in Python 2.0" appendix has been replaced with a pair regarding Python 3 and the role that Python 2.6 plays in the transition to the next generation.

Chapter Guide

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

Python is everywhere—sometimes it is amazing to discover who is using Python and what they are doing with it—and although we would have loved to produce additional chapters on such topics as Java/Jython, Win32 programming, CGI processing with `HTMLgen`, GUI programming with third-party toolkits (wxWidgets, GTK+, Qt, etc.), XML processing, numerical and scientific processing, visual and graphics image manipulation, and Web services and application frameworks (Zope, Plone, Django, TurboGears, and so on), there simply wasn't enough time to develop these topics into their own chapters. However, we are certainly glad that we were at least able to provide you with a good introduction to many of the key areas of Python development including some of the topics mentioned previously.

Here is a chapter-by-chapter guide.

Part I: Core Python

Chapter 1—Welcome to Python!

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

Chapter 2—Getting Started

If you are an experienced programmer and just want to see "how it's done" in Python, this is the right place to go. We introduce the basic Python concepts and statements, and because many of these will be familiar to you, you can simply learn the proper syntax in Python and get started right away on your projects without sacrificing too much reading time.

Chapter 3—Syntax and Style

This section gives you a good overview of Python's syntax as well as style hints. You will also be exposed to Python's keywords and its memory management ability.

Your first Python application will be presented at the end of the chapter to give you an idea of what real Python code looks like.

Chapter 4—Python Objects

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

Chapter 5—Numbers

In this chapter, we discuss Python's main numeric types: integers, floating point numbers, and complex numbers. We look at operators and built-in and factory functions which apply to all numbers, and we also briefly discuss a few other related types.

Chapter 6—Sequences: Strings, Lists, and Tuples

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

Chapter 7—Mapping and Set Types

Dictionaries are Python's mapping or hashing type. Like other data types, dictionaries also have operators and applicable built-in functions and methods. We also cover Python's set types in this chapter, discussing their operators, built-in and factory functions, and built-in methods.

Chapter 8—Conditionals and Loops

Like many other high-level languages, Python supports loops such as **`for`** and **`while`**, as well as **`if`** statements (and related). Python also has a built-in function called `range()` which enables Python's **`for`** loop to behave more like a traditional counting loop rather than the "foreach" iterative type loop that it is. Also included is coverage of auxiliary statements such as **`break`**, **`continue`**, and **`pass`**, as well as a discussion of newer constructs like iterators, list comprehensions, and generator expressions.

Chapter 9—Files and Input/Output

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

Chapter 10—Errors and Exceptions

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

Chapter 11—Functions and Functional Programming

Creating and calling functions are relatively straightforward, but Python has many other features that you will find useful, such as default arguments, named or keyword arguments, variable-length arguments, and some functional programming constructs. We also dip into variable scope and recursion briefly. We will also discuss some advanced features such as generators, decorators, inner functions, closures, and partial function application (a more generalized form of currying).

Chapter 12—Modules

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

Chapter 13—Object-Oriented Programming

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

Chapter 14—Execution Environment

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

Part II: Advanced Topics

Chapter 15—Regular Expressions

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

Chapter 16—Network Programming

So many applications today need to be network-oriented. You have to start somewhere. In this chapter, you will learn to create clients and servers, using TCP/IP and UDP/IP, as well as get an introduction to `SocketServer` and Twisted.

Chapter 17—Internet Client Programming

In Chapter 16, we introduced network programming using sockets. Most Internet protocols in use today were developed using sockets. In this chapter, we explore some of these higher-level libraries, which are used to build clients of such Internet protocols. In particular, we focus on FTP, NNTP, SMTP, and POP3 clients.

Chapter 18—Multithreaded Programming

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

Chapter 19—GUI Programming

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

Chapter 20—Web Programming

Web programming using Python takes three main forms: Web clients, Web servers, and the popular Common Gateway Interface applications that help Web servers deliver dynamically-generated Web pages. We will cover them

all in this chapter: simple and advanced Web clients and CGI applications, as well as how to build your own Web server.

Chapter 21—Database Programming

What Python does for application programming carries to database programming as well. It is simplified, and you will find it fun! We first review basic database concepts, then introduce you to the Python database application programmer's interface (API). We then show you how you can connect to a relational database and perform queries and operations with Python. Finally, if you want hands-off using the Structured Query Language (SQL) and want to just work with objects without having to worry about the underlying database layer, we will introduce you to a few object-relational managers (ORMs), which simplify database programming to yet another level.

Chapter 22—Extending Python

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

Chapter 23—Miscellaneous

This new chapter consists of bonus material that we would like to develop into full, individual chapters in the next edition. Topics covered here include Web Services, Microsoft Office (Win32 COM Client) Programming, and Java/Jython.

Optional Sections

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

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

Chapter 1

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

1.1 What Is Python?

Python is an elegant and robust programming language that delivers both the power and general applicability of traditional compiled languages with the ease of use (and then some) of simpler scripting and interpreted languages. It allows you to get the job done, and then read what you wrote later. You will be amazed at how quickly you will pick up the language as well as what kind of things you can do with Python, not to mention the things that have *already* been done. Your imagination will be the only limit.

1.2 Origins

Work on Python began in late 1989 by Guido van Rossum, then at CWI (Centrum voor Wiskunde en Informatica, the National Research Institute for Mathematics and Computer Science) in the Netherlands. It was eventually released for public distribution in early 1991. How did it all begin? Like C, C++, Lisp, Java, and Perl, Python came from a research background where the programmer was having a hard time getting the job done with the existing tools at hand, and envisioned and developed a better way.

At the time, van Rossum was a researcher with considerable language design experience with the interpreted language ABC, also developed at CWI, but he was unsatisfied with its ability to be developed into something more. Having used and partially developed a higher-level language like ABC, falling back to C was not an attractive possibility. Some of the tools he envisioned were for performing general system administration tasks, so he also wanted access to the power of system calls that were available through the Amoeba distributed operating system. Although van Rossum gave some thought to an Amoeba-specific language, a generalized language made more sense, and late in 1989, the seeds of Python were sown.

1.3 Features

Although it has been around for well over fifteen years, some feel that Python is still relatively new to the general software development industry. We should, however, use caution with our use of the word "relatively," as a few years seem like decades when developing on "Internet time."

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

1.3.1 High Level

It seems that with every generation of languages, we move to a higher level. Assembly was a godsend for those who struggled with machine code, then came FORTRAN, C, and Pascal, which took computing to another plane and created the software development industry. Through C came more modern compiled languages, C++ and Java. And further still we climb, with powerful, system-accessible, interpreted scripting languages like Tcl, Perl, and Python.

Each of these languages has higher-level data structures that reduce the "framework" development time that was once required. Useful types like Python's lists (resizeable arrays) and dictionaries (hash tables) are built into the language. Providing these crucial building blocks in the core language encourages their use and minimizes development time as well as code size, resulting in more readable code.

Because there is no one standard library for heterogeneous arrays (lists in Python) and hash tables (Python dictionaries or "dicts" for short) in C, they are often reimplemented and copied to each new project. This process is messy and error prone. C++ improves the situation with the standard template library, but the STL can hardly compare to the simplicity and readability of Python's built-in lists and dicts.

1.3.2 Object Oriented

Object-oriented programming (OOP) adds another dimension to structured and procedural languages where data and logic are discrete elements of programming. OOP allows for associating specific behaviors, characteristics, and/or capabilities with the data that they execute on or are representative of. Python is an object-oriented (OO) language, all the way down to its core. However, Python is not *just* an OO language like Java or Ruby. It is actually a pleasant mix of multiple programming paradigms. For instance, it even borrows a few things from functional languages like Lisp and Haskell.

1.3.3 Scalable

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

The term "scalable" is most often applied to measuring hardware throughput and usually refers to additional performance when new hardware is added to a system. We would like to differentiate this comparison with ours here, which tries to reflect the notion that Python provides basic building

blocks on which you can build an application, and as those needs expand and grow, Python's pluggable and modular architecture allows your project to flourish as well as maintain manageability.

1.3.4 Extensible

As the amount of Python code increases in your project, you will still be able to organize it logically by separating your code into multiple files, or modules, and be able to access code from one module and attributes from another. And what is even better is that Python's syntax for accessing modules is the same for all modules, whether you access one from the Python standard library, one you created just a minute ago, or even an extension you wrote in another language! Using this feature, you feel like you have just "extended" the language for your own needs, and you actually *have*.

The most critical portions of code, perhaps those hotspots that always show up in the profiler or areas where performance is absolutely required, are candidates for being rewritten as a Python extension written in C. But again, the interface is exactly the same as for pure Python modules. Access to code and objects occurs in exactly the same way without any code modification whatsoever. The only thing different about the code now is that you should notice an improvement in performance. Naturally, it all depends on your application and how resource-intensive it is. There are times where it is absolutely advantageous to convert application bottlenecks to compiled code because it will decidedly improve overall performance.

This type of extensibility in a language provides engineers with the flexibility to add-on or customize their tools to be more productive, and to develop in a shorter period of time. Although this feature is self-evident in mainstream third-generation languages (3GLs) such as C, C++, and even Java, the ease of writing extensions to Python in C is a real strength of Python. Furthermore, tools like PyRex, which understands a mix of C and Python, make writing extensions even easier as they compile everything to C for you.

Python extensions can be written in C and C++ for the standard implementation of Python in C (also known as CPython). The Java language implementation of Python is called Jython, so extensions would be written using Java. Finally, there is IronPython, the C# implementation for the .NET or Mono platforms. You can extend IronPython in C# or Visual Basic.NET.

1.3.5 Portable

Python can be found on a wide variety of systems, contributing to its continued rapid growth in today's computing domain. Because Python is written in C,

and because of C's portability, Python is available on practically every type of platform that has an ANSI C compiler. Although there are some platform-specific modules, any general Python application written on one system will run with little or no modification on another. Portability applies across multiple architectures as well as operating systems.

1.3.6 Easy to Learn

Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time. What may perhaps be new to beginners is the OO nature of Python. Those who are not fully versed in the ways of OOP may be apprehensive about jumping straight into Python, but OOP is neither necessary nor mandatory. Getting started is easy, and you can pick up OOP and use when you are ready to.

1.3.7 Easy to Read

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

1.3.8 Easy to Maintain

Maintaining source code is part of the software development lifecycle. Your software usually continues to evolve until it is replaced or obsoleted. Quite often it lasts longer than a programmer's stay at a company. Much of Python's success is that source code is fairly easy to maintain, dependent, of course, on size and complexity. However, this conclusion is not difficult to draw given that Python is easy to learn and easy to read. Another motivating advantage of Python is that upon reviewing a script you wrote six months ago,

you are less likely to get lost or pull out a reference book to get reacquainted with your software.

1.3.9 Robust

Nothing is more powerful than allowing a programmer to recognize error conditions and provide a software handler when such errors occur. Python provides "safe and sane" exits on errors, allowing the programmer to be in the driver's seat. When your Python crashes due to errors, the interpreter dumps out a "stack trace" full of useful information such as why your program crashed and where in the code (file name, line number, function call, etc.) the error took place. These errors are known as exceptions. Python even gives you the ability to monitor for errors and take an evasive course of action if such an error does occur during runtime.

These exception handlers can take steps such as defusing the problem, redirecting program flow, perform cleanup or maintenance measures, shutting down the application gracefully, or just ignoring it. In any case, the debugging part of the development cycle is reduced considerably. Python's robustness is beneficial for both the software designer and the user. There is also some accountability when certain errors occur that are not handled properly. The stack trace that is generated as a result of an error reveals not only the type and location of the error, but also in which module the erroneous code resides.

1.3.10 Effective as a Rapid Prototyping Tool

We've mentioned before how Python is easy to learn and easy to read. But, you say, so is a language like BASIC. What more can Python do? Unlike self-contained and less flexible languages, Python has so many different interfaces to other systems that it is powerful enough in features and robust enough that entire systems can be prototyped completely in Python. Obviously, the same systems can be completed in traditional compiled languages, but Python's simplicity of engineering allows us to do the same thing and still be home in time for supper. Also, numerous external libraries have already been developed for Python, so whatever your application is, someone may have traveled down that road before. All you need to do is "plug-and-play" (some assembly required, as usual). There are Python modules and packages that can do practically anything and everything you can imagine. The Python Standard Library is fairly complete, and if you cannot find what you need there, chances are there is a third-party module or package that can do the job.

1.3.11 A Memory Manager

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

Because memory management is performed by the Python interpreter, the application developer is able to steer clear of memory issues and focus on the immediate goal of just creating the application that was planned in the first place. This leads to fewer bugs, a more robust application, and shorter overall development time.

1.3.12 Interpreted and (Byte-) Compiled

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

CORE NOTE: File extensions

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

1.4 Downloading and Installing Python

The most obvious place to get all Python-related software is at the main Web site at http://python.org. For your convenience, you can also go to the book's Web site at http://corepython.com and click on the "Install Python" link to the left—we have organized a grid with most contemporary versions of Python for the most platforms, with a focus, of course, on the "Big Three." Unix, Win 32, MacOS X.

to NULL.) An example of a function that takes input and has a return value is the `abs()` function, which takes a number and returns its absolute value is:

```
>>> abs(4)
4
>>> abs(-4)
4
```

We will introduce both statements and expressions in this chapter. Let us continue with more about the **`print`** statement.

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

Python's **`print`** statement (2.x) or function (3.x) is the tool for displaying program output to your users, similar to C's `printf()` and `echo` for shell scripts. In fact, it even supports `printf()`-style string substitution (using the `%` string format operator):

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

CORE NOTE: Dumping variable contents in interactive interpreter

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

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

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

Notice how just giving only the name reveals quotation marks around the string. The reason for this is to allow objects other than strings to be displayed in the same manner as this string—being able to display a printable string representation of any *object, not just strings. The quotes are there to indicate that the object whose value you just dumped to the display is a string. Once you become more familiar with Python, you will recognize that* `str()` *is used for* **`print`** *statements, while* `repr()` *is what the interactive interpreter calls to display your objects.*

The underscore (_) also has special meaning in the interactive interpreter: the last evaluated expression. So after the code above has executed, _ will contain the string:

```
>>> _
'Hello World!'
```

`%s` means to substitute a string while `%d` indicates an integer should be substituted. See Section 6.4.1 for more information on the string format operator. Also see the `str.format()` method added in 2.6.

The **`print`** statement can also redirect output to a file. Debugging in 2.0, the >> symbols precede a valid file, such as these 2 examples, one to standard error and the other to a log file:

2.0

```
import sys
print >> sys.stderr, 'Fatal error: invalid input!'

logfile = open('/tmp/mylog.txt', 'a')
print >> logfile, 'Fatal error: invalid input!'
logfile.close()
```

`print` is changing to a function `[print()]` in Python 3.0—also see Python Enhancement Proposal (PEP) 3105 (more in PEPs coming up in the Core Note on p. 54). This is a significant change, so starting in Python 2.6, you can start coding against the new function by adding this special import statement:

2.6-3.0

```
from __future__ import print_function
```

The syntax of the new function is:

```
print(*args, sep=' ', end='\n', file=None)
```

For example:

```
print('Fatal error: invalid input!', file=sys.stderr)
```

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

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

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

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

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

The `int()` function converts the string `num` to an integer so that the mathematical operation can be performed. See Section 6.5.3 for more information in the `raw_input()` built-in function.

CORE NOTE: Ask for help in the interactive interpreter

If you are learning Python and need help on a new function you are not familiar with, it is easy to get that help just by calling the `help()` *built-in function and passing in the name of the function you want help with:*

```
>>> help(raw_input)
Help on built-in function raw_input in module __builtin__:

raw_input(...)
    raw_input([prompt]) -> string

Read a string from standard input. The trailing newline is stripped.
If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise
EOFError. On Unix, GNU readline is used if enabled. The prompt string,
if given, is printed without a trailing newline before reading.'
```

CORE STYLE: Keep user interaction outside of functions

It's very tempting for beginners to put **`print`** *statements and* `raw_input()` *functions wherever they need to display information to or get information from a user. However, we would like to suggest that functions should be kept "clean," meaning they should silently be used purely to take parameters and provide return values. Get all the values needed from the user, send them all to the function, retrieve the return value, and then display the results to the user. This will enable you to use the same function elsewhere without having to worry about customized output. The exception to this rule is if you create functions specifically to obtain input from the user or display output.*
More importantly, it is good practice to separate functions into two categories: those that do things (i.e. , interact with the user or set variables) and those that calculate things (usually returning results). It is surely not bad practice to put a **`print`** *statement in a function if that was its purpose.*

2.3 Comments

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

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

There are special comments called documentation strings, or "doc strings" for short. You can add a "comment" at the beginning of a module, class, or function string that serves as a doc string, a feature familiar to Java programmers:

```
def foo():
    "This is a doc string."
    return True
```

Unlike regular comments, however, doc strings can be accessed at runtime and be used to automatically generate documentation.

2.4 Operators

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

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

Addition, subtraction, multiplication, division, and modulus (remainder) are all part of the standard set of operators. Python has two division operators, a single slash character for classic division and a double-slash for "floor" division (rounds down to nearest whole number). Classic division means that if the operands are both integers, it will perform floor division, while for floating point numbers, it represents true division. If true division is enabled, then the division operator will always perform that operation, regardless of operand types. You can read more about classic, true, and floor division in Chapter 5, "Numbers."

There is also an exponentiation operator, the double star/asterisk (`**`). Although we are emphasizing the mathematical nature of these operators, please note that some of these operators are overloaded for use with other data types as well, for example, strings and lists. Let us look at an example:

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

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

Python also provides the standard comparison operators, which return a Boolean value indicating the truthfulness of the expression:

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

Trying out some of the comparison operators we get:

```
>>> 2 < 4
True
>>> 2 == 4
False
>>> 2 > 4
False
>>> 6.2 <= 6
False
>>> 6.2 <= 6.2
True
>>> 6.2 <= 6.20001
True
```

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

Python also provides the expression conjunction operators:

```
and      or      not
```

We can use these operations to chain together arbitrary expressions and logically combine the Boolean results:

```
>>> 2 < 4 and 2 == 4
False
>>> 2 > 4 or 2 < 4
True
>>> not 6.2 <= 6
True
>>> 3 < 4 < 5
True
```

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

```
>>> 3 < 4 and 4 < 5
```

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

CORE STYLE: Use parentheses for clarification

Parentheses are a good idea in many cases, such as when the outcome is altered if they are not there, if the code is difficult to read without them, or in situations that might be confusing without them. They are typically not required in Python, but remember that readability counts. Anyone maintaining your code will thank you, and you will thank you later.

2.5 Variables and Assignment

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

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

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

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

Python also supports *augmented assignment*, statements that both refer to and assign values to variables. You can take the following expression . . .

```
n = n * 10
```

. . . and use this shortcut instead:

```
n *= 10
```

Python does not support increment and decrement operators like the ones in C: `n++` or `--n`. Because `+` and `-` are also unary operators, Python will interpret `--n` as `-(-n) == n`, and the same is true for `++n`.

2.6 Numbers

Python supports five basic numerical types, three of which are integer types.

- **int** (signed integers)
  - **long** (long integers)
  - **bool** (Boolean values)
- **float** (floating point real numbers)
- **complex** (complex numbers)

Here are some examples:

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

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

Moving forward, ints and longs are in the process of becoming unified into a single integer type. Beginning in version 2.3, overflow errors are no longer reported—the result is automagically converted to a long. In Python 3, there is no distinction as both `int` and `long` have been unified into a single integer type, and the "L" will no longer be valid Python syntax.

Boolean values are a special case of integer. Although represented by the constants `True` and `False`, if put in a numeric context such as addition with other numbers, `True` is treated as the integer with value 1, and `False` has a value of 0.

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

There is also a sixth numeric type, decimal, for decimal floating numbers, but it is not a built-in type. You must import the `decimal` module to use these types of numbers. They were added to Python (version 2.4) because of a need for more accuracy. For example, the number 1.1 cannot be accurately representing with binary floating point numbers (floats) because it has a repeating fraction in binary. Because of this, numbers like 1.1 look like this as a float:

```
>>> 1.1
1.1000000000000001
```

```
>>> print decimal.Decimal('1.1')
1.1
```

All numeric types are covered in Chapter 5.

2.7 Strings

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

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

You can learn more about strings in Chapter 6.

2.8 Lists and Tuples

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

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

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

Slice access to a tuple is similar, except it cannot be modified:

```
>>> aTuple = ('robots', 77, 93, 'try')
>>> aTuple
('robots', 77, 93, 'try')
>>> aTuple[:3]
('robots', 77, 93)
>>> aTuple[1] = 5
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

You can find out a lot more about lists and tuples along with strings in Chapter 6.

2.9 Dictionaries

Dictionaries (or "dicts" for short) are Python's mapping type and work like associative arrays or hashes found in Perl; they are made up of key-value pairs. Keys can be almost any Python type, but are usually numbers or strings.

Values, on the other hand, can be any arbitrary Python object. Dicts are enclosed by curly braces ({ }).

```
>>> aDict = {'host': 'earth'}     # create dict
>>> aDict['port'] = 80            # add to dict
>>> aDict
{'host': 'earth', 'port': 80}
>>> aDict.keys()
['host', 'port']
>>> aDict['host']
'earth'
>>> for key in aDict:
...  print key, aDict[key]
...
host earth
port 80
```

Dictionaries are covered in Chapter 7.

2.10 Code Blocks Use Indentation

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

When one is new to Python, indentation may comes as a surprise. Humans generally prefer to avoid change, so perhaps after many years of coding with brace delimitation, the first impression of using pure indentation may not be completely positive. However, recall that two of Python's features are that it is simplistic in nature and easy to read. If you have a strong dislike of indentation as a delimitation device, we invite you to revisit this notion half a year from now. More than likely, you will have discovered that life without braces is not as bad as you had originally thought.

2.11 `if` Statement

The standard **`if`** conditional statement follows this syntax:

```
if expression:
   if_suite
```

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

statements. You will notice that parentheses are not required in **if** statements as they are in other languages.

```
if x < .0:
    print '"x" must be at least 0!'
```

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

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

Python has an "else-if" spelled as **elif** with the following syntax:

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

At the time of this writing, there has been some discussion pertaining to a `switch` or `case` statement, but nothing concrete. It is possible that we will see such an animal in a future version of the language. This may also seem strange and/or distracting at first, but a set of **if-elif-else** statements are not as "ugly" because of Python's clean syntax. If you really want to circumvent a set of chained **if-elif-else** statements, another elegant workaround is using a **for** loop (see Section 2.13) to iterate through your list of possible "cases."

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

2.12 `while` Loop

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

```
while expression:
     while_suite
```

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