Python 3 for Absolute Beginners

Tim Hall and J-P Stacey

Apress®

Python 3 for Absolute Beginners

ISBN-13 (pbk): 978-1-4302-1632-2

ISBN-13 (electronic): 978-1-4302-1633-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matthew Moodie
Technical Reviewer: Duncan Parkes
Additional material: Dr. J. Burton Browning
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Managers: Beth Christmas and Debra Kelly
Copy Editor: Heather Lang
Compositor: LaurelTech
Indexer: BIM Indexing and e-Services
Artist: April Milne

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

Contents at a Glance

Contents

About the Author

Tim Hall currently provides front–line support for 64 Studio. He has also written newbie tutorials for Linux User and Developer magazine in between more mundane system admin and web authoring jobs.

Tim has released albums and performed as a musician and songwriter, both solo and in collaboration with other artists. He has been further honored as the holder of the Bardic chair of Glastonbury between 2005 and 2007. Tim uses Python as his main programming language, primarily as a means for creative ends, because it is easy to read and fun to learn.

J-P Stacey has been a senior developer at Torchbox Ltd since 2005, building and maintaining (among other things) Python, Django, and Drupal applications.

He organizes the Oxford Geek Nights and gets involved in tuition and seminars at Torchbox. In his spare time he reads and writes fiction and blogs, buys too much music, and tries not to startle Cotswold lions on his bicycle.

About the Technical Reviewer

Duncan Parkes has been coding in Python, both for work and for fun, for roughly a decade. He drifted into programming largely by accident after initially taking a doctorate in Combinatorial Group Theory, a branch of Pure Mathematics. As an ex-mathematician, he loves the clarity and simplicity of Python and needs a bit of persuading to code in anything else. After completing a technical review of this book, he joined Apress as an editor in the Open Source group. He currently splits his time between editing books and coding for mySociety, a charitable organization that runs most of the UK's best-known democracy web sites.

Duncan lives in Oxford, England, with his wife, Ruth. When away from his computer, Duncan enjoys playing the guitar very badly, cycling long distances on a Brompton folding bicycle, and fiddling with old cars.

His friends and colleagues have been known to run sweepstakes on how far these vehicles can get without the assistance of a breakdown service.

CHAPTER 1

Introducing Python

So, you want to learn programming. Welcome to one of the great adventures of the twenty-first century. Programming requires little in the way of specialized equipment; the software tools can all be downloaded for free off the Internet, and it can be practiced in the safety and comfort of your own home, without having to ask anyone's permission. This chapter will ease you in gently by introducing you to the software you will need to create your programs: a command-line interface, which allows you to use Python in interactive mode, and a text editor for writing scripts—nothing more complicated than that. I will also show you where to go to find help and documentation, so you can decode the sometimes-impenetrable jargon that seems to surround this, the geekiest of all technical disciplines. To begin with, you will need to make sure that you have a decently recent version of Python installed on your machine or follow the steps later in this chapter to install it (see "Choosing the Right Python Version" for a definition of *decently recent*). This chapter explains how to make sure that you have everything set up correctly and that you have suitable references at hand before you start your journey.

Python is an excellent language with which to learn programming. There are many reasons for this, but the simple explanation is that it's easy to read and fast to write; it doesn't take long to come up with working code that does something meaningful. Python has a very human-friendly syntax, which makes writing elegant code easy. The basic language is fairly simple and therefore easy to remember, and then it has an extensive library of predefined functions that you can use to facilitate most common computer tasks. Writing effective applications in Python can be as simple as playing with conceptual building blocks. It works really well for writing a little two-line application to perform some routine system administration task or to provide interactive functions on a web page, but it has enough power and flexibility to comfortably create much larger and more complex applications with graphic interfaces indistinguishable from the programs you are used to running from your computer's main menu. If you follow the suggestions laid out in this book about writing self-explanatory code, in several months, even years, you will be able to come back to your programs and see immediately what they were supposed to do and what your original intentions were; this makes maintaining programs much simpler too.

OK, let's make sure your system is ready for you to start running Python programs.

Running Python on Various Systems

Python can be installed and run on most operating systems including Windows, Mac OS X, or OS/2, Linux, and Unix. If you are running Mac OS X or a GNU/Linux system, you will probably have it installed by default. I would recommend using a system of this kind, which already has Python set up as an integral part. The book was written using a Debian GNU/Linux system, and therefore, the examples will

work exactly the same on any Debian-based system, such as Ubuntu. In fact, the differences between Linux variants are few, so you'll be equally at home with other distributions.

Installing on Windows

If you are using Windows, you will probably have to install Python and configure certain settings correctly before you can get to grips with the examples in this book. For that, you will need to refer to the specific instructions for your operating system on the following Python web pages:

- `http://wiki.python.org/moin/BeginnersGuide/Download`
- `http://www.python.org/doc/faq/windows/`
- `http://docs.python.org/dev/3.0/using/windows.html`

First, you need to download the official installer; alternative versions for Itanium and AMD machines are available from `http://www.python.org/download/`. You should save this file, which will have a `.msi` extension, somewhere you'll be able to find again easily. You can then double-click this file to start the Python installation wizard, which will take you through the installation. It's fine to accept the default settings if you're not sure of any answer.

Installing on Other Systems

You may choose to install Python on other systems, particularly if you want to take advantage of newer versions. For Linux and other Unix-like systems, the installation instructions are here:

- `http://docs.python.org/dev/3.0/using/unix.html`

  If you're using OS X, your instructions are here:

- `http://www.python.org/download/mac/`
- `http://docs.python.org/dev/3.0/using/mac.html`

Choosing the Right Python Version

You will find the different installers include a number after the word *Python*. This is the **version number**. When I started writing this book, those numbers ranged from 2.3.7 (old but still usable) through 2.5.2 (the previous stable version) to 3.0 (the new version about to be released). At the same time as version 3.0 was released, the Python team also put out version 2.6, which is an upgraded version of Python version 2 for people who want (or need) to stick with the old way of doing things but still want to benefit from general fixes and some of the new features introduced in version 3.0.

The Python language is continuously evolving; version 3.0 has become the norm and has evolved into version 3.1.1. The *new* version, which I'll refer to as version 3.0 because all 3.x versions are simply refinements on the original plan of 3.0, includes several changes to the programming language that are incompatible with version 2.x (*x* is any number you like), which I will refer to in the rest of this book as the *old* version. Most of the language is the same, however, so the differences between the versions of Python will be noted in the text as those subjects are covered. Examples in this book are for Python 3.0 except where noted.

Further information about the latest version of Python 3.0 (also known as Py3k or Python 3000) is available at `http://www.python.org/download/releases/`.

There may be some differences running Python on other operating systems, which I will do my best to point out where relevant. Otherwise, the examples of code will work the same way. This is one of the many good points of Python. For the most part, this book will concentrate on the fun part—learning how to write programs using Python. The official Python documentation is plentiful, free, and well written, and you should read it alongside this book. It is available on at `http://www.python.org/doc/`.

Understanding the Jargon

Throughout this book, I will be doing my best to demystify, clarify and explain the various bits of technical terminology that you will frequently encounter during the process of learning the art of programming. Technical terms and jargon words that need further explanation will be highlighted in **bold** the first time you encounter them. Any terms not immediately explained in the chapter will be covered in the "Jargon Busting" section at the end of each chapter. You may also wish to study one of the most famous resources on the Internet, the Jargon File, which is part of the *New Hacker's Dictionary* (available at `http://catb.org/~esr/jargon/`).

As an example, here's the definition of the noun "code" from the Jargon File:

"The stuff that software writers write, either in source form or after translation by a compiler or assembler. Often used in opposition to 'data,' which is the stuff that code operates on. Among hackers this is a mass noun, as in 'How much code does it take to do a "bubble sort"?' or 'The code is loaded at the high end of RAM.' Among scientific programmers, it is sometimes a count noun equivalent to 'program;' thus they may speak of 'codes' in the plural. Anyone referring to software as 'the software codes' is probably a 'newbie' or a 'suit'."

For comparison's sake, here's the definition of the verb "code":

"To write code. In this sense, always refers to source code rather than compiled. 'I coded an Emacs clone in two hours!' This verb is a bit of a cultural marker associated with the Unix and minicomputer traditions (and lately Linux); people within that culture prefer v. 'code' to v. 'program,' whereas outside it, the reverse is normally true."

The noun "program" is defined as being:

- "A magic spell cast over a computer allowing it to turn one's input into error messages"
- "An exercise in experimental epistemology"
- "A form of art, ostensibly intended for the instruction of computers, which is nevertheless almost inevitably a failure if other programmers can't understand it"

As the information contained in the Jargon File is more entertaining than strictly informative, I will be providing some simpler definitions at the end of each chapter.

Learning While Having Fun

Having fun is often underestimated in the realm of technical disciplines. We learn best and are most productive when we're having fun. Developing software using Python is often an engrossing and enjoyable experience, partly because you can test out your changes as soon as you have made them without having to perform any intermediary steps.

Python also deals with many background tasks, the household chores of programming, so you don't have to bother with them as you might in languages like C. This kind of immediate gratification makes it easier to stay in the creative flow and keep being inspired.

Python is also easy to read, being closer to natural language than many other programming languages, so you spend less time trying to decode what you have just written, which in turn means more time to think about how you could improve and expand your work.

The other great thing is that you can use Python for pretty much any task you can think of, be it large or small, simple text-driven script or major graphical application. It does have its limitations, but by the time you come up against them, you will already have become a competent programmer and will be able to make an informed choice about where to go next.

Introducing the Interactive Shell

Although this book is primarily aimed at showing you how to write stand-alone scripts and **applications**—that is, fully fledged programs that don't require anything other than having Python installed for them to work—Python also works in **interactive mode**, which allows you to enter Python commands using a command-line interface. This is a great resource for immediate Python gratification. You need to locate and start up your computer's command-line interface, sometimes known as the terminal or terminal emulator, or on machines running Windows, the DOS prompt. This will provide you with a **command line** at which you can start typing commands. Interactive mode can be started up by typing `python` at the command line. If Python is properly installed, you should see something like this:

```
$ python
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12)
[GCC 4.2.3 (Debian 4.2.3-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The details may be different according to your operating system. The first line shows the Python version number and the compilation date of the version you have installed. The last line (`>>>`) is your Python prompt. This indicates that the Python interpreter is running and waiting for your input. The Python interpreter is the program that reads Python programs and carries out their instructions. Interactive mode can be useful for testing out individual commands line by line, experimenting with new ideas, and getting help (try typing `help()`, and don't forget the parentheses; they are important). To get out of the interactive Python interpreter on a Linux system, you can use the `quit()` or `exit()` functions, or else you must hold down the Ctrl key and the D key at the same time (in Windows, you need to use Ctrl+Z and then press Enter).

Choosing to Code with a Text Editor

My preferred method of creating Python scripts is to use a plain text editor, with a few basic programming features. Programs like notepad (especially notepad2/++), BBedit, gedit, NEdit and Kate are ideal for this task. A variety of editors are available that offer specific enhancements for programmers, such as syntax highlighting, which is useful for showing coding errors immediately as you type. Old-school hackers may prefer to use Vi or Emacs. Python does not dictate what software you use to create the code; the choice is up to you. Do not attempt to use word-processing software, such as Word or OpenOffice for this task; it will mess you up badly. Be warned!

Choosing to Code with an Integrated Development Environment

An integrated development environment (IDE) is a graphical interface with lots of useful features designed to make programming with Python even easier. You still have to type the code in the same way, but you can do all your coding using a single application, and these programming **environments** can provide some useful shortcuts and reminders. There are now several Python-specific IDEs. Popular applications include IDLE (which is bundled with Python itself), Eric (you may notice a theme here, nudge nudge, wink wink, say no more...), Geany, DrPython, and SPE. Python is also supported by more general programming environments like Bluefish and a whole host of others. This book doesn't cover the use of IDEs or any of the alternative distributions of Python, because each of these approaches would require a chapter unto themselves, at the very least. However, the examples contained in this book will still work if you *do* choose to explore these options. This book will take the simplest approach, using tools that come as standard with most operating systems; that is, a text editor and the Python Interactive Shell.

Getting Started with Programming

The process of writing a new program usually starts with a problem. Before you can code anything, you need to have an idea of what you want to create, the problem you wish to solve, and a fairly good idea of how to solve it. Chapter 2 will take you through the process of designing software, known as the **software development cycle**. Usually, this is something that you have to learn separately, as most programming manuals go straight into the details of the language, assuming you will have studied program design at college or in the process of learning some other programming language. This can lead you to create code that is difficult to read and understand and makes fixing problems later a nightmare. Understanding the principles of software design can dramatically speed up the process of creating new software and help ensure that important details are not missed out.

In Chapter 3, you start coding those designs in Python. You will learn about constructing the basic units of data, words and numbers, and how to combine and manipulate them. In Chapter 4, I will show you how to compare values, make choices, and deal with errors. The subsequent chapters go through the process of refining those designs, breaking them down into manageable chunks, and expanding your understanding of the Python programming language so you can turn your ideas into fully fledged, functional computer programs.

Creating Your Own Help Files

Python is described as **self-documenting**. This doesn't mean that it will write your user manual for you, but you can add blocks of text, known as **documentation strings**, to your own code, and these strings will then show up in an online help utility. This information can easily be turned into web pages to provide a useful reference. I will show you how to do this in Chapter 2; it is one of the unique features of Python and is worth learning how to use early on.

Jargon Busting

This chapter introduced several technical terms with which you may not be familiar, these are defined as follows:

- *Command-line interface (CLI)*: This mechanism is for interacting with a computer by typing in commands to a text-based interface such as the terminal or DOS prompt.
- *Compile*: This process turns a sequence of instructions in programming language, which is readable by humans (in theory) into machine code, which can be directly executed by the computer. Python does this automatically in the background when you run your script. In some other languages, this has to be done as a separate step before you can execute the program.
- *Documentation strings*: This block of text on the first line of a *function*, *module*, *class*, or *method* definition is also called a "docstring." Docstrings are used by Python's built-in tools for automated testing and to provide help documentation. See Chapter 2.
- *Interpreter*: This program turns a sequence of instructions in a script into something that a computer can execute on the fly, effectively compiling the code as the program is running rather than as a separate step.
- *Natural language*: This is language that humans speak, as opposed to a programming language.
- *Operating system*: This is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer. The operating system communicates between the applications that are run on the machine and the machine's hardware components.
- *Prompt*: This visual cue tells the user to input something.
- *Script and application*: The terms "script," "application," and "program" are used somewhat interchangeably in this book, though there are some subtle differences in meaning:
    - *Script*: This short but complete piece of code is intended to be run using an interpreter. A script will exist as a text file that you can read.
    - *Application*: This refers to any applied use of a programming language. It usually means a complete, packaged-up, and usable piece of software. It is the preferred term among Linux/Unix users.

    - *Program*: This refers to the sequence of instructions involved in a script or application; it is also used as a general term to describe an executable computer file.
- *Software development cycle*: In this structured approach to developing software, each stage can be fed back into previous stages. See Chapter 2.
- *Version number*: This number corresponds to a new development in a piece of software. These numbers may refer to major or minor changes and are generally assigned in increasing order. The method used for assigning these numbers is referred to as a *versioning system*.

Summary

This chapter has explained some of the technical terms that you will encounter as you study the art of programming and has introduced you to some sources of further information to help you as you grow in understanding.

So far, you have made sure that the right version of Python is installed on your system and you know how to get to the Python command prompt. You should have selected a text editor to use for writing and saving your scripts, but you haven't written any code yet. In the next chapter I will show you how to start designing programs from the initial seed of a problem that you wish to solve.

CHAPTER 2

Designing Software

If you are anything like me, you impatiently started reading this book at Chapter 3, became really confused about halfway through, and decided to start at the beginning and read it all properly this time. This chapter explains the bit that most manuals leave out: *how to actually turn your problem into a working program*. The first stage of the software development cycle is the creation of a program design. In this chapter, I'll demonstrate how to break down your ideas into manageable chunks, understand each problem that you are trying to solve, build up a sequence of actions, keep track of what you've done, and above all, how to go about the constant refinement and simplification of your ideas that will lead to an effective solution that is easy to understand. I'll be introducing you to pseudocode, the software design language, and showing you how to turn it into Python documentation strings to create a framework around which you can grow an effective and maintainable piece of software.

Your first challenge, which I'll be using as an example for this chapter, is to get the computer to respond to a command and print out some kind of response—fairly basic stuff, which you could probably guess at a solution for. However, it's always good to start as we mean to go on, so I'm going to break this down in terms of the software development cycle. In order to get off the starting blocks, it is necessary to identify the problem through a process of asking the right questions and then begin developing a solution, initially by creating a wish list of the things you want your new program to do and getting feedback from potential users. You will use all this information to create your initial program design.

In the next part of this chapter, you will be able to start turning that design into working code. As soon as the design is coded, you will want to begin refining and testing it, starting with defining the results you expect to get and then figuring out how to detect and correct errors.

In the final stage of the software development cycle, you will discover how to use docstrings to document the solution and see how this then feeds back into the next round of the development cycle.

Designing Your Own Software (Why Bother?)

There are many reasons that you might want to write your own software, the chief one being that the applications you have available don't do what you want in the way that you want. You may already have an idea of something you wish to achieve, which led you to pick up this book in the first place. Perhaps you need a solution for work; maybe you have a killer game idea that you'd like to realize; you might want to organize some data in a web page or perform some system administration task. You might want to control a gadget you just built or automate any number of repetitive jobs. Possibly, you already have the program you need, if only it would do this one extra thing, or maybe you just enjoy the idea that writing your own software is possible. I'm not going to try to tell you why you should want to get involved in this peculiar sport; I'll assume you have worked that out for yourself already.

Let's start by asking some questions. Never be afraid to ask questions. Why bother doing all this? Why not just go straight ahead and start writing code? Coming up with a clear design first, along with a clear idea of what answers you expect to see, cuts down on a lot of wasted effort and makes your programs easier to document, troubleshoot, and maintain. It is important to remember that you are trying to make life easier—not more complicated—for yourself and others.

Asking the Right Questions

"Why bother?" is an important programming question that should be applied to every project, module, function, or line of code before you begin. There is a strong possibility that someone, somewhere has already written the software you need, or at least something very similar, that you could convert to your needs with the addition of a couple of lines of code.

Close on its heels comes the next question, "Isn't there a simpler way to do this?" Laziness is the programmers' ally; you probably won't have to write very much from scratch, and you won't need to learn everything by heart before you begin. Generally speaking, you can look up most things as you go along. The art of programming has much more to do with figuring out a logical procedure to get something done than memorizing lines of code; indeed, one of the most common practices involved in coding is simplification. Your first task is about as simple as it gets, but as the book progresses, you will be facing much more complex problems as well as coming up with ideas of your own.

Using Python to Solve Problems

Python is similar in style to **pseudocode**, the language programmers use to design software, with the difference that you can run it on your computer and actually get results. It's easy to read (being close to English), fun to learn, and doesn't require compiling, that is, you can run your script as soon as you have saved it without having to perform any intermediate steps. Python is also a useful language in that it can be used to deal with a wide range of problems with relative ease, and it has a good selection of preconstructed modules to deal with many of the more common computer tasks. I find it similar to playing with conceptual building blocks: the elements of the language are fairly simple to grasp, but they can be combined in pretty much any way you can think of to create apparently complex and graceful structures.

One of the other great features of Python is that it is self-documenting. You still have to write the documentation yourself, but you can include blocks of help text, known as documentation strings or docstrings, within the structure of the program itself. I will be using this particular feature to keep all the design ideas together in one file at the outset and allow us to think on paper. Subsequently, you will be able to look back over old code and know what you were trying to achieve long after you have forgotten why you wrote it in the first place. Eventually, these docstrings will be used to communicate how your program works to other programmers.

Identifying the Problem

Before you start coding anything, you need to have an idea or a problem that you wish to solve. It is quite useful to start with a familiar repetitive task that you suspect could be performed more efficiently using a simple script. Your first problem is to find out how to communicate with your computer using Python and how to get your computer to communicate back. Using this as an example, I will briefly explain the stages of the software development cycle that most programs will go through in course of their development. You don't have to slavishly follow all the suggestions in this chapter, but you may find that they provide a useful structure for developing more complex applications and to get you back on track if

you find yourself getting lost. It is also useful to be able to design your program before having to think about the details of coding it in a formal computer language.

The first stage of writing any new program is to grow a design out of the initial problem, a sequence of actions that will calculate the correct result and create the required output. Once the idea has passed the "Why bother?" test, it is necessary to create a simple framework on which to hang the various bits of code that you come up with. Without such a framework, it is often difficult to know where to begin and virtually impossible to construct anything but the simplest of scripts without it getting messy very quickly. The great news is that no specialized knowledge is required at this stage. The process can be started off by answering a few simple questions:

- What do you want the software to do?
- Who will be using it?
- What system will users be running it on?

What Do You Want the Software to Do?

Open a text editor, and write out what the problem is that you want to solve and, very briefly, what you actually want the software to do. A simple text editor will do for this task (see Chapter 1). It's best to start with some simple jobs that you already know computers can do, like printing out a daft message, adding up some numbers, or updating the prices of items in a catalog. Classically, half the battle is identifying the problem, and once you have done so, the solution will begin to become clear. Save the file as plain text (with the `.txt` extension) and give it the title of your program-to-be. The first program you're going to write later in this chapter will be called Hello World! It will be designed to solve the problem of getting your computer to print out a friendly, affirmative or daft message when you ask it to, so you may as well save this as `hello_world.txt`.

■ **Tip** You might want to create a new directory to store all your Python experiments.

Who Will Be Using It?

Next, you need to specify who will be using the application, even if you're just doing it for your own personal gratification and no one is ever going to see the output. Write this down.

What other people will be using the program?

- Friends
- Relatives
- Work mates
- Registered users
- Anonymous clients that you're never likely to meet

How much interaction do you expect them to have with the application?

- Clicking a single button
- Using a text interface
- Answering a complex form

How computer-literate do you expect them to be?

- Will they need to learn the ropes?
- Should the application be self-explanatory?

What System Will Users Be Running It On?

It may make a difference whether your program will be running on a Windows, OS X, GNU/Linux, or some other operating system. It may be running on one kind of operating system and be accessed by clients using a variety of other systems, like with most Internet applications. It may use a standard computer keyboard and mouse for input or have some custom-designed interface. What other software will users need to have installed in order to be able to use your script?

All these questions may have a bearing on choices you make later on in the development process. Answer them now, and you will be better prepared for the challenges to come. At this stage, keep your answers simple and to the point; you want to produce a concise overview that neatly summarizes what the software is intended for to begin with. If you can't answer some of the questions yet, that's fine too. Later in this chapter, you will break this down into detailed steps, but before getting on to that, it's your turn to start asking questions.

Creating Your Wish List

In this first stage of defining what you want the software to do, it can be useful to create a wish-list of the specific things you want this software to achieve. This requires analyzing the procedure that might be required to perform the job in hand a little more deeply. Initially, it is quite useful to examine a task that you know how to do. It could be something that is currently not automated or computerized. The following sections outline some suggestions of things you could do to generate your wish list.

Talking to Potential Users

Interview potential users of your application to find out how they currently perform the task you have in mind. Ask them which parts of the procedure could be usefully automated and which factors they want to be able to control. Find out which aspects are confusing and how they overcome existing difficulties.

Some people find it useful to create a short questionnaire, even if only you will be answering the questions; sometimes, this can help to gain some perspective on the problem at hand. In the questionnaire, you can ask more detailed questions of what your potential users expect the application to look like and how they expect it to behave, what file formats they expect to use to read in information from and save it out to, particularly covering any unique or special considerations that you can imagine. File formats could include plain text, XML, and HTML for text documents, or your users might have images saved as JPEGs or PNGs or WAV audio files that they want to convert to MP3s. A file's extension

(the letters after the last dot in the filename) usually provides a clue as to what format a file is in, though there are other ways of finding out this information.

The purpose of this exercise is to find out how this job is usually done and what information is needed before you begin. An example calculation with realistic examples of the type and range of information you are likely to be dealing with is invaluable at this stage. A written procedure could also be useful.

Watching Users Perform the Task at Hand

If possible, watch users carry out the task, and note down any points of interest. Again, you don't need to go into a huge amount of detail at this stage, and you need to cover only the information that is relevant to your software project and your potential users. It can be useful to put yourself in the user's position and walk through the process yourself. Another technique is to collect up all the pieces of paper that go into or come out of the process and use them to create a storyboard. This stage of the design process becomes much more relevant in larger projects, but it's worth getting into the habit of doing it now.

Compiling User Stories

A **user story** consists of one or two lines in everyday language that specifies something that the user wants from the software. The idea is that stories should be brief enough to fit on an index card, on a sticky note, or into one cell of a spreadsheet. A user story should indicate the role of the user, what that user wants the software to do, and what benefit might be gained from it. User stories are prioritized by the user to indicate which features are most important and are then used to work out an estimate of the time it will take to develop those features by the developer. The developer may then create use cases out of these stories and tests to determine whether the requirements have been fulfilled. **Use cases** are longer, more detailed descriptions of features that are required by specific users in specific situations. You just need enough information to work out the basic functionality that will be required from the program you are about to write. Try to keep it concise.

Devising a Solution

Armed with all this information, let's return to the text file to complete the first stage of the software design. The design stage can often be the most time-consuming part of the programming process. It consists of several stages in which the output from one stage becomes the input to the next. Often, the output from one or more stages is fed back through the same process several times. This is known as **recursion** when it happens inside a program. The whole process of software design has a similarly recursive nature. To celebrate this fact, let's go back to the initial problem.

Understanding the Problem

This time around, you're going to fill in a little more detail and turn your text file into a very simple **user requirements document**. In a professional situation, this would become the agreement between the programmer and the client who is paying for the software to be written. For now, you're just going to use it as a means to keep a record of the original purpose of this software. Make sure that the problem is stated clearly and that you have identified the issue you wish to resolve correctly. Often, there can be hidden subtleties in the problem that look straightforward when written down in your natural language

but require additional clarification when you come to turn them into programming language, or code. If the problem is not defined well enough to make a decision at that point, you will need to return to this stage and clear up the details before you can make any further progress.

Knowing What the Software Needs to Do

The things you want the software to do are known as **functional requirements** in the trade. Simple examples include printing a message, adding up some numbers up, or sending an e-mail form. You might have additional details to add; for example, maybe the user must be able to input some text, numbers larger than 200 cannot be entered, or the update must be performed over the Internet. Functional requirements specify how a system should *behave*. For your first exercise, Hello World!, you will be asking the software to print out a simple message, so your first functional requirement is that the computer should print out a message.

Considering Other Needs or Limitations

You may also want to include other needs or limitations of the software, which are called nonfunctional requirements. **Nonfunctional requirements** define other constraints and qualities required of the software, such as accessibility, usability by the intended users, cost-effectiveness, compatibility with other systems, security, and performance. You probably won't need to worry too much about such things if you are just programming for your own amusement.

Defining Acceptable Results

You need to give some thought as to how the software will be tested. Early on, it will be enough to simply run the program and see if it works, but as your scripts grow in complexity, you will find it useful to formalize the methods used for testing. It may be worth finding or creating some data to use as an example for testing purposes, and you may find it useful to create a table of values that you expect to find at various points in the process. In Chapter 11, I'll be showing you how to create built-in test suites, which can make a large part of the process automatic. It's generally considered good practice to write your tests before you do any coding, but it often happens that the tests and code actually get written at the same time. For now, a note of the expected output will do fine.

Considering Maintenance

Any piece of software that is going to be used more than once will require some degree of maintenance. Issues that are worth thinking about early on are how the software might need to change in future and how to keep track of issues. You might also consider how the application is going to be kept in working condition while you're integrating new possibilities and responding to new challenges. In many cases, the application will be maintained by the person who wrote it, but it is good practice to organize your project and write your code in a way that makes it easy for someone else to understand what you've done and contribute fixes and new features. One day, you might want to pass on the responsibility of keeping it up to date to someone else.

To facilitate maintenance, the software should include some means of contact, such as the author's e-mail address so that users can get in touch if there are any problems. For now, just note your e-mail address. Later on, I'll show you how you can integrate this information into your code, so you don't have

to retype this information every time you want to use it. *Don't Repeat Yourself* is a very sensible principle of programming that states that *every piece of knowledge must have a single, unambiguous, authoritative representation within a system*. No piece of information should ever have to be typed in twice—if you find yourself doing this, it's a sure sign that some part of your script needs redesigning.

Great! You should now have a little text file that looks something like Listing 2-1.

Listing 2-1. *hello_world.txt*

```
Problem: Get the computer to output a message.
Target Users: Me
Target System: GNU/Linux
Interface: Command-line
Functional Requirements: Print out a message.
                         User must be able to input some text.
Testing: Simple run test - expecting a message to appear.
Maintainer: maintainer@website.com
```

Breaking Down the Solution into Steps

Next, you start the process of turning your wish list into a program. In this stage, you will design a logical flow of actions, which will hopefully produce the desired outcome. First, you need to turn the text file into something the Python interpreter can understand by saving it as a Python (`.py`) file. Use the Save As function in your text editor to save the file with the same name, but this time with a `.py` extension. This example will become `hello_world.py`.

Now, the interpreter needs to know what this text is; otherwise, it will attempt to interpret the first thing it finds as a command. There are a variety of ways of marking out blocks of text as comments or text strings so that the interpreter doesn't attempt to execute them.

Comments: The hash symbol, #, is used to mark comments. Comments are completely ignored by the Python interpreter. All text between the # sign and *the end of the line* is ignored. This formatting is a convenience so you, the programmer, can write little notes to yourself to remind yourself what this part of the code is supposed to do or to flag parts of the code that need attention. These comments will only ever be seen by people who are actually reading the code. If you are writing software as part of a team, comments are a great way of communicating your thoughts and intentions to the other programmers. Good code can often contain more comments than actual lines of code—don't be tempted to remove them once you've coded the design. In a month's time, you are likely to have completely forgotten what this script was intended to do, and you will be thankful for these little reminders.

Text strings: Text strings are delimited by quotation marks. *Delimited* means that the text is enclosed between a matching pair of the specified characters. Python uses a variety of forms of quotation marks, principally `'single'` and `"double"` quotation marks. There is a subtle difference between the two forms, which I'll cover in detail in the section on text strings in Chapter 3. The main reason for the different forms is to allow nesting of quotes. **Nesting** means putting one inside the other like this: "What on earth does 'nested delimiters' mean?" she asked.

Like comments, quotation marks only work if the text is all on one line. Fortunately, there are a variety of ways to get the interpreter to ignore line breaks. In this example, I shall use the

technique of *triple quoting*. Triple quotes can be either `'''single'''` or `"""double"""`, and they allow us to mark out an entire block of text along with all the line breaks. I'm going to use triple quotes in a specialized way to create a documentation string

The block of text in Listing 2-1 is going to form the basis of your documentation, so let's enclose it in triple quotes. A text string that occurs as the first statement of any module, function or class becomes a Python docstring, which makes the text available as part of the built-in documentation for this application. It won't do anything yet, but Listing 2-2 is now a legal Python file; the interpreter won't complain if you try to run it.

Listing 2-2. *Creating a Framework for hello_world.py*

```
"""
Problem: Get the computer to output a message.
Target Users: Me
Target System: GNU/Linux
Interface: Command-line
Functional Requirements: Print out a message.
                        User must be able to input some text.
Testing: Simple run test - expecting a message to appear.
Maintainer: maintainer@website.com
"""

# This is just a comment
```

Organizing the Tasks into Steps

Now, you are prepared to start designing your program. The program design is initially sketched out in pseudocode, which is the design language that can be used as an intermediary step to turn the user requirements document into a piece of Python code. There is nothing special about the form of this language; your own natural way of expressing ideas is exactly what is needed in this situation. Pseudocode is simply a series of logical statements in your own words that describe the actions that the program needs to take in order to produce the desired result.

If you are working as part of a team, your company may have a standard way of writing pseudocode, but you don't need to worry about such considerations while you are learning. Over time, you are likely to find that your pseudocode naturally adopts a Python-like syntax. Initially, the important thing is that you can express your ideas clearly without having to worry about the rules of the language yet. The pseudocode is presented in comments # in the order you want it to happen, as shown in Listing 2-3.

Listing 2-3. *Pseudocode Example*

```
# Do the first thing
# Do the next thing
# Do the last thing
```

Each line is commented out so that the interpreter knows not to execute any of these statements. This is your **top-level design**, which is a general description of how you are going to solve this problem. Each statement is referred to as a *step*. When a step is performed, it is said to be *executed*.

Using Indentation and Subgroups

Python is very strict about indentation. Python regards any space or tab at the start of a line as an indentation. Every indentation means something. Without going into too much detail at this stage, pay careful attention to how indentation is used in the examples. The most important thing to remember is not to mix tabs with spaces, as the interpreter thinks these are two different things (even if they look the same to you).

In your text editor's Preferences window, set it to insert four spaces instead of using tabs, and you'll avoid so much future grief by doing this. It is always best to keep your designs clear and readable: use blank lines to separate the different sections of your pseudocode, and use indentation to show how statements are grouped together. As long as each line is a comment preceded by a #, you are free to format it how you like.

■ **Caution** The use of indentation in your actual code follows very specific rules. This is one aspect of Python that takes a little bit of getting used to. I will explain the rules fully in the chapters on control flow statements and functions.

Refining Each Line of Your Pseudocode Until a Solution Is Reached

Each line is then further refined by breaking it down into steps in the same way, until each step comprises a single action and you have a complete design. Subsections should be indented by four spaces, as shown in Listing 2-4.

While you are working in pseudocode, your groupings and indentation can be quite arbitrary; all you are doing is providing a framework for your code to live in. As you learn more about Python, your pseudocode will naturally adopt more Pythonic structures. The process of refining your designs will be covered in greater detail from Chapter 4 onward.

Listing 2-4. *Indentation of Subsections*

```
# Do the first thing.
    # Set the first value.
    # Check whether the first value is some text or not.
    # Print the first value
```

Save it! Every time you sit back from the screen and wonder "What next?" you should save your work. Get into this habit if you haven't already; it will save so much brain-ache and frustration. You may also wish to copy your top-level design along with the finished design back into the original text file, so that you can refer to it when you have finished the coding stage.

Coding the Design

Now, you are ready to start turning your design into proper Python code. Great!

The data your software will be manipulating will take various forms; these values are assigned to things called **variables**, which you can think of as a box or a bucket that contains a single piece of information, often a word or a number. Some people find it useful to think of them as slates that can be written on with chalk. The important thing is that the information in the container can be changed by putting a new value into it; this automatically wipes out the old value and replaces it with a new one. The fact that these values can be changed gives rise to the term *variable*.

There are two stages to creating a variable, the first is to create the container and stick an identifying label on it: this is called **initialization**. The second is to put a value into it: this is called **assignment**. In Python, both these things happen in the same statement. Assignment (and therefore initialization) is performed using the = sign like this: `variable = value`. One of the specific features of Python is that, once a value has been assigned to a variable, the interpreter will then decide what sort of value it is (i.e., a number, some text, or some other relevant piece of information).

Each variable is referred to by a name, known as an **identifier**, which is rather like a name tag that identifies the variable to the rest of the program. It's a good idea to choose names that give an idea of what sort of values they represent. Python will regard any word that has not been commented out, delimited, or **escaped** in some other way as either a command or an identifier; for this reason, you need to be careful to avoid choosing words that are already being used as part of the Python language to identify your new variables. Details of which words you cannot use will be covered in the next chapter.

Turning the Design into Executable Code

Now, you have to come up with a strategy for solving the problems you have agreed to tackle.

First, you want to print out a message, and you want the user to be able to input some text. OK, so you're expecting your message and users' text to be strings of text. It would be a very good idea to add this information to the `Testing` section of your docstring at the start of the file. Text strings are represented by enclosing the text in quotes.

Now, you just need to know the function that prints things out to the screen, which is called `print()` in Python. So your first problem translates fairly easily into executable code as you can see in Listing 2-5. You can print any text you like; "Hello World!" is the default.

Listing 2-5. *First Steps for hello_world.py*

```
"""
Problem: Get the computer to output a message.
Target Users: Me
Target System: GNU/Linux
Interface: Command-line
Functional Requirements: Print out a message.
                         User must be able to input some text.
Testing: Simple run test - expecting a message to appear.
                         - expecting: message == input text
Maintainer: maintainer@website.com
"""
```

```
# 1. Print out a friendly message
print("Hello World!")

# 2. Input some text

# 3. Print out the text we just entered
```

This script can be run by typing

```
$ python hello_world.py
```

on the command line. You did remember to save it, didn't you?

The only line in this script that is actually executed by the interpreter is `print("Hello World!")`. Everything else is either ignored as a comment or assumed to be a docstring, in the case of the block of text at the beginning. If you change the text between the quotes in the call to the `print()` function, the Python interpreter will print out whatever you tell it to.

Well done! You just wrote your first Python program. It's about as basic as can be, but you can now bask in the satisfaction of having got Python to do something.

Further Refining the Design

Constantly editing the script to get it to say something different quickly becomes rather tedious. Wouldn't it be great if the program were interactive? In that case, you would need to find a way to get some user input.

The quick-and-dirty method for doing this is to use the built-in `input()` function. This function takes one argument, a text string that is printed to the screen to prompt the user for input. The function then returns the user's input. All requests for input need a message to tell the user that input is required; this is known as a **prompt**. I assign this input to a variable called `some_text`. Then, I can use the `print()` function to print out the user's input. Notice that this time `some_text` isn't in quotes, because I want the value contained in the variable called `some_text` rather than the literal text string `"some_text"`. Variable names aren't allowed to contain spaces, so you'll notice that I've replaced the space with an underscore. It's worth remembering that variable names can't contain dashes either, because Python will assume that the dash is a minus sign. Details of how to name your variables will be covered fully in Chapter 3.

Using functions

Most of the actual work in programs is performed by functions. You can spot functions quite easily, as they usually have parentheses, (), immediately following the function name. Any information that the function needs to work on is placed inside the parentheses; these pieces of information, called arguments, have to be separated by commas if there is more than one of them.

Again, text strings have to be enclosed in quotes or else the interpreter will treat the string as if it is a variable. Functions often return a value of some kind, so it is usual to catch this value by assigning it to a variable like this: `variable = function()`.

Don't forget the brackets after the function name.

Functions are covered in much greater detail in Chapter 6.

Your final script should look something like Listing 2-6.

Listing 2-6. *A Refined hello_world.py*

```
"""

Problem: Get the computer to output a message.
Target Users: Me
Target System: GNU/Linux
Interface: Command-line
Functional Requirements: Print out a message.
                         User must be able to input some text.
Testing: Simple run test - expecting a message to appear.
                         - expecting: message == input text
Maintainer: maintainer@website.com
"""

# 1. Print out a friendly message
print("Hello World!")

# 2. Input some text
some_text = input('Type in some words: ')

# 3. Print out the text we just entered
print(some_text)
```

■ **Note** The behavior of the `input()` function has changed in Python 3 . You need to use the `raw_input()` function to get the same effect in older versions.

Testing the Design

Now, it's time to save the file again and test it by running the script with the Python interpreter as before. If you get any error messages at this stage, you probably made a mistake typing the script. Common errors include missing quotation marks or spaces in the wrong places. In this case, you will need to go back to the coding stage and correct these errors before you test the code again.

The testing stage involves making sure that the design does what it is supposed to and that it matches the specification that you started with. Compare the actual output from running the program to the output you said you were expecting in your original specification. Are they the same? If not, why not? What could be done better?

Apart from the initial design stage, you will probably spend most of your time cycling around this testing stage. Don't be tempted to view error messages as annoyances or irritations: pay great attention to what they tell you, as they can be one of the most useful aids on the path of learning Python. On some

occasions, you will want to deliberately produce error messages to prove that something is working. Try it now. Find out what you have to mess up to get an error message. Take note of what sort of message you get, and fix your script so that it runs cleanly again.

Detecting and correcting coding errors

Let's take a brief tour of a few common errors you're likely to see. For example, the following error indicates that you've left off the quotation marks:

```
$ python hello_world.py
  File "hello_world.py", line 16
    print("Hello World!)
                        ^
SyntaxError: EOL while scanning string literal
```

Turning on the line numbers in the preferences of your text editor will help locate the relevant line. Many editors also allow you to search for the specific line number. The fix for this is easy: go back and add the matching pair of quotation marks. The little arrowhead shows you the place where the interpreter realized all was not well. That won't always be exactly the same place as you made the error, but it will probably have some logical relationship.

The following error means you misspelled `some_text` the first time you mentioned it:

```
$ python hello_world.py
Hello World!
Type in some words: Some Text
Traceback (most recent call last):
  File "hello_world.py", line 22, in <module>
    print(some_text)
NameError: name 'some_text' is not defined
```

The program runs fine up to that point, but the misspelled word is considered to be an error in your logic. Go back to the Python file, and make sure your logic is sane and that you have spelled the names of your variables consistently.

In the following script, you forgot to put quotes around the argument to `input()`, your prompt string:

```
$ python hello_world.py
  File "hello_world.py", line 19
    some_text = input(Type in some words: )
                                          ^
SyntaxError: invalid syntax
```

Notice that the interpreter doesn't mind the space between the function name and the parentheses, but it does mind the spaces in between the words in the argument because you didn't enclose them with quotes. Syntax errors mean that you got the punctuation or grammar wrong. The fix for this is usually to go and look up the correct form for this function in the official documentation. It might have been acceptable to have a one-word argument if that word was the name of a variable containing a text string. If no variable of that name existed you would be accused of calling an undefined identifier.

The final example is slightly more obscure. This error was actually caused by leaving off the comment marker # at the beginning of the line and leaving a space:

```
$ python hello_world.py
  File "hello_world.py", line 18
    2. Input some text
    ^
IndentationError: unexpected indent
```

The interpreter wasn't expecting to hit an indented line here; it was expecting another statement at the same level of indentation. I told you Python was fussy about indentation. If the space hadn't been left, you would simply receive a syntax error because `2.` isn't a command that Python recognizes.

Noting Modifications, Extensions, and Enhancements

As you read through your code, make notes of any improvements that could be made. I like to use comments in the script for this. It is a useful convention to include the word `TODO` or `FIXME` in the line, so you can search the source file later for things that you have decided need doing. You could create a separate `TODO` file in the same directory if you want to keep track of multiple files.

Documenting the Solution

Documentation is the last good habit I'd like to introduce you to, before I move on to discussing the Python language in depth. As mentioned earlier in this chapter, Python is self-documenting to a certain extent. Modules, classes, and functions may be documented by placing a text string as their first statement, just as I have done with the specification details in this example. The simplest way to test this is to start Python in interactive mode from the same directory as the `hello_world.py` script:

```
$ python
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12)
[GCC 4.2.3 (Debian 4.2.3-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

The three chevrons are the Python prompt, which means the interpreter is ready to receive your first command. Next, you need to import the script you have just written as a module using the `import` statement:

```
>>> import hello_world
Hello World!
Type in some words: Some Text
Some Text
```

You will see that the program runs in exactly the same way. However, this method allows us to easily access the documentation you have created. `help()` is a built-in function that takes one argument—the name of the module, class, function, keyword or topic that you want to find out about. Notice that I didn't use the `.py` extension or enclose the name of the module in quotes; this is because it became a