Mark Summerfield

Second Edition

Programming in Python 3

A Complete Introduction to the Python Language

Developer's Library

Library of Congress Cataloging-in-Publication Data

Summerfield, Mark.
Programming in Python 3 : a complete introduction to the Python language / Mark Summerfield.—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-68056-3 (pbk. : alk. paper)
1. Python (Computer program language) 2. Object-oriented programming (Computer science) I. Title.

QA76.73.P98S86 2010
005.13′3—dc22

2009035430

ISBN-13: 978-0-321-68056-3
ISBN-10: 0-321-68056-1
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2009

Introduction

Python is probably the easiest-to-learn and nicest-to-use programming language in widespread use. Python code is clear to read and write, and it is concise without being cryptic. Python is a very expressive language, which means that we can usually write far fewer lines of Python code than would be required for an equivalent application written in, say, C++ or Java.

Python is a cross-platform language: In general, the same Python program can be run on Windows and Unix-like systems such as Linux, BSD, and Mac OS X, simply by copying the file or files that make up the program to the target machine, with no "building" or compiling necessary. It is possible to create Python programs that use platform-specific functionality, but this is rarely necessary since almost all of Python's standard library and most third-party libraries are fully and transparently cross-platform.

One of Python's great strengths is that it comes with a very complete standard library—this allows us to do such things as download a file from the Internet, unpack a compressed archive file, or create a web server, all with just one or a few lines of code. And in addition to the standard library, thousands of third-party libraries are available, some providing more powerful and sophisticated facilities than the standard library—for example, the Twisted networking library and the NumPy numeric library—while others provide functionality that is too specialized to be included in the standard library—for example, the SimPy simulation package. Most of the third-party libraries are available from the Python Package Index, `pypi.python.org/pypi`.

Python can be used to program in procedural, object-oriented, and to a lesser extent, in functional style, although at heart Python is an object-oriented language. This book shows how to write both procedural and object-oriented programs, and also teaches Python's functional programming features.

The purpose of this book is to show you how to write Python programs in good idiomatic Python 3 style, and to be a useful reference for the Python 3 language after the initial reading. Although Python 3 is an evolutionary rather than revolutionary advance on Python 2, some older practices are no longer appropriate or necessary in Python 3, and new practices have been introduced to take advantage of Python 3 features. Python 3 is a better language than Python 2—it builds on the many years of experience with Python 2 and adds lots of new features (and omits Python 2's misfeatures), to make it even more of a pleasure to use than Python 2, as well as more convenient, easier, and more consistent.

The book's aim is to teach the Python *language*, and although many of the standard Python libraries are used, not all of them are. This is not a problem, because once you have read the book, you will have enough Python knowledge to be able to make use of any of the standard libraries, or any third-party Python library, and be able to create library modules of your own.

The book is designed to be useful to several different audiences, including self-taught and hobbyist programmers, students, scientists, engineers, and others who need to program as part of their work, and of course, computing professionals and computer scientists. To be of use to such a wide range of people without boring the knowledgeable or losing the less-experienced, the book assumes at least some programming experience (in any language). In particular, it assumes a basic knowledge of data types (such as numbers and strings), collection data types (such as sets and lists), control structures (such as if and while statements), and functions. In addition, some examples and exercises assume a basic knowledge of HTML markup, and some of the more specialized chapters at the end assume a basic knowledge of their subject area; for example, the databases chapter assumes a basic knowledge of SQL.

The book is structured in such a way as to make you as productive as possible as quickly as possible. By the end of the first chapter you will be able to write small but useful Python programs. Each successive chapter introduces new topics, and often both broadens and deepens the coverage of topics introduced in earlier chapters. This means that if you read the chapters in sequence, you can stop at any point and you'll be able to write complete programs with what you have learned up to that point, and then, of course, resume reading to learn more advanced and sophisticated techniques when you are ready. For this reason, some topics are introduced in one chapter, and then are explored further in one or more later chapters.

Two key problems arise when teaching a new programming language. The first is that sometimes when it is necessary to teach one particular concept, that concept depends on another concept, which in turn depends either directly or indirectly on the first. The second is that, at the beginning, the reader may know little or nothing of the language, so it is very difficult to present interesting or useful examples and exercises. In this book, we seek to solve both of these problems, first by assuming some prior programming experience, and second by presenting Python's "beautiful heart" in Chapter 1—eight key pieces of Python that are sufficient on their own to write decent programs. One consequence of this approach is that in the early chapters some of the examples are a bit artificial in style, since they use only what has been taught up to the point where they are presented; this effect diminishes chapter by chapter, until by the end of Chapter 7, all the examples are written in completely natural and idiomatic Python 3 style.

The book's approach is wholly practical, and you are encouraged to try out the examples and exercises for yourself to get hands-on experience. Wherever

possible, small but complete programs and modules are used as examples to provide realistic use cases. The examples, exercise solutions, and the book's errata are available online at `www.qtrac.eu/py3book.html`.

Two sets of examples are provided. The standard examples work with *any* Python 3.*x* version—use these if you care about Python 3.0 compatibility. The "eg31" examples work with Python 3.1 or later—use these if you don't need to support Python 3.0 because your programs' users have Python 3.1 or later. All of the examples have been tested on Windows, Linux, and Mac OS X.

While it is best to use the most recent version of Python 3, this is not always possible if your users cannot or will not upgrade. Every example in this book works with Python 3.0 except where stated, and those examples and features that are specific to Python 3.1 are clearly indicated as such.

Although it is possible to use this book to develop software that uses only Python 3.0, for those wanting to produce software that is expected to be in use for many years and that is expected to be compatible with later Python 3.*x* releases, it is best to use Python 3.1 as the oldest Python 3 version that you support. This is partly because Python 3.1 has some very nice new features, but mostly because the Python developers *strongly* recommend using Python 3.1 (or later). The developers have decided that Python 3.0.1 will be the last Python 3.0.*y* release, and that there will be no more Python 3.0.*y* releases even if bugs or security problems are discovered. Instead, they want all Python 3 users to migrate to Python 3.1 (or to a later version), which will have the usual bugfix and security maintenance releases that Python versions normally have.

The Structure of the Book

Chapter 1 presents eight key pieces of Python that are sufficient for writing complete programs. It also describes some of the Python programming environments that are available and presents two tiny example programs, both built using the eight key pieces of Python covered earlier in the chapter.

Chapters 2 through 5 introduce Python's procedural programming features, including its basic data types and collection data types, and many useful built-in functions and control structures, as well as very simple text file handling. Chapter 5 shows how to create custom modules and packages and provides an overview of Python's standard library so that you will have a good idea of the functionality that Python provides out of the box and can avoid reinventing the wheel.

Chapter 6 provides a thorough introduction to object-oriented programming with Python. All of the material on procedural programming that you learned in earlier chapters is still applicable, since object-oriented programming is

built on procedural foundations—for example, making use of the same data types, collection data types, and control structures.

Chapter 7 covers writing and reading files. For binary files, the coverage includes compression and random access, and for text files, the coverage includes parsing manually and with regular expressions. This chapter also shows how to write and read XML files, including using element trees, DOM (Document Object Model), and SAX (Simple API for XML).

Chapter 8 revisits material covered in some earlier chapters, exploring many of Python's more advanced features in the areas of data types and collection data types, control structures, functions, and object-oriented programming. This chapter also introduces many new functions, classes, and advanced techniques, including functional-style programming and the use of coroutines—the material it covers is both challenging and rewarding.

Chapter 9 is different from all the other chapters in that it discusses techniques and libraries for debugging, testing, and profiling programs, rather than introducing new Python features.

The remaining chapters cover various advanced topics. Chapter 10 shows techniques for spreading a program's workload over multiple processes and over multiple threads. Chapter 11 shows how to write client/server applications using Python's standard networking support. Chapter 12 covers database programming (both simple key–value "DBM" files and SQL databases).

Chapter 13 explains and illustrates Python's regular expression mini-language and covers the regular expressions module. Chapter 14 follows on from the regular expressions chapter by showing basic parsing techniques using regular expressions, and also using two third-party modules, PyParsing and PLY. Finally, Chapter 15 introduces GUI (Graphical User Interface) programming using the `tkinter` module that is part of Python's standard library. In addition, the book has a very brief epilogue, a selected bibliography, and of course, an index.

Most of the book's chapters are quite long to keep all the related material together in one place for ease of reference. However, the chapters are broken down into sections, subsections, and sometimes subsubsections, so it is easy to read at a pace that suits you; for example, by reading one section or subsection at a time.

Obtaining and Installing Python 3

If you have a modern and up-to-date Mac or other Unix-like system you may already have Python 3 installed. You can check by typing `python -V` (note the capital *V*) in a console (Terminal.app on Mac OS X)—if the version is 3.*x* you've already got Python 3 and don't have to install it yourself. If Python wasn't found at all it may be that it has a name which includes a version number. Try

typing `python3 -V`, and if that does not work try `python3.0 -V`, and failing that try `python3.1 -V`. If any of these work you now know that you already have Python installed, what version it is, and what it is called. (In this book we use the name `python3`, but use whatever name worked for you, for example, `python3.1`.) If you don't have any version of Python 3 installed, read on.

For Windows and Mac OS X, easy-to-use graphical installer packages are provided that take you step-by-step through the installation process. These are available from `www.python.org/download`. For Windows, download the "Windows x86 MSI Installer", unless you know for sure that your machine has a different processor for which a separate installer is supplied—for example, if you have an AMD64, get the "Windows AMD64 MSI Installer". Once you've got the installer, just run it and follow the on-screen instructions.

For Linux, BSD, and other Unixes (apart from Mac OS X for which a `.dmg` installation file is provided), the easiest way to install Python is to use your operating system's package management system. In most cases Python is provided in several separate packages. For example, in Ubuntu (from version 8), there is `python3.0` for Python, `idle-python3.0` for IDLE (a simple development environment), and `python3.0-doc` for the documentation—as well as many other packages that provide add-ons for even more functionality than that provided by the standard library. (Naturally, the package names will start with `python-3.1` for the Python 3.1 versions, and so on.)

If no Python 3 packages are available for your operating system you will need to download the source from `www.python.org/download` and build Python from scratch. Get either of the source tarballs and unpack it using `tar xvfz Python-3.1.tgz` if you got the gzipped tarball or `tar xvfj Python-3.1.tar.bz2` if you got the bzip2 tarball. (The version numbers may be different, for example, `Python-3.1.1.tgz` or `Python-3.1.2.tar.bz2`, in which case simply replace `3.1` with your actual version number throughout.) The configuration and building are standard. First, change into the newly created `Python-3.1` directory and run `./configure`. (You can use the `--prefix` option if you want to do a local install.) Next, run `make`.

It is possible that you may get some messages at the end saying that not all modules could be built. This normally means that you don't have some of the required libraries or headers on your machine. For example, if the `readline` module could not be built, use the package management system to install the corresponding development library; for example, `readline-devel` on Fedora-based systems and `readline-dev` on Debian-based systems such as Ubuntu. Another module that may not build straight away is the `tkinter` module—this depends on both the Tcl and Tk development libraries, `tcl-devel` and `tk-devel` on Fedora-based systems, and `tcl8.5-dev` and `tk8.5-dev` on Debian-based systems (and where the minor version may not be 5). Unfortunately, the relevant package names are not always so obvious, so you might need to ask for help on

Python's mailing list. Once the missing packages are installed, run `./configure` and `make` again.

After successfully making, you could run `make test` to see that everything is okay, although this is not necessary and can take many minutes to complete.

If you used `--prefix` to do a local installation, just run `make install`. For Python 3.1, if you installed into, say, `~/local/python31`, then by adding the `~/local/python31/bin` directory to your `PATH`, you will be able to run Python using `python3` and IDLE using `idle3`. Alternatively, if you already have a local directory for executables that is already in your `PATH` (such as `~/bin`), you might prefer to add soft links instead of changing the `PATH`. For example, if you keep executables in `~/bin` and you installed Python in `~/local/python31`, you could create suitable links by executing `ln -s ~/local/python31/bin/python3 ~/bin/python3`, and `~/local/python31/bin/idle3 ~/bin/idle3`. For this book we did a local install and added soft links on Linux and Mac OS X exactly as described here—and on Windows we used the binary installer.

If you did not use `--prefix` and have root access, log in as root and do `make install`. On sudo-based systems like Ubuntu, do `sudo make install`. If Python 2 is on the system, `/usr/bin/python` won't be changed, and Python 3 will be available as `python3.0` (or `python3.1` depending on the version installed) and from Python 3.1, in addition, as `python3`. Python 3.0's IDLE is installed as `idle`, so if access to Python 2's IDLE is still required the old IDLE will need to be renamed—for example, to `/usr/bin/idle2`—*before* doing the install. Python 3.1 installs IDLE as `idle3` and so does not conflict with Python 2's IDLE.

Acknowledgments

I would first like to acknowledge with thanks the feedback I have received from readers of the first edition, who gave corrections, or made suggestions, or both.

My next acknowledgments are of the book's technical reviewers, starting with Jasmin Blanchette, a computer scientist, programmer, and writer with whom I have cowritten two C++/Qt books. Jasmin's involvement with chapter planning and his suggestions and criticisms regarding all the examples, as well as his careful reading, have immensely improved the quality of this book.

Georg Brandl is a leading Python developer and documentor responsible for creating Python's new documentation tool chain. Georg spotted many subtle mistakes and very patiently and persistently explained them until they were understood and corrected. He also made many improvements to the examples.

Phil Thompson is a Python expert and the creator of PyQt, probably the best Python GUI library available. Phil's sharp-eyed and sometimes challenging feedback led to many clarifications and corrections.

Trenton Schulz is a senior software engineer at Nokia's Qt Software (formerly Trolltech) who has been a valuable reviewer of all my previous books, and has once again come to my aid. Trenton's careful reading and the numerous suggestions that he made helped clarify many issues and have led to considerable improvements to the text.

In addition to the aforementioned reviewers, all of whom read the whole book, David Boddie, a senior technical writer at Nokia's Qt Software and an experienced Python practitioner and open source developer, has read and given valuable feedback on portions of it.

For this second edition, I would also like to thank Paul McGuire (author of the PyParsing module), who was kind enough to review the PyParsing examples that appear in the new chapter on parsing, and who gave me a lot of thoughtful and useful advice. And for the same chapter, David Beazley (author of the PLY module) reviewed the PLY examples and provided valuable feedback. In addition, Jasmin, Trenton, Georg, and Phil read most of this second edition's new material, and provided very valuable feedback.

Thanks are also due to Guido van Rossum, creator of Python, as well as to the wider Python community who have contributed so much to make Python, and especially its libraries, so useful and enjoyable to use.

As always, thanks to Jeff Kingston, creator of the Lout typesetting language that I have used for more than a decade.

Special thanks to my editor, Debra Williams Cauley, for her support, and for once again making the entire process as smooth as possible. Thanks also to Anna Popick, who managed the production process so well, and to the proofreader, Audrey Doyle, who did such fine work once again. And for this second edition I also want to thank Jennifer Lindner for helping me keep the new material understandable, and the first edition's Japanese translator Takahiro Nagao 長尾 高弘, for spotting some subtle mistakes which I've been able to correct in this edition.

Last but not least, I want to thank my wife, Andrea, both for putting up with the 4 a.m. wake-ups when book ideas and code corrections often arrived and insisted upon being noted or tested there and then, and for her love, loyalty, and support.

13

- Python's Regular Expression Language
- The Regular Expression Module

Regular Expressions

A regular expression is a compact notation for representing a collection of strings. What makes regular expressions so powerful is that a single regular expression can represent an unlimited number of strings—providing they meet the regular expression's requirements. Regular expressions (which we will mostly call "regexes" from now on) are defined using a mini-language that is completely different from Python—but Python includes the `re` module through which we can seamlessly create and use regexes.*

Regexes are used for five main purposes:

- Parsing: identifying and extracting pieces of text that match certain criteria—regexes are used for creating ad hoc parsers and also by traditional parsing tools
- Searching: locating substrings that can have more than one form, for example, finding any of "pet.png", "pet.jpg", "pet.jpeg", or "pet.svg" while avoiding "carpet.png" and similar
- Searching and replacing: replacing everywhere the regex matches with a string, for example, finding "bicycle" or "human powered vehicle" and replacing either with "bike"
- Splitting strings: splitting a string at each place the regex matches, for example, splitting everywhere colon-space or equals (": " or "=") occurs
- Validation: checking whether a piece of text meets some criteria, for example, contains a currency symbol followed by digits

The regexes used for searching, splitting, and validation are often fairly small and understandable, making them ideal for these purposes. However, although

* A good book on regular expressions is *Mastering Regular Expressions* by Jeffrey E. F. Friedl, ISBN 0596528124. It does not explicitly cover Python, but Python's `re` module offers very similar functionality to the Perl regular expression engine that the book covers in depth.

regexes are widely and successfully used to create parsers, they do have a limitation in that area: They are only able to deal with recursively structured text if the maximum level of recursion is known. Also, large and complex regexes can be difficult to read and maintain. So apart from simple cases, for parsing the best approach is to use a tool designed for the purpose—for example, use a dedicated XML parser for XML. If such a parser isn't available, then an alternative to using regexes is to use a generic parsing tool, an approach that is covered in Chapter 14.

Parsing XML files 312 ◀

At its simplest a regular expression is an expression (e.g., a literal character), optionally followed by a quantifier. More complex regexes consist of any number of quantified expressions and may include assertions and may be influenced by flags.

This chapter's first section introduces and explains all the key regular expression concepts and shows pure regular expression syntax—it makes minimal reference to Python itself. Then the second section shows how to use regular expressions in the context of Python programming, drawing on all the material covered in the earlier sections. Readers familiar with regular expressions who just want to learn how they work in Python could skip to the second section (➤ 499). The chapter covers the complete regex language offered by the `re` module, including all the assertions and flags. We indicate regular expressions in the text using **`bold`**, show where they match using `underlining`, and show captures using `shading`.

Python's Regular Expression Language

In this section we look at the regular expression language in four subsections. The first subsection shows how to match individual characters or groups of characters, for example, match *a*, or match *b*, or match either *a* or *b*. The second subsection shows how to quantify matches, for example, match once, or match at least once, or match as many times as possible. The third subsection shows how to group subexpressions and how to capture matching text, and the final subsection shows how to use the language's assertions and flags to affect how regular expressions work.

Characters and Character Classes

The simplest expressions are just literal characters, such as **`a`** or **`5`**, and if no quantifier is explicitly given it is taken to be "match one occurrence". For example, the regex **`tune`** consists of four expressions, each implicitly quantified to match once, so it matches one *t* followed by one *u* followed by one *n* followed by one *e*, and hence matches the strings `tune` and `attuned`.

Although most characters can be used as literals, some are "special characters"—these are symbols in the regex language and so must be escaped by preceding them with a backslash (\) to use them as literals. The special characters are `\.^$?+*{}[]()|`. Most of Python's standard string escapes can also be used within regexes, for example, `\n` for newline and `\t` for tab, as well as hexadecimal escapes for characters using the `\x`*HH*, `\u`*HHHH*, and `\U`*HHHHHHHH* syntaxes.

String escapes 66 ◀

In many cases, rather than matching one particular character we want to match any one of a set of characters. This can be achieved by using a *character class*—one or more characters enclosed in square brackets. (This has nothing to do with a Python class, and is simply the regex term for "set of characters".) A character class is an expression, and like any other expression, if not explicitly quantified it matches exactly one character (which can be any of the characters in the character class). For example, the regex **`r[ea]d`** matches both `red` and `radar`, but not `read`. Similarly, to match a single digit we can use the regex **`[0123456789]`**. For convenience we can specify a range of characters using a hyphen, so the regex **`[0-9]`** also matches a digit. It is possible to negate the meaning of a character class by following the opening bracket with a caret, so **`[^0-9]`** matches any character that is *not* a digit.

Note that inside a character class, apart from \, the special characters lose their special meaning, although in the case of ^ it acquires a new meaning (negation) if it is the first character in the character class, and otherwise is simply a literal caret. Also, - signifies a character range unless it is the first character, in which case it is a literal hyphen.

Since some sets of characters are required so frequently, several have shorthand forms—these are shown in Table 13.1. With one exception the shorthands can be used inside character sets, so for example, the regex **`[\dA-Fa-f]`** matches any hexadecimal digit. The exception is **`.`** which is a shorthand outside a character class but matches a literal . inside a character class.

Quantifiers

A quantifier has the form **`{`***m*`,`*n*`}`** where *m* and *n* are the minimum and maximum times the expression the quantifier applies to must match. For example, both **`e{1,1}e{1,1}`** and **`e{2,2}`** match `feel`, but neither matches `felt`.

Writing a quantifier after every expression would soon become tedious, and is certainly difficult to read. Fortunately, the regex language supports several convenient shorthands. If only one number is given in the quantifier it is taken to be both the minimum and the maximum, so **`e{2}`** is the same as **`e{2,2}`**. And as we noted in the preceding section, if no quantifier is explicitly given, it is assumed to be one (i.e., **`{1,1}`** or **`{1}`**); therefore, **`ee`** is the same as **`e{1,1}e{1,1}`** and **`e{1}e{1}`**, so both **`e{2}`** and **`ee`** match `feel` but not `felt`.

Table 13.1 *Character Class Shorthands*

Symbol	Meaning
`.`	Matches any character except newline; or any character at all with the `re.DOTALL` flag; or inside a character class matches a literal `.`
`\d`	Matches a Unicode digit; or `[0-9]` with the `re.ASCII` flag
`\D`	Matches a Unicode nondigit; or `[^0-9]` with the `re.ASCII` flag
`\s`	Matches a Unicode whitespace; or `[ \t\n\r\f\v]` with the `re.ASCII` flag
`\S`	Matches a Unicode nonwhitespace; or `[^ \t\n\r\f\v]` with the `re.ASCII` flag
`\w`	Matches a Unicode "word" character; or `[a-zA-Z0-9_]` with the `re.ASCII` flag
`\W`	Matches a Unicode non-"word" character; or `[^a-zA-Z0-9_]` with the `re.ASCII` flag

Meaning of the flags ➤ 496

Having a different minimum and maximum is often convenient. For example, to match `travelled` and `traveled` (both legitimate spellings), we could use either **`travel{1,2}ed`** or **`travell{0,1}ed`**. The **`{0,1}`** quantification is so often used that it has its own shorthand form, **`?`**, so another way of writing the regex (and the one most likely to be used in practice) is **`travell?ed`**.

Two other quantification shorthands are provided: **`+`** which stands for **`{1,`***n***`}`** ("at least one") and `*` which stands for **`{0,`***n***`}`** ("any number of"); in both cases *n* is the maximum possible number allowed for a quantifier, usually at least 32 767. All the quantifiers are shown in Table 13.2.

The **`+`** quantifier is very useful. For example, to match integers we could use **`\d+`** since this matches one or more digits. This regex could match in two places in the string `4588.91`, for example, `4588`.`91` and `4588.91`. Sometimes typos are the result of pressing a key too long. We could use the regex **`bevel+ed`** to match the legitimate `beveled` and `bevelled`, and the incorrect `bevellled`. If we wanted to standardize on the one *l* spelling, and match only occurrences that had two or more *l*s, we could use **`bevell+ed`** to find them.

The `*` quantifier is less useful, simply because it can so often lead to unexpected results. For example, supposing that we want to find lines that contain comments in Python files, we might try searching for **`#*`**. But this regex will match any line whatsoever, including blank lines because the meaning is "match any number of #s"—and that includes none. As a rule of thumb for those new to regexes, avoid using `*` at all, and if you do use it (or if you use **`?`**), make sure there is at least one other expression in the regex that has a nonzero quantifier—so at least one quantifier other than `*` or **`?`** since both of these can match their expression zero times.

Table 13.2 *Regular Expression Quantifiers*

Syntax	Meaning
e`?` or *e*`{0,1}`	Greedily match zero or one occurrence of expression *e*
e`??` or *e*`{0,1}?`	Nongreedily match zero or one occurrence of expression *e*
e`+` or *e*`{1,}`	Greedily match one or more occurrences of expression *e*
e`+?` or *e*`{1,}?`	Nongreedily match one or more occurrences of expression *e*
e`*` or *e*`{0,}`	Greedily match zero or more occurrences of expression *e*
e`*?` or *e*`{0,}?`	Nongreedily match zero or more occurrences of expression *e*
e`{`*m*`}`	Match exactly *m* occurrences of expression *e*
e`{`*m*`,}`	Greedily match at least *m* occurrences of expression *e*
e`{`*m*`,}?`	Nongreedily match at least *m* occurrences of expression *e*
e`{,`*n*`}`	Greedily match at most *n* occurrences of expression *e*
e`{,`*n*`}?`	Nongreedily match at most *n* occurrences of expression *e*
e`{`*m*`,`*n*`}`	Greedily match at least *m* and at most *n* occurrences of expression *e*
e`{`*m*`,`*n*`}?`	Nongreedily match at least *m* and at most *n* occurrences of expression *e*

It is often possible to convert `*` uses to `+` uses and vice versa. For example, we could match "tasselled" with at least one *l* using `tassell*ed` or `tassel+ed`, and match those with two or more *l*s using `tasselll*ed` or `tassell+ed`.

If we use the regex `\d+` it will match 136. But why does it match all the digits, rather than just the first one? By default, all quantifiers are *greedy*—they match as many characters as they can. We can make any quantifier nongreedy (also called *minimal*) by following it with a `?` symbol. (The question mark has two different meanings—on its own it is a shorthand for the `{0,1}` quantifier, and when it follows a quantifier it tells the quantifier to be nongreedy.) For example, `\d+?` can match the string 136 in three different places: 136, 136, and 136. Here is another example: `\d??` matches zero or one digits, but prefers to match none since it is nongreedy—on its own it suffers the same problem as * in that it will match nothing, that is, any text at all.

Nongreedy quantifiers can be useful for quick and dirty XML and HTML parsing. For example, to match all the image tags, writing `<img.*>` (match one "<", then one "i", then one "m", then one "g", then zero or more of any character apart from newline, then one ">") will not work because the `.*` part is greedy and will match everything including the tag's closing >, and will keep going until it reaches the last > in the entire text.

Three solutions present themselves (apart from using a proper parser). One is **<img[^>]*>** (match <img, then any number of non-> characters and then the tag's closing > character), another is **<img.*?>** (match <img, then any number of characters, but nongreedily, so it will stop immediately before the tag's closing >, and then the >), and a third combines both, as in **<img[^>]*?>**. None of them is correct, though, since they can all match <img>, which is not valid. Since we know that an image tag must have a src attribute, a more accurate regex is **<img\s+[^>]*?src=\w+[^>]*?>**. This matches the literal characters <img, then one or more whitespace characters, then nongreedily zero or more of anything except > (to skip any other attributes such as alt), then the src attribute (the literal characters src= then at least one "word" character), and then any other non-> characters (including none) to account for any other attributes, and finally the closing >.

Grouping and Capturing

In practical applications we often need regexes that can match any one of two or more alternatives, and we often need to capture the match or some part of the match for further processing. Also, we sometimes want a quantifier to apply to several expressions. All of these can be achieved by grouping with (), and in the case of alternatives using alternation with |.

Alternation is especially useful when we want to match any one of several quite different alternatives. For example, the regex **aircraft|airplane|jet** will match any text that contains "aircraft" or "airplane" or "jet". The same thing can be achieved using the regex **air(craft|plane)|jet**. Here, the parentheses are used to group expressions, so we have two outer expressions, **air(craft|plane)** and **jet**. The first of these has an inner expression, **craft|plane**, and because this is preceded by **air** the first outer expression can match only "aircraft" or "airplane".

Parentheses serve two different purposes—to group expressions and to capture the text that matches an expression. We will use the term *group* to refer to a grouped expression whether it captures or not, and *capture* and *capture group* to refer to a captured group. If we used the regex **(aircraft|airplane|jet)** it would not only match any of the three expressions, but would also capture whichever one was matched for later reference. Compare this with the regex **(air(craft|plane)|jet)** which has two captures if the first expression matches ("aircraft" or "airplane" as the first capture and "craft" or "plane" as the second capture), and one capture if the second expression matches ("jet"). We can switch off the capturing effect by following an opening parenthesis with **?:**, so for example, **(air(?:craft|plane)|jet)** will have only one capture if it matches ("aircraft" or "airplane" or "jet").

A grouped expression is an expression and so can be quantified. Like any other expression the quantity is assumed to be one unless explicitly given. For

example, if we have read a text file with lines of the form *key*=*value*, where each *key* is alphanumeric, the regex **(\w+)=(.+)** will match every line that has a nonempty key and a nonempty value. (Recall that . matches anything except newlines.) And for every line that matches, two captures are made, the first being the key and the second being the value.

For example, the *key*=*value* regular expression will match the entire line `topic= physical geography` with the two captures shown shaded. Notice that the second capture includes some whitespace, and that whitespace before the = is not accepted. We could refine the regex to be more flexible in accepting whitespace, and to strip off unwanted whitespace using a somewhat longer version:

```
[ \t]*(\w+)[ \t]*=[ \t]*(.+)
```

This matches the same line as before and also lines that have whitespace around the = sign, but with the first capture having no leading or trailing whitespace, and the second capture having no leading whitespace. For example: `topic = physical geography`. We have been careful to keep the whitespace matching parts outside the capturing parentheses, and to allow for lines that have no whitespace at all. We did not use **\s** to match whitespace because that matches newlines (\n) which could lead to incorrect matches that span lines (e.g., if the `re.MULTILINE` flag is used). And for the value we did not use **\S** to match nonwhitespace because we want to allow for values that contain whitespace (e.g., English sentences). To avoid the second capture having trailing whitespace we would need a more sophisticated regex; we will see this in the next subsection.

Regex flags ➤ 502

Captures can be referred to using *backreferences*, that is, by referring back to an earlier capture group.* One syntax for backreferences inside regexes themselves is *i* where *i* is the capture number. Captures are numbered starting from one and increasing by one going from left to right as each new (capturing) left parenthesis is encountered. For example, to simplistically match duplicated words we can use the regex **(\w+)\s+\1** which matches a "word", then at least one whitespace, and then the same word as was captured. (Capture number 0 is created automatically without the need for parentheses; it holds the entire match, that is, what we show underlined.) We will see a more sophisticated way to match duplicate words later.

In long or complicated regexes it is often more convenient to use names rather than numbers for captures. This can also make maintenance easier since adding or removing capturing parentheses may change the numbers but won't affect names. To name a capture we follow the opening parenthesis with **?P<***name***>**. For example, **(?P<key>\w+)=(?P<value>.+)** has two captures called `"key"` and `"value"`. The syntax for backreferences to named captures inside a

*Note that backreferences cannot be used inside character classes, that is, inside [].

regex is **(?P=*name*)**. For example, **(?P<word>\w+)\s+(?P=word)** matches duplicate words using a capture called "word".

Assertions and Flags

One problem that affects many of the regexes we have looked at so far is that they can match more or different text than we intended. For example, the regex **aircraft|airplane|jet** will match "waterjet" and "jetski" as well as "jet". This kind of problem can be solved by using assertions. An assertion does not match any text, but instead says something about the text at the point where the assertion occurs.

One assertion is **\b** (word boundary), which asserts that the character that precedes it must be a "word" (**\w**) and the character that follows it must be a non-"word" (**\W**), or vice versa. For example, although the regex **jet** can match twice in the text the jet and jetski are noisy, that is, the jet and jetski are noisy, the regex **\bjet\b** will match only once, the jet and jetski are noisy. In the context of the original regex, we could write it either as **\baircraft\b|\bairplane\b|\bjet\b** or more clearly as **\b(?:aircraft|airplane|jet)\b**, that is, word boundary, noncapturing expression, word boundary.

Many other assertions are supported, as shown in Table 13.3. We could use assertions to improve the clarity of a *key*=*value* regex, for example, by changing it to **^(\w+)=([^\n]+)** and setting the re.MULTILINE flag to ensure that each *key*=*value* is taken from a single line with no possibility of spanning lines—providing no part of the regex matches a newline, so we can't use, say, **\s**. (The flags are shown in Table 13.5; ➤ 502; their syntaxes are described at the end of this subsection, and examples are given in the next section.) And if we want to strip whitespace from the ends and use named captures, the regex becomes:

```
^[ \t]*(?P<key>\w+)[ \t]*=[ \t]*(?P<value>[^\n]+)(?<![ \t])
```

Even though this regex is designed for a fairly simple task, it looks quite complicated. One way to make it more maintainable is to include comments in it. This can be done by adding inline comments using the syntax **(?#*the comment*)**, but in practice comments like this can easily make the regex even more difficult to read. A much nicer solution is to use the re.VERBOSE flag—this allows us to freely use whitespace and normal Python comments in regexes, with the one constraint that if we need to match whitespace we must either use **\s** or a character class such as [ ]. Here's the *key*=*value* regex with comments:

Regex flags ➤ 502

```
^[ \t]*             # start of line and optional leading whitespace
(?P<key>\w+)        # the key text
[ \t]*=[ \t]*       # the equals with optional surrounding whitespace
(?P<value>[^\n]+)   # the value text
(?<![ \t])          # negative lookbehind to avoid trailing whitespace
```

Table 13.3 *Regular Expression Assertions*

Symbol	Meaning
`^`	Matches at the start; also matches after each newline with the `re.MULTILINE` flag
`$`	Matches at the end; also matches before each newline with the `re.MULTILINE` flag
`\A`	Matches at the start
`\b`	Matches at a "word" boundary; influenced by the `re.ASCII` flag—inside a character class this is the escape for the backspace character
`\B`	Matches at a non-"word" boundary; influenced by the `re.ASCII` flag
`\Z`	Matches at the end
`(?=e)`	Matches if the expression `e` matches at this assertion but does not advance over it—called *lookahead* or *positive lookahead*
`(?!e)`	Matches if the expression `e` does not match at this assertion and does not advance over it—called *negative lookahead*
`(?<=e)`	Matches if the expression `e` matches immediately before this assertion—called *positive lookbehind*
`(?<!e)`	Matches if the expression `e` does not match immediately before this assertion—called *negative lookbehind*

Regex flags ➤ 502

Raw strings 67 ➤

In the context of a Python program we would normally write a regex like this inside a raw triple quoted string—raw so that we don't have to double up the backslashes, and triple quoted so that we can spread it over multiple lines.

In addition to the assertions we have discussed so far, there are additional assertions which look at the text in front of (or behind) the assertion to see whether it matches (or does not match) an expression we specify. The expressions that can be used in lookbehind assertions must be of fixed length (so the quantifiers `?`, `+`, and `*` cannot be used, and numeric quantifiers must be of a fixed size, for example, `{3}`).

In the case of the *key=value* regex, the negative lookbehind assertion means that at the point it occurs the *preceding* character must not be a space or a tab. This has the effect of ensuring that the last character captured into the `"value"` capture group is not a space or tab (yet without preventing spaces or tabs from appearing inside the captured text).

Let's consider another example. Suppose we are reading a multiline text that contains the names "Helen Patricia Sharman", "Jim Sharman", "Sharman Joshi", "Helen Kelly", and so on, and we want to match "Helen Patricia", but only when referring to "Helen Patricia Sharman". The easi-

est way is to use the regex **\b(Helen\s+Patricia)\s+Sharman\b**. But we could also achieve the same thing using a lookahead assertion, for example, **\b(Helen\s+Patricia)(?=\s+Sharman\b)**. This will match "Helen Patricia" only if it is preceded by a word boundary and followed by whitespace and "Sharman" ending at a word boundary.

To capture the particular variation of the forenames that is used ("Helen", "Helen P.", or "Helen Patricia"), we could make the regex slightly more sophisticated, for example, **\b(Helen(?:\s+(?:P\.|Patricia))?)\s+(?=Sharman\b)**. This matches a word boundary followed by one of the forename forms—but only if this is followed by some whitespace and then "Sharman" and a word boundary.

Note that only two syntaxes perform capturing, **(***e***)** and **(?P<***name***>***e***)**. None of the other parenthesized forms captures. This makes perfect sense for the lookahead and lookbehind assertions since they only make a statement about what follows or precedes them—they are not part of the match, but rather affect whether a match is made. It also makes sense for the last two parenthesized forms that we will now consider.

We saw earlier how we can backreference a capture inside a regex either by number (e.g., **\1**) or by name (e.g., **(?P=***name***)**). It is also possible to match conditionally depending on whether an earlier match occurred. The syntaxes are **(?(***id***)***yes_exp***)** and **(?(***id***)***yes_exp***|***no_exp***)**. The *id* is the name or number of an earlier capture that we are referring to. If the capture succeeded the *yes_exp* will be matched here. If the capture failed the no_exp will be matched if it is given.

Let's consider an example. Suppose we want to extract the filenames referred to by the src attribute in HTML img tags. We will begin just by trying to match the src attribute, but unlike our earlier attempt we will account for the three forms that the attribute's value can take: single quoted, double quoted, and unquoted. Here is an initial attempt: **src=(["'])([^"'>]+)\1**. The **([^"'>]+)** part captures a greedy match of at least one character that isn't a quote or >. This regex works fine for quoted filenames, and thanks to the **\1** matches only when the opening and closing quotes are the same. But it does not allow for unquoted filenames. To fix this we must make the opening quote optional and therefore match only if it is present.

Here is a revised regex: **src=(["'])?([^"'>]+)(?(1)\1)**. We did not provide a *no_exp* since there is nothing to match if no quote is given. Unfortunately, this doesn't work quite right. It will work fine for quoted filenames, but for unquoted filenames it will work only if the src attribute is the last attribute in the tag; otherwise it will incorrectly match text into the next attribute. The solution is to treat the two cases (quoted and unquoted) separately, and to use alternation: **src=((["'])([^\1>]+?)\1|([^"' >]+))**. Now let's see the regex in context, complete with named groups, nonmatching parentheses, and comments:

```
<img\s+                     # start of the tag
[^>]*?                      # any attributes that precede the src
src=                        # start of the src attribute
(?:
    (?P<quote>["'])         # opening quote
    (?P<qimage>[^\1>]+?)    # image filename
    (?P=quote)              # closing quote matching the opening quote
|                           # ---or alternatively---
    (?P<uimage>[^"' >]+)    # unquoted image filename
)
[^>]*?                      # any attributes that follow the src
>                           # end of the tag
```

The indentation is just for clarity. The noncapturing parentheses are used for alternation. The first alternative matches a quote (either single or double), then the image filename (which may contain any characters except for the quote that matched or >), and finally, another quote which must be the same as the matching quote. We also had to use minimal matching, **+?**, for the filename, to ensure that the match doesn't extend beyond the first matching closing quote. This means that a filename such as `"I'm here!.png"` will match correctly. Note also that to refer to the matching quote inside the character class we had to use a numbered backreference, **\1**, instead of **(?P=quote)**, since only numbered backreferences work inside character classes. The second alternative matches an unquoted filename—a string of characters that don't include quotes, spaces, or >. Due to the alternation, the filename is captured in `"qimage"` (capture number 2) or in `"uimage"` (capture number 3, since **(?P=quote)** matches but doesn't capture), so we must check for both.

The final piece of regex syntax that Python's regular expression engine offers is a means of setting the flags. Usually the flags are set by passing them as additional parameters when calling the `re.compile()` function, but sometimes it is more convenient to set them as part of the regex itself. The syntax is simply **(?***flags***)** where *flags* is one or more of `a` (the same as passing `re.ASCII`), `i` (`re.IGNORECASE`), `m` (`re.MULTILINE`), `s` (`re.DOTALL`), and `x` (`re.VERBOSE`).* If the flags are set this way they should be put at the start of the regex; they match nothing, so their effect on the regex is only to set the flags.

Regex flags ➤ 502

The Regular Expression Module

The `re` module provides two ways of working with regexes. One is to use the functions listed in Table 13.4 (➤ 502), where each function is given a regex as its first argument. Each function converts the regex into an internal format—a

*The letters used for the flags are the same as the ones used by Perl's regex engine, which is why `s` is used for `re.DOTALL` and `x` is used for `re.VERBOSE`.

process called *compiling*—and then does its work. This is very convenient for one-off uses, but if we need to use the same regex repeatedly we can avoid the cost of compiling it at each use by compiling it once using the `re.compile()` function. We can then call methods on the compiled regex object as many times as we like. The compiled regex methods are listed in Table 13.6 (➤ 503).

```
match = re.search(r"#[\dA-Fa-f]{6}\b", text)
```

This code snippet shows the use of an `re` module function. The regex matches HTML-style colors (such as `#C0C0AB`). If a match is found the `re.search()` function returns a match object; otherwise, it returns `None`. The methods provided by match objects are listed in Table 13.7 (➤ 507).

If we were going to use this regex repeatedly, we could compile it once and then use the compiled regex whenever we needed it:

```
color_re = re.compile(r"#[\dA-Fa-f]{6}\b")
match = color_re.search(text)
```

As we noted earlier, we use raw strings to avoid having to escape backslashes. Another way of writing this regex would be to use the character class **`[\dA-F]`** and pass the `re.IGNORECASE` flag as the last argument to the `re.compile()` call, or to use the regex **`(?i)#[\dA-F]{6}\b`** which starts with the ignore case flag.

If more than one flag is required they can be combined using the OR operator (`|`), for example, `re.MULTILINE|re.DOTALL`, or **`(?ms)`** if embedded in the regex itself.

We will round off this section by reviewing some examples, starting with some of the regexes shown in earlier sections, so as to illustrate the most commonly used functionality that the `re` module provides. Let's start with a regex to spot duplicate words:

```
double_word_re = re.compile(r"\b(?P<word>\w+)\s+(?P=word)(?!\w)",
                            re.IGNORECASE)
for match in double_word_re.finditer(text):
    print("{0} is duplicated".format(match.group("word")))
```

The regex is slightly more sophisticated than the version we made earlier. It starts at a word boundary (to ensure that each match starts at the beginning of a word), then greedily matches one or more "word" characters, then one or more whitespace characters, then the same word again—but only if the second occurrence of the word is not followed by a word character.

If the input text was "win in vain", *without* the first assertion there would be one match and one capture: `win in` `vain`. There aren't two matches because while **`(?P<word>)`** matches and captures, the **`\s+`** and **`(?P=word)`** parts only match. The use of the word boundary assertion ensures that the first word matched is a whole word, so we end up with no match or capture since there is no du-

plicate whole word. Similarly, if the input text was "one and and two let's say", *without* the last assertion there would be two matches and two captures: `one and and two let's s`ay. The use of the lookahead assertion means that the second word matched is a whole word, so we end up with one match and one capture: `one and and two let's say`.

The `for` loop iterates over every match object returned by the `finditer()` method and we use the match object's `group()` method to retrieve the captured group's text. We could just as easily (but less maintainably) have used `group(1)`—in which case we need not have named the capture group at all and just used the regex **`\b(\w+)\s+\1(?!\w)`**. Another point to note is that we could have used a word boundary **`\b`** at the end, instead of **`(?!\w)`**.

Another example we presented earlier was a regex for finding the filenames in HTML image tags. Here is how we would compile the regex, adding flags so that it is not case-sensitive, and allowing us to include comments:

```
image_re = re.compile(r"""
            <img\s+                     # start of tag
            [^>]*?                      # non-src attributes
            src=                        # start of src attribute
            (?:
                (?P<quote>["'])         # opening quote
                (?P<qimage>[^\1>]+?)    # image filename
                (?P=quote)              # closing quote
            |                           # ---or alternatively---
                (?P<uimage>[^"' >]+)    # unquoted image filename
            )
            [^>]*?                      # non-src attributes
            >                           # end of the tag
            """, re.IGNORECASE|re.VERBOSE)
image_files = []
for match in image_re.finditer(text):
    image_files.append(match.group("qimage") or
                       match.group("uimage"))
```

Again we use the `finditer()` method to retrieve each match and the match object's `group()` function to retrieve the captured texts. Each time a match is made we don't know which of the image groups (`"qimage"` or `"uimage"`) has matched, but using the `or` operator provides a neat solution for this. Since the case insensitivity applies only to **`img`** and **`src`**, we could drop the `re.IGNORECASE` flag and use **`[Ii][Mm][Gg]`** and **`[Ss][Rr][Cc]`** instead. Although this would make the regex less clear, it might make it faster since it would not require the text being matched to be set to upper- (or lower-) case—but it is likely to make a difference only if the regex was being used on a very large amount of text.

Table 13.4 *The Regular Expression Module's Functions*

Syntax	Description
`re.compile(r, f)`	Returns compiled regex `r` with its flags set to `f` if specified. (The flags are described in Table 13.5.)
`re.escape(s)`	Returns string `s` with all nonalphanumeric characters backslash-escaped—therefore, the returned string has no special regex characters
`re.findall(r, s, f)`	Returns all nonoverlapping matches of regex `r` in string `s` (influenced by the flags `f` if given). If the regex has captures, each match is returned as a tuple of captures.
`re.finditer(r, s, f)`	Returns a match object for each nonoverlapping match of regex `r` in string `s` (influenced by the flags `f` if given)
`re.match(r, s, f)`	Returns a match object if the regex `r` matches at the start of string `s` (influenced by the flags `f` if given); otherwise, returns `None`
`re.search(r, s, f)`	Returns a match object if the regex `r` matches anywhere in string `s` (influenced by the flags `f` if given); otherwise, returns `None`
`re.split(r, s, m, f)`	Returns the list of strings that results from splitting string `s` on every occurrence of regex `r` doing up to `m` splits (or as many as possible if no `m` is given, and for Python 3.1 influenced by flags `f` if given). If the regex has captures, these are included in the list between the parts they split.
`re.sub(r, x, s, m, f)`	Returns a copy of string `s` with every (or up to `m` if given, and for Python 3.1 influenced by flags `f` if given) match of regex `r` replaced with `x`—this can be a string or a function; see text
`re.subn(r, x, s m, f)`	The same as `re.sub()` except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

3.*x*

Table 13.5 *The Regular Expression Module's Flags*

Flag	Meaning
`re.A` or `re.ASCII`	Makes **`\b`**, **`\B`**, **`\s`**, **`\S`**, **`\w`**, and **`\W`** assume that strings are ASCII; the default is for these character class shorthands to depend on the Unicode specification
`re.I` or `re.IGNORECASE`	Makes the regex match case-insensitively
`re.M` or `re.MULTILINE`	Makes **`^`** match at the start and after each newline and **`$`** match before each newline and at the end
`re.S` or `re.DOTALL`	Makes **`.`** match every character including newlines
`re.X` or `re.VERBOSE`	Allows whitespace and comments to be included

Table 13.6 *Regular Expression Object Methods*

Syntax	Description
`rx.findall(s, start, end)`	Returns all nonoverlapping matches of the regex in string `s` (or in the *`start`*`:`*`end`* slice of `s`). If the regex has captures, each match is returned as a tuple of captures.
`rx.finditer(s, start, end)`	Returns a match object for each nonoverlapping match in string `s` (or in the *`start`*`:`*`end`* slice of `s`)
`rx.flags`	The flags that were set when the regex was compiled
`rx.groupindex`	A dictionary whose keys are capture group names and whose values are group numbers; empty if no names are used
`rx.match(s, start, end)`	Returns a match object if the regex matches at the start of string `s` (or at the start of the *`start`*`:`*`end`* slice of `s`); otherwise, returns `None`
`rx.pattern`	The string from which the regex was compiled
`rx.search(s, start, end)`	Returns a match object if the regex matches anywhere in string `s` (or in the *`start`*`:`*`end`* slice of `s`); otherwise, returns `None`
`rx.split(s, m)`	Returns the list of strings that results from splitting string `s` on every occurrence of the regex doing up to *`m`* splits (or as many as possible if no *`m`* is given). If the regex has captures, these are included in the list between the parts they split.
`rx.sub(x, s, m)`	Returns a copy of string `s` with every (or up to *`m`* if given) match replaced with `x`—this can be a string or a function; see text
`rx.subn(x, s m)`	The same as `re.sub()` except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

One common task is to take an HTML text and output just the plain text that it contains. Naturally we could do this using one of Python's parsers, but a simple tool can be created using regexes. There are three tasks that need to be done: delete any tags, replace entities with the characters they represent, and insert blank lines to separate paragraphs. Here is a function (taken from the `html2text.py` program) that does the job:

```
def html2text(html_text):
    def char_from_entity(match):
        code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
        return chr(code)
```

```
text = re.sub(r"<!--(?:.|\n)*?-->", "", html_text)          #1
text = re.sub(r"<[Pp][^>]*?>", "\n\n", text)                #2
text = re.sub(r"<[^>]*?>", "", text)                        #3
text = re.sub(r"&#(\d+);", lambda m: chr(int(m.group(1))), text)
text = re.sub(r"&([A-Za-z]+);", char_from_entity, text)     #5
text = re.sub(r"\n(?:[ \xA0\t]+\n)+", "\n", text)           #6
return re.sub(r"\n\n+", "\n\n", text.strip())               #7
```

The first regex, **<!--(?:.|\n)*?-->**, matches HTML comments, including those with other HTML tags nested inside them. The `re.sub()` function replaces as many matches as it finds with the replacement—deleting the matches if the replacement is an empty string, as it is here. (We can specify a maximum number of matches by giving an additional integer argument at the end.)

We are careful to use nongreedy (minimal) matching to ensure that we delete one comment for each match; if we did not do this we would delete from the start of the first comment to the end of the last comment.

In Python 3.0, the `re.sub()` function does not accept any flags as arguments, and since `.` means "any character except newline", we must look for `.` or `\n`. And we must look for these using alternation rather than a character class, since inside a character class `.` has its literal meaning, that is, period. An alternative would be to begin the regex with the flag embedded, for example, **(?s)<!--.*?-->**, or we could compile a regex object with the `re.DOTALL` flag, in which case the regex would simply be **<!--.*?-->**.

3.0

From Python 3.1, `re.split()`, `re.sub()`, and `re.subn()`, can all accept a flags argument, so we could simply use **<!--.*?-->** and pass the `re.DOTALL` flag.

3.1

The second regex, **<[Pp][^>]*?>**, matches opening paragraph tags (such as `<P>` or `<p align="center">`). It matches the opening `<p` (or `<P`), then any attributes (using nongreedy matching), and finally the closing `>`. The second call to the `re.sub()` function uses this regex to replace opening paragraph tags with two newline characters (the standard way to delimit a paragraph in a plain text file).

The third regex, **<[^>]*?>**, matches any tag and is used in the third `re.sub()` call to delete all the remaining tags.

HTML entities are a way of specifying non-ASCII characters using ASCII characters. They come in two forms: `&`*name*`;` where *name* is the name of the character—for example, `&copy;` for ©—and `&#`*digits*`;` where *digits* are decimal digits identifying the Unicode code point—for example, `&#165;` for ¥. The fourth call to `re.sub()` uses the regex **&#(\d+);**, which matches the digits form and captures the digits into capture group 1. Instead of a literal replacement text we have passed a `lambda` function. When a function is passed to `re.sub()` it calls the function once for each time it matches, passing the match object as the function's sole argument. Inside the `lambda` function we retrieve the digits (as a

string), convert to an integer using the built-in `int()` function, and then use the built-in `chr()` function to obtain the Unicode character for the given code point. The function's return value (or in the case of a `lambda` expression, the result of the expression) is used as the replacement text.

The fifth `re.sub()` call uses the regex **`&([A-Za-z]+);`** to capture named entities. The standard library's `html.entities` module contains dictionaries of entities, including `name2codepoint` whose keys are entity names and whose values are integer code points. The `re.sub()` function calls the local `char_from_entity()` function every time it has a match. The `char_from_entity()` function uses `dict.get()` with a default argument of `0xFFFD` (the code point of the standard Unicode replacement character—often depicted as ❓). This ensures that a code point is always retrieved and it is used with the `chr()` function to return a suitable character to replace the named entity with—using the Unicode replacement character if the entity name is invalid.

The sixth `re.sub()` call's regex, **`\n(?:[ \xA0\t]+\n)+`**, is used to delete lines that contain only whitespace. The character class we have used contains a space, a nonbreaking space (which ` ` entities are replaced with in the preceding regex), and a tab. The regex matches a newline (the one at the end of a line that precedes one or more whitespace-only lines), then at least one (and as many as possible) lines that contain only whitespace. Since the match includes the newline, from the line preceding the whitespace-only lines we must replace the match with a single newline; otherwise, we would delete not just the whitespace-only lines but also the newline of the line that preceded them.

The result of the seventh and last `re.sub()` call is returned to the caller. This regex, **`\n\n+`**, is used to replace sequences of two or more newlines with exactly two newlines, that is, to ensure that each paragraph is separated by just one blank line.

In the HTML example none of the replacements were directly taken from the match (although HTML entity names and numbers were used), but in some situations the replacement might need to include all or some of the matching text. For example, if we have a list of names, each of the form *Forename Middlename1 … MiddlenameN Surname*, where there may be any number of middle names (including none), and we want to produce a new version of the list with each item of the form *Surname*, *Forename Middlename1 … MiddlenameN*, we can easily do so using a regex:

```
new_names = []
for name in names:
    name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
    new_names.append(name)
```

The first part of the regex, **`(\w+(?:\s+\w+)*)`**, matches the forename with the first **`\w+`** expression and zero or more middle names with the **`(?:\s+\w+)*`** ex-

pression. The middle name expression matches zero or more occurrences of whitespace followed by a word. The second part of the regex, **\s+(\w+)**, matches the whitespace that follows the forename (and middle names) and the surname.

If the regex looks a bit too much like line noise, we can use named capture groups to improve legibility and make it more maintainable:

```
name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>", name)
```

Captured text can be referred to in a `sub()` or `subn()` function or method by using the syntax *****i* or **\g<***id***>** where *i* is the number of the capture group and *id* is the name or number of the capture group—so **\1** is the same as **\g<1>**, and in this example, the same as **\g<forenames>**. This syntax can also be used in the string passed to a match object's `expand()` method.

Why doesn't the first part of the regex grab the entire name? After all, it is using greedy matching. In fact it will, but then the match will fail because although the middle names part can match zero or more times, the surname part must match exactly once, but the greedy middle names part has grabbed everything. Having failed, the regular expression engine will then backtrack, giving up the last "middle name" and thus allowing the surname to match. Although greedy matches match as much as possible, they stop if matching more would make the match fail.

For example, if the name is "John le Carré", the regex will first match the entire name, that is, `John le Carré`. This satisfies the first part of the regex but leaves nothing for the surname part to match, and since the surname is mandatory (it has an implicit quantifier of 1), the regex has failed. Since the middle names part is quantified by `*`, it can match zero or more times (currently it is matching twice, " le" and " Carré"), so the regular expression engine can make it give up some of its match without causing it to fail. Therefore, the regex backtracks, giving up the last `\s+\w+` (i.e., " Carré"), so the match becomes `John le Carré` with the match satisfying the whole regex and with the two match groups containing the correct texts.

There's one weakness in the regex as written: It doesn't cope correctly with forenames that are written using an initial, such as "James W. Loewen", or "J. R. R. Tolkein". This is because **\w** matches word characters and these don't include period. One obvious—but incorrect—solution is to change the forenames part of the regex's **\w+** expression to **[\w.]+**, in both places that it occurs. A period in a character class is taken to be a literal period, and character class shorthands retain their meaning inside character classes, so the new expression matches word characters or periods. But this would allow for names like ".", "..", ".A", ".A.", and so on. In view of this, a more subtle approach is required.

Table 13.7 *Match Object Attributes and Methods*

Syntax	Description
`m.end(g)`	Returns the end position of the match in the text for group *g* if given (or for group 0, the whole match); returns -1 if the group did not participate in the match
`m.endpos`	The search's end position (the end of the text or the *end* given to `match()` or `search()`)
`m.expand(s)`	Returns string `s` with capture markers (`\1`, `\2`, `\g<name>`, and similar) replaced by the corresponding captures
`m.group(g, ...)`	Returns the numbered or named capture group `g`; if more than one is given a tuple of corresponding capture groups is returned (the whole match is group 0)
`m.groupdict(default)`	Returns a dictionary of all the named capture groups with the names as keys and the captures as values; if a *default* is given this is the value used for capture groups that did not participate in the match
`m.groups(default)`	Returns a tuple of all the capture groups starting from 1; if a *default* is given this is the value used for capture groups that did not participate in the match
`m.lastgroup`	The name of the highest numbered capturing group that matched or `None` if there isn't one or if no names are used
`m.lastindex`	The number of the highest capturing group that matched or `None` if there isn't one
`m.pos`	The start position to look from (the start of the text or the *start* given to `match()` or `search()`)
`m.re`	The regex object which produced this match object
`m.span(g)`	Returns the start and end positions of the match in the text for group *g* if given (or for group 0, the whole match); returns `(-1, -1)` if the group did not participate in the match
`m.start(g)`	Returns the start position of the match in the text for group *g* if given (or for group 0, the whole match); returns -1 if the group did not participate in the match
`m.string`	The string that was passed to `match()` or `search()`

```
name = re.sub(r"(?P<forenames>\w+\.?(?:\s+\w+\.?)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>", name)
```

Here we have changed the forenames part of the regex (the first line). The first part of the forenames regex matches one or more word characters optionally followed by a period. The second part matches at least one whitespace charac-

ter, then one or more word characters optionally followed by a period, with the whole of this second part itself matching zero or more times.

When we use alternation (|) with two or more alternatives capturing, we don't know which alternative matched, so we don't know which capture group to retrieve the captured text from. We can of course iterate over all the groups to find the nonempty one, but quite often in this situation the match object's `lastindex` attribute can give us the number of the group we want. We will look at one last example to illustrate this and to give us a little bit more regex practice.

Suppose we want to find out what encoding an HTML, XML, or Python file is using. We could open the file in binary mode, and read, say, the first 1 000 bytes into a `bytes` object. We could then close the file, look for an encoding in the `bytes`, and reopen the file in text mode using the encoding we found or using a fallback encoding (such as UTF-8). The regex engine expects regexes to be supplied as strings, but the text the regex is applied to can be a `str`, `bytes`, or `bytearray` object, and when `bytes` or `bytearray` objects are used, all the functions and methods return `bytes` instead of strings, and the `re.ASCII` flag is implicitly switched on.

For HTML files the encoding is normally specified in a `<meta>` tag (if specified at all), for example, `<meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'/>`. XML files are UTF-8 by default, but this can be overridden, for example, `<?xml version="1.0" encoding="Shift_JIS"?>`. Python 3 files are also UTF-8 by default, but again this can be overridden by including a line such as `# encoding: latin1` or `# -*- coding: latin1 -*-` immediately after the shebang line.

Here is how we would find the encoding, assuming that the variable `binary` is a `bytes` object containing the first 1 000 bytes of an HTML, XML, or Python file:

```
match = re.search(r"""(?<![-\w])                    #1
                      (?:(?:en)?coding|charset)     #2
                      (?:=(["']?)([-\w]+)(?(1)\1)   #3
                      |:\s*([-\w]+))""".encode("utf8"),
                  binary, re.IGNORECASE|re.VERBOSE)
encoding = match.group(match.lastindex) if match else b"utf8"
```

To search a `bytes` object we must specify a pattern that is also a `bytes` object. In this case we want the convenience of using a raw string, so we use one and convert it to a `bytes` object as the `re.search()` function's first argument.

The first part of the regex itself is a lookbehind assertion that says that the match cannot be preceded by a hyphen or a word character. The second part matches "encoding", "coding", or "charset" and could have been written as **`(?:encoding|coding|charset)`**. We have made the third part span two lines to emphasise the fact that it has two alternating parts, **`=(["']?)([-\w]+)(?(1)\1)`**

Conditional matching 498 ◀

and `:\s*([-\w]+)`, only one of which can match. The first of these matches an equals sign followed by one or more word or hyphen characters (optionally enclosed in matching quotes using a conditional match), and the second matches a colon and then optional whitespace followed by one or more word or hyphen characters. (Recall that a hyphen inside a character class is taken to be a literal hyphen if it is the first character; otherwise, it means a range of characters, for example, `[0-9]`.)

We have used the `re.IGNORECASE` flag to avoid having to write `(?:(?:[Ee][Nn])?[Cc][Oo][Dd][Ii][Nn][Gg]|[Cc][Hh][Aa][Rr][Ss][Ee][Tt])` and we have used the `re.VERBOSE` flag so that we can lay out the regex neatly and include comments (in this case just numbers to make the parts easy to refer to in this text).

There are three capturing match groups, all in the third part: `(["'])?` which captures the optional opening quote, `([-\w]+)` which captures an encoding that follows an equals sign, and the second `([-\w]+)` (on the following line) that captures an encoding that follows a colon. We are only interested in the encoding, so we want to retrieve either the second or third capture group, only one of which can match since they are alternatives. The `lastindex` attribute holds the index of the last *matching* capture group (either 2 or 3 when a match occurs in this example), so we retrieve whichever matched, or use a default encoding if no match was made.

We have now seen all of the most frequently used `re` module functionality in action, so we will conclude this section by mentioning one last function. The `re.split()` function (or the regex object's `split()` method) can split strings based on a regex. One common requirement is to split a text on whitespace to get a list of words. This can be done using `re.split(r"\s+", text)` which returns a list of words (or more precisely a list of strings, each of which matches `\S+`). Regular expressions are very powerful and useful, and once they are learned, it is easy to see all text problems as requiring a regex solution. But sometimes using string methods is both sufficient and more appropriate. For example, we can just as easily split on whitespace by using `text.split()` since the `str.split()` method's default behavior (or with a first argument of `None`) is to split on `\s+`.

Summary

Regular expressions offer a powerful way of searching texts for strings that match a particular pattern, and for replacing such strings with other strings which themselves can depend on what was matched.

In this chapter we saw that most characters are matched literally and are implicitly quantified by `{1}`. We also learned how to specify character classes—sets of characters to match—and how to negate such sets and include

ranges of characters in them without having to write each character individually.

We learned how to quantify expressions to match a specific number of times or to match from a given minimum to a given maximum number of times, and how to use greedy and nongreedy matching. We also learned how to group one or more expressions together so that they can be quantified (and optionally captured) as a unit.

The chapter also showed how what is matched can be affected by using various assertions, such as positive and negative lookahead and lookbehind, and by various flags, for example, to control the interpretation of the period and whether to use case-insensitive matching.

The final section showed how to put regexes to use within the context of Python programs. In this section we learned how to use the functions provided by the `re` module, and the methods available from compiled regexes and from match objects. We also learned how to replace matches with literal strings, with literal strings that contain backreferences, and with the results of function calls or `lambda` expressions, and how to make regexes more maintainable by using named captures and comments.

Exercises

1. In many contexts (e.g., in some web forms), users must enter a phone number, and some of these irritate users by accepting only a specific format. Write a program that reads U.S. phone numbers with the three-digit area and seven-digit local codes accepted as ten digits, or separated into blocks using hyphens or spaces, and with the area code optionally enclosed in parentheses. For example, all of these are valid: 555-123-1234, (555) 1234567, (555) 123 1234, and 5551234567. Read the phone numbers from `sys.stdin` and for each one echo the number in the form "(999) 999 9999" or report an error for any that are invalid, or that don't have exactly ten digits.

   The regex to match these phone numbers is about ten lines long (in verbose mode) and is quite straightforward. A solution is provided in `phone.py`, which is about twenty-five lines long.

2. Write a small program that reads an XML or HTML file specified on the command line and for each tag that has attributes, outputs the name of the tag with its attributes shown underneath. For example, here is an extract from the program's output when given one of the Python documentation's `index.html` files:

```
html
    xmlns = http://www.w3.org/1999/xhtml
```

```
meta
    http-equiv = Content-Type
    content = text/html; charset=utf-8
li
    class = right
    style = margin-right: 10px
```

One approach is to use two regexes, one to capture tags with their attributes and another to extract the name and value of each attribute. Attribute values might be quoted using single or double quotes (in which case they may contain whitespace and the quotes that are not used to enclose them), or they may be unquoted (in which case they cannot contain whitespace or quotes). It is probably easiest to start by creating a regex to handle quoted and unquoted values separately, and then merging the two regexes into a single regex to cover both cases. It is best to use named groups to make the regex more readable. This is not easy, especially since backreferences cannot be used inside character classes.

A solution is provided in `extract_tags.py`, which is less than 35 lines long. The tag and attributes regex is just one line. The attribute name–value regex is half a dozen lines and uses alternation, conditional matching (twice, with one nested inside the other), and both greedy and nongreedy quantifiers.

Index

All functions and methods are listed under their class or module, and in most cases also as top-level terms in their own right. For modules that contain classes, look under the class for its methods. Where a method or function name is close enough to a concept, the concept is not usually listed. For example, there is no entry for "splitting strings", but there are entries for the `str.split()` *method.*

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z