsger Dijkstra. *A Discipline of Programming*. 1976. Prentice-Hall. 0613924118.

[Gries81] David Gries. *The Science of Programming*. 1981. Springer-Verlag. 0387964800.

[Holt78] R. C. Holt, G. S. Graham, E. D. Lazowska, M. A. Scott. *Structured Concurrent Programming with Operating Systems Applications*. 1978. Addison-Wesley. 0201029375.

[Knuth73] Donald Knuth. *The Art of Computer Programming*. Fundamental Algorithms.. 1973. Addison-Wesley. 0201896834.

[Meyer88] Bertrand Meyer. *Object-Oriented Software Construction*. 1988. Prentice Hall. 0136290493.

[Parnas72] D. Parnas. *On the Criteria to Be Used in Decomposing Systems into Modules*. 1053-1058. 1972. Communications of the ACM.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Elements of Object-Oriented Software. 1995. Addison-Wesley Professional. 0201633612.

[Larman98] Craig Larman. *Applying UML and Patterns*. An Introduction to Object-Oriented Analysis and Design. 1998. Prentice-Hall. 0137488807.

[Lott05] Steven Lott. *Building Skills in Object-Oriented Design*. Step-by-Step Construction of A Complete Application. 2005. Steven F. Lott.

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design*. 1991. Prentice Hall. 0136298419.

[Geurts91] Leo Geurts, Lambert Meertens, Steven Pemberton. *The ABC Programmer's Handbook*. 1991. Prentice-Hall. 0-13-000027-2.

[Gosling96] Gosling, McGilton. *Java Language Environment White Paper*. 1996. Sun Microsystems.

[Harbison92] Samuel P. Harbison. *Modula-3*. 1992. Prentice-Hall. 0-13-596396-6.

[PythonTut] Python Tutorial. http://www.python.org/doc/2.5.4/tut/tut.html

[PythonRef] Python Reference Manual. http://www.python.org/doc/2.5.4/ref/ref.html

[PythonLib] Python Library Reference. http://www.python.org/doc/2.5.4/lib/lib.html

[Sun04] *Java Technology Reference Documentation*. 2004. Sun Microsystems.

[Wirth74] *Proceedings of the IFIP Congress 74*. 1974. North-Holland. *On the Design of Programming Languages*. 386-393.

[Royce98] Walker Royce. *Software Project Management*. A Unified Framework. 1998. Addison-Wesley Longman. 0201309580.

[Aceten04] *Ace-Ten*. 2004.

[Banks02] Robert B. Banks. *Slicing Pizzas, Racing Turtles, and Further Adventures in Applied Mathematics*. 2002. Princeton University Press. 0-691-10284-8.

[Dershowitz97] Nachum Dershowitz, Edward M. Reingold. *Calendrical Calculations*. 1997. Cambridge University Press. 0-521-56474-3.

[Latham98] Lance Latham. *Standard C Date/Time Library*. 1998. Miller-Freeman. 0-87930-496-0.

[Meeus91] Jean Meeus. *Astronomical Algorithms*. 1991. Willmann-Bell Inc. 0-943396-35-2.

[Neter73] John Neter, William Wasserman, G. A. Whitmore. *Fundamental Statistics for Business and Economics*. 4th edition. 1973. Allyn and Bacon, Inc.. 020503853.

[OBeirne65] T. M. O'Beirne. *Puzzles and Paradoxes*. 1965. Oxford University Press.

[Shackleford04] Michael Shackleford. *The Wizard Of Odds*. 2004.

[Silberstang05] Edwin Silberstang. *The Winner's Guide to Casino Gambling*. 4th edition. 2005. Owl Books. 0805077650.

[Skiena01] Steven Skiena. *Calculated Bets*. Computers, Gambling, and Mathematical Modeling to Win. 2001. Cambridge University Press. 0521009626.

PYTHON MODULE INDEX