Learning to Program Using Python

Cody Jackson

More information about this book, as well as source code files and to contact the author, can be found online at http://python-ebook.blogspot.com.

For those curious, this book was initially drafted on a Mac using Scrivener (http://www.literatureandlatte.com/). It was imported to LyX (http://www.lyx.org) for editing, typesetting, page layout, and other "book writing" tasks. LaTeX (http://www.latex-project.org/) was used to create the PDF and HTML versions of this book.

Contents

Part I

The Core Language

Chapter 1

Introduction

I originally wanted to learn Python because I wanted to make a computer game. I had taken several programming classes in college (C, C++, and Java) but nothing really serious. I'm not a Computer Science major and I don't program on a professional level.

I didn't really like the low-level work involved with C/C++. Things like pointers, memory management, and other concepts were difficult for me to grasp, much less effectively use. Java, as my first programming class in school, didn't make any sense. I had never used an object-oriented language before and object-oriented programming (OOP) concepts gave me fits. It probably didn't help that my Java class wasn't actually real Java; it was actually Microsoft's "custom" version: J++. So not only was I learning a language that had little practical use (J++ added and cut many features found in real Java), but the programs didn't work correctly. Ultimately the class was canceled near the end of the semester and everyone received full credit.

These problems, and issues learning other programming languages, left a bad taste in my mouth for programming. I never thought I learned a language well enough to feel comfortable using it, much less actually enjoy programming. But then I heard about Python on a computer forum, and noticed several other mentions of the language at other sites around the Internet. People were talking about how great the language was for personal projects and how versatile it is. I decided to give programming one more try and see if Python was the

language for me.

To give me more incentive to learn the language, I decided to recreate a role playing game from my childhood as a computer game. Not only would I have a reason to learn the language but, hopefully, I would have something useful that I could give to others for their enjoyment.

1.1 Why Python?

Python is regarded as being a great hobbyist language, yet it is also an extremely powerful language. It has bindings for C/C++ and Java so it can be used to tie large projects together or for rapid prototyping. It has a built-in GUI (graphical user interface) library via Tkinter, which lets the programmer make simple graphical interfaces with little effort. However, other, more powerful and complete GUI builders are available, such as Qt and GTK+. IronPython, a Python version for Windows using the .NET framework, is also available for those using Microsoft's Visual Studio products. Python can also be used in a real-time interpreter for testing code snippets before adding them into a normal "executable".

Python is classified as a scripting language. Generally speaking, this just means that it's not compiled to create the machine-readable code and that the code is "tied-into" another program as a control routine. Compiled languages, such as C++, require the programmer to run the source code through a compiler before the software is can be used by a computer. Depending on the program's size, the compilation process can take minutes to hours.

Using Python as a control routine means Python can act as a "glue" between different programs. For example, Python is often used as the scripting language for video games; while the heavy-duty work is performed by pre-compiled modules, Python can act in a call/response fashion, such as taking controller input and passing it to the appropriate module.

Python is also considered a high-level language, meaning it takes care of a lot of the grunt work involved in programming. For example, Python has a built-in garbage collector so you, as a programmer, don't really need to worry about memory management and memory leaks, a common occurrence when using older languages such as C.

The main emphasis of Python is readable code and enhancing programmer productivity. This is accomplished by enforcing a strict way of structuring the code to ensure the reader can follow the logic flow and by having an "everything's included" mentality; the programmer doesn't have to worry about including a lot of different libraries or other source code to make his program work.

One of the main arguments against Python is the use of whitespace. As shown in Chapter 3, many other languages require the programmer to use brackets, typically curly braces, i.e. "{}", to identify different blocks of code. With Python, these code blocks are identified by different amounts of indentation.

People who have spent a lot of time with "traditional" languages feel the lack of brackets is a bad thing, while others prefer the white space. Ultimately, it comes down to personal preference. I happen to like the lack of brackets because it's one less thing I have to troubleshoot when there is a coding problem. Imagine that one missing bracket in several dozen to hundreds lines of code is the reason your program won't work. Now imagine having to go through your code line by line to find the missing bracket. (Yes, programming environments can help but it's still one extra thing to consider).

1.2 Why Another Tutorial?

Even though there are several great tutorials at the Python web site (http://www.python.org), in addition to many books, my emphasis will be on the practical features of the language, i.e. I won't go into the history of the language or the esoteric ways it can be used. Though it will help if you have programmed before, or at least can understand programming logic and program flow, I will try to make sure that things start out slow so you don't get confused.

The main purpose of this book is to teach people how to program; Python just happens to be the language I have chosen to use. As mentioned above, it is a very friendly language which lets you learn how to program without getting in your way. Most people, when they decide to learn programming, want to jump into C, C++, or Java. However, these languages have little "gotchas" that can make learning difficult and dissuade people from continuing with programming. My

goal is to present programming in a fun, friendly manner so you will have the desire to learn more.

1.3 Getting Python

As of this revision, Python 3.x has been out for several years. Most of my experience is with Python 2.4 and 2.5, though much of my knowledge was gained from reading books written for version 2.2. As you can see, it doesn't necessarily mean your knowledge is obsolete when a new version comes out. Usually the newer versions simply add new features, often features that a beginner won't have a need for.

Python 3.x breaks compatibility with programs written in 2.x versions. However, much of the knowledge you gain from learning a 2.x version will still carry over. It just means you have to be aware of the changes to the language when you start using version 3.0. Plus, the install base of 2.x is quite large and won't be going away for quite some time. Due to the fact that most Linux distributions (and Mac OS X) still have older Python versions installed by default (some as old as v2.4), many of the code examples in this book are written for Python 2.x. Special note is made of significant changes between 2.x and 3.x in the chapter about 19, but learning either version won't harm you.

You can download Python from the Python web site (for Windows) or it may already be installed on your system if you're using a Mac, Linux, or *BSD. However, the Unix-like operating systems, including OS X, may not have the latest version so you may wish to upgrade, at least to version 2.6. Version 2.6 is a modification of 2.5 that allows use of both 2.x code and certain 3.0 functions. Essentially it lets you code in "legacy" style while still being able to use the latest features as desired, and testing which legacy features will be broken when moving to 3.0.

For those interested in using Python via a USB thumbdrive, you may be interested in Portable Python. This is a self-contained Python environment that you can either run from the thumbdrive or install to your computer. This is useful for people who can't or don't want to install Python but would still like to use it.

I'll assume you can figure out how to get the interactive interpreter running; if you need help, read the help pages on the web site. Gener-

ally speaking though, you open up a command prompt (or terminal) and type "python" at the prompt. This will open a Python session, allowing you to work with the Python interpreter in an interactive manner. In Windows, typically you just go to the Python file in All Programs and click it.

1.4 Conventions Used in this Book

The latest version of Python is 3.2 while the most current "legacy" version is 2.7. I will use the term 3.x to signify anything in the Python 3 family and 2.x for anything in the Python 2 family, unless explicitly stated.

The term "*nix" is used to refer to any Unix-like language, including Linux and the various flavors of BSD (FreeBSD, OpenBSD, NetBSD). Though Mac OS X is built upon FreeBSD, it is different enough to not be lumped in the *nix label.

Due to the word-wrap formatting for this book, some lines are automatically indented when they are really on the same line. It may make some of the examples confusing, especially because Python uses "white space" like tabs and spaces as significant areas. Thus, a word-wrapped line may appear to be an indented line when it really isn't. Hopefully you will be able to figure out if a line is intentionally tabbed over or simply wrapped.

Chapter 2

How is Python Different?

So what is Python? Chances you are asking yourself this. You may have found this book because you want to learn to program but don't know anything about programming languages. Or you may have heard of programming languages like C, C++, C#, or Java and want to know what Python is and how it compares to "big name" languages. Hopefully I can explain it for you.

2.1 Python Concepts

If your not interested in the the hows and whys of Python, feel free to skip to the next chapter. In this chapter I will try to explain to the reader why I think Python is one of the best languages available and why it's a great one to start programming with.

2.1.1 Dynamic vs Static Types

Python is a dynamic-typed language. Many other languages are static typed, such as C/C++ and Java. A static typed language requires the programmer to explicitly tell the computer what type of "thing" each data value is. For example, in C if you had a variable that was to contain the price of something, you would have to declare the variable as a "float" type. This tells the compiler that the only data that can be

used for that variable must be a floating point number, i.e. a number with a decimal point. If any other data value was assigned to that variable, the compiler would give an error when trying to compile the program.

Python, however, doesn't require this. You simply give your variables names and assign values to them. The interpreter takes care of keeping track of what kinds of objects your program is using. This also means that you can change the size of the values as you develop the program. Say you have another decimal number (a.k.a. a floating point number) you need in your program. With a static typed language, you have to decide the memory size the variable can take when you first initialize that variable. A double is a floating point value that can handle a much larger number than a normal float (the actual memory sizes depend on the operating environment). If you declare a variable to be a float but later on assign a value that is too big to it, your program will fail; you will have to go back and change that variable to be a double.

With Python, it doesn't matter. You simply give it whatever number you want and Python will take care of manipulating it as needed. It even works for derived values. For example, say you are dividing two numbers. One is a floating point number and one is an integer. Python realizes that it's more accurate to keep track of decimals so it automatically calculates the result as a floating point number. Here's what it would look like in the Python interpreter.

```
>>>6.0 / 2
3.0
>>>6 / 2.0
3.0
```

As you can see, it doesn't matter which value is on top or bottom; Python "sees" that a float is being used and gives the output as a decimal value.

2.1.2 Interpreted vs. Compiled

Many "traditional" languages are compiled, meaning the source code the developer writes is converted into machine language by the compiler. Compiled languages are usually used for low-level programming

(such as device drivers and other hardware interaction) and faster processing, e.g. video games.

Because the language is pre-converted to machine code, it can be processed by the computer much quicker because the compiler has already checked the code for errors and other issues that can cause the program to fail. The compiler won't catch all errors but it does help. The caveat to using a compiler is that compiling can be a time consuming task; the actual compiling time can take several minutes to hours to complete depending on the program. If errors are found, the developer has to find and fix them then rerun the compiler; this cycle continues until the program works correctly.

Python is considered an interpreted language. It doesn't have a compiler; the interpreter processes the code line by line and creates a *bytecode*. Bytecode is an in-between "language" that isn't quite machine code but it isn't the source code. Because of this in-between state, bytecode is more transferable between operating systems than machine code; this helps Python be cross-platform. Java is another language that uses bytecodes.

However, because Python uses an interpreter rather than compiler, the code processing can be slower. The bytecode still has to be "deciphered" for use by the processor, which takes additional time. But the benefit to this is that the programmer can immediately see the results of his code. He doesn't have to wait for the compiler to decide if there is a syntax error somewhere that causes the program to crash.

2.1.3 Prototyping

Because of interpretation, Python and similar languages are used for rapid application development and program prototyping. For example, a simple program can be created in just a few hours and shown to a customer in the same visit.

Programmers can repeatedly modify the program and see the results quickly. This allows them to try different ideas and see which one is best without investing a lot of time on dead-ends. This also applies to creating graphical user interfaces (GUIs). Simple "sketches" can be laid out in minutes because Python not only has several different GUI libraries available but also includes a simple library (Tkinter) by default.

Another benefit of not having a compiler is that errors are immediately generated by the Python interpreter. Depending on the developing environment, it will automatically read through your code as you develop it and notify you of syntax errors. Logic errors won't be pointed out but a simple mouse click will launch the program and show you final product. If something isn't right, you can simply make a change and click the launch button again.

2.1.4 Procedural vs. Object-Oriented Programming

Python is somewhat unique in that you have two choices when developing your programs: procedural programming or object-oriented. As a matter of fact, you can mix the two in the same program.

Briefly, procedural programming is a step-by-step process of developing the program in a somewhat linear fashion. Functions (sometimes called subroutines) are called by the program at times to perform some processing, then control is returned back to the main program. C and BASIC are procedural languages.

Object-oriented programming (OOP) is just that: programming with objects. Objects are created by distinct units of programming logic; variables and methods (an OOP term for functions) are combined into objects that do a particular thing. For example, you could model a robot and each body part would be a separate object, capable of doing different things but still part of the overall object. OOP is also heavily used in GUI development.

Personally, I feel procedural programming is easier to learn, especially at first. The thought process is mostly straightforward and essentially linear. I never understood OOP until I started learning Python; it can be a difficult thing to wrap your head around, especially when you are still figuring out how to get your program to work in the first place.

Procedural programming and OOP will be discussed in more depth later in the book. Each will get their own chapters and hopefully you will see how they build upon familiar concepts.

Chapter 3

Comparison of Programming Languages

For the new programmer, some of the terms in this book will probably be unfamiliar. You should have a better idea of them by the time you finish reading this book. However, you may also be unfamiliar with the various programming languages that exist. This chapter will display some of the more popular languages currently used by programmers. These programs are designed to show how each language can be used to create the same output. You'll notice that the Python program is significantly simpler than the others.

The following code examples all display the song, "99 Bottles of Beer on the Wall" (they have been reformatted to fit the pages). You can find more at the official 99 Bottles website: http://99-bottles-of-beer.net. I can't vouch that each of these programs is valid and will actually run correctly, but at least you get an idea of how each one looks. You should also realize that, generally speaking, white space is not significant to a programming language (Python being one of the few exceptions). That means that all of the programs below could be written one one line, put into one huge column, or any other combination. This is why some people hate Python because it forces them to have structured, readable code.

3.1 C

```
/*
* 99 bottles of beer in ansi c
*
* by Bill Wein: bearheart@bearnet.com
*
*/
#define MAXBEER (99)
void chug(int beers);

main()
{
register beers;
for(beers = MAXBEER; beers; chug(beers--))
    puts("");
puts("\nTime to buy more beer!\n");
exit(0);
}

void chug(register beers)
{
char howmany[8], *s;
s = beers != 1 ? "s" : "";
printf("%d bottle%s of beer on the wall,\n", beers
    , s);
printf("%d bottle%s of beeeeer . . . ,\n", beers,
    s);
printf("Take one down, pass it around,\n");

if(--beers) sprintf(howmany, "%d", beers); else
    strcpy(howmany, "No more");
s = beers != 1 ? "s" : "";
printf("%s bottle%s of beer on the wall.\n",
    howmany, s);
}
```

3.2 C++

```
//C++ version of 99 Bottles of Beer, object
    oriented paradigm
//programmer: Tim Robinson timtroyr@ionet.NET

#include <fstream.h>

enum Bottle { BeerBottle };

class Shelf {
    unsigned BottlesLeft;
public:
    Shelf( unsigned bottlesbought )
        : BottlesLeft( bottlesbought )
        {}
    void TakeOneDown()
        {
        if (!BottlesLeft)
            throw BeerBottle;
        BottlesLeft--;
        }
    operator int () { return BottlesLeft; }
    };

int main( int, char ** )
    {
    Shelf Beer(99);
    try {
        for (;;) {
            char *plural = (int)Beer !=1 ? "s" : "
                ";
            cout << (int)Beer << " bottle" <<
                plural
                << " of beer on the wall," << endl
                    ;
            cout << (int)Beer << " bottle" <<
```

```
                plural
                << " of beer," << endl;
            Beer.TakeOneDown();
            cout << "Take one down, pass it around
                ," << endl;
            plural = (int)Beer !=1 ? "s":"";
            cout << (int)Beer << " bottle" <<
                plural
                << " of beer on the wall." << endl
                    ;
            }
        }
    catch ( Bottle ) {
        cout << "Go to the store and buy some more
            ," << endl;
        cout << "99 bottles of beer on the wall."
           << endl;
        }
    return 0;
    }
```

3.3 Java

```
/**
* Java 5.0 version of the famous "99 bottles of
    beer on the wall".
* Note the use of specific Java 5.0 features and
    the strictly correct output.
*
* @author kvols
*/
import java.util.*;
class Verse {
    private final int count;
    Verse(int verse) {
    count= 100-verse;
    }
    public String toString() {
```

```
        String c=
            "{0,choice,0#no more bottles|1#1
                bottle|1<{0} bottles} of beer";
        return java.text.MessageFormat.format(
            c.replace("n","N")+" on the wall, "+c+
                ".\n"+
            "{0,choice,0#Go to the store and buy
                some more"+
            "|0<Take one down and pass it around},
                 "+c.replace("{0","{1")+
            " on the wall.\n", count, (count+99)
                %100);
    }
}
class Song implements Iterator<Verse> {
    private int verse=1;
    public boolean hasNext() {
        return verse <= 100;
    }
    public Verse next() {
        if(!hasNext())
            throw new NoSuchElementException("End
                of song!");
        return new Verse(verse++);
    }
    public void remove() {
        throw new UnsupportedOperationException
            ("Cannot remove verses!");
    }
}
    public class Beer {
        public static void main(String[] args ) {
            Iterable<Verse> song= new Iterable<
                Verse>() {
                 public Iterator<Verse> iterator()
                     {
                      return new Song();
                 }
```

```
            };

            // All this work to utilize this
                feature:
            // "For each verse in the song..."
            for(Verse verse : song) {
                System.out.println(verse);
            }
        }
    }
```

3.4 C#

```
/// Implementation of Ninety-Nine Bottles of Beer
    Song in C#.
/// What's neat is that .NET makes the Binge class
     a full-fledged component that may be called
    from any other  .NET component.
///
/// Paul M. Parks
/// http://www.parkscomputing.com/
/// February 8, 2002
///

using System;

namespace NinetyNineBottles
{
    /// <summary>
    /// References the method of output.
    /// </summary>
    public delegate void Writer(string format,
        params object[] arg);

    /// <summary>
    /// References the corrective action to take
        when we run out.
    /// </summary>
```

```
public delegate int MakeRun();

/// <summary>
/// The act of consuming all those beverages.
/// </summary>
public class Binge
{
    /// <summary>
    /// What we'll be drinking.
    /// </summary>
    private string beverage;

    /// <summary>
    /// The starting count.
    /// </summary>
    private int count = 0;

    /// <summary>
    /// The manner in which the lyrics are
        output.
    /// </summary>
    private Writer Sing;

    /// <summary>
    /// What to do when it's all gone.
    /// </summary>
    private MakeRun RiskDUI;

    public event MakeRun OutOfBottles;

    /// <summary>
    /// Initializes the binge.
    /// </summary>
    /// <param name="count">How many we're
        consuming.
    /// </param>
    /// <param name="disasterWaitingToHappen">
    /// Our instructions, should we succeed.
```

```
        /// </param>
        /// <param name="writer">How our drinking
            song will be heard.</param>
        /// <param name="beverage">What to drink
            during this binge.</param>
        public Binge(string beverage, int count,
            Writer writer)
        {
            this.beverage = beverage;
            this.count = count;
            this.Sing = writer;
        }

        /// <summary>
        /// Let's get started.
        /// </summary>
        public void Start()
        {
            while (count > 0)
            {
                Sing(
                    @"
{0} bottle{1} of {2} on the wall,
{0} bottle{1} of {2}.
Take one down, pass it around,",
                    count, (count == 1)
                        ? "" : "s", beverage);

                count--;

                if (count > 0)
                {
                    Sing("{0} bottle{1} of {2}
                        on the wall.",
                        count, (count == 1)
                            ? "" : "s", beverage);
                }
                else
```

```
                {
                    Sing("No more bottles of
                        {0} on the wall.",
                            beverage, null);
                }

            }

            Sing(
                @"
    No more bottles of {0} on the wall,
    No more bottles of {0}.", beverage, null);

            if (this.OutOfBottles != null)
            {
                count = this.OutOfBottles();
                Sing("{0} bottles of {1} on
                    the wall.", count, beverage);
            }
            else
            {
                Sing("First we weep, then we sleep
                    .");
                Sing("No more bottles of {0} on
                    the wall.",
                    beverage, null);
            }
        }
    }

    /// <summary>
    /// The song remains the same.
    /// </summary>
    class SingTheSong
    {
        /// <summary>
        /// Any other number would be strange.
        /// </summary>
```

```
        const int bottleCount = 99;

        /// <summary>
        /// The entry point. Sets the parameters
            of the Binge and starts it.
        /// </summary>
        /// <param name="args">unused</param>
        static void Main(string[] args)
        {
            Binge binge =
                new Binge("beer", bottleCount,
                new Writer(Console.WriteLine));
            binge.OutOfBottles += new MakeRun(
                SevenEleven);
            binge.Start();
        }

        /// <summary>
        /// There's bound to be one nearby.
        /// </summary>
        /// <returns>Whatever would fit in the
            trunk.</returns>
        static int SevenEleven()
        {
            Console.WriteLine("Go to the store,
                get some more...");
            return bottleCount;
        }
    }
}
```

3.5 Python

```
#!/usr/bin/env python
# -*- coding: iso-8859-1 -*-
"""
99 Bottles of Beer (by Gerold Penz)
Python can be simple, too :-)
```

```
"""
for quant in range(99, 0, -1):
    if quant > 1:
        print (quant, "bottles of beer on the wall
            ,", quant, "bottles of beer.")
        if quant > 2:
            suffix = str(quant - 1) + " bottles of
                beer on the wall."
    else:
        suffix = "1 bottle of beer on the wall."
elif quant == 1:
        print "1 bottle of beer on the wall, 1
            bottle of beer."
        suffix = "no more beer on the wall!"
    print "Take one down, pass it around,", suffix
    print "--"
```

Chapter 4

The Python Interpreter

4.1 Launching the Python interpreter

Python can be programmed via the interactive command line (aka the interpreter or IDE) but anything you code won't be saved. Once you close the session it all goes away. To save your program, it's easiest to just type it in a text file and save it (be sure to use the *.py* extension, i.e. *foo.py*)

To use the interpreter, type "python" at the command prompt (*nix and Mac) or launch the Python IDE (Windows and Mac). If you're using Windows and installed the Python *.msi* file, you should be able to also type Python on the command prompt. The main difference between the IDE and the command prompt is the command prompt is part of the operating system while the IDE is part of Python. The command prompt can be used for other tasks besides messing with Python; the IDE can only be used for Python. Use whichever you're more comfortable with.

If you're using Linux, BSD, or another *nix operating system, I'll assume you technologically-inclined enough to know about the terminal; you probably even know how to get Python up and running already. For those who aren't used to opening Terminal or Command Prompt (same thing, different name on different operating systems), here's how to do it.

4.1.1 Windows

1. Open the Start menu.
2. Click on "Run..."
3. Type "cmd" in the text box (without the quotes) and hit Return.
4. You should now have a black window with white text. This is the command prompt.
5. If you type "python" at the prompt, you should be dropped into the Python interpreter prompt.

4.1.2 Mac

1. Open Applications
2. Open Utilities
3. Scroll down and open Terminal
4. You should have a similar window as Windows users above.
5. Type "python" at the prompt and you will be in the Python interpreter.

Here is what your terminal should look like now:

Listing 4.1: Example Python interpreter

```
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license"
    for more information.
>>>
```

You may notice that the Python version used in the above examples is 2.5.1; the current version is 3.2. Don't panic. The vast majority of what you will learn will work regardless of what version you're using. My favorite Python book, Python How to Program, is based on version 2.2 but I still use it nearly every day as a reference while coding.

For the most part, you won't even notice what version is in use, unless you are using a library, function, or method for a specific version. Then, you simply add a checker to your code to identify what version the user has and notify him to upgrade or modify your code so it is backwards-compatible.

(A later chapter in this book will cover all the major differences you need to be aware of when using Python 3.x. Right now you're just learning the basics that you will use regardless of which version of the language you end up using.)

The >>> in Listing 4.1 is the Python command prompt; your code is typed here and the result is printed on the following line, without a prompt. For example, Listing 4.2 shows a simple print statement from Python 2.5:

Listing 4.2: Python command prompt

```
>>>print "We are the knights who say, 'Ni'."
We are the knights who say, 'Ni'.
```

4.2 Python Versions

As an aside, Python 3.x has changed *print* from a simple statement, like Listing 4.2, into a bona-fide function (Listing 4.3). Since we haven't talked about functions yet, all you need to know is that Python 3.x simply requires you to use the *print* statement in a slightly different way: parenthesis need to surround the quoted words. Just so you're aware.

Listing 4.3: Print function (Python 3.x)

```
>>>print ("We are the knights who say, 'Ni'.")
We are the knights who say, 'Ni'.
```

The vast majority of the code in this book will be written for Python 2.6 or earlier, since those versions are installed by default on many *nix systems and is therefore still very popular; there is also a lot of older code in the wild, especially open-source programs, that haven't been (and probably never will be) upgraded to Python 3.x. If you decide to use other Python programs to study from or use, you

will need to know the "old school" way of Python programming; many open-source programs are still written for Python 2.4.

If you are using Python 3.x and you want to type every example into your computer as we go along, please be aware that the print statements, as written, won't work. They will have to be modified to use a **print()** function like in Listing 4.3.

4.3 Using the Python Command Prompt

If you write a statement that doesn't require any "processing" by Python, it will simply return you to the prompt, awaiting your next order. The next code example shows the user assigning the value "spam" to the variable **can**. Python doesn't have to do anything with this, in regards to calculations or anything, so it accepts the statement and then waits for a new statement.

Listing 4.4: Python statements

```
>>>can = "spam"
>>>
```

(By the way, Python was named after Monty Python, not the snake. Hence, much of the code you'll find on the Internet, tutorials, and books will have references to Monty Python sketches.)

The standard Python interpreter can be used to test ideas before you put them in your code. This is a good way to hash out the logic required to make a particular function work correctly or see how a conditional loop will work. You can also use the interpreter as a simple calculator. This may sound geeky, but I often launch a Python session for use as a calculator because it's often faster than clicking through Windows' menus to use its calculator.

Here's an example of the "calculator" capabilities:

Listing 4.5: Python as a calculator

```
>>>2+2
4
>>>4*4
16
>>>5**2 #five squared
```

25

Python also has a math library that you can import to do trigonometric functions and many other higher math calculations. Importing libraries will be covered later in this book.

4.4 Commenting Python

One final thing to discuss is that comments in Python are marked with the "#" symbol. Comments are used to annotate notes or other information without having Python try to perform an operation on them. For example,

Listing 4.6: Python comments

```
>>>dict = {"First phonetic":"Able", "Second
    phonetic":"Baker"}  #create a dictionary
>>>print dict.keys()     #dictionary values aren't
    in order
['Second phonetic', 'First phonetic']
>>>print dict["First phonetic"] #print the key's
    value
Able
```

You will see later on that, even though Python is a very readable language, it still helps to put comments in your code. Sometimes it's to explicitly state what the code is doing, to explain a neat shortcut you used, or to simply remind yourself of something while you're coding, like a "todo" list.

4.5 Launching Python programs

If you want to run a Python program, simply type python at the shell command prompt (not the IDE or interactive interpreter) followed by the program name.

Listing 4.7: Launching a Python program

```
$python foo.py
```

Files saved with the .py extension are called modules and can be called individually at the command line or within a program, similar to header files in other languages. If your program is going to import other modules, you will need to make sure they are all saved in the same directory on the computer. More information on working with modules can be found later in this book or in the Python documentation.

Depending on the program, certain arguments can be added to the command line when launching the program. This is similar to adding switches to a Windows DOS prompt command. The arguments tell the program what exactly it should do. For example, perhaps you have a Python program that can output it's processed data to a file rather than to the screen. To invoke this function in the program you simply launch the program like so:

Listing 4.8: Launching a Python program with arguments

```
$python foo.py -f
```

The "-f" argument is received by the program and calls a function that prints the data to a designated location within the computer's file system instead of printing it to the screen.

If you have multiple versions of Python installed on your computer, e.g. the system default is version 2.5 but you want to play around with Python 3.x, you simply have to tell the OS which version to use (see Listing 4.9). This is important since many older Python programs aren't immediately compatible with Python 3.x. In many cases, an older version of Python must be retained on a computer to enable certain programs to run correctly; you don't want to completely overwrite older Python versions.

Listing 4.9: Selecting a Python version

```
$python2.5 sample.py    #force use of Python 2.5
$python3.0 sample.py    #force use of Python 3.0
```

4.6 Integrated Development Environments

I should take the time now to explain more about programming environments. Throughout this book, most of the examples involve using

the Python interactive interpreter, which is used by typing "python" at the operating system command prompt. This environment is is really more for testing ideas or simple "one-shot" tasks. Because the information isn't stored anywhere, when the Python session is finished, all the data you worked with goes away.

To make a reusable program, you need to use some sort of source code editor. These editors can be an actual programming environment application, a.k.a. IDEs (integrated development environments), or they can be a simple text editor like Windows Notepad. There is nothing special about source code; regardless of the programming language, it is simply text. IDEs are basically enhanced text editors that provide special tools to make programming easier and quicker.

Python has many IDEs available and nearly all of them are free. Typically, an IDE includes a source code editor, a debugger, a compiler (not necessary for Python), and often a graphical user interface builder; different IDEs may include or remove certain features. Below is a list of some common Python IDEs:

- Eric-a free application that acts as a front-end to other programs and uses plug-ins
- IDLE-a free application included in the base Python installation; includes an integrated debugger
- Komodo-a full-featured, proprietary application that can be used for other programming languages
- PyDev-a plug-in for the Eclipse development environment
- Stani's Python Editor (SPE)-a free application that includes many development aids

Chapter 5

Types and Operators

Python is based on the C programming language and is written in C, so much of the format Python uses will be familiar to C and C++ programmers. However, it makes life a little easier because it's not made to be a low-level language (it's difficult to interact heavily with hardware or perform memory allocation) and it has built-in "garbage collection" (it tracks references to objects and automatically removes objects from memory when they are no longer referenced), which allows the programmer to worry more about how the program will work rather than dealing with the computer.

5.1 Python Syntax

5.1.1 Indentation

Python forces the user to program in a structured format. Code blocks are determined by the amount of indentation used. As you'll recall from the Comparison of Programming Languages chapter, brackets and semicolons were used to show code grouping or end-of-line termination for the other languages. Python doesn't require those; indentation is used to signify where each code block starts and ends. Here is an example (line numbers are added for clarification):

Listing 5.1: White space is significant

```
1 x = 1
2 if x:    #if x is true
3     y = 2
4     if y:    #if y is true
5         print "block 2"
6     print "block 1"
7 print "block 0"
```

Each indented line demarcates a new code block. To walk through the above code snippet, line 1 is the start of the main code block. Line 2 is a new code section; if "x" has a value not equal to 0, then indented lines below it will be evaluated. Hence, lines 3 and 4 are in another code section and will be evaluated if line 2 is true. Line 5 is yet another code section and is only evaluated if "y" is not equal to 0. Line 6 is part of the same code block as lines 3 and 4; it will also be evaluated in the same block as those lines. Line 7 is in the same section as line 1 and is evaluated regardless of what any indented lines may do.

You'll notice that compound statements, like the *if* comparisons, are created by having the header line followed by a colon (":"). The rest of the statement is indented below it. The biggest thing to remember is that indentation determines grouping; if your code doesn't work for some reason, double-check which statements are indented.

A quick note: the act of saying "x = 1" is assigning a value to a variable. In this case, "x" is the variable; by definition its value varies. That just means that you can give it any value you want; in this case the value is "1". Variables are one of the most common programming items you will work with because they are what store values and are used in data manipulation.

5.1.2 Multiple Line Spanning

Statements can span more than one line if they are collected within braces (parenthesis "()", square brackets "[]", or curly braces "{}"). Normally parentheses are used. When spanning lines within braces, indentation doesn't matter; the indentation of the initial bracket used to determine which code section the whole statement belongs to. String

statements can also be multi-line if you use triple quotes. For example:

Listing 5.2: Use of triple quotes

```
>>> big = """This is
... a multi-line block
... of text; Python puts
... an end-of-line marker
... after each line. """
>>>
>>> big
'This is\012a multi-line block\012of text; Python
    puts\012an end-of-line marker \012after each
    line.'
```

Note that the \012 is the octal version of \n, the "newline" indicator. The ellipsis (...) above are blank lines in the interactive Python prompt used to indicate the interpreter is waiting for more information.

5.2 Python Object Types

Like many other programming languages, Python has built-in data types that the programmer uses to create his program. These data types are the building blocks of the program. Depending on the language, different data types are available. Some languages, notably C and C++, have very primitive types; a lot of programming time is simply used up to combine these primitive types into useful data structures. Python does away with a lot of this tedious work. It already implements a wide range of types and structures, leaving the developer more time to actually create the program. Trust me; this is one of the things I hated when I was learning C/C++. Having to constantly recreate the same data structures for every program is not something to look forward to.

Python has the following built-in types: numbers, strings, lists, dictionaries, tuples, and files. Naturally, you can build your own types if needed, but Python was created so that very rarely will you have to "roll your own". The built-in types are powerful enough to cover the vast majority of your code and are easily enhanced. We'll finish up

this section by talking about numbers; we'll cover the others in later chapters.

Before I forget, I should mention that Python doesn't have strong coded types; that is, a variable can be used as an integer, a float, a string, or whatever. Python will determine what is needed as it runs. See below:

Listing 5.3: Weak coding types

```
>>> x = 12
>>> y = "lumberjack"
>>> x
12
>>> y
'lumberjack'
```

Other languages often require the programmer to decide what the variable must be when it is initially created. For example, C would require you to declare "x" in the above program to be of type *int* and "y" to be of type *string*. From then on, that's all those variables can be, even if later on you decide that they should be a different type.

That means you have to decide what each variable will be when you start your program, i.e. deciding whether a number variable should be an integer or a floating-point number. Obviously you could go back and change them at a later time but it's just one more thing for you to think about and remember. Plus, anytime you forget what type a variable is and you try to assign the wrong value to it, you get a compiler error.

5.3 Python Numbers

Python can handle normal long integers (max length determined based on the operating system, just like C), Python long integers (max length dependent on available memory), floating point numbers (just like C doubles), octal and hex numbers, and complex numbers (numbers with an imaginary component). Here are some examples of these numbers:

- integer: 12345, -32

- Python integer: 999999999L (In Python 3.x, all integers are Python integers)
- float: 1.23, 4e5, 3e-4
- octal: 012, 0456
- hex: 0xf34, 0X12FA
- complex: 3+4j, 2J, 5.0+2.5j

Python has the normal built-in numeric tools you'd expect: expression operators (*, >>, +, <, etc.), math functions (pow, abs, etc.), and utilities (rand, math, etc.). For heavy number-crunching Python has the Numeric Python (NumPy) extension that has such things as matrix data types. If you need it, it has to be installed separately. It's heavily used in science and mathematical settings, as it's power and ease of use make it equivalent to Mathematica, Maple, and MatLab.

Though this probably doesn't mean much to non-programmers, the expression operators found in C have been included in Python, however some of them are slightly different. Logic operators are spelled out in Python rather than using symbols, e.g. logical AND is represented by "and", not by "&&"; logical OR is represented by "or", not "||"; and logical NOT uses "not" instead of "!". More information can be found in the Python documentation.

Operator level-of-precedence is the same as C, but using parentheses is highly encouraged to ensure the expression is evaluated correctly and enhance readability. Mixed types (float values combined with integer values) are converted up to the highest type before evaluation, i.e. adding a float and an integer will cause the integer to be changed to a float value before the sum is evaluated.

Following up on what I said earlier, variable assignments are created when first used and do not have to be pre-declared like in C.

Listing 5.4: Generic C++ example

```
int a = 3;  //inline initialization of integer
float b;    //sequential initialization of
    floating point number
b = 4.0f;
```

Listing 5.5: Generic Python example

```
>>>a = 3      #integer
>>>b = 4.0    #floating point
```

As you can see, “a” and “b” are both numbers but Python can figure out what type they are without being told. In the C++ example, a float value has to be “declared” twice; first the variable is given a type (“float”) then the actual value is given to the variable. You’ll also note that comments in Python are set off with a hash/pound sign (#) and are used exactly like the "//" comments in C++ or Java.

That’s about it for numbers in Python. It can also handle bit-wise manipulation such as left-shift and right-shift, but if you want to do that, then you’ll probably not want to use Python for your project. As also stated, complex numbers can be used but if you ever need to use them, check the documentation first.

Chapter 6

Strings

Strings in programming are simply text, either individual characters, words, phrases, or complete sentences. They are one of the most common elements to use when programming, at least when it comes to interacting with the user. Because they are so common, they are a native data type within Python, meaning they have many powerful capabilities built-in. Unlike other languages, you don't have to worry about creating these capabilities yourself. This is good because the built-in ones have been tested many times over and have been optimized for performance and stability.

Strings in Python are different than most other languages. First off, there are no char types, only single character strings (char types are single characters, separate from actual strings, used for memory conservation). Strings also can't be changed in-place; a new string object is created whenever you want to make changes to it, such as concatenation. This simply means you have to be aware that you are not manipulating the string in memory; it doesn't get changed or deleted as you work with it. You are simply creating a new string each time.

Here's a list of common string operations:

- s1 = ' ' : empty string
- s2 = "knight's" : double quotes
- block = """ - """ : triple-quoted block

- s1 + s2 : concatenate (combine)
- s2 * 3 : repeat the string a certain number of times
- s2[n] : index (the position of a certain character)
- len(s2) : get the length of a string
- "a %s parrot" % 'dead' : string formatting (deprecated in Python 3.x)
- "a {0} parrot".format("dead") : string formatting (Python 3.x)
- for x in s2 : iteration (sequentially move through the string's characters)
- 'm' in s2 : membership (is there a given character in the string?)

Empty strings are written as two quotes with nothing in between. The quotes used can be either single or double; my preference is to use double quotes since you don't have to escape the single quote to use it in a string. That means you can write a statement like

```
''And then he said, 'No way' when I told him.''
```

If you want to use just one type of quote mark all the time, you have to use the backslash character to "escape" the desired quote marks so Python doesn't think it's at the end of the phrase, like this:

```
''And then he said, \''No way\'' when I told him.''
```

Triple quoted blocks are for strings that span multiple lines, as shown last chapter. Python collects the entire text block into a single string with embedded newline characters. This is good for things like writing short paragraphs of text, e.g. instructions, or for formatting your source code for clarification.

6.1 Basic string operations

The "+" and "*" operators are overloaded in Python, letting you concatenate and repeat string objects, respectively. Overloading is just using the same operator to do multiple things, based on the situation where it's used. For example, the "+" symbol can mean addition when two numbers are involved or, as in this case, combining strings.

Concatenation combines two (or more) strings into a new string object whereas repeat simply repeats a given string a given number of times. Here are some examples:

Listing 6.1: Operator overloading

```
>>> len('abc') #length: number items
3
>>> 'abc' + 'def' #concatenation: a new string
'abcdef'
>>> 'Ni!' * 4    #multiple concatentation: "Ni!" +
    "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

You need to be aware that Python doesn't automatically change a number to a string, so writing "spam" + 3 will give you an error. To explicitly tell Python that a number should be a string, simply tell it. This is similar to casting values in C/C++. It informs Python that the number is not an integer or floating point number but is, in reality, a text representation of the number. Just remember that you can no longer perform mathematical functions with it; it's strictly text.

Listing 6.2: Casting a number to a string

```
>>>str(3) #converts number to string
```

Iteration in strings is a little different than in other languages. Rather than creating a loop to continually go through the string and print out each character, Python has a built-in type for iteration. Here's an example followed by an explanation:

Listing 6.3: Iteration through a string

```
>>> myjob = "lumberjack"
>>> for c in myjob: print c, #step through items
...
l u m b e r j a c k
>>> "k" in myjob     #1 means true
1
```

Essentially what is happening is that Python is sequentially going through the variable "myjob" and printing each character that exists in the string. *for* statements will be covered in depth later in the book

but for now just be aware that they are what you use to step through a range of values. As you can see they can be used for strings or, more often, numbers.

The second example is simply a comparison. Does the letter "k" exist in the value stored by "myjob"? If yes, then Python will return a numeric value of 1, indicating yes. If "k" didn't exist, it would return a 0. This particular case is most often used in word processing applications, though you can probably think of other situations where it would be useful.

6.2 Indexing and slicing strings

Strings in Python are handled similar to arrays in C. Unlike C arrays, characters within a string can be accessed both front and backwards. Front-ways, a string starts of with a position of 0 and the character desired is found via an offset value (how far to move from the end of the string). However, you also can find this character by using a negative offset value from the end of the string. I won't go deeply into it, but here's a quick example:

Listing 6.4: String indexing

```
>>>S = "spam"
>>>S[0], S[-2] #indexing from the front and rear
('s', 'a')
```

Indexing is simply telling Python where a character can be found within the string. Like many other languages, Python starts counting at 0 instead of 1. So the first character's index is 0, the second character's index is 1, and so on. It's the same counting backwards through the string, except that the last letter's index is -1 instead of 0 (since 0 is already taken). Therefore, to index the final letter you would use -1, the second to the last letter is -2, etc. Knowing the index of a character is important for slicing.

Slicing a string is basically what it sounds like: by giving upper and lower index values, we can pull out just the characters we want. A great example of this is when processing an input file where each line is terminated with a newline character; just slice off the last character

and process each line. You could also use it to process command-line arguments by "filtering" out the program name. Again, here's an example:

Listing 6.5: String slicing

```
>>>S = "spam"
>>>S[1:3], S[1:], S[:-1]     #slicing: extract section
('pa', 'pam', 'spa')
```

You'll notice that the colon symbol is used when slicing. The colon acts as a separator between the upper and lower index values. If one of those values is not given, Python interprets that to mean that you want everything from the index value to the end of the string. In the example above, the first slice is from index 1 (the second letter, inclusive) to index 3 (the 4th letter, exclusive). You can consider the index to actually be the space before each letter; that's why the letter "m" isn't included in the first slice but the letter "p" is.

The second slice is from index 1 (the second letter) to the end of the string. The third slice starts at the end of the string and goes backwards.

6.3 String Formatting

Formatting strings is simply a way of presenting the information on the screen in a way that conveys the information best. Some examples of formatting are creating column headers, dynamically creating a sentence from a list or stored variable, or stripping extraneous information from the strings, such as excess spaces. (Python 3.x has a new way of formatting strings; this will be discussed in the 19 section below.)

Python supports the creation of dynamic strings. What this means is that you can create a variable containing a value of some type (such as a string or number) then "call" that value into your string. You can process a string the same way as in C if you choose to, such as %d for integers and %f for floating point numbers. Here's an example:

Listing 6.6: Dynamic string creation

```
>>>S = "parrot"
```

```
>>>d = 1
>>>print 'That is %d dead %s!' % (d, s)
That is 1 dead parrot!
```

Python also has a string utility module for tools such as case conversion, converting strings to numbers, etc. Here's yet another example:

Listing 6.7: String utilities

```
>>> import string    #standard utilities module
>>> S = "spammify"
>>> string.upper(S) #convert to uppercase
'SPAMMIFY'
>>> string.find(S, "mm")     #return index of substring
3
>>> string.atoi("42"), `42` #convert from/to string
(42, '42')
>>> string.join(string.split(S, "mm"), "XX")
'spaXXify'
```

Notice the example of the second to last line. Backquotes are used to convert an object into a string. This is one way around the "don't mix strings and numbers" problem from earlier. I'll leave the last line example above as a mental test. See if you can figure out what the statement is doing.

Though it's not strictly a string operation (it can be used with just about anything that can be measured), the **len()** method can be used to give you the length of a string. For example,

Listing 6.8: Finding the length of a string

```
>>>string = "The Life of Brian"
>>>print len(string)
17
>>>len("The Meaning of Life")
19
```

As shown in the second example above, you don't necessarily have to use a print statement (or **print()** function in Python 3.x) to display a value. Simply writing what you want will print out the result. However, this doesn't always work in your favor. Sometimes the object

will only return a memory address, as we will see later in the book. Generally speaking, it's simply easier to explicitly state "print" if you want a statement evaluated and printed out. Otherwise you don't know exactly what value it will return.

6.4 Combining and Separating Strings

Strings can be combined (joined) and separated (split) quite easily. Tokenization is the process of splitting something up into individual tokens; in this case, a sentence is split into individual words. When a web page is parsed by a browser, the HTML, Javascript, and any other code in the page is tokenized and identified as a keyword, operator, variable, etc. The browser then uses this information to display the web page correctly, or at least as well as it can.

Python does much the same thing (though with better results). The Python interpreter tokenizes the source code and identifies the parts that are part of the actual programming language and the parts that are data. The individual tokens are separated by delimiters, characters that actually separate one token from another.

In strings, the main delimiter is a whitespace character, such as a tab, a newline, or an actual space. These delimiters mark off individual characters or words, sentences, and paragraphs. When special formatting is needed, other delimiters can be specified by the programmer.

Joining strings combines the separate strings into one string. Because string operations always create a new string, you don't have to worry about the original strings being overwritten. The catch is that it doesn't concatenate the strings, i.e. joining doesn't combine them like you would expect. Here's an example:

Listing 6.9: Joining strings

```
>>>string1 = "1 2 3"
>>>string2= "A B C"
>>>string3 = string2.join(string1)
>>>print string3
1A B C A B C2A B C A B C3
```

As you can see, the results are not what you expect. However, when creating a complex string, it can be better to put the pieces into a list and then simply join them, rather than trying to concatenate them.

Mentioned briefly before, I will speak a little more about concatenation. Concatenation combines two or more strings into a new, complete string. This is probably what you were thinking when I talked about joining strings together.

Listing 6.10: String concatenation

```
>>>string1 + string2
'1 2 3A B C'
>>>"Navy gravy. " + "The finest gravy in the Navy."
Navy gravy. The finest gravy in the Navy.
```

Chances are you will use concatenation more often than joining. To me, it simply makes more sense than messing with join(). But, with practice, you may find joining to be easier or more efficient.

Finally, splitting strings separates them into their component parts. The result is a list containing the individual words or characters. Here are some examples:

Listing 6.11: Splitting strings

```
>>>string = "My wife hates spam."
>>>string.split() #split string at spaces
['My', 'wife', 'hates', 'spam.']
>>>new_string = "1, 2, 3"
>>>new_string.split(",") #split string at commas
['1', ' 2', ' 3']
```

Note how the *new_string* was split exactly at the commas; the leading spaces before the numbers was included in the output. You should be aware of how your output will be when defining the separating characters.

As we move further into the Python language, we will look at these and other features of strings. Console programs will benefit most from learning how to use strings, however, word processors are obviously another place where knowing how to manipulate strings will come in handy.

One final note: adding a comma at the end of a *print* line prevents Python from automatically creating a newline. This is most practical when making tables and you don't want everything in a single column.

A handy reference of the most common string methods can be found on page 154 in the appendix. These methods perform operations, such as **split()** shown above, reducing the amount of work you have to do manually and providing a larger toolset for you to use.

6.5 Regular Expressions

I'm not going to delve into regular expressions in this book. They are a little too complicated for an introductory book. However, I will briefly explain so people new to programming understand how powerful of a tool regular expressions are.

Regular expressions (regex) are standardized expressions that allow you to search, replace, and parse text. Essentially, it's like using the find/replace tool in a word processor. However, regex is a complex, formal language format that allows you to do a lot more with strings than the normal methods allow you to do. To be honest, though, I have never used regular expressions, simply because my programs so far haven't required it.

Actually, if you can get by with the normal string methods, then by all means use them. They are quick, easy, and make it easy to understand your code. However, regex statements can make more sense than long, complex *if/else* conditions or daisy-chained string methods.

If you feel the need to use regular expressions, please consult the Python documentationdiscussing regular expressions. There is a lot of information there and regex is a little too advanced for this book; there are books solely dedicated to regex and I don't think I can do the topic justice.

Chapter 7

Lists

Lists in Python are one of the most versatile collection object types available. The other two types are dictionaries and tuples, but they are really more like variations of lists.

Python lists do the work of most of the collection data structures found in other languages and since they are built-in, you don't have to worry about manually creating them. Lists can be used for any type of object, from numbers and strings to more lists. They are accessed just like strings (e.g. slicing and concatenation) so they are simple to use and they're variable length, i.e. they grow and shrink automatically as they're used. In reality, Python lists are C arrays inside the Python interpreter and act just like an array of pointers.

Just so you know what exactly I'm talking about, Listing 7.1 shows a couple of quick examples that creates a list and then does a couple of manipulations to it.

Listing 7.1: Generic list examples

```
>>>list = [1, 2, 3, 4, 5]
>>>print list
[1, 2, 3, 4, 5]
>>>print list[0] #print the list item at index 0
1
>>>list.pop() #remove and print the last item
5
```

```
>>>print list #show that the last item was removed
[1, 2, 3, 4]
```

Here's a list of common list operations:

- L1 = [] An empty list
- L2 = [0, 1, 2, 3] Four items
- L3 = ['abc', ['def', 'ghi']] Nested sublists
- L2 [n], L3[n][j] L2[n:j], len(L2) Index, slice, length
- L1 + L2, L2 * 3 Concatenate, repeat
- for x in L2, 3 in L2 Iteration, membership
- L2.append(4), L2.sort(), L2.index(1), L2.reverse() Methods: grow, sort, search, reverse, etc.
- del L2[k], L2[n:j] = [] Shrinking
- L2[n] = 1, L2[n:j] = [4,5,6] Index assignment, slice assignment
- range(4), xrange(0, 4) Make lists/tuples of integers

The biggest thing to remember is that lists are a series of objects written inside square brackets, separated by commas. Dictionaries and tuples will look similar except they have different types of brackets.

7.1 List usage

Lists are most often used to store homogeneous values, i.e. a list usually holds names, numbers, or other sequences that are all one data type. They don't have to; they can be used with whatever data types you want to mix and match. It's just usually easier to think of a list as holding a "standard" sequence of items.

The most common use of a list is to iterate over the list and perform the same action to each object within the list, hence the use of similar data types. Time for an example:

Listing 7.2: Iterating through a list

```
>>> mylist = ["one", "two", "three"]
>>> for x in mylist:
...     print "number " + x
...
number one
number two
number three
```

In the above example, a list of text strings was created. Next, a simple *for* loop was used to iterate through the list, pre-pending the word "number" to each list object and printing them out. We will talk about *for* loops later but this is a common use of them.

One thing to note right now, however, is that you can use whatever value for "x" that you want, i.e. you can use whatever name you want instead of "x". I mention this because it kind of threw me for a loop when I first encountered it in Python. In other languages, loops like this are either hard-wired into the language and you have to use its format or you have to expressly create the "x" value beforehand so you can call it in the loop. Python's way is much easier because you can use whatever name makes the most sense, or you can simply use a "generic variable" like I did. For example, I could have used "for num in mylist:" or any other variation to iterate through the list. It's all up to you.

I won't go into the simple actions for lists since they work just like string operations. You can index, slice, and manipulate the list like you can for strings. In reality, a string is more like a modified list that only handles alphanumeric characters.

If you have questions, look at the chapter on Strings (Chapter 6); if you still have questions, look at the official Python documentation. Just remember that the resulting object will be a new list (surrounded by square brackets) and not a string, integers, etc.

7.2 Adding List Elements

Adding new items to a list is extremely easy. You simply tell the list to add it. Same thing with sorting a list.

Listing 7.3: Adding items to a list

```
>>>newlist = [1, 2, 3]
>>>newlist.append(54)
>>>newlist
[1, 2, 3, 54]
>>>a_list = ["eat", "ham", "Spam", "eggs", "and"]
>>>a_list.sort() #sort list items (capital letters
    come first)
>>>a_list
['Spam', 'and', 'eat', 'eggs', 'ham']
```

The **append()** method simply adds a single item to the end of a list; it's different from concatenation since it takes a single object and not a list. **append()** and **sort()** both change the list in-place and don't create a brand new list object, nor do they return the modified list. To view the changes, you have to expressly call the list object again, as shown in Listing 7.3. So be aware of that in case you are confused about whether the changes actually took place.

If you want to put the new item in a specific position in the list, you have to tell the list which position it should be in, i.e. you have to use the index of what the position is. Remember, the index starts at 0, not 1, as shown in Listing 7.4.

Listing 7.4: Adding items via indexing

```
>>>newlist.insert(1, 69)     #insert '69' at index '1'
>>>newlist
[1, 69, 2, 3, 54]
```

You can add a second list to an existing one by using the **extend()** method. Essentially, the two lists are concatenated (linked) together, like so:

Listing 7.5: Combining lists

```
>>>newerlist = ["Mary", "had, "a", "little", "spam
    ."]
>>>newlist.extend(newerlist) #extending with named
     list
>>> newlist
```

```
[1, 69, 2, 3, 54, 'Mary', 'had', 'a', 'little', '
    spam.']
>>>newerlist.extend(["It's", "grease", "was", "
    white", "as", "snow."]) #extending inline
>>> newerlist
['Mary', 'had', 'a', 'little', 'spam.', "It's", '
    grease', 'was', 'white', 'as', 'snow.']
```

Be aware, there is a distinct difference between extend and append. **extend()** takes a single argument, which is always a list, and adds each of the elements of that list to the original list; the two lists are merged into one. **append()** takes one argument, which can be any data type, and simply adds it to the end of the list; you end up with a list that has one element which is the appended object. Here's an example from "Dive into Python":

Listing 7.6: Extend vs. append

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f'])
>>> li
['a', 'b', 'c', 'd', 'e', 'f']  #merged list
>>> len(li)      #list length
6
>>> li[-1]  #reverse index
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f'])    #list object used
    as an element
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)
4
>>> li[-1]  #the single list object
['d', 'e', 'f']
```

7.3 Mutability

One of the special things about lists is that they are mutable, i.e. they can be modified in-place without creating a new object. The big concern with this is remembering that, if you do this, it can affect other references to it. However, this isn't usually a large problem so it's more of something to keep in mind if you get program errors.

Here's an example of changing a list using offset and slicing:

Listing 7.7: Changing a list

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'    #index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more'] #slice assignment:
    delete+insert
>>> L    #replaces items indexed at 0 and 1
['eat', 'more', 'SPAM!']
```

Because lists are mutable, you can also use the **del** statement to delete an item or section. Here's an example:

Listing 7.8: Deleting list items

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0]     #delete one item
>>> L
['eat', 'more', 'please']
>>> del L[1:]    #delete an entire section
>>> L    #same as L[1:] = []
['eat']
```

7.4 Methods

I've mentioned methods previously and I'll talk about methods more in the object-oriented programming chapter, but for the curious, a method works like a function, in that you have the method name followed by arguments in parentheses. The big difference is that a

method is qualified to a specific object with the period punctuation mark. In some of the examples above, the list object was affected by a method using the "." (dot) nomenclature. The "." told the Python interpreter to look for the method name that followed the dot and perform the actions in the method on the associated list object.

Because everything in Python is an object, nearly everything has some sort of a method. You probably won't be able to remember all the methods for every single object type, but remembering the most common and useful ones will speed up development. Having a list of the methods for each object type is very handy. You'll find lists of the most common methods for each object type in the appendices of this book.

Finally, you need to remember that only mutable objects can be changed in-place; strings, tuples, and other objects will always have to create new objects if you change them. You also need to remember that modifying an object in place can affect other objects that refer to it.

A complete listing of list methods can be found on page 160 in the appendix.

Chapter 8

Dictionaries

Next to lists, dictionaries are one of the most useful data types in Python. Python dictionaries are unordered collections of objects, matched to a keyword. Python lists, on the other hand, are ordered collections that use a numerical offset.

Because of their construction, dictionaries can replace many "typical" search algorithms and data structures found in C and related languages. For those coming from other languages, Python dictionaries are just like a hash table, where an object is mapped to a key name.

Dictionaries include the following properties:

1. Accessed by keyword, not an offset. Dictionaries are similar to associative arrays. Each item in the dictionary has a corresponding keyword; the keyword is used to “call” the item.

2. Stored objects are in a random order to provide faster lookup. When created, a dictionary stores items in any order it chooses. To get a value, simply supply the key. If you need to order the items within a dictionary, you have to do it yourself; there are no built-in methods for it.

3. Dictionaries are variable length, can hold objects of any type (including other dictionaries), and support deep nesting (multiple levels of items can be in a dictionary, such as a list within a dictionary within another dictionary).

4. They are mutable but can't be modified like lists or strings; they are the only data type that supports mapping.
5. Internally, dictionaries are implemented as a hash table.

Here's the (now standard) list of common operations:

- d1 = {} Empty dictionary
- d2 = {'spam' : 2, 'eggs' : 3} Two-item dictionary
- d3 = {'food' : {'ham' : 1, 'egg' : 2}} Nesting
- d2['eggs'], d3['food']['ham'] Indexing by key
- d2.has_key('eggs'), d2.keys(), d2.values() Methods: membership test, keys list, values list, etc.
- len(d1) Length (number stored entries)
- d2[key] = new, del d2[key] Adding/changing, deleting

8.1 Making a dictionary

As previously stated, you create dictionaries and access items via a key. The key can be of any immutable type, like a string, number, or tuple. The values can be any type of object, including other dictionaries. The format for making a dictionary is shown in Listing 8.1:

Listing 8.1: Dictionary format

```
>>>dictionary = {"key name":"value}
```

However, the key name and value can be anything allowed, such as:

Listing 8.2: Dictionary keys example

```
>>>dictionary = {"cow":"barn", 1:"pig", 2:["spam",
    "green", "corn"]}
```

Notice that the brackets for dictionaries are curly braces, the separator between a key word and it's associated value is a colon, and that each key/value is separated by a comma. This are just some of the things that can cause syntax errors in your program.

8.2 Basic operations

The **len()** function can be used to give the number of items stored in a dictionary or the length of the key list. The **keys()** method returns all the keys in the dictionary as a list. Here's a few examples:

Listing 8.3: Some dictionary methods

```
>>> d2 = {'spam':2, 'ham':1, 'eggs':3}
>>> d2['spam']  #fetch value for key
2
>>> len(d2) #number of entries in dictionary
3
>>> d2.has_key('ham')   #does the key exist? (1
    means true)
1
>>> d2.keys()   #list of keys
['eggs', 'spam', 'ham']
```

Since dictionaries are mutable, you can add and delete values to them without creating a new dictionary object. Just assign a value to a key to change or create an entry and use **del** to delete an object associated with a given key.

Listing 8.4: Modifying dictionaries

```
>>> d2['ham'] = ['grill', 'bake', 'fry']    #
    change entry
>>> d2
{'eggs' : 3, 'spam': 2, 'ham': ['grill', 'bake', '
    fry']}
>>> del d2['eggs']  #delete entry based on keyword
>>> d2
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> d2['brunch'] = 'Bacon'  #add new entry
>>> d2
{'brunch': 'Bacon', 'ham': ['grill', 'bake', 'fry'
    ],
'spam': 2}
```

To compare with lists, adding a new object to a dictionary only requires making a new keyword and value. Lists will return an "index

out-of-bounds" error if the offset is past the end of the list. Therefore you must append or slice to add values to lists.

Here is a more realistic dictionary example. The following example creates a table that maps programming language names (the keys) to their creators (the values). You fetch a creator name by indexing on language name:

Listing 8.5: Using a dictionary

```
>>> table = {'Python': 'Guido van Rossum',
... 'Perl': 'Larry Wall',
... 'Tcl': 'John Ousterhout' }
...
>>> language = 'Python'
>>> creator = table[language]
>>> creator
'Guido van Rossum'
>>> for lang in table.keys(): print lang, '\t',
    table[lang]
...
Tcl John Ousterhout
Python Guido van Rossum
Perl Larry Wall
```

From this example, you might notice that the last command is similar to string and list iteration using the *for* command. However, you'll also notice that, since dictionaries aren't sequences, you can't use the standard *for* statement. You must use the **keys()** method to return a list of all the keywords which you can then iterate through like a normal list.

You may have also noticed that dictionaries can act like lightweight databases. The example above creates a table, where the programming language "column" is matched by the creator's "row". If you have a need for a database, you might want to consider using a dictionary instead. If the data will fit, you will save yourself a lot of unnecessary coding and reduce the headaches you would get from dealing with a full-blown database. Granted, you don't have the flexibility and power of a true database, but for quick-and-dirty solutions, dictionaries will suffice.

8.3 Dictionary details

1. Sequence operations don't work. As previously stated, dictionaries are mappings, not sequences. Because there's no order to dictionary items, functions like concatenation and slicing don't work.

2. Assigning new indexes adds entries. Keys can be created when making a dictionary (i.e. when you initially create the dictionary) or by adding new values to an existing dictionary. The process is similar and the end result is the same.

3. Keys can be anything immutable. The previous examples showed keys as string objects, but any non-mutable object (like lists) can be used for a keyword. Numbers can be used to create a list-like object but without the ordering. Tuples (covered later) are sometimes used to make compound keys; class instances (also covered later) that are designed not to change can also be used if needed.

Well, we're nearly done with Python types. The next chapter will cover tuples, which are basically immutable lists.

8.4 Operation

Like the other chapters, a list of common dictionary operations can be found in the appendix on page 162.

Chapter 9

Tuples

The final built-in data type is the tuple. Python tuples work exactly like Python lists except they are immutable, i.e. they can't be changed in place. They are normally written inside parentheses to distinguish them from lists (which use square brackets), but as you'll see, parentheses aren't always necessary. Since tuples are immutable, their length is fixed. To grow or shrink a tuple, a new tuple must be created.

Here's a list of common operations for tuples:

- () An empty tuple
- t1 = (0,) A one-item tuple (not an expression)
- t2 = (0, 1, 2, 3) A four-item tuple
- t3 = 0, 1, 2, 3 Another four-item tuple (same as prior line, just minus the parenthesis)
- t3 = ('abc', ('def', 'ghi')) Nested tuples
- t1[n], t3[n][j] Index
- t1[i:j], Slice
- len(tl) Length

- t1 + t2 Concatenate
- t2 * 3 Repeat
- for x in t2, Iteration
- 3 in t2 Membership

The second entry shows how to create a one item tuple. Since parentheses can surround expressions, you have to show Python when a single item is actually a tuple by placing a comma after the item. The fourth entry shows a tuple without parentheses; this form can be used when a tuple is unambiguous. However, it's easiest to just use parentheses than to figure out when they're optional.

9.1 Why Use Tuples?

Tuples typically store heterogeneous data, similar to how lists typically hold homogeneous data. It's not a hard-coded rule but simply a convention that some Python programmers follow. Because tuples are immutable, they can be used to store different data about a certain thing. For example, a contact list could conceivably stored within a tuple; you could have a name and address (both strings) plus a phone number (integer) within on data object.

The biggest thing to remember is that standard operations like slicing and iteration return new tuple objects. In my programming, I like use lists for everything except when I don't want a collection to change. It cuts down on the number of collections to think about, plus tuples don't let you add new items to them or delete data. You have to make a new tuple in those cases.

There are a few times when you simply have to use a tuple because your code requires it. However, a lot of times you never know exactly what you're going to do with your code and having the flexibility of lists can be useful.

So why use tuples? Apart from sometimes being the only way to make your code work, there are few other reasons to use tuples:

- Tuples are processed faster than lists. If you are creating a constant set of values that won't change, and you need to simply iterate through them, use a tuple.

- The sequences within a tuple are essentially protected from modification. This way, you won't accidentally change the values, nor can someone misuse an API to modify the data. (An API is an application programming interface. It allows programmers to use a program without having to know the details of the whole program.)
- Tuples can be used as keys for dictionaries. Honestly, I don't think I've ever used this, nor can I think of a time when you would need to. But it's there if you ever need to use it.
- Tuples are used in string formatting, by holding multiple values to be inserted into a string. In case you don't remember, here's a quick example:

Listing 9.1: String formatting with tuples

```
>>>val1 = "integer"
>>>val2 = 2
>>>"The %s value is equal to %d" % (val1, val2)
'The integer value is equal to 2'
```

9.2 Sequence Unpacking

So, to create a tuple, we treat it like a list (just remembering to change the brackets).

Listing 9.2: Packing a tuple

```
>>>tuple = (1, 2, 3, 4)
```

The term for this is packing a tuple, because the data is "packed into" the tuple, all wrapped up and ready to go. So, to remove items from a tuple you simply unpack it.

Listing 9.3: Unpacking a tuple

```
>>>first, second, third, fourth = tuple
>>> first
1
>>> second
```

```
2
>>> third
3
>>> fourth
4
```

Neat, huh? One benefit of tuple packing/unpacking is that you can swap items in-place. With other languages, you have to create the logic to swap variables; with tuples, the logic is inherent in the data type.

Listing 9.4: In-place variable swapping

```
>>> bug = "weevil"
>>> bird = "African swallow"
>>> bug, bird = bird, bug
>>> bug
'African swallow'
>>> bird
'weevil'
```

Tuple unpacking and in-place swapping are one of the neatest features of Python, in my opinion. Rather than creating the logic to pull each item from a collection and place it in its own variable, tuple unpacking allows you to do everything in one step. In-place swapping is also a shortcut; you don't need to create temporary variables to hold the values as you switch places.

9.3 Methods

Tuples have no methods. Sorry.

The best you can do with tuples is slicing, iteration, packing and unpacking. However, Python has a neat little trick if you need more flexibility with tuples: you can change them into lists. Simply use the **list()** function call on a tuple and it magically becomes a list. Contrarily, you can call **tuple()** on a list and it becomes a tuple.

Listing 9.5: Converting lists and tuples

```
>>> my_list = ["moose", "Sweden", "llama"]
```

```
>>> my_tuple = ("Norwegian Blue", "parrot", "pet
    shop")
>>> tuple(my_list)
('moose', 'Sweden', 'llama')
>>> list(my_tuple)
['Norwegian Blue', 'parrot', 'pet shop']
```

Obviously the benefit to this is that you can arbitrarily switch between the two, depending on what you need to do. If, halfway through your program, you realize that you need to be able to manipulate a tuple but you don't want it to be always modifiable, you can make a new variable that calls the **list()** function on the tuple and then use the new list as needed.

So, now that we have the fundamental building blocks down, we can move on to how you use them in practice. However, we'll cover one last essential tool that all programmers need to know how to use: files.

Chapter 10

Files

The final built-in object type of Python allows us to access files. The **open()** function creates a Python file object, which links to an external file. After a file is opened, you can read and write to it like normal.

Files in Python are different from the previous types I've covered. They aren't numbers, sequences, nor mappings; they only export methods for common file processing. Technically, files are a pre-built C extension that provides a wrapper for the C *stdio* (standard input/output) filesystem. If you already know how to use C files, you pretty much know how to use Python files.

Files are a way to save data permanently. Everything you've learned so far is resident only in memory; as soon as you close down Python or turn off your computer, it goes away. You would have to retype everything over if you wanted to use it again.

The files that Python creates are manipulated by the computer's file system. Python is able to use operating system specific functions to import, save, and modify files. It may be a little bit of work to make certain features work correctly in cross-platform manner but it means that your program will be able to be used by more people. Of course, if you are writing your program for a specific operating system, then you only need to worry about the OS-specific functions.

10.1 File Operations

To keep things consistent, here's the list of Python file operations:

- output = open('/tmp/spam', 'w') Create output file ('w' means write)
- input = open('data', 'r') Create input file ('r' means read)
- S = input.read() Read entire file into a single string
- S = input.read(N) Read N number of bytes (1 or more)
- S = input.readline() Read next line (through end-line marker)
- L = input.readlines() Read entire file into list of line strings
- output.write(S) Write string S onto file
- output.writelines(L) Write all line strings in list L onto file
- output.close() Manual close (or it's done for you when automatically collected)

Because Python has a built-in garbage collector, you don't really need to manually close your files; once an object is no longer referenced within memory, the object's memory space is automatically reclaimed. This applies to all objects in Python, including files. However, it's recommended to manually close files in large systems; it won't hurt anything and it's good to get into the habit in case you ever have to work in a language that doesn't have garbage collection.

10.2 Files and Streams

Coming from a Unix-background, Python treats files as a data stream, i.e. each file is read and stored as a sequential flow of bytes. Each file has an end-of-file (EOF) marker denoting when the last byte of data has been read from it. This is useful because you can write a program that reads a file in pieces rather than loading the entire file into memory at one time. When the end-of-file marker is reached, your

program knows there is nothing further to read and can continue with whatever processing it needs to do.

When a file is read, such as with a **readline()** method, the end of the file is shown at the command line with an empty string; empty lines are just strings with an end-of-line character. Here's an example:

Listing 10.1: End of File example

```
>>> myfile = open('myfile', 'w') #open/create file
    for input
>>> myfile.write('hello text file') #write a line
    of text
>>> myfile.close()
>>> myfile = open('myfile', 'r') #open for output
>>> myfile.readline() #read the line back
'hello text file'
>>> myfile.readline()
'' #empty string denotes end of file
```

10.3 Creating a File

Creating a file is extremely easy with Python. As shown in the example above, you simply create the variable that will represent the file, open the file, give it a filename, and tell Python that you want to write to it.

If you don't expressly tell Python that you want to write to a file, it will be opened in read-only mode. This acts as a safety feature to prevent you from accidentally overwriting files. In addition to the standard "w" to indicate writing and "r" for reading, Python supports several other file access modes.

- "a": Appends all output to the end of the file; does not overwrite information currently present. If the indicated file does not exist, it is created.
- "r": Opens a file for input (reading). If the file does not exist, an IOError exception is raised. (Exceptions are covered in Chapter 14.)

- "r+": Opens a file for input and output. If the file does not exist, causes an IOError exception.
- "w": Opens a file for output (writing). If the file exists, it is overwritten. If the file does not exist, one is created.
- "w+": Opens a file for input and output. If the file exists, it is overwritten; otherwise one is created.
- "ab", "rb", "r+b", "wb", "w+b": Opens a file for binary (i.e., non-text) input or output. [Note: These modes are supported only on the Windows and Macintosh platforms. Unix-like systems don't care about the data type.]

When using standard files, most of the information will be alphanumeric in nature, hence the extra binary-mode file operations. Unless you have a specific need, this will be fine for most of your tasks. In a later section, I will talk about saving files that are comprised of lists, dictionaries, or other data elements.

10.4 Reading From a File

If you notice in the above list, the standard read-modes produce an I/O (input/output) error if the file doesn't exist. If you end up with this error, your program will halt and give you an error message, like below:

Listing 10.2: Input/Output error example

```
>>> file = open("myfile", "r")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '
    myfile'
>>>
```

To fix this, you should always open files in such a way as to catch the error before it kills your program. This is called "catching the exception", because the IOError given is actually an exception given by the Python interpreter. There is a chapter dedicated to exception handling (Chapter 14) but here is a brief overview.

When you are performing an operation where there is a potential for an exception to occur, you should wrap that operation within a *try/except* code block. This will try to run the operation; if an exception is thrown, you can catch it and deal with it gracefully. Otherwise, your program crashes and burns.

So, how do you handle potential exception errors? Just give it a *try*. (Sorry, bad joke.)

Listing 10.3: Catching errors, part 1

```
1  >>> f = open("myfile", "w")
2  >>> f.write("hello there, my text file.\nWill you
       fail gracefully?")
3  >>> f.close()
4  >>> try:
5  ...     file = open("myfile", "r")
6  ...     file.readlines()
7  ...     file.close()
8  ...     except IOError:
9  ...         print "The file doesn't exist"
10 ...
11 ['hello there, my text file.\n', 'Will you fail
       gracefully?']
12 >>>
```

What's going on here? Well, the first few lines are simply the same as first example in this chapter, then we set up a try/except block to gracefully open the file. The following steps apply to Listing 10.3.

1. We open the file to allow writing to it.
2. The data is written to the file.
3. The file is closed.
4. The try block is created; the lines below that are indented are part of this block. Any errors in this block are caught by the except statement.
5. The file is opened for reading.

6. The whole file is read and output. (The **readlines()** method returns a list of the lines in the file, separated at the newline character.)

7. The file is closed again.

8. If an exception was raised when the file is opened, the *except* line should catch it and process whatever is within the exception block. In this case, it simply prints out that the file doesn't exist.

So, what happens if the exception does occur? This:

Listing 10.4: Catching errors, part 2

```
>>> try:
...         file3 = open("file3", "r")
...         file3.readlines()
...         file3.close()
...         except IOError:
...             print "The file doesn't exist. Check
    filename."
...
The file doesn't exist. Check filename.
>>>
```

The file "file3" hasn't been created, so of course there is nothing to open. Normally you would get an IOError but since you are expressly looking for this error, you can handle it. When the exception is raised, the program gracefully exits and prints out the information you told it to.

One final note: when using files, it's important to close them when you're done using them. Though Python has built-in garbage collection, and it will usually close files when they are no longer used, occasionally the computer "loses track" of the files and doesn't close them when they are no longer needed. Open files consume system resources and, depending on the file mode, other programs may not be able to access open files.

10.5 Iterating Through Files

I've talked about iteration before and we'll talk about it in later chapters. Iteration is simply performing an operation on data in a sequential fashion, usually through the *for* loop. With files, iteration can be used to read the information in the file and process it in an orderly manner. It also limits the amount of memory taken up when a file is read, which not only reduces system resource use but can also improve performance.

Say you have a file of tabular information, e.g. a payroll file. You want to read the file and print out each line, with "pretty" formatting so it is easy to read. Here's an example of how to do that. (We're assuming that the information has already been put in the file. Also, the normal Python interpreter prompts aren't visible because you would actually write this as a full-blown program, as we'll see later. Finally, the print statements are not compatible with Python 3.x.)

Listing 10.5: Inputting tabular data

```
try:
    file = open("payroll", "r")
except IOError:
    print "The file doesn't exist. Check filename.
        "
individuals = file.readlines()
print "Account".ljust(10),  #comma prevents
    newline
print "Name".ljust(10),
print "Amount".rjust(10)
for record in individuals:
    columns = record.split()
    print columns[0].ljust(10)
    print columns[1].ljust(10)
    print columns[2].rjust(10)
file.close()
```

Here is how the output should be displayed on the screen (or on paper if sent to a printer).

Account	Name	Balance
101	Jeffrey	100.50
105	Patrick	325.49
110	Susan	210.50

A shortcut would be rewriting the *for* block so it doesn't have to iterate through the variable *individuals* but to simply read the file directly, as such:

```
for record in file:
```

This will iterate through the file, read each line, and assign it to "record". This results in each line being processed immediately, rather than having to wait for the entire file to be read into memory. The **readlines()** method requires the file to be placed in memory before it can be processed; for large files, this can result in a performance hit.

10.6 Seeking

Seeking is the process of moving a pointer within a file to an arbitrary position. This allows you to get data from anywhere within the file without having to start at the beginning every time.

The **seek()** method can take several arguments. The first argument (offset) is starting position of the pointer. The second, optional argument is the seek direction from where the offset starts. 0 is the default value and indicates an offset relative to the beginning of the file, 1 is relative to the current position within the file, and 2 is relative to the end of the file.

Listing 10.6: File seeking

```
file.seek(15) #move pointer 15 bytes from
    beginning of file
file.seek(12, 1) #move pointer 12 bytes from
    current location
file.seek(-50, 2) #move pointer 50 bytes backwards
    from end of file
file.seek(0, 2) #move pointer at end of file
```

The **tell()** method returns the current position of the pointer within the file. This can be useful for troubleshooting (to make sure

the pointer is actually in the location you think it is) or as a returned value for a function.

10.7 Serialization

Serialization (pickling) allows you to save non-textual information to memory or transmit it over a network. Pickling essentially takes any data object, such as dictionaries, lists, or even class instances (which we'll cover later), and converts it into a byte set that can be used to "reconstitute" the original data.

Listing 10.7: Pickling data

```
>>>import cPickle    #import cPickle library
>>>a_list = ["one", "two", "buckle", "my", "shoe"]
>>>save_file = open("pickled_list", "w")
>>>cPickle.dump(a_list, save_file)  #serialize
    list to file
>>>file.close()
>>>open_file = open("pickled_list", "r")
>>>b_list = cPickle.load(open_file)
```

There are two different pickle libraries for Python: cPickle and pickle. In the above example, I used the cPickle library rather than the pickle library. The reason is related to the information discussed in Chapter 2. Since Python is interpreted, it runs a bit slower compared to compiled languages, like C. Because of this, Python has a pre-compiled version of pickle that was written in C; hence cPickle. Using cPickle makes your program run faster.

Of course, with processor speeds getting faster all the time, you probably won't see a significant difference. However, it is there and the use is the same as the normal pickle library, so you might as well use it. (As an aside, anytime you need to increase the speed of your program, you can write the bottleneck code in C and bind it into Python. I won't cover that in this book but you can learn more in the official Python documentation.)

Shelves are similar to pickles except that they pickle objects to an access-by-key database, much like dictionaries. Shelves allow you to

simulate a random-access file or a database. It's not a true database but it often works well enough for development and testing purposes.

Listing 10.8: Shelving data

```
>>>import shelve     #import shelve library
>>>a_list = ["one", "two", "buckle", "my", "shoe"]
>>>dbase = shelve.open("filename")
>>>dbase["rhyme"] = a_list  #save list under key name
>>>b_list = dbase["rhyme"]  #retrieve list
```

Chapter 11

Statements

Now that we know how Python uses it's fundamental data types, let's talk about how to use them. Python is nominally a procedure-based language but as we'll see later, it also functions as an object-oriented language. As a matter of fact, it's similar to C++ in this aspect; you can use it as either a procedural or OO language or combine them as necessary.

The following is a listing of many Python statements. It's not all-inclusive but it gives you an idea of some of the features Python has.

Statement	*Role*	*Examples*
Assignment	Creating references	new_car = “Audi”
Calls	Running functions	stdout.write("eggs, ham, toast\n")
Print	Printing objects	print “The Killer”, joke
Print()	Python 3.x print function	print(“Have you seen my baseball?”)
If/elif/else	Selecting actions	if "python" in text: print “yes”
For/else	Sequence iteration	for X in mylist: print X
While/else	General loops	while 1: print ’hello’
Pass	Empty placeholder	while 1: pass
Break, Continue	Loop jumps	while 1: if not line: break
Try/except/finally	Catching exceptions	try: action() except: print ’action error’
Raise	Trigger exception	raise locationError
Import, From	Module access	import sys; from wx import wizard
Def, Return	Building functions	def f(a, b, c=1, *d): return a+b+c+d[0]
Class	Building objects	class subclass: staticData = []

11.1 Assignment

I’ve already talked about assignment before. To reiterate, assignment is basically putting the target name on the left of an equals sign and the object you’re assigning to it on the right. There’s only a few things you need to remember:

- Assignment creates object references.
  - Assignment acts like pointers in C since it doesn’t copy objects, just refers to an object. Hence, you can have multiple assignments of the same object, i.e. several different names referring to one object.
- Names are created when first assigned

  - Names don't have to be "pre-declared"; Python creates the variable name when it's first created. But as you'll see, this doesn't mean you can call on a variable that hasn't been assigned an object yet. If you call a name that hasn't been assigned yet, you'll get an exception error.
  - Sometimes you may have to declare a name and give it an empty value, simply as a "placeholder" for future use in your program. For example, if you create a class to hold global values, these global values will be empty until another class uses them.

- Assignment can be created either the standard way (food = "SPAM"), via multiple target (spam = ham = "Yummy"), with a tuple (spam, ham = "lunch", "dinner"), or with a list ([spam, ham] = ["blech", "YUM"]).

  - This is another feature that Python has over other languages. Many languages require you to have a separate entry for each assignment, even if they are all going to have the same value. With Python, you can keep adding names to the assignment statement without making a separate entry each time.

The final thing to mention about assignment is that a name can be reassigned to different objects. Since a name is just a reference to an object and doesn't have to be declared, you can change it's "value" to anything. For example:

Listing 11.1: Variable values aren't fixed

```
>>>x = 0      #x is linked to an integer
>>>x = "spam"    #now it's a string
>>>x = [1, 2, 3]      #now it's a list
```

11.2 Expressions/Calls

Python expressions can be used as statements but since the result won't be saved, expressions are usually used to call functions/methods and for printing values at the interactive prompt.

Here's the typical format:

Listing 11.2: Expression examples

```
spam(eggs, ham) #function call using parenthesis
spam.ham(eggs) #method call using dot operator
spam #interactive print
spam < ham and ham != eggs #compound expression
spam < ham < eggs #range test
```

The range test above lets you perform a Boolean test but in a "normal" fashion; it looks just like a comparison from math class. Again, another handy Python feature that other languages don't necessarily have.

11.3 Printing

Printing in Python is extremely simple. Using **print** writes the output to the C stdout stream and normally goes to the console unless you redirect it to another file.

Now is a good time to mention that Python has 3 streams for input/output (I/O). *sys.stdout* is the standard output stream; it is normally send to the monitor but can be rerouted to a file or other location. *sys.stdin* is the standard input stream; it normally receives input from the keyboard but can also take input from a file or other location. *sys.stderr* is the standard error stream; it only takes errors from the program.

The print statement can be used with either the sys.stdout or sys.stderror streams. This allows you to maximize efficiency. For example, you can print all program errors to a log file and normal program output to a printer or another program.

Printing, by default, adds a space between items separated by commas and adds a linefeed at the end of the output stream. To suppress the linefeed, just add a comma at the end of the print statement:

Listing 11.3: Print example (no line feed)

```
print lumberjack, spam, eggs,
```

To suppress the space between elements, just concatenate them when printing:

Listing 11.4: Printing concatenation

```
print "a" + "b"
```

Python 3.x replaces the simple print statement with the **print()** function. This is to make it more powerful, such as allowing overloading, yet it requires very little to change. Instead of using the print statement like I have throughout the book so far, you simply refer to it as a function. Here are some examples from the Python documentation page:

```
Old:  print "The answer is", 2*2
New:  print("The answer is", 2*2)

Old:  print x, #Trailing comma suppresses newline
New:  print(x, end=" ") #Appends a space instead of
a newline

Old:  print #Prints a newline
New:  print() #You must call the function!

Old:  print >> sys.stderr, "fatal error"
New:  print("fatal error", file=sys.stderr)

Old:  print (x, y) #prints repr((x, y))
New:  print((x, y)) #Not the same as print(x, y)!
```

11.4 *if* Tests

One of the most common control structures you'll use, and run into in other programs, is the *if* conditional block. Simply put, you ask a yes or no question; depending on the answer different things happen. For example, you could say, "If the movie selected is 'The Meaning of Life', then print 'Good choice.' Otherwise, randomly select a movie from the database."

If you've programmed in other languages, the *if* statement works the same as other languages. The only difference is the *else/if* as shown below:

Listing 11.5: Using *if* statements

```
if item == "magnet":
    kitchen_list = ["fridge"]
elif item == "mirror": #optional condition
    bathroom_list = ["sink"]
elif item == "shrubbery": #optional condition
    landscape_list = ["pink flamingo"]
else: #optional final condition
    print "No more money to remodel"
```

Having the *elif* (else/if) or the *else* statement isn't necessary but I like to have an *else* statement in my blocks. It helps clarify to me what the alternative is if the *if* condition isn't met. Plus, later revisions can remove it if it's irrelevant.

Unlike C, Pascal, and other languages, there isn't a *switch* or *case* statement in Python. You can get the same functionality by using *if/elif* tests, searching lists, or indexing dictionaries. Since lists and dictionaries are built at runtime, they can be more flexible. Here's an equivalent switch statement using a dictionary:

Listing 11.6: Dictionary as a *switch* statement

```
>>>choice = 'ham'
>>>print {'spam': 1.25, #a dictionary-based '
    switch'
... 'ham': 1.99,
... 'eggs': 0.99,
... 'bacon': 1.10}[choice]
1.99
```

Obviously, this isn't the most intuitive way to write this program. A better way to do it is to create the dictionary as a separate object, then use something like **has_key()** or otherwise find the value corresponding to your choice.

To be honest, I don't think about this way of using dictionaries when I'm programming. It's not natural for me yet; I'm still used to

using *if/elif* conditions. Again, you can create your program using *if/elif* statements and change them to dictionaries or lists when you revise it. This can be part of normal refactoring (rewriting the code to make it easier to manage or read), part of bug hunting, or to speed it up.

11.5 *while* Loops

while loops are a standard workhorse of many languages. Essentially, the program will continue doing something while a certain condition exists. As soon as that condition is not longer true, the loop stops.

The Python *while* statement is, again, similar to other languages. Here's the main format:

Listing 11.7: *while* loops, part 1

```
while <test>:    #loop test
    <code block>     #loop body
else:    #optional else statement
    <code block>     #run if didn't exit loop with
        break
```

break and *continue* work the exact same as in C. The equivalent of C's empty statement (a semicolon) is the *pass* statement, and Python includes an *else* statement for use with breaks. Here's a full-blown *while* example loop:

Listing 11.8: *while* loops, part 2

```
while <test>:
    <statements>
    if <test>: break     #exit loop now if true
    if <test>: continue #return to top of loop now
        if true
else:
    <statements>     #if we didn't hit a 'break'
```

break statements simply force the loop to quit early; when used with nested loops, it only exits the smallest enclosing loop. *continue*

statements cause the loop to start over, regardless of any other statements further on in the loop. The *else* code block is ran "on the way out" of the loop, unless a *break* statement causes the loop to quit early.

For those who are still confused, the next section will show how these statements are used in a real-world program with prime numbers.

11.6 *for* Loops

The *for* loop is a sequence iterator for Python. It will work on nearly anything: strings, lists, tuples, etc. I've talked about *for* loops before, and we will see a lot of them in future chapters, so I won't get into much more detail about them. The main format is below in Listing 11.9. Notice how it's essentially the same as a *while* loop.

Listing 11.9: *for* loops

```
for <target> in <object>:   #assign object items
    to target
    <statements>
     if <test>: break     #exit loop now, skip else
     if <test>: continue #go to top of loop now
else:
    <statements>     #if we didn't hit a 'break'
```

From *Learning Python* from O'Reilly publishing:

> "*When Python runs a* for *loop, it assigns items in the sequence object to the target, one by one, and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence, as though it were a cursor stepping through the sequence. Technically, the* for *works by repeatedly indexing the sequence object on successively higher indexes (starting at zero), until an index out-of-bounds exception is raised. Because* for *loops automatically manage sequence indexing behind the scenes, they replace most of the counter style loops you may be used to coding in languages like C.*"

In other words, when the *for* loop starts, it looks at the first item in the list. This item is given a value of 0 (many programming languages

start counting at 0, rather than 1). Once the code block is done doing its processing, the *for* loop looks at the second value and gives it a value of 1. Again, the code block does it's processing and the *for* loop looks at the next value and gives it a value of 2. This sequence continues until there are no more values in the list. At that point the *for* loop stops and control proceeds to the next statement in the program.

Listing 11.10 shows a practical version of a *for* loop that implements *break* and *else* statements, as explained in the Python documentation.

Listing 11.10: *break* and *else* statements

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:      #if the remainder
    of n/x is 0
...             print n, 'equals', x, '*', n/x
...             break   #exit immediately
...     else:
...         # loop fell through without finding a
    factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Related to *for* loops are *range* and counter loops. The **range()** function auto-builds a list of integers for you. Typically it's used to create indexes for a *for* statement but you can use it anywhere.

Listing 11.11: Using the **range()** function

```
>>>range(5) #create a list of 5 numbers, starting
    at 0
```

```
[0, 1, 2, 3, 4]
>>>range(2, 5) #start at 2 and end at 5 (remember
    the index values)
[2, 3, 4]
>>>range(0, 10, 2) #start at 0, end at 10 (index
   value), with an increment of 2
[0, 2, 4, 6, 8]
```

As you can see, a single argument gives you a list of integers, starting from 0 and ending at one less than the argument (because of the index). Two arguments give a starting number and the max value while three arguments adds a stepping value, i.e. how many numbers to skip between each value.

Counter loops simply count the number of times the loop has been processed. At the end of the loop, a variable is incremented to show that the loop has been completed. Once a certain number of loops have occurred, the loop is executed and the rest of the program is executed.

11.7 *pass* Statement

The *pass* statement is simply a way to tell Python to continue moving, nothing to see here. Most often, the *pass* statement is used while initially writing a program. You may create a reference to a function but haven't actually implemented any code for it yet. However, Python will be looking for something within that function. Without having something to process, the Python interpreter will give an exception and stop when it doesn't find anything. If you simply put a *pass* statement in the function, it will continue on without stopping.

Listing 11.12: *pass* statements

```
if variable > 12:
    print "Yeah, that's a big number."
else: pass
```

11.8 *break* and *continue* Statements

Already mentioned, these two statements affect the flow control within a loop. When a particular condition is met, the *break* statement "breaks" out of the loop, effectively ending the loop prematurely (though in an expected manner). The *continue* statement "short circuits" the loop, causing flow control to return to the top of the loop immediately.

I rarely use these statements but they are good to have when needed. They help ensure you don't get stuck in a loop forever and also ensure that you don't keep iterating through the loop for no good reason.

11.9 *try*, *except*, *finally* and *raise* Statements

I've briefly touched on some of these and will talk about them more in the Exceptions chapter. Briefly, *try* creates a block that attempts to perform an action. If that action fails, the *except* block catches any exception that is raised and does something about it. *finally* performs some last minute actions, regardless of whether an exception was raised or not. The *raise* statement manually creates an exception.

11.10 *import* and *from* Statements

These two statements are used to include other Python libraries and modules that you want to use in your program. This helps to keep your program small (you don't have to put all the code within a single module) and "isolates" modules (you only import what you need). *import* actually calls the other libraries or modules while *from* makes the import statement selective; you only import subsections of a module, minimizing the amount of code brought into your program.

11.11 *def* and *return* Statements

These are used in functions and methods. Functions are used in procedural-based programming while methods are used in object-oriented

programming. The *def* statement defines the function/method. The *return* statement returns a value from the function or method, allowing you to assign the returned value to a variable.

Listing 11.13: Defining functions

```
>>> a = 2
>>> b = 5
>>> def math_function():
...     return a * b
...
>>> product = math_function()
>>> product
10
```

11.12 Class Statements

These are the building blocks of OOP. *class* creates a new object. This object can be anything, whether an abstract data concept or a model of a physical object, e.g. a chair. Each class has individual characteristics unique to that class, including variables and methods. Classes are very powerful and currently "the big thing" in most programming languages. Hence, there are several chapters dedicated to OOP later in the book.

Chapter 12

Documenting Your Code

Some of this information is borrowed from Dive Into Python, a free Python programming book for experienced programmers. Other info is from the Python Style Guide and the Python Enhancement Proposal (PEP) 257. (Note that in this section, the information presented may be contrary to the official Python guides. This information is presented in a general format regarding docstrings and uses the conventions that I have developed. The reader is encouraged to review the official documentation for further details.)

You can document a Python object by giving it a *docstring*. A docstring is simply a triple-quoted sentence giving a brief summary of the object. The object can be a function, method, class, etc. (In this section, the term "function" is used to signify an actual function or a method, class, or other Python object.)

Listing 12.1: docstring example

```
def buildConnectionString(params):
"""Build a connection string from a dictionary of
    parameters.
Returns string."""
```

As noted previously, triple quotes signify a multi-line string. Everything between the start and end quotes is part of a single string, including carriage returns and other quote characters. You'll see them most often used when defining a docstring.

Everything between the triple quotes is the function's docstring, which documents what the function does. A docstring, if it exists, must be the first thing defined in a function (that is, the first thing after the colon).

You don't technically need to give your function a docstring, but you should; the docstring is available at runtime as an attribute of the function. Many Python IDEs use the docstring to provide context-sensitive documentation, so that when you type a function name, its docstring appears as a tooltip.

From the Python Style Guide:

> *"The docstring of a script should be usable as its 'usage' message, printed when the script is invoked with incorrect or missing arguments (or perhaps with a "-h" option, for "help"). Such a docstring should document the script's function and command line syntax, environment variables, and files. Usage messages can be fairly elaborate (several screenfuls) and should be sufficient for a new user to use the command properly, as well as a complete quick reference to all options and arguments for the sophisticated user."*

To be honest, I don't adhere to this rule all the time. I normally write a short statement about what the function, method, class, etc. is supposed to accomplish. However, as my programs evolve I try to enhance the docstring, adding what inputs it gets and what the output is, if any.

There are two forms of docstrings: one-liners and multi-line docstrings. One-liners are exactly that: information that doesn't need a lot of descriptive text to explain what's going on. Triple quotes are used even though the string fits on one line to make it easy to later expand it. The closing quotes are on the same line as the opening quotes, since it looks better. There's no blank line either before or after the docstring. The docstring is a phrase ending in a period. It describes the function's effect as a command ("Do this", "Return that"). It should not restate the function's parameters (or arguments) but it can state the expected return value, if present.

Listing 12.2: Good use of docstring

```
def kos_root():
```

```
"""Return the pathname of the KOS root directory."""
global _kos_root
if _kos_root: return _kos_root
#...
```

Again, I have to admit I'm not the best about this. I usually put the end quotes on a separate line and I have a space between the docstring and the start of the actual code; it makes it easier to simply add information and helps to delineate the docstring from the rest of the code block. Yes, I'm a bad person. However, as long as you are consistent throughout your projects, blind adherence to "the Python way" isn't necessary.

As a side note, it's not totally wrong to have the end quotes on a separate line; the multi-line docstring should (according to PEP 257) have them that way while a one-line docstring should have the end quotes on the same line. I've just gotten in the habit of using one method when writing my docstrings so I don't have to think about it.

Multi-line docstrings start out just like a single line docstring. The first line is a summary but is then followed by a blank line. After the blank line more descriptive discussion can be made. The blank line is used to separate the summary from descriptive info for automatic indexing tools. They will use the one-line summary to create a documentation index, allowing the programmer to do less work.

When continuing your docstring after the blank line, make sure to follow the indentation rules for Python, i.e. after the blank line all of your docstring info is indented as far as the initial triple-quote. Otherwise you will get errors when you run your program.

More info from the Python Style Guide:

> *"The docstring for a module should generally list the classes, exceptions and functions (and any other objects) that are exported by the module, with a one-line summary of each. (These summaries generally give less detail than the summary line in the object's docstring.)"*

The docstring for a function or method should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called (all if applicable). Optional arguments should be indicated. It should be documented whether keyword arguments are part of the interface.

The docstring for a class should summarize its behavior and list the public methods and instance variables. If the class is intended to be subclassed, and has an additional interface for subclasses, this interface should be listed separately (in the docstring). The class constructor should be documented in the docstring for its __init__ method (the "initialization" method that is invoked when the class is first called). Individual methods should be documented by their own docstring.

If a class subclasses another class and its behavior is mostly inherited from that class, its docstring should mention this and summarize the differences. Use the verb "override" to indicate that a subclass method replaces a superclass method and does not call the superclass method; use the verb "extend" to indicate that a subclass method calls the superclass method (in addition to its own behavior).

Python is case sensitive and the argument names can be used for keyword arguments, so the docstring should document the correct argument names. It is best to list each argument on a separate line, with two dashes separating the name from the description

If you've made it this far, I'll help you out and summarize what you just learned. Python has a documentation feature called "docstring" that allows you to use comments to create self-documenting source code. Several Python IDEs, such as Stani's Python Editor (SPE), can use these docstrings to create a listing of your source code structures, such as classes and modules. This makes it easier on the programmer since less work is required when you create your help files and other program documentation. Documentation indexers can pull the docstrings out of your code and make a listing for you, or you could even make your own script to create it for you. You are also able to manually read the docstrings of objects by calling the __doc__ method for an object; this is essentially what the above IDEs and indexers are doing. Listing 12.3 shows how a docstring for Python's *random* module.

Listing 12.3: docstring for Python's *random* module

```
>>>import random
>>>print random.__doc__
Random variable generators.
        integers
        ________
```

```
            uniform within range

        sequences
        ---------

            pick random element
            pick random sample
            generate random permutation

        distributions on the real line:
        -------------------------------

            uniform
            triangular
            normal (Gaussian)
            lognormal
            negative exponential
            gamma
            beta
            pareto
            Weibull

        distributions on the circle (angles 0 to 2
           pi)
        ---------------------------------------

            circular uniform
            von Mises

General notes on the underlying Mersenne Twister
   core generator:
* The period is 2**19937-1.
* It is one of the most extensively tested
   generators in existence.
* Without a direct way to compute N steps forward,
    the semantics of jumpahead(n) are weakened to
   simply jump to another distant state and rely
   on the large period to avoid overlapping
   sequences.
* The random() method is implemented in C,
```

```
    executes in a single Python step, and is,
    therefore, threadsafe.
```

This doesn't really tell you everything you need to know about the module; this is simply the description of the *random* module. To get a comprehensive listing of the module, you would have to type *"help(random)"* at the Python interpreter prompt. Doing this will give you 22 pages of formatted text, similar to *nix *man* pages, that will tell you everything you need to know about the module.

Alternatively, if you only want to know the functions a module provides, you can use the **dir()** function, as shown in Listing 12.4.

Listing 12.4: Functions for *random* module

```
>>>dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', '
    Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', 'WichmannHill', '
    _BuiltinMethodType', '_MethodType',
'__all__', '__builtins__', '__doc__', '__file__',
    '__name__', '__package__',
'_acos', '_ceil', '_cos', '_e', '_exp', '_hexlify
    ', '_inst', '_log', '_pi',
'_random', '_sin', '_sqrt', '_test', '
    _test_generator', '_urandom', '_warn',
'betavariate', 'choice', 'division', 'expovariate
    ', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'jumpahead', '
    lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange',
     'sample', 'seed', 'setstate',
'shuffle', 'triangular', 'uniform', '
    vonmisesvariate', 'weibullvariate']
```

Naturally, the only way to harness the power of docstrings is to follow the style rules Python expects, meaning you have to use triple-quotes, separate your summary line from the full-blown description, etc. You can document your Python code without following these rules but then it's up to you to create a help file or whatever. I haven't had any problems with my docstrings yet but only because I slightly

modify how they are formatted (having a space between the docstring and code, putting end quotes on a separate line, etc.) Be careful if you desire to not follow the Style Guide.

Not only will it make your life easier when you finish your project, but it also makes your code easier to read and follow. (Wish the people at my work could learn how to document their code. Even just a few comments explaining what a function does would help. :))

Chapter 13

Making a Program

After having used it for many years now, I've come to find that Python is an extremely capable language, equal in power to C++, Java, et al. If you don't need the "finesse" the major languages provide, I highly recommend learning Python or another dynamic language like Ruby. You'll program faster with fewer errors (like memory management) and can harness the power of a built-in GUI for rapid prototyping of applications. You can also use these languages for quick scripts to speed repetitive tasks. Plus, they are inherently cross-platform so you can easily switch between operating systems or find a larger market for your programs. Heck, Python is used extensively by Google, NASA, and many game publishers, so it can't be all that bad.

One of the biggest complaints people have is the forced use of white space and indentation. But if you think about it, that's considered a "good coding practice"; it makes it easier to follow the flow of the program and reduces the chance of errors. Plus, since brackets aren't required, you don't have to worry about your program not working because you forgot to close a nested *if* statement. After a few days of using Python, you won't even notice, though I imagine you'll notice how "sloppy" other languages look.

Now, on with the show...

13.1 Making Python Do Something

So far I've talked about how Python is structured and how it differs from other languages. Now it's time to make some real programs. To begin with, Python programs are comprised of functions, classes, modules, and packages.

1. Functions are programmer created code blocks that do a specific task.
2. Classes are object-oriented structures that I'll talk about later; suffice to say they are pretty powerful structures that can make programming life easier, though they can be difficult to learn and wield well.
3. Modules are generally considered normal program files, i.e. a file comprised of functions/classes, loops, control statements, etc.
4. Packages are programs made up of many different modules.

In reality, I consider modules and packages to be "programs". It just depends on how many separate files are required to make the program run. Yes, it is possible to have a single, monolithic file that controls the entire program but it's usually better to have different parts in different files. It's actually easier to keep track of what's going on and you can cluster bits of code that have common goals, e.g. have a file that holds library functions, one that handles the GUI, and one that processes data entry.

An important module to know is the Python standard library. There are two versions: Python 2.6 and Python 3.0. The library is a collection of common code blocks that you can call when needed. This means you don't have to "rebuild the wheel" every time you want to do something, such as calculate the tangent of a function. All you have to do is import the portion of the standard library you need, e.g. the math block, and then use it like regular Python code. Knowing what's in the standard library separates the good programmers from the great ones, at least in my book.

That being said, let's make a simple Python program. This program can be made in IDLE (the standard Python programming environment that comes with the Python install), a third-party programming environment (such as SPE, Komodo, Eclipse, BoaConstructor,

etc.), or a simple text editor like Window's Notepad, vim, emacs, BBEdit, etc. (More programs can be found in the appendix on page 168).

Listing 13.1: First example program

```
def square(x):  #define the function; "x" is the
    argument
     return x * x     #pass back to caller the
         square of a number

for y in range(1, 11):  #cycle through a list of
    numbers
     print square(y) #print the square of a number
```

Listing 13.1 is about as simple as it gets. First we define the function called **square()** (the parenthesis indicates that it's a function rather than a statement) and tell it that the argument called "x" will be used for processing. Then we actually define what the function will do; in this case, it will multiply "x" times itself to produce a square. By using the keyword *return*, the square value will be given back to whatever actually called the function (in this case, the *print* statement).

Next we create a *for* loop that prints the squared value of each number as it increases from 1 to 11. This should be fairly easy to follow, especially with the comments off to the side. Realize that many programs you'll see aren't commented this much; quite often, the programs aren't commented at all. I like to think that I have a sufficient amount of documentation in my code (you'll see later) so that it's pretty easy for even new programmers to figure out what's going on.

To run this program, simply save it with a filename, such as "first_program.py". Then, at the command prompt simply type "python first_program.py". The results should look like this:

Listing 13.2: First example program output

```
$python first_program.py     #your command prompt
    by differ from "$"
1
4
9
```

```
16
25
36
49
64
81
100
```

Let's look at another program, this one a little more complex.

Listing 13.3: Second example program and output

```
def adder(*args):    #accept multiple arguments
    sum = args[0]    #create a blank list
    for next in args[1:]:    #iterate through
        arguments
        sum = sum + next     #add arguments
    return sum
>>> adder(2, 3)
5
>>> adder(4, 5, 56)
65
>>> adder("spam", "eggs")
'spameggs'
>>> adder([1,2,3], [4,5,6])
[1, 2, 3, 4, 5, 6]
```

This little program is pretty powerful, as you can see. Essentially, it takes a variable number of arguments and either adds them or concatenates them together, depending on the argument type. These arguments can be anything: numbers, strings, lists, tuples, etc.

A note about the **args* keyword. This is a special feature of Python that allows you to enter undesignated arguments and do things to them (like add them together). The "*" is like a wildcard; it signifies that a variable number of arguments can be given. A similar argument keyword is ***kwargs*. This one is related (it takes an unlimited number of arguments) but the arguments are set off by keywords. This way, you can match variables to the arguments based on the keywords. More information can be seen in Section 13.3 (Default Arguments) below.

13.2 Scope

What? You didn't know snakes got bad breath? (I know, bad joke.) Seriously though, scope describes the area of a program where an identifier (a name for something, like a variable) can access it's associated value. Scope ties in with namespaces because namespaces pretty much define where an identifier's scope is.

In simple terms, namespaces store information about an identifier and it's value. Python has three namespaces: local, global, and built-in. When an identifier is first accessed, Python looks for it's value locally, i.e. it's surrounding code block. In the example above, "x" is defined within the function **square()**. Every function is assigned its own local namespace. Functions can't use identifiers defined in other functions; they're simply not seen. If a function tries to call a variable defined in another function, you'll get an error. If a function tried to define a previously defined variable, you'll just get a brand new variable that happens to have the same name but a different value.

However, if an identifier isn't defined locally, Python will check if it's in the global namespace. The global namespace is different from the local one in that global identifiers can be used by other functions. So if you made global variable cars_in_shop = 2, all functions in the program can access that variable and use it as needed. So you can define a variable in one location and have it used in multiple places without having to make it over and over. However, this isn't recommended because it can lead to security issues or programming problems. For instance, making a variable global means any function can access them. If you start having multiple functions using the same variable, you don't know what is happening to the variable at any given time; there is no guarantee its value will be what you expect it to be when you use it.

This isn't to say that global variables are a strict no-no. They are useful and can make your life easier, when used appropriately. But they can limit the scalability of a program and often lead to unexplained logic errors, so I tend to stay away from them.

The built-in namespace is set aside for Python's built-in functions. (Kinda convenient, huh?) So keywords and standard function calls like **range()** are already defined when the Python interpreter starts up and you can use them "out of the box".

As you may have figured out, namespaces are nested:

```
built-in
    ↪ global
        ↪ local
```

If an identifier isn't found locally, Python will check the global namespace. If it's not there Python will check the built-in namespace. If it still can't find it, it coughs up an error and dies.

One thing to consider (and I touched on slightly) is that you can hide identifiers as you go down the namespace tree. If you have cars_in_shop = 2 defined globally, you can make a function that has the exact same name with a different value, e.g. cars_in_shop = 15. When the function calls this variable, it will use the value of 15 vs. 2 to calculate the result. This is another problem of global variables; they can cause problems if you don't have good variable names since you may forget which variable you're actually using.

13.3 Default Arguments

When you create a function, you can set it up to use default values for it's arguments, just in case the item calling it doesn't have any arguments. For example:

Listing 13.4: Default arguments

```
def perimeter(length = 1, width = 1):
    return length * width
```

If you want to call this particular function, you can supply it with the necessary measurements [perimeter(15, 25)] or you can supply one [perimeter(7)] or you can just use the defaults [perimeter()]. Each argument is matched to the passed in values, in order, so if you're going to do this make sure you know which arguments are going to be matched, i.e. if you supply just one argument, it will replace the first default value but any other values will remain as defaults.

You can also use keyword arguments, which match arguments based on a corresponding keyword. This way, you don't have to worry about the order they are given. So for the "perimeter()" example above, you could simply say "perimeter(width = 12)". This will make the function use 1 for the length and 12 for the width. This is easier

than remembering the order of the arguments; however, it also means more typing for you. If you have a lot of functions with these types of arguments, it can become tedious.

Additionally, once you give a keyword for an argument, you can't go back to not naming them then try to rely on position to indicate the matchup. For example:

Listing 13.5: Default arguments and position

```
def abstract_function(color = "blue", size = 30,
    range = 40, noodle = True):
     pass

#call the function
abstract_function("red", noodle = False, 45, range
    = 50)     #not allowed
```

Trying it call it this way will give you an error. Once you start using keywords, you have to continue for the rest of the argument set.

That's about it for programming with functions. They're pretty simple and the more examples you see, the more they'll make sense. Python is cool since you can mix functions and classes (with methods) in the same module without worrying about errors. This way you aren't constrained to one way of programming; if a short function will work, you don't have to take the time to make a full-blown class with a method to do the same thing.

If you don't want to deal with object-oriented programming, you can stick with functions and have a good time. However, I'll start to cover OOP in later chapters to show you why it's good to know and use. And with Python, it's not as scary as OOP implementation in other languages.

Chapter 14

Exceptions

I've talked about exceptions before but now I will talk about them in depth. Essentially, exceptions are events that modify program's flow, either intentionally or due to errors. They are special events that can occur due to an error, e.g. trying to open a file that doesn't exist, or when the program reaches a marker, such as the completion of a loop. Exceptions, by definition, don't occur very often; hence, they are the "exception to the rule" and a special class has been created for them.

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances. Here are just a few examples:

- Accessing a non−existent dictionary key will raise a KeyError exception.
- Searching a list for a non−existent value will raise a ValueError exception.
- Calling a non−existent method will raise an AttributeError exception.
- Referencing a non−existent variable will raise a NameError exception.
- Mixing datatypes without coercion will raise a TypeError exception.

One use of exceptions is to catch a fault and allow the program to continue working; we have seen this before when we talked about files. This is the most common way to use exceptions. When programming with the Python command line interpreter, you don't need to worry about catching exceptions. Your program is usually short enough to not be hurt too much if an exception occurs. Plus, having the exception occur at the command line is a quick and easy way to tell if your code logic has a problem. However, if the same error occurred in your real program, it will fail and stop working.

Exceptions can be created manually in the code by raising an exception. It operates exactly as a system-caused exceptions, except that the programmer is doing it on purpose. This can be for a number of reasons. One of the benefits of using exceptions is that, by their nature, they don't put any overhead on the code processing. Because exceptions aren't supposed to happen very often, they aren't processed until they occur.

Exceptions can be thought of as a special form of the *if/elif* statements. You can realistically do the same thing with *if* blocks as you can with exceptions. However, as already mentioned, exceptions aren't processed until they occur; *if* blocks are processed all the time. Proper use of exceptions can help the performance of your program. The more infrequent the error might occur, the better off you are to use exceptions; using *if* blocks requires Python to always test extra conditions before continuing. Exceptions also make code management easier: if your programming logic is mixed in with error-handling *if* statements, it can be difficult to read, modify, and debug your program.

Here is a simple program that highlights most of the important features of exception processing. It simply produces the quotient of 2 numbers.

Listing 14.1: Exceptions

```
first_number = raw_input("Enter the first number."
    ) #gets input from keyboard
sec_number = raw_input("Enter the second number.")
try:
    num1 = float(first_number)
    num2 = float(sec_number)
    result = num1/num2
```

```
except ValueError:  #not enough numbers entered
    print "Two numbers are required."
except ZeroDivisionError:  #tried to divide by 0
    print "Zero can't be a denominator."
else:
    print str(num1) + "/" + str(num2) + "=" + str(
        result)
    #alternative format
    #a tuple is required for multiple values
    #printed values have floating numbers with one
        decimal point
    print "%.1f/%.1f=%.1f" % (num1, num2, result)
```

As you can see, you can have several "exception catchers" in the same *try* block. You can also use the *else* statement at the end to denote the logic to perform if all goes well; however, it's not necessary. As stated before, the whole *try* block could also have been written as *if/elif* statements but that would have required Python to process each statement to see if they matched. By using exceptions, the "default" case is assumed to be true until an exception actually occurs. These speeds up processing.

One change you could make to this program is to simply put it all within the *try* block. The **raw_input()** variables (which capture input from the user's keyboard) could be placed within the *try* block, replacing the "num1" and "num2" variables by forcing the user input to a float value, like so:

Listing 14.2: User input with *try* statements

```
try:
    numerator = float(raw_input("Enter the
        numerator."))
    denominator = float(raw_input("Enter the
        denominator."))
```

This way, you reduce the amount of logic that has to be written, processed, and tested. You still have the same exceptions; you're just simplifying the program.

Finally, it's better to include error-checking, such as exceptions, in your code as you program rather than as an afterthought. A special

"category" of programming involves writing test cases to ensure that most possible errors are accounted for in the code, especially as the code changes or new versions are created. By planning ahead and putting exceptions and other error-checking code into your program at the outset, you ensure that problems are caught before they can cause problems. By updating your test cases as your program evolves, you ensure that version upgrades maintain compatibility and a fix doesn't create an error condition.

14.1 Exception Class Hierarchy

Table 14.1 shows the hierarchy of exceptions from the Python Library Reference. When an exception occurs, it starts at the lowest level possible (a child) and travels upward (through the parents), waiting to be caught. This means a couple of things to a programmer:

1. If you don't know what exception may occur, you can always just catch a higher level exception. For example, if you didn't know that ZeroDivisionError from the previous example was a "stand-alone" exception, you could have used the ArithmeticError for the exception and caught that; as the diagram shows, ZeroDivisionError is a child of ArithmeticError, which in turn is a child of StandardError, and so on up the hierarchy.

2. Multiple exceptions can be treated the same way. Following on with the above example, suppose you plan on using the ZeroDivisionError and you want to include the FloatingPointError. If you wanted to have the same action taken for both errors, simply use the parent exception ArithmeticError as the exception to catch. That way, when either a floating point or zero division error occurs, you don't have to have a separate case for each one. Naturally, if you have a need or desire to catch each one separately, perhaps because you want different actions to be taken, then writing exceptions for each case is fine.

Table 14.1: Exception Hierarchy

```
BaseException
+- SystemExit
+- KeyboardInterrupt
+- GeneratorExit
+- Exception
    +- StopIteration
    +- StandardError
    |  +- BufferError
    |  +- ArithmeticError
    |  |  +- FloatingPointError
    |  |  +- OverflowError
    |  |  +- ZeroDivisionError
    |  +- AssertionError
    |  +- AttributeError
    |  +- EnvironmentError
    |  |  +- IOError
    |  |  +- OSError
    |  |  +- WindowsError (Windows)
    |  |  +- VMSError (VMS)
    |  +- EOFError
    |  +- ImportError
    |  +- LookupError
    |  |  +- IndexError
    |  |  +- KeyError
    |  +- MemoryError
    |  +- NameError
    |  |  +- UnboundLocalError
    |  +- ReferenceError
    |  +- RuntimeError
    |  |  +- NotImplementedError
    |  +- SyntaxError
    |  |  +- IndentationError
    |  |  +- TabError
    |  +- SystemError
    |  +- TypeError
    |  +- ValueError
    |  +- UnicodeError
    |  +- UnicodeDecodeError
    |  +- UnicodeEncodeError
    |  +- UnicodeTranslateError
    +- Warnings (various)
```

14.2 User-Defined Exceptions

I won't spend too much time talking about this, but Python does allow for a programmer to create his own exceptions. You probably won't have to do this very often but it's nice to have the option when necessary. However, before making your own exceptions, make sure there isn't one of the built-in exceptions that will work for you. They have been "tested by fire" over the years and not only work effectively, they have been optimized for performance and are bug-free.

Making your own exceptions involves object-oriented programming, which will be covered in the next chapter. To make a custom exception, the programmer determines which base exception to use as the class to inherit from, e.g. making an exception for negative numbers or one for imaginary numbers would probably fall under the ArithmeticError exception class. To make a custom exception, simply inherit the base exception and define what it will do. Listing 14.3 gives an example of creating a custom exception:

Listing 14.3: Defining custom exceptions

```
import math #necessary for square root function

class NegativeNumberError(ArithmeticError):
"""Attempted improper operation on negative number
    ."""
    pass

def squareRoot(number):
"""Computes square root of number. Raises
    NegativeNumberError
    if number is less than 0."""
    if number < 0:
        raise NegativeNumberError, \
        "Square root of negative number not
            permitted"

    return math.sqrt(number)
```

The first line creates the custom exception NegativeNumberError, which inherits from ArithmeticError. Because it inherits all the fea-

tures of the base exception, you don't have to define anything else, hence the pass statement that signifies that no actions are to be performed. Then, to use the new exception, a function is created (**squareRoot()**) that calls NegativeNumberError if the argument value is less than 0, otherwise it gives the square root of the number.

Chapter 15

Object Oriented Programming

15.1 Learning Python Classes

The class is the most basic component of object-oriented programming. Previously, you learned how to use functions to make your program do something. Now will move into the big, scary world of Object-Oriented Programming (OOP).

To be honest, it took me several months to get a handle on objects. When I first learned C and C++, I did great; functions just made sense for me. Having messed around with BASIC in the early '90s, I realized functions were just like subroutines so there wasn't much new to learn. However, when my C++ course started talking about objects, classes, and all the new features of OOP, my grades definitely suffered.

Once you learn OOP, you'll realize that it's actually a pretty powerful tool. Plus many Python libraries and APIs use classes, so you should at least be able to understand what the code is doing.

One thing to note about Python and OOP: it's not mandatory to use objects in your code. As you've already seen, Python can do just fine with functions. Unlike languages such as Java, you aren't tied down to a single way of doing things; you can mix functions and classes as necessary in the same program. This lets you build the code

in a way that works best; maybe you don't need to have a full-blown class with initialization code and methods to just return a calculation. With Python, you can get as technical as you want.

15.2 How Are Classes Better?

Imagine you have a program that calculates the velocity of a car in a two-dimensional plane using functions. If you want to make a new program that calculates the velocity of an airplane in three dimensions, you can use the concepts of your car functions to make the airplane model work, but you'll have to rewrite the many of the functions to make them work for the vertical dimension, especially want to map the object in a 3-D space. You may be lucky and be able to copy and paste some of them, but for the most part you'll have to redo much of the work.

Classes let you define an object once, then reuse it multiple times. You can give it a base function (called a method in OOP parlance) then build upon that method to redefine it as necessary. It also lets you model real-world objects much better than using functions.

For example, you could make a tire class that defines the size of the tire, how much pressure it holds, what it's made of, etc. then make methods to determine how quickly it wears down based on certain conditions. You can then use this tire class as part of a car class, a bicycle class, or whatever. Each use of the tire class (called instances) would use different properties of the base tire object. If the base tire object said it was just made of rubber, perhaps the car class would "enhance" the tire by saying it had steel bands or maybe the bike class would say it has an internal air bladder. This will make more sense later.

15.3 Improving Your Class Standing

Several concepts of classes are important to know.

1. Classes have a definite namespace, just like modules. Trying to call a class method from a different class will give you an error unless you qualify it, e.g. spamClass.eggMethod().

2. Classes support multiple copies. This is because classes have two different objects: class objects and instance objects. Class objects give the default behavior and are used to create instance objects. Instance objects are the objects that actually do the work in your program. You can have as many instance objects of the same class object as you need. Instance objects are normally marked by the keyword *self*, so a class method could be **Car.Brake()** while a specific instance of the **Brake()** method would be marked as **self.Brake()**. (I'll cover this in more depth later).

3. Each instance object has its own namespace but also inherits from the base class object. This means each instance has the same default namespace components as the class object, but additionally each instance can make new namespace objects just for itself.

4. Classes can define objects that respond to the same operations as built-in types. So class objects can be sliced, indexed, concatenated, etc. just like strings, lists, and other standard Python types. This is because everything in Python is actually a class object; we aren't actually doing anything new with classes, we're just learning how to better use the inherent nature of the Python language.

Here's a brief list of Python OOP ideas:

- The *class* statement creates a class object and gives it a name. This creates a new namespace.

- Assignments within the class create class attributes. These attributes are accessed by qualifying the name using dot syntax: ClassName.Attribute.

- Class attributes export the state of an object and its associated behavior. These attributes are shared by all instances of a class.

- Calling a class (just like a function) creates a new instance of the class. This is where the multiple copies part comes in.

- Each instance gets ("inherits") the default class attributes and gets its own namespace. This prevents instance objects from overlapping and confusing the program.
- Using the term *self* identifies a particular instance, allowing for per-instance attributes. This allows items such as variables to be associated with a particular instance.

15.4 So What Does a Class Look Like?

Before we leave this particular tutorial, I'll give you some quick examples to explain what I've talked about so far. Assuming your using the Python interactive interpreter, here's how a simple class would look like.

Listing 15.1: Defining a class

```
>>> class Hero: #define a class object
...     def setName(self, value):    #define class
    methods
...         self.name = value    #self identifies a
     particular instance
...     def display(self):
...         print self.name #print the data for a
    particular instance
```

There are a few things to notice about this example:

1. When the class object is defined, there are no parenthesis at the end; parenthesis are only used for functions and methods. However, see Section 15.5 for a caveat.
2. The first argument in the parentheses for a class method must be *self*. This is used to identify the instance calling the method. The Python interpreter handles the calls internally. All you have to do is make sure *self* is where it's supposed to be so you don't get an error. Even though you must use *self* to identify each instance, Python is smart enough to know which particular instance is being referenced, so having multiple instances at the same time is not a problem. (*self* is similar to *this*, which is used in several other languages like Java).

3. When you are assigning variables, like “self.name”, the variable must be qualified with the "self" title. Again, this is used to identify a particular instance.

So, lets make a few instances to see how this works:

Listing 15.2: Creating class instances

```
>>> x = Hero()
>>> y = Hero()
>>> z = Hero()
```

Here you'll notice that parenthesis make an appearance. This is to signify that these are instance objects created from the Hero class. Each one of these instances has the exact same attributes, derived from the Hero class. (Later on I'll show you how to customize an instance to do more than the base class).

Now, lets add some information.

Listing 15.3: Adding data to instances

```
>>> x.setName("Arthur, King of the Britons")
>>> y.setName("Sir Lancelot, the Brave")
>>> z.setName("Sir Robin, the Not-Quite-So-Brave-
    As-Sir-Lancelot")
```

These call the **setName()** method that sits in the Hero class. However, as you know by now, each one is for a different instance; not only do x, y, and z each have a different value, but the original value in Hero is left untouched.

If you now call the **display()** method for each instance, you should see the name of each hero.

Listing 15.4: Displaying instance data

```
>>> x.display()
Arthur, King of the Britons
>>> y.display()
Sir Lancelot, the Brave
>>> z.display()
Sir Robin, the Not-Quite-So-Brave-As-Sir-Lancelot
```

You can change instance attributes "on the fly" simply by assigning to *self* in methods inside the class object or via explicitly assigning to instance objects.

Listing 15.5: Modifying instances

```
>>> x.name = "Sir Galahad, the Pure"
>>> x.display()
Sir Galahad, the Pure
```

That's probably enough for this lesson. I'll cover the rest of classes in the next chapter but this is hopefully enough to give you an idea of how useful classes and OOP in general can be when programming. The vast majority of languages in current use implement OOP to one extent or another, so learning how to use classes and objects will help you out as you gain knowledge. Thankfully Python implements OOP in a reasonable way, so it's relatively painless to learn in Python rather than something like C++, at least in my experience.

15.5 "New-style" classes

Starting with Python 2.2, a new type of class was developed for Python. This new class provided a way for programmers to create a class derived, directly or indirectly, from a built-in Python type, such as a list, string, etc.

You can make a new class like the above examples, where no parenthesis are used. However, you can also expressly inherit your new class from the *object* class, or you can derive a new class from one of the built-in types. Listing 15.6 shows how this would look. Deriving your custom classes from *object* is a good idea, since Python 3.x only uses the "new-style" and omitting *object* can cause problems. More information can be found at Introduction To New-Style Classes In Python and Python's New-style Classes.

Listing 15.6: New-style classes

```
class NewStyleUserDefinedClass(object):
    pass
class DerivedFromBuiltInType(list):
    pass
```

```
class IndirectlyDerivedFromType(DerivedFromBuiltInType):
    pass
```

15.6 A Note About Style

As mentioned previously, Python has a certain "style-syntax" that is considered the "right" way to write Python programs. PEP 8 is the document that describes all of the "approved" ways of writing. One of these is how to identify classes, functions, methods, etc.

Classes should be written with the first letter capitalized; any additional words in the class name should also be capitalized: class SpamAndEggs(object). Functions and methods should be written in lower-case, with each word separated by underscores: def bunny_vicious_bite(). Constants (variables that don't change value) should be written in all upper case: MAX_VALUE = 22.

There are many other stylistic ideas to be concerned about. I'll admit, I'm not the best about following Python's stylistic conventions but I try to follow them as best I can remember. Even if I don't follow "the Python way", I do try to be consistent within my own programs. My personal suggestion is to read PEP 8 and look at the source code for different Python programs, then pick a style that works best for you. The most important thing is to have a consistent style.

Chapter 16

More OOP

Last chapter I told you some of the basics about using Python classes and object-oriented programming. Time to delve more into classes and see how they make programming life better.

16.1 Inheritance

First off, classes allow you to modify a program without really making changes to it. To elaborate, by subclassing a class, you can change the behavior of the program by simply adding new components to it rather than rewriting the existing components.

As we've seen, an instance of a class inherits the attributes of that class. However, classes can also inherit attributes from other classes. Hence, a subclass inherits from a superclass allowing you to make a generic superclass that is specialized via subclasses. The subclasses can override the logic in a superclass, allowing you to change the behavior of your classes without changing the superclass at all.

Let's make a simple example. First make a class:

Listing 16.1: Defining a superclass

```
>>>class FirstClass:      #define the superclass
... def setdata(self, value):    #define methods
...         self.data = value    #'self' refers to an
    instance
```

```
... def display(self):
...     print self.data
...
```

Then we make a subclass:

Listing 16.2: Defining a subclass

```
>>>class SecondClass(FirstClass):    #inherits from
    FirstClass
... def display(self):  #redefines 'display'
...     print "Current value = '%s'" % self.data
...
```

As you can see, SecondClass "overwrites" the display method. When a FirstClass instance is created, all of its actions will be taken from the methods defined in FirstClass. When a SecondClass instance is created, it will use the inherited **setdata()** method from FirstClass but the display method will be the one from SecondClass.

To make this easier to understand, here are some examples in practice.

Listing 16.3: More inheritance examples

```
>>>x=FirstClass() #instance of FirstClass
>>>y=SecondClass() #instance of SecondClass
>>>x.setdata("The boy called Brian.")
>>>y.setdata(42)
>>>x.display()
The boy called Brian.
>>>y.display()
Current value = '42'
```

Both instances (x and y) use the same **setdata()** method from FirstClass; x uses it because it's an instance of FirstClass while y uses it because SecondClass inherits **setdata()** from FirstClass. However, when the display method is called, x uses the definition from FirstClass but y uses the definition from SecondClass, where display is overridden.

Because changes to program logic can be made via subclasses, the use of classes generally supports code reuse and extension better than

traditional functions do. Functions have to be rewritten to change how they work whereas classes can just be subclassed to redefine methods.

On a final note, you can use multiple inheritance (adding more than one superclass within the parenthesis) if you need a class that belongs to different groups. In theory this is good because it should cut down on extra work. For example, a person could be a chef, a musician, a store owner, and a programmer; the person could inherit the properties from all of those roles. But in reality it can be a real pain to manage the multiple inheritance sets. You have to ask yourself, "Is it really necessary that this class inherit from all of these others?"; often the answer is, "No".

Using multiple inheritance is considered an "advanced technique" and therefore I won't discuss it. Actually, I don't use it; if I encounter a situation where I could use it, I try and rethink the program's structure to avoid using it. It's kind of like normalizing databases; you keep breaking it down until it's as simple as you can get it. If you still need multiple inheritance, then I recommend getting a more advanced Python book.

16.2 Operator Overloads

Operator overloading simply means that objects that you create from classes can respond to actions (operations) that are already defined within Python, such as addition, slicing, printing, etc. Even though these actions can be implemented via class methods, using overloading ties the behavior closer to Python's object model and the object interfaces are more consistent to Python's built-in objects, hence overloading is easier to learn and use.

User-made classes can override nearly all of Python's built-in operation methods. These methods are identified by having two underlines before and after the method name, like this: __add__. These methods are automatically called when Python evaluates operators; if a user class overloads the __add__ method, then when an expression has "+" in it, the user's method will be used instead of Python's built-in method.

Using an example from the Learning Python book, here is how operator overloading would work in practice:

Listing 16.4: Operator overloading example

```
>>>class ThirdClass(SecondClass):    #is-a
    SecondClass
... def __init__(self, value):  #on "ThirdClass(
    value)"
... self.data = value
... def __add__(self, other):   # on "self + other
    "
... return ThirdClass(self.data + other)
... def __mul__(self, other):   #on "self * other"
... self.data = self.data * other
...
>>>a = ThirdClass("abc")     #new __init__ called
>>>a.display() #inherited method
Current value = 'abc'
>>>b = a + "xyz"     #new __add__ called: makes a
    new instance
>>>b.display()
Current value = 'abcxyz'
>>>a*3  #new __mul__ called: changes instance in-
    place
>>>a.display()
Current value = 'abcabcabc'
```

ThirdClass is technically a subclass of SecondClass but it doesn't override any of SecondClass' methods. If you wanted, you could put the methods from ThirdClass in SecondClass and go from there. However, creating a new subclass allows you flexibility in your program.

When a new instance of ThirdClass is made, the __init__ method takes the instance-creation argument and assigns it to self.data. ThirdClass also overrides the "+" and "*" operators; when one of these is encountered in an expression, the instance object on the left of the operator is passed to the self argument and the object on the right is passed to other. These methods are different from the normal way Python deals with "+" and "*" but they only apply to instances of ThirdClass. Instances of other classes still use the built-in Python methods.

One final thing to mention about operator overloading is that you

can make your custom methods do whatever you want. However, common practice is to follow the structure of the built-in methods. That is, if a built-in method creates a new object when called, your overriding method should too. This reduces confusion when other people are using your code. Regarding the example above, the built-in method for resolving "*" expressions creates a new object (just like how the "+" method does), therefore the overriding method we created should probably create a new object too, rather than changing the value in place as it currently does. You're not obligated to "follow the rules" but it does make life easier when things work as expected.

16.3 Class Methods

Instance methods (which is what we've been using so far) and class methods are the two ways to call Python methods. As a matter of fact, instance methods are automatically converted into class methods by Python.

Here's what I'm talking about. Say you have a class:

Listing 16.5: Class methods, part 1

```
class PrintClass:
    def printMethod(self, input):
        print input
```

Now we'll call the class' method using the normal instance method and the "new" class method:

Listing 16.6: Class methods, part 2

```
>>>x = PrintClass()
>>>x.printMethod("Try spam!")    #instance method
Try spam!
>>>PrintClass.printMethod(x, "Buy more spam!")  #
    class method
Buy more spam!
```

So, what is the benefit of using class methods? Well, when using inheritance you can extend, rather than replace, inherited behavior by calling a method via the class rather than the instance.

Here's a generic example:

Listing 16.7: Class methods and inheritance

```
>>>class Super:
...     def method(self):
...         print "now in Super.method"
...
>>>class Subclass(Super):
...     def method(self):   #override method
...         print "starting Subclass.method"    #
    new actions
...         Super.method(self)  #default action
...         print "ending Subclass.method"
...
>>>x = Super()   #make a Super instance
>>>x.method()    #run Super.method
now in Super.method
>>>x = Subclass()    #make a Subclass instance
>>>x.method()    #run Subclass.method which calls
    Super.method
starting Subclass.method
now in Super.method
ending Subclass.method
```

Using class methods this way, you can have a subclass extend the default method actions by having specialized subclass actions yet still call the original default behavior via the superclass. Personally, I haven't used this yet but it is nice to know that it's available if needed.

16.4 Have you seen my class?

There is more to classes than I have covered here but I think I've covered most of the basics. Hopefully you have enough knowledge to use them; the more you work with them the easier they are to figure out. I may have mentioned it before, but it took me almost six months to get my head around using classes. Objects were a new area for me and I couldn't figure out how everything worked. It didn't help that my first exposure to them was Java and C++; my two textbooks just jumped right into using objects and classes without explaining the

how's and why's of them. I hope I did better explaining them than my text books did.

There are several "gotchas" when using classes, such as learning the difference between "is-a" and "has-a" relationships, but most of them are pretty obvious, especially when you get error messages. If you really get stumped, don't be afraid to ask questions. Remember, we were all beginners once and so many of us have encountered the same problem before.

Chapter 17

Databases

Databases are popular for many applications, especially for use with web applications or customer-oriented programs. There is a caveat though; databases don't have the performance that file-system based applications do.

Normal files, such as text files, are easy to create and use; Python has the tools built-in and it doesn't take much to work with files. File systems are more efficient (most of the time) in terms of performance because you don't have the overhead of database queries or other things to worry about. And files are easily portable between operating systems (assuming you aren't using a proprietary format) and are often editable/usable with different programs.

Databases are good when discrete "structures" are to be operated on, e.g. a customer list that has phone numbers, addresses, past orders, etc. A database can store a lump of data and allow the user or developer to pull the necessary information, without regard to how the data is stored. Additionally, databases can be used to retrieve data randomly, rather than sequentially. For pure sequential processing, a standard file is better.

Obviously, there is more to the file-system vs. database battle than what I just covered. But, generally speaking, you will be better suited using a file-system structure than a database unless there is a reason to use a database. My personal recommendation is that, unless you are creating a server-based application, try using a local file rather

than a database. If that doesn't work, then you can try a database.

17.1 How to Use a Database

A database (DB) is simply a collection of data, placed into an arbitrary structured format. The most common DB is a relational database; tables are used to store the data and relationships can be defined between different tables. SQL (Structured Query Language) is the language used to work with most DBs. (SQL can either be pronounced as discrete letters "S-Q-L" or as a word "sequel". I personally use "sequel".)

SQL provides the commands to query a database and retrieve or manipulate information. The format of a query is one of the most powerful forces when working with DBs; an improper query won't return the desired information, or worse, it will return the wrong information. SQL is also used to input information into a DB.

While you can interact directly with a DB using SQL, as a programmer you have the liberty of using Python to control much of the interactions. You will still have to know SQL so you can populate and interact with the DB, but most of the calls to the DB will be with the Python DB-API (database application programming interface).

17.2 Working With a Database

This book is not intended to be a database or SQL primer. However, I will provide you with enough information to create simple database and an application that uses it. First I will cover the basic principles of databases and SQL queries then we will use Python to make and manipulate a small database.

First off, consider a database to be one or more tables, just like a spreadsheet. The vertical columns comprise different fields or categories; they are analogous to the fields you fill out in a form. The horizontal rows are individual records; each row is one complete record entry. Here's a pictorial summary, representing a customer list. The table's name is "Customers_table":

Index	Last_Name	First_Name	Address	City	State
0	Johnson	Jack	123 Easy St.	Anywhere	CA
1	Smith	John	312 Hard St.	Somewhere	NY

The only column that needs special explanation is the Index field. This field isn't required but is highly recommended. You can name it anything you want but the purpose is the same. It is a field that provides a unique value to every record; it's often called the primary key field. The primary key is a special object for most databases; simply identifying which field is the primary key will automatically increment that field as new entries are made, thereby ensuring a unique data object for easy identification. The other fields are simply created based on the information that you want to include in the database.

To make a true relational database, you have one table that refers to one or more tables in some fashion. If I wanted to make a order-entry database, I could make another table that tracks an order and relate that order to the above customer list, like so:

Key	Item_title	Price	Order_Number	Customer_ID
0	Boots	55.50	4455	0
1	Shirt	16.00	4455	0
2	Pants	33.00	7690	0
3	Shoes	23.99	3490	1
4	Shoes	65.00	5512	1

This table is called "Orders_table". This table shows the various orders made by each person in the customer table. Each entry has a unique key and is related to Customers_table by the Customer_ID field, which is the Index value for each customer.

17.3 Using SQL to Query a Database

To query a table using SQL, you simply tell the database what it is your are trying to do. If you want to get a list of the customers or a list of orders in the system, just select what parts of the table you want to get. (Note: the following code snippets are not Python specific; additionally, SQL statements are not case-sensitive but are usually written in uppercase for clarity.)

Listing 17.1: Returning data with SQL

```
SELECT * FROM Customers_table
```

The result should pretty look just like the table above; the command simply pulls everything from Customers_table and prints it. The printed results may be textual or have grid lines, depending on the environment you are using but the information will all be there.

You can also limit the selection to specific fields, such as:

Listing 17.2: Limiting results with SQL

```
SELECT Last_name, First_name FROM Customers_table
SELECT Address FROM Customers_table WHERE State == "NY"
```

The second SQL query above uses the "WHERE" statement, which returns a limited set of information based on the condition specified. If you used the statement as written, you should only get back the addresses of customers who live in New York state. Obviously this is a good idea because it limits the results you have to process and it reduces the amount of memory being used. Many system slowdowns can be traced to bad DB queries that return too much information and consume too many resources.

To combine the information from two tables, i.e. to harness the power of relational databases, you have to join the tables in the query.

Listing 17.3: Joining database tables

```
SELECT Last_name, First_name, Order_Number FROM
    Customers_table, Orders_table WHERE
    Customers_table.Index = Orders_table.
    Customer_ID
```

This should give you something that looks like this:

Listing 17.4: SQL query results

```
Johnson Jack 4455
Johnson Jack 4455
Johnson Jack 7690
Smith John 3490
Smith John 5512
```

Again, the formatting may be different depending on the system you are working with but it's the information that counts.

17.4 Python and SQLite

Starting with v2.5, Python has included SQLite, a light-weight SQL library. SQLite is written in C, so it's quick. It also creates the database in a single file, which makes implementing a DB fairly simple; you don't have to worry about all the issues of having a DB spread across a server. However, it does mean that SQLite is better suited to either development purposes or small, stand-alone applications. If you are planning on using your Python program for large-scale systems, you'll want to move to a more robust database, such as PostgreSQL or MySQL.

However, this doesn't mean SQLite isn't useful. It's good for prototyping your application before you throw in a full-blown DB; that way you know your program works and any problems are most likely with the DB implementation. It's also good for small programs that don't need a complete DB package with its associated overhead.

So, how do you use SQLite with Python? I'll show you.

17.5 Creating an SQLite DB

Because SQLite is built into Python, you simply import it like any other library. Once imported, you have to make a connection to it; this creates the database file. A cursor is the object within SQLite that performs most of the functions you will be doing with the DB.

Listing 17.5: Creating a SQLite database

```
import sqlite3  #SQLite v3 is the version
    currently included with Python
connection = sqlite3.connect("Hand_tools.db")    #
    The .db extension is optional
cursor = connection.cursor()

#Alternative DB created only in memory
#mem_conn = sqlite3.connect(":memory:")
#cursor = mem_conn.cursor()

cursor.execute("""CREATE TABLE Tools
```

```
    (id INTEGER PRIMARY KEY,
    name TEXT,
    size TEXT,
    price INTEGER)""")

for item in (
    (None, "Knife", "Small", 15),    #The end comma
        is required to separate tuple items
    (None, "Machete", "Medium", 35),
    (None, "Axe", "Large", 55),
    (None, "Hatchet", "Small", 25),
    (None, "Hammer", "Small", 25)
    (None, "Screwdriver", "Small", 10),
    (None, "Prybar", "Large", 60),
    ):cursor.execute("INSERT INTO Tools VALUES (?,
        ?, ?, ?)", item)

connection.commit()      #Write data to database
cursor.close()  #Close database
```

The above code makes a simple, single-table database of a collection of hand tools. Notice the question marks used to insert items into the table. The question marks are used to prevent a SQL injection attack, where a SQL command is passed to the DB as a legitimate value. The DB program will process the command as a normal, legitimate command which could delete data, change data, or otherwise compromise your DB. The question marks act as a substitution value to prevent this from occurring.

You'll also note the ability to create a DB in memory. This is good for testing, when you don't want to take the time to write to disc or worry about directories. If you have enough memory, you can also create the DB completely in memory for your final product; however, if you lose power or otherwise have memory problems, you lose the complete DB. I only use a RAM DB when I'm testing the initial implementation to make sure I have the syntax and format correct. Once I verify it works the way I want, then I change it to create a disc-based DB.

17.6 Pulling Data from a DB

To retrieve the data from an SQLite DB, you just use the SQL commands that tell the DB what information you want and how you want it formatted.

Listing 17.6: Retrieving data from SQLite

```
cursor.execute("SELECT name, size, price FROM Tools")
toolsTuple = cursor.fetchall()
for tuple in toolsTuple:
    name, size, price = tuple    #unpack the tuples
    item = ("%s, %s, %d" % (name, size, price))
    print item
```

Which returns the following list:

Listing 17.7: Returned data

```
Knife, Small, 15
Machete, Medium, 35
Axe, Large, 55
Hatchet, Small, 25
Hammer, Small, 25
Screwdriver, Small, 10
Prybar, Large, 60
Knife, Small, 15
Machete, Medium, 35
Axe, Large, 55
Hatchet, Small, 25
Hammer, Small, 25
Screwdriver, Small, 10
Prybar, Large, 60
```

Alternatively, if you want to print out pretty tables, you can do something like this:

Listing 17.8: "Pretty printing" returned data

```
cursor.execute("SELECT * FROM Tools")
for row in cursor:
    print "-" * 10
```

```
    print "ID:", row[0]
    print "Name:", row[1]
    print "Size:", row[2]
    print "Price:", row[3]
    print "-" * 10
```

Which gives you this:

Listing 17.9: Output of "pretty printed" data

```
----------
ID: 1
Name: Knife
Size: Small
Price: 15
----------
----------
ID: 2
Name: Machete
Size: Medium
Price: 35
----------
----------
ID: 3
Name: Axe
Size: Large
Price: 55
----------
----------
ID: 4
Name: Hatchet
Size: Small
Price: 25
----------
----------
ID: 5
Name: Hammer
Size: Small
Price: 25
----------
```

```
----------
ID: 6
Name: Screwdriver
Size: Small
Price: 10
----------
----------
ID: 7
Name: Prybar
Size: Large
Price: 60
----------
----------
ID: 8
Name: Knife
Size: Small
Price: 15
----------
----------
ID: 9
Name: Machete
Size: Medium
Price: 35
----------
----------
ID: 10
Name: Axe
Size: Large
Price: 55
----------
----------
ID: 11
Name: Hatchet
Size: Small
Price: 25
----------
----------
ID: 12
```

```
Name: Hammer
Size: Small
Price: 25
----------
----------
ID: 13
Name: Screwdriver
Size: Small
Price: 10
----------
----------
ID: 14
Name: Prybar
Size: Large
Price: 60
----------
```

Obviously, you can mess around with the formatting to present the information as you desire, such as giving columns with headers, including or removing certain fields, etc.

17.7 SQLite Database Files

SQLite will try to recreate the database file every time you run the program. If the DB file already exists, you will get an "OperationalError" exception stating that the file already exists. The easiest way to deal with this is to simply catch the exception and ignore it.

Listing 17.10: Dealing with existing databases

```
try:
    cursor.execute("CREATE TABLE Foo (id INTEGER
        PRIMARY KEY, name TEXT)")
except sqlite3.OperationalError:
    pass
```

This will allow you to run your database program multiple times (such as during creation or testing) without having to delete the DB file after every run.

You can also use a similar try/except block when testing to see if the DB file already exists; if the file doesn't exist, then you can call the DB creation module. This allows you to put the DB creation code in a separate module from your "core" program, calling it only when needed.

Chapter 18

Distributing Your Program

This will be a short chapter because distributing your Python program to others is generally pretty easy. The main way for users to get your program is by getting the raw .py files and saving them to a storage location. Assuming the user has Python installed on his system, all he has to do is call your Python program like normal, e.g. python foo.py. This is the same method you have been using in this book and it is the easiest way to deal with Python files.

Another way of distributing your source code is providing the byte-code compiled versions; the files that end with .pyc that are created after you first run your program. These compiled versions are not the programmer-readable files of your program; they are the machine-ready files that are actually used by the computer. If you want a modicum of security with your program but don't want the whole hassle of dealing with obfuscation or other ways of hiding your work, you may want to use .pyc files.

If you want to distribute your program like a "normal" Python application, i.e. invoking it with the setup.py command, you can use Python's **distutils** module. I won't go into specifics on this utility; suffice to say that many installable Python programs use this same method to package themselves. It helps to keep everything together

and reduce the number of individual files a user needs to keep track of when running your program. More information about distutils can be found at the Python distutils documentation.

Alternatively, you can package your program as a Python egg. Eggs are analogous to the JAR files in Java; they are simply a way of bundling information with a Python project, enabling run-time checking of program dependencies and allowing a program to provide plugins for other projects. A good example of a Python program that uses eggs is Django, the web framework project. The nice thing about Python eggs is that you can use the "Easy Install" package manager, which takes care of finding, downloading, and installing the egg files for you.

For people who don't have (or want) Python installed on their systems, you can compile your program to a binary executable, e.g. a .exe file. For Unix and Linux systems, you can use the Freeze utility that is included with the Python language. Note that, for the most part, this is really unnecessary because Python is nearly always a default installed application on *nix operating systems and, generally speaking, people who are using these systems can probably handle working with Python files. But, the capability is there if you want to do it.

For Windows, which doesn't have Python installed by default, you are more likely to want to create a standard executable file (though you can still offer the raw .py files for the more adventurous or power users). The utility for this is called **py2exe**. py2exe is a utility that converts Python scripts to normal executable Windows programs. Again, Python isn't required to be installed on the computer so this is one of the most popular ways to create Python programs for Windows. However, the file size can be considerably larger for a converted program vs. the raw .py files.

Finally, for Mac users, you can use the **py2app** utility. It functions just like py2exe, creating a compiled version of your Python program for Mac users. OS X, the current operating system for Macs, includes Python with the OS install so technically you could just distribute the source code. But most Mac users don't know how to use Python so creating a "normal" program for them is probably preferable.

Chapter 19

Python 3

As mentioned at the beginning of this book, the newest version of Python is version 3.2, as of this writing. Python 2.7 is the final version for Python 2.x; no new releases for Python 2 will be released.

Python 3.x is the future of the language. It is not fully backwards-compatible with Python 2.x, breaking several features in order clean up the language, make some things easier, and in general improve the consistency of the language. (More information can be found at the Python web site).

As mentioned previously, most *nix distributions and Mac OS X include Python 2.x by default. The vast majority of information on the Internet is geared towards Python 2.x, including packages and libraries. So, unless you are sure that all users of your programs are using Python 3.x, you may want to learn 2.x.

This is a short summary of the major changes between the Python 2.x versions and Python 3.2, the latest version. Not all changes to the language are mentioned here; more comprehensive information can be found at the official Python web site or the "What's New in Python 3.2" page. Some of the information here has already been talked about previously in this book; it is mentioned here again for easier reference.

- The *print* statement has been replaced with a *print()* function, with keyword arguments to replace most of the special syntax of the old *print* statement.

- Certain APIs don't use lists as return types but give views or iterators instead. For example, the dictionary methods *dict.keys()*, *dict.items()*, and *dict.values()* return views instead of lists.
- Integers have only one type now ("long" integers for those who know them in other languages)
- The default value for division (e.g. $1/2$) is to return a float value. To have truncating behavior, i.e. return just the whole number, you must use a double forward slash: $1//2$.
- All text (a.k.a. "strings") is Unicode; 8-bit strings are no longer.
- "!=" is the only way to state "not equal to"; the old way ("<>") has been removed.
- Calling modules from within functions is essentially gone. The statement "from module import *" can only be used at the module level, not in functions.
- String formatting has been changed, removing the need for the "%" formatting operator.
- The exception APIs have been changed to clean them up and add new, powerful features.
- *raw_input()* has been changed to just *input()*.
- The C language API has been modified, such as removing support for several operating systems.

If you need to port Python 2.x code to version 3.0, there is a simple utility called "2to3" that will automatically correct and modify the legacy code to the new version. Generally speaking, you simply call the 2to3 program and let it run; it is robust enough to handle the vast majority of code changes necessary. However, the library is flexible enough to allow you to write your own "fixer" code if something doesn't work correctly.

Part II

Graphical User Interfaces

Chapter 20

Overview of Graphical User Interfaces

20.1 Introduction

Graphical user interfaces (GUIs) are very popular for computers nowadays. Rarely will you find a program that doesn't have some sort of graphical interface to use it. Most non-graphical programs will be found in *nix operating systems; even then, those programs are usually older ones where the programmer didn't want (or see the need for) a graphical interface.

Unless you are a power-user or feel very comfortable using command-line tools, you will interact with your computer via a graphical interface. A graphical interface entails the program creating what is commonly called a "window". This window is populated by "widgets", the basic building blocks of a GUI. Widgets are the pieces of a GUI that make it usable, e.g. the close button, menu items, sliders, scrollbars, etc. Basically, anything the user can see is a widget.

Because most people are used to using windows to manipulate their computer, it's worthwhile to know how to create a GUI. Though it's easier and often faster to make a command-line program, very few people will use it. Humans are visual creatures and people like "flashy" things. Hence, putting a nice graphical interface on your program

makes it easier for people to use, even if it's faster/easier to use the command line.

There are many GUI development tools available. Some of the most popular are Qt, GTK, wxWidgets, and .NET. All but MFC are cross-platform, meaning you can develop your program in any operating system and know that it will work in any other operating system. However, some programs created in Microsoft's Visual Studio will only run on Microsoft Windows.

20.2 Popular GUI Frameworks

Though this tutorial will focus on wxPython, a GUI library based on wxWidgets, I will give a brief rundown of several other graphical libraries. As a disclaimer, the only GUI libraries I am familiar with are wxPython and Tkinter; some information below may be incorrect or outdated but is correct to the best of my knowledge.

- Qt- Based upon technology from Trolltech (and subsequently purchased by Nokia), Qt is one of the most popular libraries for cross-platform development. It is the main graphical library for the KDE desktop. Qt has several licenses available, with the main ones being for commercial and open-source use. If your application will be commercial, i.e. closed-source, you have to pay for a license to use it. If your application will be open-sourced, then you can use the GPL license; however, you will be unable to commercialize your program without "upgrading" the license. Qt includes a graphical layout utility, allowing the user to drag & drop graphical widgets for quick design of the interface. Alternatively, the developer can hand-code the interface layout. Qt also support non-GUI features, including SQL database access, XML parsing, network support, et al. Qt is written in C++ but a Python binding is available via PyQt.

- GTK+- Originally created for development of the GIMP image software (GTK stands for GIMP Tool Kit), GTK+ evolved into the graphical library that powers the GNOME desktop. Many of the applications written for KDE will run on GNOME and vice versa. GTK+ is open-sourced under the GPL license. This

was intentional by the creators. When Qt was chosen to power the KDE desktop, Trolltech had a proprietary license for it. Because this goes against the Free Software philosophy, GTK was created to allow people to use the GNOME desktop and GTK applications as open-source software. GTK doesn't necessarily have a built-in drag & drop interface, though the wxGlade utility provides this functionality. The original GTK was written in C while GTK+ is written in C++, using OOP practices. PyGTK is the Python binding for GTK+.

- wxPython- wxPython is a Python-implementation of wxWidgets, meaning Python programs interact with Python wrappers of the underlying C/C++ code. wxWidgets is a general purpose GUI library originally developed for use with C/C++ programs. wxPython is Free/Open Source Software (FOSS), licensed under the GPL. Programs written with wxPython use the native widgets of an operating system, so the programs don't look out of place.

- Tkinter- A graphical library created for the Tcl/Tk language. Tkinter is the default GUI library for Python development due to it's inclusion in the core Python language. Tkinter programs don't necessarily "blend in" with other applications, depending on operating system. Some of the widgets can look like older versions of Windows, which some developers feel is an issue. The Tkinter toolset is fairly limited but easy to learn, making it easy to make simple GUI programs in Python.

- MFC/.NET- These two libraries were created by Microsoft for Windows development. MFC is the GUI library used with C++ programs while .NET is used with .NET languages, such as C# and VB.NET. Unlike the previous libraries mentioned, MFC is not cross-platform. Some open-source projects, like Mono, have been started to use the .NET framework on non-Windows computers but some additional effort is required to ensure true cross-platform compatibility. MFC is considered a legacy framework; .NET is the new way of creating Windows programs. IronPython is a .NET enabled version of Python, for those interested in using Python with Visual Studio. The Visual Studio development

suites include the GUI libraries required for widget layout.

20.3 Before You Start

For purposes of this book, I will be talking about wxPython. Though it requires you to download additional software (unlike Tkinter which is included with Python), it has more widgets available to you (making it more versatile) and it allows you to use graphical layout tools, like wxGlade, to design the GUI. Otherwise you have to manually place each item in the code, run the program to see how it looks, and go back into your code to make changes. Graphical layout tools let you place widgets wherever you want; when done, you "run" the program which creates the source code template that auto-populates all the widgets. Then you only have to write the underlying logic that make your program work.

Before you start creating a GUI with wxPython, you need to know how to program in Python. This may seem pretty obvious but some people may expect wxPython is self-contained; no additional knowledge required. If you don't already have Python installed on your computer, you will have to download and install it from the Python web site; this is most common for Windows users. Linux and Mac users already have Python installed.

Additionally, you should have made a few command-line programs in Python so you know how to handle user input and terminal output. My personal suggestion for learning wxPython is to take a command-line program and turn it into a graphical program. This way, you already know how the program works and what the necessary input and expected output should be. All you need to do is create the graphical elements.

I will make a plug for my preference of Python development environments: SPE. SPE (Stani's Python Editor) is an integrated development environment that provides code completion (reduces typing), integrated Python shell (for testing bits of code before you put them into your program), Python calltips (displays expected arguments for the method/function), and various helper tools like notes and todo lists.

Additionally, SPE includes wxPython, which is nice because it was

created with wxPython. It also includes wxGlade, so you can make quick GUIs without having to install wxGlade separately.

Chapter 21

A Simple Graphical Dice Roller

I'm going to show a simple wxPython program to highlight the various portions you should be familiar with. But first, I'm giving you a program that I wrote a few years ago so you can see how a command-line program can be easily converted to a GUI program.

Listing 21.1 is a program I created that turns the command-line dice roller in Appendix E.1 into a GUI. You'll notice that there are comments in the program indicating that the program was generated by wxGlade. You want to make sure any additions or changes to the wxGlade-generated code is outside of the begin/end comments. Whenever you make changes to your program using wxGlade, the code within those sections is generated by wxGlade; any code you put in there will be overwritten.

Note: the following program isn't fully functional. It displays the required information but only the 1d6 button works. The reader is encouraged to revise the code to make the other buttons to work.

Listing 21.1: Graphical Dice Rolling Program

```
1 #!/usr/bin/env python
2 # -*- coding: iso-8859-15 -*-
3 # generated by wxGlade 0.6.2 on Fri Aug 29
      09:24:23 2008
```

```
4
5  import wx
6  from dice_roller import multiDie
7  # begin wxGlade: extracode
8  # end wxGlade
9
10 class MyFrame ( wx.Frame ) :
11         def __init__ ( self, *args, **kwds ) :
12         # begin wxGlade: MyFrame.__init__
13                 kwds["style"] = wx.
                       DEFAULT_FRAME_STYLE
14                 wx.Frame.__init__ ( self, *args,
                       **kwds )
15                 self.panel_1 = wx.Panel ( self, -1
                       )
16                 self.label_1 = wx.StaticText (
                       self.panel_1, -1, "Dice␣Roll␣
                       Simulator" )
17                 self.text_ctrl_1 = wx.TextCtrl (
                       self.panel_1, -1, "" )
18                 self.button_1 = wx.Button ( self.
                       panel_1, -1, "1d6" )
19                 self.button_2 = wx.Button ( self.
                       panel_1, -1, "1d10" )
20                 self.button_3 = wx.Button ( self.
                       panel_1, -1, "2d6" )
21                 self.button_4 = wx.Button ( self.
                       panel_1, -1, "2d10" )
22                 self.button_5 = wx.Button ( self.
                       panel_1, -1, "3d6" )
23                 self.button_6 = wx.Button ( self.
                       panel_1, -1, "d100" )
24
25                 self.__set_properties ( )
26                 self.__do_layout ( )
27
28                 self.Bind ( wx.EVT_BUTTON, self.
                       pressed1d6, self.button_1 )
```

```
29          # end wxGlade
30
31          def __set_properties ( self ) :
32          # begin wxGlade: MyFrame.__set_properties
33                  self.SetTitle ( "frame_1" )
34          # end wxGlade
35
36          def __do_layout ( self ) :
37          # begin wxGlade: MyFrame.__do_layout
38                  sizer_1 = wx.BoxSizer ( wx.
                        VERTICAL )
39                  grid_sizer_1 = wx.GridSizer ( 4,
                        2, 0, 0 )
40                  grid_sizer_1.Add ( self.label_1,
                        0, 0, 0 )
41                  grid_sizer_1.Add ( self.
                        text_ctrl_1, 0, 0, 0 )
42                  grid_sizer_1.Add ( self.button_1,
                        0, 0, 0 )
43                  grid_sizer_1.Add ( self.button_2,
                        0, 0, 0 )
44                  grid_sizer_1.Add ( self.button_3,
                        0, 0, 0 )
45                  grid_sizer_1.Add ( self.button_4,
                        0, 0, 0 )
46                  grid_sizer_1.Add ( self.button_5,
                        0, 0, 0 )
47                  grid_sizer_1.Add ( self.button_6,
                        0, 0, 0 )
48                  self.panel_1.SetSizer (
                        grid_sizer_1 )
49                  sizer_1.Add ( self.panel_1, 1, wx.
                        EXPAND, 0 )
50                  self.SetSizer ( sizer_1 )  sizer_1
                        .Fit ( self )
51                  self.Layout ( )
52          # end wxGlade
53
```

```
54          def pressed1d6 ( self, event ) :
55                  """Roll one 6-sided die."""
56
57                  self.text_ctrl_1.SetValue ( "" )
                            #clears any value in text
                        box
58                  val = str ( multiDie ( 1, 1 ) )
59                  self.text_ctrl_1.SetValue ( val )
60
61  # end of class MyFrame
62
63  if __name__ == "__main__":
64          app = wx.PySimpleApp ( 0 )
65          wx.InitAllImageHandlers ( )
66          frame_1 = MyFrame ( None, -1, "" )
67          app.SetTopWindow ( frame_1 )
68          frame_1.Show ( )
69          app.MainLoop ( )
```

Now we will walk through the wxDiceRoller program, discussing the various sections. Line 1 is the normal "she-bang" line added to Python programs for use in Unix-like operating systems. It simply points towards the location of the Python interpreter in the computer system.

Lines 2, 3, 6, and 7 are auto-generated by wxGlade. Lines 6 and 7 delineate any additional code you want your program to use; normally you won't have anything here.

Lines 4 and 5 import necessary modules for the program. Obviously, *wx* is the library for wxPython; *dice_roller* is the program I wrote (found in Appendix E.1). Line 5 imports just the **multiDie()** function from the dice_roller program, rather than the entire program.

Line 10 creates the class that will create the GUI. wxGlade allows you specify the name for your program's objects; the default for the class is MyFrame. The more complex your program, the more classes you will want to have in your program. However, wxGlade creates one class (normally) and simply populates it with the elements you add. This particular class is a child of the wx.Frame class, one of the most common arrangements you will use.

Line 11 initializes the class, defining initial values and creating the

widgets that are seen by the user. It's a standard Python initialization statement, accepting various arguments and keywords so you can pass in whatever information you need. The arguments for this program will be discussed later.

Line 12 is another default comment generated by wxGlade, simply telling you where the auto-generated code starts so you don't "step" on it.

Line 13 is one of the keywords passed into the class, in this case causing the frame to be created in the default style, which adds several standard widgets such as minimize, maximize, and close buttons.

Line 14 is simply the initialization statement for the wx.Frame class. What you have essentially done is make an instance of wx.Frame by creating the MyFrame class. However, before MyFrame can be created/used, an instance of wx.Frame has to be created first.

Line 15 creates a panel within the frame. A panel is the most common item for placing widgets in. You can add widgets to the frame directly but using a panel adds certain inherent features, such as allowing the Tab key to cycle through fields and buttons.

Lines 16-23 simply add widgets to the panel. These particular widgets simply create the dice rolling form by creating a label, an output field, and the "dice" buttons.

Lines 24 & 25 simply call their respective methods, which are explained below.

Line 28 binds the clicking of the 1d6 button to the event that calculates and returns the "die roll".

Line 29 indicates the end of the auto-generated wxGlade code.

Lines 31-34 are the **set_properties()** method. This method sets the properties of your program, in this case the title of the window that is created upon running the program.

Lines 36-52 are the **do_layout()** method. This method actually places the various widgets within the window upon creation.

Line 38 creates a sizer, an object that holds other objects and can automatically resize itself as necessary. When using wxGlade, this sizer is automatically created when creating your frame.

Line 39 is a different type of sizer, this one making a grid. The buttons and other widgets are added to the grid in a sequential fashion, starting in the top left cell. This is good to know when you are hand-coding a grid or trying to auto-populate a grid from a list. Lines 40-47

simply add the various widgets to the grid's cells.

Lines 48 & 49 add the sizers to their respective containers, in this case the BoxSizer (Line 48) and the panel (Line 49).

Line 50 calls the Fit method, which tells the object (sizer_1) to resize itself to match the minimum size it thinks it needs.

Line 51 lays out and displays the widgets when the window is created.

Lines 54-59 comprise the method that calculates the die roll and returns that value to the output field in the window.

Line 61 is the end of the "core" logic, i.e. the part that creates the GUI and calculates the results when a button is pushed.

Lines 63-69 come from standard Python programming. This block tests whether the program is being imported into another program or is being run by itself. This gives the developer a chance to modify the program's behavior based on how it is being executed. In this case, if the program is being called directly, the code is configured to create an instance of the MyFrame class and run it. The MainLoop() method, when invoked, waits for user input and responds appropriately.

This "initial" program is quite long, longer than you would normally expect for an introductory program. Most other tutorials or introductory books would start out with a much smaller program (not more than 10-20 lines). I decided to use this program because the actual logic flow is quite simple; most of the code is taken up by widget creation and placement. It's not only a functional and reasonably useful program, it shows a relatively wide range of wxPython code.

You can compare this program to the command-line program in the Appendix to see what changes are made for a graphical version. You can see that much of the program is taken up with the "fluff" of widgets. Obviously, if a program isn't expected to be used often by regular users, there is little need to make a GUI for it. You'll probably spend more time getting the widgets placed "just so" than you will designing the logic to make it work. Graphical tools like wxGlade can make it easier but it still takes time.

Chapter 22

What Can wxPython Do?

wxPython is a stable, mature graphical library. As such, it has widgets for nearly everything you plan on creating. Generally speaking, if you can't do it with wxPython, you'll probably have to create a custom GUI, such as used in video games.

I won't cover everything wxPython can do for you; looking through the demonstration code that comes with wxPython will show you all the current widgets included in the toolkit. The demonstration also shows the source code in an interactive environment; you can test different ideas within the demonstration code and see what happens to the resulting GUI.

wxPython has several standard, built-in frames and dialogs. Frames include a multiple document interface (having files of the same type contained within the parent window, rather than separate windows) and a wizard class for making simple user walk-throughs.

Included dialogs range from simple "About" boxes and file selections to color pickers and print dialogs. Simple modifications to the source code makes them plug & play-ready for your application.

The main group of objects you will be using are the core widgets and controls. These are what you will use to build your GUI. This category includes things like buttons, check boxes, radio buttons, list boxes, menus, labels, and text boxes. As before, the wxPython demo shows how to use these items.

There are also many different tools shown in the wxPython demo.

Most of them you will probably never use, but it's nice to know wxPython includes them and there is some source code for you to work with.

One thing I've noticed, however, is that the sample demonstrations don't always show how to best to use the widgets. For example, the wizard demo certainly displays a simple wizard with previous/next buttons. But it doesn't have any functionality, such as accepting input from the user for file names or dynamically changing the data displayed. This makes it extremely difficult to make your applications work well if you are coding off the beaten path, such as writing your program without the help of wxGlade or incorporating many different widgets into a program.

If you really want to learn wxPython, you pretty much have to either keep messing with the sample programs to figure out how they work and how to modify them to your needs, or look for a book about wxPython. Unfortunately, there are very few books on the subject and they can be hard to find. The Amazon website is probably your best bet. Alternatively, you can move to Qt, which has very extensive documention since it is marketed towards commercial developers.

Appendix A

String Methods

This list is an abbreviated version of the Python Language Library's section of string methods. It lists the most common string methods you'll probably be using.

- str.**capitalize()**
  - Return a copy of the string with only its first character capitalized.
- str.**center(**width[, fillchar]**)**
  - Return centered in a string of length width. Padding is done using the specified fillchar (default is a space).
- str.**count(**sub[, start[, end]]**)**
  - Return the number of non-overlapping occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.
- str.**endswith(**suffix[, start[, end]]**)**
  - Return True if the string ends with the specified suffix, otherwise return False. suffix can also be a tuple of suffixes to look for. With optional start, test beginning at that position. With optional end, stop comparing at that position.

- str.**expandtabs**([tabsize])
  - Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. If tabsize is not given, a tab size of 8 characters is assumed. This doesn't understand other non-printing characters or escape sequences.
- str.**find**(sub[, start[, end]])
  - Return the lowest index in the string where substring sub is found, such that sub is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found. An alternative method is **index()**, which uses the same parameters but raises ValueError when the substring isn't found.
- str.**format**(format_string, *args, **kwargs)
  - Perform a string formatting operation. The format_string argument can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of format_string where each replacement field is replaced with the string value of the corresponding argument. >>> "The sum of 1 + 2 is {0}".format(1+2) 'The sum of 1 + 2 is 3'
  - See Format String Syntax for a description of the various formatting options that can be specified in format strings.
  - This method of string formatting is the new standard in Python 3.x, and should be preferred to the % formatting described in the text. However, if you are using a version of Python before 2.6, you will have to use the % method.
- str.**isalnum**()

  - Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

- str.**isalpha()**

  - Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

- str.**isdigit()**

  - Return true if all characters in the string are digits and there is at least one character, false otherwise.

- str.**islower()**

  - Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

- str.**isspace()**

  - Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

- str.**isupper()**

  - Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

- str.**join(**seq**)**

  - Return a string which is the concatenation of the strings in the sequence seq. The separator between elements is the string providing this method.

- str.**ljust(**width[, fillchar]**)**

  - Return the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s). There is also a right justify method [**rjust()**] with the same parameters.

- str.**lower()**
  - Return a copy of the string converted to lowercase.
- str.**lstrip(**[chars]**)**
  - Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped. Naturally there is an opposite method [**rstrip()**] that removes the trailing characters.
- str.**replace(**old, new[, count]**)**
  - Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced. str.rfind(sub[, start[, end]]) Return the highest index in the string where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure. str.rindex(sub[, start[, end]]) Like rfind() but raises ValueError when the substring sub is not found.
- str.**split(**[sep[, maxsplit]]**)**
  - Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified, then there is no limit on the number of splits (all possible splits are made).
  - If sep is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']). The sep argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns ['].

  - If sep is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns [].
  - For example, ' 1 2 3 '.split() returns ['1', '2', '3'], and ' 1 2 3 '.split(None, 1) returns ['1', '2 3 ']. str.splitlines([keepends]) 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.

- str.**startswith(**prefix[, start[, end]]**)**
  - Return True if string starts with the prefix, otherwise return False. prefix can also be a tuple of prefixes to look for. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

- str.**strip(**[chars]**)**
  - Return a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped.

- str.**swapcase()**
  - Return a copy of the string with uppercase characters converted to lowercase and vice versa.

- str.**title()**
  - Return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase. There is also a method to determine if a string is a title [**istitle()**].

- str.**upper()**
  - Return a copy of the string converted to uppercase.

Appendix B

List Methods

Even though this is a complete list of methods, more information about Python lists can be found at the Python web site's discussion of data structures.

- list.**append**(x)
  - Add an item to the end of the list; equivalent to a[len(a):] = [x].
- list.**extend**(L)
  - Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.
- list.**insert**(i, x)
  - Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
- list.**remove**(x)
  - Remove the first item from the list whose value is x. It is an error if there is no such item.

- list.**pop**([i])
  - Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
- list.**index**(x)
  - Return the index in the list of the first item whose value is x. It is an error if there is no such item.
- list.**count**(x)
  - Return the number of times x appears in the list. list.sort() Sort the items of the list, in place.
- list.**reverse**()
  - Reverse the elements of the list, in place.

Appendix C

Dictionary operations

Full documentation of these operations and dictionaries in general can be found in the Python documentation.

- len(d)
  - Return the number of items in the dictionary d.
- d[key]
  - Return the item of d with key key. Raises a KeyError if key is not in the map.
- d[key] = value
  - Set d[key] to value.
- del d[key]
  - Remove d[key] from d. Raises a KeyError if key is not in the map.
- key in d
  - Return True if d has a key key, else False.
- key not in d

  - Equivalent to not key in d.

- clear()
  - Remove all items from the dictionary.

- copy()
  - Return a shallow copy of the dictionary.

- fromkeys(seq[, value])
  - Create a new dictionary with keys from seq and values set to value.
  - fromkeys() is a class method that returns a new dictionary. value defaults to None.

- get(key[, default])
  - Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

- items()
  - Return a copy of the dictionary's list of (key, value) pairs. Note: Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If items(), keys(), values(), iteritems(), iterkeys(), and itervalues() are called with no intervening modifications to the dictionary, the lists will directly correspond. This allows the creation of (value, key) pairs using zip(): pairs = zip(d.values(), d.keys()). The same relationship holds for the iterkeys() and itervalues() methods: pairs = zip(d.itervalues(), d.iterkeys()) provides the same value for pairs. Another way to create the same list is pairs = [(v, k) for (k, v) in d.iteritems()].

- iteritems()

  - Return an iterator over the dictionary's (key, value) pairs. See the note for dict.items().

- iterkeys()

  - Return an iterator over the dictionary's keys. See the note for dict.items().

- itervalues()

  - Return an iterator over the dictionary's values. See the note for dict.items().

- keys()

  - Return a copy of the dictionary's list of keys. See the note for dict.items().

- pop(key[, default])

  - If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

- popitem()

  - Remove and return an arbitrary (key, value) pair from the dictionary.
  - popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling popitem() raises a KeyError. setdefault(key[, default]) If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.

- update([other])

  - Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.

  - update() accepts either another dictionary object or an iterable of key/value pairs (as a tuple or other iterable of length two). If keyword arguments are specified, the dictionary is then is updated with those key/value pairs: d.update(red=1, blue=2).

- values()
  - Return a copy of the dictionary's list of values. See the note for dict.items().

Appendix D

Operators

Table D.1 is a list of the operators found in Python. These include standard mathematics functions, logical operators, etc.

Table D.1: Python Operators

Symbol	Type	What It Does
+	Math	Addition
-	Math	Subtraction
*	Math	Multiplication
/	Math	Division (floating point)
//	Math	Division (truncation)
**	Math	Powers
%	Modulos	Returns the remainder from division
<<	Shift	Left bitwise shift
>>	Shift	Right bitwise shift
&	Logical	And
|	Logical	Or
^	Logical	Bitwise XOR
~	Logical	Bitwise Negation
<	Comparison	Less than
>	Comparison	Greater than
==	Comparison	Exactly equal to

!=	Comparison	Not equal to
>=	Comparison	Greater than or equal to
<=	Comparison	Less than or equal to
=	Assignment	Assign a value
+=	Assignment	Add and assign
-=	Assignment	Subtract and assign
*=	Assignment	Multiply and assign
/=	Assignment	Divide and assign
//=	Assignment	Truncate divide and assign
**=	Assignment	Power and assign
%=	Assignment	Modulus and assign
>>	Assignment	Right shift and assign
<<	Assignment	Left shift and assign
And	Boolean	
Or	Boolean	
Not	Boolean	

Appendix E

Sample programs

All of the programs in this section are written for Python 2.6 or below. However, these programs are available electronically from this book's web site in both Python 2.x and Python 3.x versions. If you downloaded the torrent file, then all the software versions are included.

E.1 Dice rolling simulator

Listing D.1 is one of the first programs I ever wrote. I have the source code listed in its entirety to show the various features that I suggest should be included in a well-written program. Take a look at it and I will discuss the various sections afterward.

Listing E.1: Random dice roller

```
1 ##########################################
2 #Dice_roller.py
3 #
4 #Purpose: A random number generation program that
      simulates
5 # various dice rolls.
6 #Author: Cody Jackson
7 #Date: 4/11/06
8 #
9 #Copyright 2006 Cody Jackson
```

```
10 #This program is free software; you can
       redistribute it and/or modify it
11 #under the terms of the GNU General Public License
        as published by the Free
12 #Software Foundation; either version 2 of the
       License, or (at your option)
13 #any later version.
14 #
15 #This program is distributed in the hope that it
       will be useful, but
16 #WITHOUT ANY WARRANTY; without even the implied
       warranty of
17 #MERCHANTABILITY or FITNESS FOR A PARTICULAR
       PURPOSE. See the GNU
18 #General Public License for more details.
19 #
20 #You should have received a copy of the GNU
       General Public License
21 #along with this program; if not, write to the
       Free Software Foundation,
22 #Inc., 59 Temple Place, Suite 330, Boston, MA
       02111-1307 USA
23 #-------------------------------------
24 #Version 1.0
25 #    Initial build
26 #Version 2.0
27 #    Added support for AD&D 1st edition dice (d4,
       d8, d12, d20)
28 ########################################
29
30 import random #randint
31
32 def randomNumGen(choice):
33     """Get a random number to simulate a d6, d10,
           or d100 roll."""
34
35     if choice == 1: #d6 roll
36         die = random.randint(1, 6)
```

```
37      elif choice == 2: #d10 roll
38          die = random.randint(1, 10)
39      elif choice == 3: #d100 roll
40          die = random.randint(1, 100)
41      elif choice == 4: #d4 roll
42          die = random.randint(1, 4)
43      elif choice == 5: #d8 roll
44          die = random.randint(1, 8)
45      elif choice == 6: #d12 roll
46          die = random.randint(1, 12)
47      elif choice == 7: #d20 roll
48          die = random.randint(1, 20)
49      else:   #simple error message
50          print "Shouldn't be here.  Invalid choice"
51      return die
52
53  def multiDie(dice_number, die_type):
54      """Add die rolls together, e.g. 2d6, 4d10, etc
            """
55
56  #---Initialize variables
57      final_roll = 0
58      val = 0
59
60      while val < dice_number:
61          final_roll += randomNumGen(die_type)
62          val += 1
63      return final_roll
64
65  def test():
66      """Test criteria to show script works."""
67
68      _1d6 = multiDie(1,1)    #1d6
69      print "1d6 = ", _1d6,
70      _2d6 = multiDie(2,1)    #2d6
71      print "\n2d6 = ", _2d6,
72      _3d6 = multiDie(3,1)    #3d6
73      print "\n3d6 = ", _3d6,
```

```
74      _4d6 = multiDie(4,1)    #4d6
75      print "\n4d6 = ", _4d6,
76      _1d10 = multiDie(1,2)   #1d10
77      print "\n1d10 = ", _1d10,
78      _2d10 = multiDie(2,2)   #2d10
79      print "\n2d10 = ", _2d10,
80      _3d10 = multiDie(2,2)   #3d10
81      print "\n3d10 = ", _3d10,
82      _d100 = multiDie(1,3)   #d100
83      print "\n1d100 = ", _d100,
84
85  if __name__ == "__main__":  #run test() if calling
        as a separate program
86      test()
```

The first section (lines 1-28) is the descriptive header and license information. It gives the name of the file, a purpose statement, author, date created, and the license under which it is distributed. In this case, I am releasing it under the GNU General Public License, which basically means that anyone is free to take and use this software as they see fit as long as it remains Free (as in freedom) under the terms of the GPL. For convenience, I have included a copy of the GPL in Appendix F. Obviously, you are free to choose whatever type of license to use with your software, whether release it as open-source or lock it down as completely proprietary.

Line 30 imports the **random** module from the Python standard library. This is used to call the **randint()** function to do the randomization part of the program. Lines 32-51 are where the random number generator is made. The user determines what type of die to roll, e.g. a 6-sided or a 10-sided, and it returns a random number. The random number is an integer based on the range of values as determined by the user. As currently written, this generator program can be used with several different games, from board games to role-playing games.

Lines 53-63 define the function that actually returns a value simulating a dice roll. It receives the die type and the quantity to "roll" from the user then calls the random number function in the previous block. This function has a *while* loop that will continue to roll dice until the number specified by the user is reached, then it returns the

final value.

Lines 65-83 constitute a **test()** function. This is a common feature of Python programs; it allows the programmer to verify that the program is working correctly. For example, if this program was included in a computer game but the game appeared to be having problems with dice rolls, by invoking this program by itself the **test()** function is automatically called (by lines 85 & 86). Thus, a developer could quickly determine if the results from this program make sense and decide if it was the cause of the game's problems.

If this program is called separately without arguments, the **test()** function is called by lines 85 & 86 and displays the various results. However, if the program is called with arguments, i.e. from the command line or as part of a separate program, then lines 85 & 86 are ignored and the **test()** function is never called.

This is a very simple program but it features many of the basic parts of Python. It has *if/else* statements, a *while* loop, returns, functions, module imports, et al. Feel free to modify this program or try your hand at writing a similar program, such as a card dealing simulator. The more comfortable you are at writing simple programs, the easier it will be when your programs get larger and encompass more features.

E.2 Temperature conversion

Listing E.2: Fahrenheit to Celsius converter

```
#########################################
#Create a chart of fahrenheit to celsius
    conversions from 0 to 100 degrees.
#Author:  Cody Jackson
#Date:  4/10/06
#########################################

def fahrenheit(celsius):
    """Converts celsius temperature to fahrenheit
       """
```

```
    fahr = (9.0/5.0)*celsius + 32
    return fahr

#---Create table
print "Celsius | Fahrenheit\n"  #header

for temp_c in range (0, 101):
    temp_f = fahrenheit(temp_c)
    print temp_c, " | %.1f\n" % temp_f
```

Listing E.2 is a short, simple program to convert Fahrenheit to Celsius. Again, it was one of the first programs I wrote. This one isn't nearly as complicated as Listing D.1 but you can see that it gets the job done. I didn't include the GPL information in this one since Listing E.1 already showed you what it looks like.

This particular program doesn't take any user input; it simply creates a table listing the values from 0 to 100 degrees Celsius. As practice, you might want to convert it to accept user input and display only one temperature conversion.

E.3 Game character attribute generator

Listing E.3 shows a short program I wrote to generate character attributes for a role-playing game. It's pretty self-explanatory but I will add comments below.

Listing E.3: Attribute generator

```
1  ########################################
2  #Attribute_generator.py
3  #
4  #Purpose:  Create the attributes for a character.
5  #Author:  Cody Jackson
6  #Date:  4/17/06
7  #
8  #Version: 2
9  #Changes - changed attribute variables to
       dictionaries
10 ########################################
11
```

```
12 import dice_roller #multiDie
13
14 def setAttribute():
15     """Generate value for an attribute."""
16
17     attrib = dice_roller.multiDie(2, 2)
18     return attrib
19
20 def hitPoints(build, body_part = 3):
21     """Generate hit points for each body part."""
22
23     if body_part == 1:  #head
24         hp = build
25     elif body_part == 2:    #torso
26         hp = build * 4
27     elif body_part == 3:   #arms & legs
28         hp = build * 2
29     return hp
30
31 def makeAttrib():
32     #--Make dictionary of attributes
33     attrib_dict = {}
34
35     #--Get values for core attributes
36     attrib_dict["str"] = setAttribute()     #
           strength
37     attrib_dict["intel"] = setAttribute()  #
           intelligence
38     attrib_dict["will"] = setAttribute()    #
           willpower
39     attrib_dict["charisma"] = setAttribute()
40     attrib_dict["build"] = setAttribute()
41     attrib_dict["dex"] = setAttribute()      #
           dexterity
42
43     #--Get values for secondary attributes
44     attrib_dict["mass"] = (attrib_dict.get("build"
           ) * 5) + 15 #kilograms
```

```
45      attrib_dict["throw"] = attrib_dict.get("str")
            * 2     #meters
46      attrib_dict["load"] = (attrib_dict.get("str")
47          + attrib_dict.get("build")) * 2    #
                kilograms
48
49      return attrib_dict
50
51  ##Get build value from attribute dictionary to
        calculate hp
52  attribs = makeAttrib()
53  build = attribs.get("build")
54
55  def makeHP(build):
56      #--Make dictionary of hit points
57      hp_dict = {}
58
59      #--Get hit points
60      hp_dict["head"] = hitPoints(build, 1)
61      hp_dict["torso"] = hitPoints(build, 2)
62      hp_dict["rarm"] = hitPoints(build)
63      hp_dict["larm"] = hitPoints(build)
64      hp_dict["rleg"] = hitPoints(build)
65      hp_dict["lleg"] = hitPoints(build)
66
67      return hp_dict
68
69  def test():
70      """Show test values."""
71
72      attribs = makeAttrib()
73      build = attribs.get("build")
74      hp = makeHP(build)
75      print attribs
76      print hp
77
78  if __name__ == "__main__":
79      test()
```

Line 12 imports the dice rolling program from Listing E.1. This is probably the most important item in this program; without the dice roller, no results can be made.

Line 14 is a function that calls the dice roller module and returns a value. The value will be used as a character's physical or mental attribute.

Line 20 starts a function that generates a character's hit points, based on individual body parts, e.g. head, arm, torso, etc. This function uses a default value for *body_part*; this way I didn't have to account for every body part individually.

Line 31 is a function to store each attribute. A dictionary is used to link an attribute name to its corresponding value. Each attribute calls the **setAttribute()** function (Line 14) to calculate the value. The completed dictionary is returned when this function is done.

Line 51 shows a "block comment", a comment I use to describe what a block of related code will be doing. It's similar to a doc string but is ignored by the Python interpreter. It's simply there to help the programmer understand what the following lines of code will be doing. The code following the block comment is delimited by a blank line from any non-related code.

Line 55 is a function that puts hit point values into a dictionary, and follows the pattern of **makeAttrib()** (Line 31) function above.

Line 69 is a testing function. When this program is called on its own (not part of another program), such as from the command line, it will make a sample character and print out the values to show that the program is working correctly. Line 78 has the command that determines whether the **test()** function will run or not.

E.4 Text-based character creation

The next few pages (Listing E.4) are a console-based (non-graphical) character creation program that I wrote shortly after the attribute generator above. This was the basis for a game I was making based on the Colonial Marines from the Aliens movie. It's not perfect, barely complete enough for real use, but it is useful for several reasons:

1. It's object-oriented, showing how inheritance, "self", and many of the other features of OOP actually work in practice.

2. It takes user input and outputs data, so you can see how input/output functions work.

3. Nearly all the main features of Python are used, notably lists and dictionaries.

4. It's reasonably well documented, at least in my opinion, so following the logic flow should be somewhat easy.

Many tutorial books I've read don't have an in-depth program that incorporates many of the features of the language; you usually just get a bunch of small, "one shot" code snippets that only explain a concept. The reader is usually left on his own to figure out how to put it all together.

I've included this program because it shows a good number of concepts of the Python language. If ran "as is", the built-in test function lets the user create a character. It doesn't include file operations but I'll leave it as a challenge for the reader to modify it to save a complete character. It's also not written in Python 3.x, so I can't guarantee that it will run correctly if you have that version installed.

You'll note the "commented out" print statements in the **test()** function. Using print statements in this manner can be a good way of making sure your code is doing what it should be, by showing what various values are within the program.

Listing E.4: Non-graphical character creation

```
#########################################
#BasicCharacter.py
#
#Purpose: A revised version of the Marine
    character using classes.
#Author: Cody Jackson
#Date: 6/16/06
#
#Copyright 2006 Cody Jackson
#----------------------------------------
#Version 0.3
#   Removed MOS subset catagories; added values to
    MOS dictionary
```

```
#    Added default MOS skills
#    Added Tank Commander and APC Commander MOS's
#    Combined separate Character and Marine class
    tests into one test method
#    Added character initiative methods
#Version 0.2.1
#    Corrected Marine rank methods
#Version 0.2
#    Added Marine subclass
#Version 0.1
#    Initial build
##########################################

#TODO: make I/O operations (save to file)
from dice_roller import multiDie

class Character:
    """Creation of a basic character type."""

    #---Class attributes
    default_attribs = {}

    def __init__(self):
        """Constructor to initialize each data
            member to zero."""

        #---General info
        self.name = ""
        self.gender = ""
        self.age = 0

        #---Attribute info
        self.attrib = 0
        self.attrib_dict = self.default_attribs.
            copy()  #instance version
        self.setAttribute("str")
        self.setAttribute("intel")
        self.setAttribute("build")
```

```
        self.setAttribute("char")
        self.setAttribute("will")
        self.setAttribute("throw")
        self.setAttribute("mass")
        self.setAttribute("load")
        self.setAttribute("dex")
        self.coreInitiative = 0

        #---Hit point info
        self.hp_dict = {}
        self.body_part = 0
        self.setHitPoints("head")
        self.setHitPoints("torso")
        self.setHitPoints("rarm")
        self.setHitPoints("larm")
        self.setHitPoints("lleg")
        self.setHitPoints("rleg")

        #---Skills info
        self.chosenSkills = {}
        self.subskills = {}
        self.skillPoints = 0
        self.skillLevel = 0
        self.skillChoice = 0
        self.maximum = 0
        self.subskillChoices = []
        self.baseSkills = ["Armed Combat", "
            Unarmed Combat", "Throwing", "Small
                        Arms", "Heavy Weapons", "
            Vehicle Weapons", "Combat Engineer", "
            First                Aid", "Wheeled
            Vehicles", "Tracked Vehicles", "
            Aircraft",              "Interrogation", "
            Swimming", "Reconnaissance", "Mechanic"
            , "Forward                  Observer", "
            Indirect Fire", "Electronics", "
            Computers", "Gunsmith",
```

```
            "Supply", "Mountaineering", "Parachute
                ", "Small Boats",
            "Harsh environments", "Xenomorphs", "
                Navigation", "Stealth",
            "Chemistry", "Biology"]

        #---Equipment list
        self.equipment = []

#---Get marine info---------------------
    def setName(self):
        """Get the name of the character."""

        self.name = raw_input("Please enter your
            character's name.\n")

    def setGender(self):
        """Get the gender of the character."""

        while 1:
            self.gender = raw_input("Select a
                gender:  1=Male, 2=Female: ")
            if int(self.gender) == 1:
                self.gender = "Male"
                break
            elif int(self.gender) == 2:
                self.gender = "Female"
                break
            else:
                print "Invalid choice.  Please
                    choose 1 or 2."
                continue

    def setAge(self):
        """Calculate the age of character, between
            18 and 45 years old."""

        self.age = 15 + multiDie(3,2) #3d10
```

```
#---Create attributes--------------------
    def setAttribute(self, attr):
        """Generate value for an attribute,
           between 2 and 20."""

        #---Get secondary attributes
        if attr == "throw":
            self.attrib_dict[attr] = self.
                attrib_dict["str"] * 2    #meters
        elif attr == "mass":
            self.attrib_dict[attr] = (self.
                attrib_dict["build"] * 5) + 15 #kg
        elif attr == "load":
            self.attrib_dict[attr] = (self.
                attrib_dict["str"] +
                self.attrib_dict["build"]) #kg
        #---Get core attributes
        else:
            self.attrib_dict[attr] = multiDie(2,
                2)   #2d10

    def setHitPoints(self, body_part):
        """Generate hit points for each body part,
            based on 'build' attribute."""

        if body_part == "head":
            self.hp_dict[body_part] = self.
                attrib_dict["build"]
        elif body_part == "torso":
            self.hp_dict[body_part] = self.
                attrib_dict["build"] * 4
        else:   #arms & legs
            self.hp_dict[body_part] = self.
                attrib_dict["build"] * 2

    def setInitiative(self):
```

```
        """Establishes the core initiative level
           of a character.

        This initiative level can be modified for
           different character types."""

        self.coreInitiative = multiDie(1,1)

#---Choose skills-----------------
    def printSkills(self):
        """Print list of skills available for a
           character."""

        print "Below is a list of available skills
            for your character."
        print "Some skills may have
           specializations and will be noted when
           chosen."
        for i in range(len(self.baseSkills)):
            print "     ", i, self.baseSkills[i],
                  #5 spaces between columns
            if i % 3 == 0:
                print "\n"

    def setSkills(self):
        """Calculate skill points and pick base
           skills and levels.

        Skill points are randomly determined by
           dice roll. Certain skills have
           specialty areas available and are
           chosen separately."""

        self.skillPoints = multiDie(2,2) * 10
        while self.skillPoints > 0:
            #---Choose skill
            self.printSkills()
```

```
            print "\n\nYou have", self.skillPoints
                , "skill level points available."
            self.skillChoice = int(raw_input("
                Please pick a skill: "))
            self.pick = self.baseSkills[self.
                skillChoice]    #Chosen skill
            print "\nYou chose", self.skillChoice,
                 self.baseSkills[self.skillChoice],
                 "."

            #---Determine maximum skill level
            if self.skillPoints > 98:
                 self.maximum = 98
            else: self.maximum = self.skillPoints

            #---Choose skill level
            print "The maximum points you can use
                are", self.maximum, "."
            self.skillLevel = int(raw_input("What
                skill level would you like? "))
            if self.skillLevel > self.skillPoints:
                  #Are available points exceeded?
                 print "Sorry, you don't have that
                     many points."
                 continue
            self.chosenSkills[self.pick] = self.
                skillLevel  #Chosen skill level
            self.skillPoints -= self.skillLevel #
                Decrement skill points

            #---Pick specialty
            if self.skillLevel >= 20:    #Minimum
                level for a specialty
                 self.pickSubSkill(self.pick)

    def pickSubSkill(self, skill):
        """If a base skill level is 20 or greater,
             a specialty may be available,
```

```
        depending on the skill.

        Chosen skill text string passed in as
            argument."""

        self.skill = skill

        #---Set speciality lists
        if self.skill == "Armed Combat":
            self.subskillChoices = ["Blunt Weapons
                ", "Edged Weapons", "Chain Weapons"
                ]
        elif self.skill == "Unarmed Combat":
            self.subskillChoices = ["Grappling", "
                Pummeling", "Throttling"]
        elif self.skill == "Throwing":
            self.subskillChoices = ["Aerodynamic",
                 "Non-aerodynamic"]
        elif self.skill == "Small Arms":
            self.subskillChoices = ["Flamer", "
                Pulse Rifle", "Smartgun", "Sniper
                rifle",
                 "Pistol"]
        elif self.skill == "Heavy Weapons":
            self.subskillChoices = ["PIG", "RPG",
                "SADAR", "HIMAT", "Remote Sentry"]
        elif self.skill == "Vehicle Weapons":
            self.subskillChoices = ["Aircraft", "
                Land Vehicles", "Water Vehicles"]
        elif self.skill == "Combat Engineer":
            self.subskillChoices = ["Underwater
                demolitions", "EOD", "Demolitions",
                 "Land structures", "Vehicle use",
                    "Bridges"]
        elif self.skill == "Aircraft":
            self.subskillChoices = ["Dropship", "
                Conventional", "Helicopter"]
        elif self.skill == "Swimming":
```

```
            self.subskillChoices = ["SCUBA", "
                Snorkel"]
        elif self.skill == "Mechanic":
            self.subskillChoices = ["Wheeled", "
                Tracked", "Aircraft"]
        elif self.skill == "Electronics":
            self.subskillChoices = ["Radio", "ECM"
                ]
        elif self.skill == "Computers":
            self.subskillChoices = ["Hacking", "
                Programming"]
        elif self.skill == "Gunsmith":
            self.subskillChoices = ["Small Arms",
                "Heavy Weapons", "Vehicles"]
        elif self.skill == "Parachute":
            self.subskillChoices = ["HALO", "HAHO"
                ]
        elif self.skill == "Harsh Environments":
            self.subskillChoices = ["No atmosphere
                ", "Non-terra"]
        else:
            return

        self.statement(self.skill)
        for i in range(len(self.subskillChoices)):
            print i, self.subskillChoices[i]
        self.choice = int(raw_input())
        if self.choice == -1:   #Specialization
            not desired
            return
        else:   #Speciality chosen
            print "You chose the", self.
                subskillChoices[self.choice], "
                specialty.\n\
It has an initial skill level of 10."
            self.subskills[self.subskillChoices[
                self.choice]] = 10
        print self.subskills
```

```
            return

        def statement(self, skill):
            """Prints a generic statement for choosing
                a skill specialty."""

            self.skill = skill
            print "\n", self.skill, "has
                specializations.  If you desire to
                specialize in\
 a field,\nplease choose from the following list.
     Enter -1 if you don't want a\n\
specialization.\n"

#---Equipment access methods--------------------
        def setEquipment(self, item):
            """Add equipment to character's inventory.
                """

            self.equipment.append(item)

        def getEquipment(self):
            """Display equipment in inventory."""

            print self.equipment

class Marine(Character):
    """Specializiation of a basic character with
       MOS and default skills."""

    def __init__(self):
        """Initialize Marine specific values."""

        #---Class attributes
        self.marineInit = 0
        #---Rename attributes
        Character.__init__(self)
        self.intel = self.attrib_dict["intel"]
```

```
        self.char = self.attrib_dict["char"]
        self.str = self.attrib_dict["str"]
        self.dex = self.attrib_dict["dex"]
        self.will = self.attrib_dict["will"]

        #---Rank attributes
        self.rank = ""
        self.rankName = ""
        self.modifier = self.intel * .1 #rank
            modifier

        #---MOS attributes
        self.mos = ""
        self.mosSpecialty = ""
        self.mosSelection = {0:"Supply", 1:"Crew
            Chief", 2:"Infantry"}

#---Determine rank---------------------
    def setRank(self):
        """Determine rank of Marine."""

        if (self.intel + self.char) > 23:    #
            senior ranks
             self.roll = multiDie(1,2) + self.
                modifierValue()
             self.rank = self.seniorRank(self.roll)
        else:    #junior ranks
             self.roll = multiDie(1,2)
             self.rank = self.lowerRank(self.roll)

        #---Convert numerical rank to full name
        if self.rank == "E1":
             self.rankName = "Private"
        elif self.rank == "E2":
             self.rankName = "Private First Class"
        elif self.rank == "E3":
             self.rankName = "Lance Corporal"
        elif self.rank == "E4":
```

```
            self.rankName = "Corporal"
        elif self.rank == "E5":
            self.rankName = "Sergeant"
        elif self.rank == "E6":
            self.rankName = "Staff Sergeant"
        elif self.rank == "E7":
            self.rankName = "Gunnery Sergeant"
        elif self.rank == "E8":
            self.rankName = "Master Sergeant"
        elif self.rank == "O1":
            self.rankName = "2nd Lieutenant"
        elif self.rank == "O2":
            self.rankName = "1st Lieutenant"
        elif self.rank == "O3":
            self.rankName = "Captain"
        elif self.rank == "O4":
            self.rankName = "Major"
        else: print "Invalid rank."

    def modifierValue(self):
        """Determine rank modifier value."""

        if self.modifier % 1 < .5:  #round down
            self.modifier = int(self.modifier)
        elif self.modifier %1 >= .5:    #round up
            self.modifier = int(self.modifier) + 1
        return self.modifier

    def lowerRank(self, roll):
        """Determine rank of junior marine."""

        if self.roll == 1:
            self.rank = "E1"
        elif self.roll == 2:
            self.rank = "E2"
        elif self.roll == 3 or self.roll == 4:
            self.rank = "E3"
        elif self.roll == 5 or self.roll == 6:
```

```
            self.rank = "E4"
        elif 7 <= self.roll <= 9:
            self.rank = "E5"
        elif self.roll == 10:
            self.rank = "E6"
        else:   #Shouldn't reach here
            print "Ranking roll invalid"
        return self.rank

    def seniorRank(self, roll):
        """Determine rank and if character is
           senior enlisted or officer."""

        if self.roll > 5:   #Character is officer
            self.new_roll = multiDie(1,2) + self.
                modifierValue()
            if 1 <= self.new_roll <= 3:
                self.rank = "O1"
            elif 4 <= self.new_roll <= 7:
                self.rank = "O2"
            elif 8 <= self.new_roll <= 10:
                self.rank = "O3"
            elif self.new_roll > 10:
                self.rank = "O4"
            else:   #Shouldn't reach here
                print "Ranking roll invalid"
        else:   #Character is senior enlisted
            self.new_roll = multiDie(1,2)
            if 1 <= self.new_roll <= 4:
                self.rank = "E6"
            elif 5 <= self.new_roll <= 8:
                self.rank = "E7"
            elif 9 <= self.new_roll <= 11:
                self.rank = "E8"
            elif self.new_roll > 11:
                self.rank = "E9"
            else: #Shouldn't reach here
                print "Ranking roll invalid"
```

```
        return self.rank

#---Get MOS-------------------
    def requirements(self):
        """Adds eligible MOS's to selection
            dictionary.

        Takes various character attributes and
            determines if additional MOS choices
        are available."""

        if self.intel >= 11:
            self.mosSelection[3] = "Medical"
        ##Value "4" is missing due to changing
            positions
        if self.str >= 12:
            self.mosSelection[5] = "Heavy Weapons"
        if (self.dex + self.intel + self.will) >=
            40:
            if multiDie(1,3) >= 85:
                self.mosSelection[6] = "Scout/
                    Sniper"
        if (self.str + self.dex + self.intel +
            self.will) >= 50:
            if multiDie(1,3) >= 85:
                self.mosSelection[7] = "Recon"
        if self.str >= 10 and self.intel >= 10:
            self.mosSelection[8] = "Combat
                Engineer"
        if self.dex >= 11:  #Armor MOS's
            self.mosSelection[9] = "Tank Driver"
            self.mosSelection[10] = "Tank Gunner"
            self.mosSelection[11] = "APC Driver"
            self.mosSelection[12] = "APC Gunner"
        if self.intel + self.will >= 27:    #
            Intelligence MOS's
            self.mosSelection[15] = "Intelligence
                Analyst"
```

```
            self.mosSelection[16] = "Interrogator"
            self.mosSelection[17] = "Intelligence:
                Xenomorphs"
        if self.intel >= 10 and self.dex >= 10 and
            ("O1" <= self.rank <= "O4"):
            self.mosSelection[18] = "Pilot"
            self.mosSelection[19] = "Co-Pilot/
                Gunner"
            ##Tank and APC Commander MOS have the
                same requirements as pilots
            ##  but are added after the 'Armor'
                MOS's.
            self.mosSelection[13] = "Tank
                Commander"
            self.mosSelection[14] = "APC Commander
                "

    def eligibleMOS(self):
        """Displays MOS's available to character.
            """

        print "You are eligible for the following
            MOS's:"
        for key in self.mosSelection.keys():
            print key, self.mosSelection[key]
        self.mosChoice = int(raw_input("""\nPlease
            choose the number of
                the MOS you want: """))
        self.mos = self.mosSelection[self.
            mosChoice]     #Rename selection
        print "You chose the", self.mos, "
            specialty."
        ##Subclasses removed for possible later
            use
        #if self.mosChoice == 2 or self.mosChoice
            == 4 or self.mosChoice == 9 or
        #       self.mosChoice == 10:
```

```
        #     self.mosSpecialty = self.subClass(
            self.mosChoice)
        #if self.mosSpecialty == "":
        #     self.mos = self.mosSelection[self.
            mosChoice]
        #else:
        #     self.mos = self.mosSelection[self.
            mosChoice] +
        #          " (" + self.mosSpecialty + ")"
        return self.mos

    def mosSkills(self, mosChoice):
        """Determine default skill levels based on
            MOS."""

        if self.mosChoice == 0: #Supply
            self.chosenSkills["Supply"] = 20
        elif self.mosChoice == 1:    #Crew Chief
            self.chosenSkills["Mechanic"] = 20
        elif self.mosChoice == 2:    #Infantry
            self.chosenSkills["Small Arms"] = 20
            self.chosenSkills["Armed Combat"] = 10
            self.chosenSkills["Unarmed Combat"] =
                10
            self.chosenSkills["Swimming"] = 10
        elif self.mosChoice == 3:    #Medical
            self.chosenSkills["First Aid"] = 20
        elif self.mosChoice == 4:    #Not a value
            pass
        elif self.mosChoice == 5:    #Heavy Weapons
            self.chosenSkills["Heavy Weapons"] =
                20
            self.chosenSkills["Small Arms"] = 10
            self.chosenSkills["Armed Combat"] = 10
            self.chosenSkills["Unarmed Combat"] =
                10
        elif self.mosChoice == 6:    #Scout/Sniper
            self.chosenSkills["Recon"] = 15
```

```
            self.chosenSkills["Forward Observer"]
                = 10
            self.chosenSkills["Small Arms"] = 25
            self.chosenSkills["Armed Combat"] = 10
            self.chosenSkills["Unarmed Combat"] =
                10
            self.chosenSkills["Swimming"] = 10
            self.chosenSkills["Navigation"] = 10
        elif self.mosChoice == 7:    #Recon
            self.chosenSkills["Recon"] = 20
            self.chosenSkills["Forward Observer"]
                = 10
            self.chosenSkills["Small Arms"] = 20
            self.chosenSkills["Armed Combat"] = 10
            self.chosenSkills["Unarmed Combat"] =
                10
            self.chosenSkills["Swimming"] = 10
            self.chosenSkills["Mountaineering"] =
                10
            self.chosenSkills["Parachute"] = 10
            self.chosenSkills["Navigation"] = 10
        elif self.mosChoice == 8:    #Combat
            Engineer
            self.chosenSkills["Combat Engineer"] =
                20
        elif self.mosChoice == 9:    #Tank Driver
            self.chosenSkills["Tracked Vehicles"]
                = 20
        elif self.mosChoice == 11:  #APC Driver
            self.chosenSkills["Wheeled Vehicles"]
                = 20
        elif self.mosChoice == 10 or self.
            mosChoice == 12:  #Vehicle Gunners
            self.chosenSkills["Vehicle Weapons"] =
                20
            self.chosenSkills["Indirect Fire"] =
                10
```

```
        elif self.mosChoice == 13 or self.
            mosChoice == 14:  #Vehicle Commanders
             self.chosenSkills["Navigation"] = 15
             self.chosenSkills["Tracked Vehicles"]
                = 10
             self.chosenSkills["Wheeled Vehicles"]
                = 10
             self.chosenSkills["Vehicle Weapons"] =
                 10
             self.chosenSkills["Indirect Fire"] =
                10
             self.chosenSkills["Forward Observer"]
                = 10
        elif self.mosChoice == 15:  #Analyst
             self.chosenSkills["Electronics"] = 20
             self.chosenSkills["Computers"] = 20
        elif self.mosChoice == 16:  #Interrogation
             self.chosenSkills["Electronics"] = 10
             self.chosenSkills["Computers"] = 10
             self.chosenSkills["Interrogation"] =
                20
        elif self.mosChoice == 17:  #Xenomorphs
             self.chosenSkills["Electronics"] = 10
             self.chosenSkills["Computers"] = 10
             self.chosenSkills["Xenomorphs"] = 20
             self.chosenSkills["Biology"] = 10
             self.chosenSkills["Chemistry"] = 10
        elif self.mosChoice == 18:  #Pilot
             self.chosenSkills["Aircraft"] = 20
             self.chosenSkills["Vehicl Weapons"] =
                10
        elif self.mosChoice == 19:  #Co-Pilot/
            Gunner
             self.chosenSkills["Aircraft"] = 10
             self.chosenSkills["Vehicle Weapons"] =
                 20
        else: print "**Invalid option**"     #
            Shouldn't reach
```

```
#---Base initiative--------------------
    def marineInitiative(self):
        """Creates the base level for a Marine
           character type.

        Marines with an MOS of Sniper/Scout, Recon
           , or Pilot gain +1 to initiative.
        Marines with an MOS of Suppy, Intelligence
           , or Medical lose -1 to initiative."""

        if self.mosChoice == 6 or self.mosChoice
           == 7 or self.mosChoice == 18:
            self.marineInit = self.coreInitiative
               + 1
        elif self.mosChoice == 0 or self.mosChoice
            == 3 or 15 <= self.mosChoice <= 17:
            self.marineInit = self.coreInitiative
               -1
        else: self.marineInit = self.
           coreInitiative

def test():
    marine = Character()
    marine = Marine()
    marine.setName()
    marine.setGender()
    marine.setAge()
    marine.setSkills()
    marine.setEquipment("Rucksack")
    marine.setEquipment("Knife")
    marine.setEquipment("Flashlight")
    #print "\nName = ", marine.name
    #print "Gender = ", marine.gender
    #print "Age = ", marine.age
    #for key in marine.attrib_dict.keys():
    #    print key, marine.attrib_dict[key]
    #for key in marine.hp_dict.keys():
```

```
    #     print key, marine.hp_dict[key]
    #print marine.chosenSkills
    #print marine.subskills
    #marine.getEquipment()
    marine.setRank()
    print "Rank: ", marine.rank
    #print "Rank name: ", marine.rankName
    marine.requirements()
    marine.eligibleMOS()
    print "Your MOS is: ", marine.mos
    marine.mosSkills(marine.mosChoice)
    print marine.chosenSkills
    marine.setInitiative()
    marine.marineInitiative()
    print "Your Initiative is: ", marine.
        marineInit

if __name__ == "__main__":
    test()
```

Appendix F

GNU General Public License

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

1. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

2. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

3. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

4. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention

to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

5. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

6. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

(d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

7. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A

product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

8. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified

versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

9. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you

have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

10. Acceptance Not Required for Having Copies.

    You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

11. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

12. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of

this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

13. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

14. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

15. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

16. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

17. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER

PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

18. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
>
> Copyright (C) <textyear> <name of author>
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> <program> Copyright (C) <year> <name of author>
> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.