



Mark Summerfield

Second Edition

Programming in Python 3

A Complete Introduction to the
Python Language

Developer's Library



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

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

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

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 `\xHH`, `\uHHHH`, and `\UHHHHHHHH` syntaxes.

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

Mean-
ing 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 `ls`, 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 non-zero 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 $e\{0,1\}$	Greedy match zero or one occurrence of expression e
$e??$ or $e\{0,1\}?$	Nongreedy match zero or one occurrence of expression e
$e+$ or $e\{1,\}$	Greedy match one or more occurrences of expression e
$e+?$ or $e\{1,\}?$	Nongreedy match one or more occurrences of expression e
e^* or $e\{0,\}$	Greedy match zero or more occurrences of expression e
$e*?$ or $e\{0,\}?$	Nongreedy match zero or more occurrences of expression e
$e\{m\}$	Match exactly m occurrences of expression e
$e\{m,\}$	Greedy match at least m occurrences of expression e
$e\{m,\}?$	Nongreedy match at least m occurrences of expression e
$e\{,n\}$	Greedy match at most n occurrences of expression e
$e\{,n\}?$	Nongreedy match at most n occurrences of expression e
$e\{m,n\}$	Greedy match at least m and at most n occurrences of expression e
$e\{m,n\}?$	Nongreedy 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 ``, 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.

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 `[\\t]`. Here's the `key=value` regex with comments:

```
^[ \\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
```

Regex
flags

► 502

Table 13.3 Regular Expression Assertions

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

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|([^\>]+))`. 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
|
  (?P<uimage>[^\1 >]+) # 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 say. 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
    |
        (?P<uimage>[^\1 >]+) # 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
<code>re.compile(r, f)</code>	Returns compiled regex <code>r</code> with its flags set to <code>f</code> if specified. (The flags are described in Table 13.5.)
<code>re.escape(s)</code>	Returns string <code>s</code> with all nonalphanumeric characters backslash-escaped—therefore, the returned string has no special regex characters
<code>re.findall(r, s, f)</code>	Returns all nonoverlapping matches of regex <code>r</code> in string <code>s</code> (influenced by the flags <code>f</code> if given). If the regex has captures, each match is returned as a tuple of captures.
<code>re.finditer(r, s, f)</code>	Returns a match object for each nonoverlapping match of regex <code>r</code> in string <code>s</code> (influenced by the flags <code>f</code> if given)
<code>re.match(r, s, f)</code>	Returns a match object if the regex <code>r</code> matches at the start of string <code>s</code> (influenced by the flags <code>f</code> if given); otherwise, returns <code>None</code>
<code>re.search(r, s, f)</code>	Returns a match object if the regex <code>r</code> matches anywhere in string <code>s</code> (influenced by the flags <code>f</code> if given); otherwise, returns <code>None</code>
<code>re.split(r, s, m, f)</code>	Returns the list of strings that results from splitting string <code>s</code> on every occurrence of regex <code>r</code> doing up to <code>m</code> splits (or as many as possible if no <code>m</code> is given, and for Python 3.1 influenced by flags <code>f</code> if given). If the regex has captures, these are included in the list between the parts they split.
<code>re.sub(r, x, s, m, f)</code>	Returns a copy of string <code>s</code> with every (or up to <code>m</code> if given, and for Python 3.1 influenced by flags <code>f</code> if given) match of regex <code>r</code> replaced with <code>x</code> —this can be a string or a function; see text
<code>re.subn(r, x, s, m, f)</code>	The same as <code>re.sub()</code> except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

Table 13.5 *The Regular Expression Module's Flags*

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

Table 13.6 *Regular Expression Object Methods*

Syntax	Description
<code>rx.findall(s, start, end)</code>	Returns all nonoverlapping matches of the regex in string <i>s</i> (or in the <i>start:end</i> slice of <i>s</i>). If the regex has captures, each match is returned as a tuple of captures.
<code>rx.finditer(s, start, end)</code>	Returns a match object for each nonoverlapping match in string <i>s</i> (or in the <i>start:end</i> slice of <i>s</i>)
<code>rx.flags</code>	The flags that were set when the regex was compiled
<code>rx.groupindex</code>	A dictionary whose keys are capture group names and whose values are group numbers; empty if no names are used
<code>rx.match(s, start, end)</code>	Returns a match object if the regex matches at the start of string <i>s</i> (or at the start of the <i>start:end</i> slice of <i>s</i>); otherwise, returns <code>None</code>
<code>rx.pattern</code>	The string from which the regex was compiled
<code>rx.search(s, start, end)</code>	Returns a match object if the regex matches anywhere in string <i>s</i> (or in the <i>start:end</i> slice of <i>s</i>); otherwise, returns <code>None</code>
<code>rx.split(s, m)</code>	Returns the list of strings that results from splitting string <i>s</i> on every occurrence of the regex doing up to <i>m</i> splits (or as many as possible if no <i>m</i> is given). If the regex has captures, these are included in the list between the parts they split.
<code>rx.sub(x, s, m)</code>	Returns a copy of string <i>s</i> with every (or up to <i>m</i> if given) match replaced with <i>x</i> —this can be a string or a function; see text
<code>rx.subn(x, s, m)</code>	The same as <code>re.sub()</code> 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) #4
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 `<!--.*?-->`.

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.

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, `©`; for ©—and `&#digits`; where `digits` are decimal digits identifying the Unicode code point—for example, `¥`; 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

3.0

3.1

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

```
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 1000 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 1000 bytes of an HTML, XML, or Python file:

```
match = re.search(r"""(?![\w])                #1
                  (?:(?:en)?coding|charset)   #2
                  (?:=["'])?([\w+])?(?1)\1    #3
                  |:\s*([\w+])""",
                  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`

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

- `!=` (not equal operator), 23, 241, 242, 259, 379
- `#` comment character, 10
- `%` (modulus/remainder operator), 55, 253
- `%=` (modulus augmented assignment operator), 253
- `&` (bitwise AND operator), 57, 122, 123, 130, 253
- `&=` (bitwise AND augmented assignment operator), 123, 253
- `()` (tuple creation operator, function and method call operator, expression operator), 341, 377, 383
- `*` (multiplication operator, replication operator, sequence unpacker, from ... import operator), 55, 72, 90, 108, 110, 114, 140, 197, 200–201, 253, 336, 379, 460
- `*=` (multiplication augmented assignment operator, replication augmented assignment operator), 72, 108, 114, 253
- `**` (power/exponentiation operator, mapping unpacker), 55, 179, 253, 304, 379
- `**=` (power/exponentiation augmented assignment operator), 253
- `+` (addition operator, concatenation operator), 55, 108, 114, 140, 253
- `+=` (addition augmented assignment operator, append/extend operator), 108, 114, 115, 144, 253
- `-` (subtraction operator, negation operator), 55, 122, 123, 253
- `-=` (subtraction augmented assignment operator), 123, 253
- `/` (division operator), 31, 55, 253
- `/=` (division augmented assignment operator), 253
- `//` (truncating division operator), 55, 253, 330
- `//=` (truncating division augmented assignment operator), 253
- `<` (less than operator), 123, 145, 242, 259, 379
- `<<` (int shift left operator), 57, 253
- `<<=` (int shift left augmented assignment operator), 253
- `<=` (less than or equal to operator), 123, 242, 259, 379
- `=` (name binding operator, object reference creation and assignment operator), 16, 146
- `==` (equal to operator), 23, 241, 242, 254, 259, 379
- `>` (greater than operator), 123, 242, 259, 379
- `>=` (greater than or equal to operator), 123, 242, 259, 379
- `>>` (int shift right operator), 57, 253

>>= (int shift right augmented assignment operator), 253
 @ (decorator operator), 246–248
 [] (indexing operator, item access operator, slicing operator), 69, 108, 110, 113, 114, 116, 117, 262, 264, 273, 274, 278, 279, 293
 \n (newline character, statement terminator), 66
 ^ (bitwise XOR operator), 57, 122, 123, 253
 ^= (bitwise XOR augmented assignment operator), 123, 253
 _ (underscore), 53
 | (bitwise OR operator), 57, 122, 123, 253
 |= (bitwise OR augmented assignment operator), 123, 253
 ~ (bitwise NOT operator), 57, 253

A

abc module

ABCMeta type, 381, 384, 387
 @abstractmethod(), 384, 387
 abstractproperty(), 384, 387
 __abs__(), 253
 abs() (built-in), 55, 56, 96, 145, 253
 abspath() (os.path module), 223, 406
 abstract base class (ABC), 269, 380–388
 see also collections and numbers modules
 Abstract.py (example), 386
 Abstract Syntax Tree (AST), 515
 @abstractmethod() (abc module), 384, 387
 abstractproperty() (abc module), 384, 387
 accelerator, keyboard, 574, 580, 592
 access control, 238, 249, 270, 271
 acos() (math module), 60
 acosh() (math module), 60
 __add__() (+), 55, 253

add() (set type), 123
 aggregating data, 111
 aggregation, 269
 aifc module, 219
 algorithm, for searching, 217, 272
 algorithm, for sorting, 145, 282
 algorithm, MD5, 449, 452
 __all__ (attribute), 197, 200, 201
 all() (built-in), 140, 184, 396, 397
 alternation, regex, 494–495
 __and__() (&), 57, 251, 253, 257
 and (logical operator), 58
 annotations, 360–363
 __annotations__ (attribute), 360
 anonymous functions; *see* lambda statement
 any() (built-in), 140, 205, 396, 397
 append()
 bytearray type, 299
 list type, 115, 117, 118, 271
 archive files, 219
 arguments, command-line, 215
 arguments, function, 379
 default, 173, 174, 175
 immutable, 175
 keyword, 174–175, 178, 179, 188, 189, 362
 mutable, 175
 positional, 173–175, 178, 179, 189, 362
 unpacking, 177–180
 arguments, interpreter, 185, 198, 199
 argv list (sys module), 41, 343
 array module, 218
 arraysize attribute (cursor object), 482
 as_integer_ratio() (float type), 61
 as (binding operator), 163, 196, 369
 ascii() (built-in), 68, 83
 ASCII encoding, 9, 68, 91–94, 220, 293, 504
 see also character encodings
 asin() (math module), 60
 asinh() (math module), 60

askopenfilename() (tkinter.filedialog module), 586
 asksaveasfilename() (tkinter.filedialog module), 585
 askyesno() (tkinter.messagebox module), 589
 askyesnocancel() (tkinter.messagebox module), 584
 assert (statement), 184–185, 205, 208, 247
 AssertionError (exception), 184
 assertions, regex, 496–499
 associativity, 517–518, 551, 565
 AST (Abstract Syntax Tree), 515
 asynchat module, 225
 asyncore module, 225
 atan() (math module), 60
 atan2() (math module), 60
 atanh() (math module), 60
 attrgetter() (operator module), 369, 397
 attribute
 __all__, 197, 200, 201
 __annotations__, 360
 __call__, 271, 350, 392
 __class__, 252, 364, 366
 __dict__, 348, 363, 364
 __doc__, 357
 __file__, 441
 __module__, 243
 __name__, 206, 252, 357, 362, 377
 private, 238, 249, 270, 271, 366
 __slots__, 363, 373, 375, 394
 attribute access methods, table of, 365
 AttributeError (exception), 240, 241, 275, 350, 364, 366
 attributes, 197, 200, 201, 206, 246–248, 252, 271, 351, 363–367
 attributes, mutable and immutable, 264
 audio-related modules, 219
 audioop module, 219

augmented assignment, 31–33, 56, 108, 114

B

-B option, interpreter, 199
 backreferences, regex, 495
 backtrace; *see* traceback
 backups, 414
 base64 module, 219, 220–221
 basename() (os.path module), 223
 Berkeley DB, 475
 bigdigits.py (example), 39–42
 BikeStock.py (example), 332–336
 bin() (built-in), 55, 253
 binary data, 220
 binary files, 295–304, 324–336
 binary numbers, 56
 binary search, 272
 see also bisect module
 BinaryRecordFile.py (example), 324–332
 bindings, event, 576
 bindings, keyboard, 576
 bisect module, 217, 272
 bit_length() (int type), 57
 bitwise operators, table of, 57
 block structure, using indentation, 27
 blocks.py (example), 525–534, 543–547, 559–562
 BNF (Backus–Naur Form), 515–518
 bookmarks-tk.pyw (example), 578–593
 __bool__(), 250, 252, 258
 bool() (built-in), 250
 bool type, 58
 bool() (built-in), 58, 250, 309
 conversion, 58
 Boolean expressions, 26, 54
 branching; *see* if statement
 branching, with dictionaries, 340–341
 break (statement), 161, 162

built-in

abs(), 55, 56, 96, 145, 253
 all(), 140, 184, 396, 397
 any(), 140, 205, 396, 397
 ascii(), 68, 83
 bin(), 55, 253
 bool(), 58, 250, 309
 chr(), 67, 90, 504
 @classmethod(), 257, 278
 compile(), 349
 complex(), 63, 253
 delattr(), 349
 dict(), 127, 147
 dir(), 52, 172, 349, 365
 divmod(), 55, 253
 enumerate(), 139–141, 398, 524
 eval(), 242, 243, 258, 266, 275, 344, 349, 379
 exec(), 260, 345–346, 348, 349, 351
 filter(), 395, 397
 float(), 61, 154, 253
 format(), 250, 254
 frozenset(), 125
 getattr(), 349, 350, 364, 368, 374, 391, 409
 globals(), 345, 349
 hasattr(), 270, 349, 350, 391
 hash(), 241, 250, 254
 help(), 61, 172
 hex(), 55, 253
 id(), 254
 __import__(), 349, 350
 input(), 34, 96
 int(), 55, 61, 136, 253, 309
 isinstance(), 170, 216, 242, 270, 382, 390, 391
 issubclass(), 390
 iter(), 138, 274, 281
 len(), 71, 114, 122, 140, 265, 275
 list(), 113, 147
 locals(), 81, 82, 97, 154, 188, 189, 190, 345, 349, 422, 423, 484
 map(), 395, 397, 539
 max(), 140, 154, 396, 397

built-in (cont.)

min(), 140, 396, 397
 next(), 138, 343, 401
 oct(), 55, 253
 ord(), 67, 90, 364
 pow(), 55
 print(), 11, 180, 181, 214, 422
 @property(), 246–248, 376, 385, 394
 range(), 115, 118, 119, 140, 141–142, 365
 repr(), 242, 250
 reversed(), 72, 140, 144, 265, 274
 round(), 55, 56, 61, 252, 253, 258
 set(), 122, 147
 setattr(), 349, 379, 409
 sorted(), 118, 133, 140, 144–146, 270
 @staticmethod(), 255
 str(), 65, 136, 243, 250
 sum(), 140, 396, 397
 super(), 241, 244, 256, 276, 282, 381, 385
 tuple(), 108
 type(), 18, 348, 349
 vars(), 349
 zip(), 127, 140, 143–144, 205, 389
 builtins module, 364
 Button type (tkinter module), 581, 591
 byte-code, 198
 byte order, 297
 bytearray type, 293, 301, 383, 418–419, 462
 append(), 299
 capitalize(), 299
 center(), 299
 count(), 299
 decode(), 93, 94, 299, 326, 336, 443
 endswith(), 299
 expandtabs(), 299
 extend(), 299, 301, 462
 find(), 299

bytearray type (*cont.*)

- fromhex(), 293, 299
- index(), 299
- insert(), 293, 299
- isalnum(), 299
- isalpha(), 299
- isdigit(), 299
- islower(), 299
- isspace(), 299
- istitle(), 300
- isupper(), 300
- join(), 300
- ljust(), 300
- lower(), 300
- lstrip(), 300
- methods, table of, 299, 300, 301
- partition(), 300
- pop(), 293, 300
- remove(), 300
- replace(), 293, 300
- reverse(), 300
- rfind(), 299
- rindex(), 299
- rjust(), 300
- rpartition(), 300
- rsplit(), 300
- rstrip(), 300
- split(), 300
- splitlines(), 300
- startswith(), 300
- strip(), 300
- swapcase(), 300
- title(), 300
- translate(), 300
- upper(), 293, 301
- zfill(), 301

bytes type, 93, 293, 297, 383,
418–419

- capitalize(), 299
- center(), 299
- count(), 299
- decode(), 93, 94, 226, 228, 299,
302, 326, 336, 418, 443
- endswith(), 299
- expandtabs(), 299

bytes type (*cont.*)

- find(), 299
- fromhex(), 293, 299
- index(), 299
- isalnum(), 299
- isalpha(), 299
- isdigit(), 299
- islower(), 299
- isspace(), 299
- istitle(), 300
- isupper(), 300
- join(), 300
- literal, 93, 220
- ljust(), 300
- lower(), 300
- lstrip(), 300
- methods, table of, 299, 300, 301
- partition(), 300
- replace(), 293, 300
- rfind(), 299
- rindex(), 299
- rjust(), 300
- rpartition(), 300
- rsplit(), 300
- rstrip(), 300
- split(), 300
- splitlines(), 300
- startswith(), 300
- strip(), 300
- swapcase(), 300
- title(), 300
- translate(), 300
- upper(), 293, 301
- zfill(), 301
- .bz2 (extension), 219
- bz2 module, 219

C

- c option, interpreter, 198
- calcsize() (struct module), 297
- calendar module, 216
- __call__ (attribute), 271, 350, 392
- __call__ (), 367, 368

- call() (subprocess module), 209
- callable; *see* functions and methods
- Callable ABC (collections module), 383, 391
- callable objects, 271, 367
- capitalize()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73
- captures, regex, 494–495, 506
- car_registration_server.py (example), 464–471
- car_registration.py (example), 458–464
- case statement; *see* dictionary branching
- category() (unicodedata module), 361
- ceil() (math module), 60
- center()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73
- cgi module, 225
- cglib module, 225
- chaining exceptions, 419–420
- changing dictionaries, 128
- changing lists, 115
- character class, regex, 491
- character encodings, 9, 91–94, 314
 - see also* ASCII encoding, Latin 1 encoding, Unicode
- CharGrid.py (example), 207–212
- chdir() (os module), 223
- checktags.py (example), 169
- choice() (random module), 142
- chr() (built-in), 67, 90, 504
- class (statement), 238, 244, 378, 407
- __class__ (attribute), 252, 364, 366
- class, mixin, 466
- class decorators, 378–380, 407–409
- class methods, 257
- class variables, 255, 465
- classes, immutable, 256, 261
- @classmethod(), 257, 278
- clear()
 - dict type, 129
 - set type, 123
- close()
 - connection object, 481
 - coroutines, 399, 401, 402
 - cursor object, 482
 - file object, 131, 167, 325
- closed attribute (file object), 325
- closures, 367, 369
- cmath module, 63
- code comments, 10
- collation order (Unicode), 68–69
- collections; *see* dict, list, set, and tuple types
- collections, copying, 146–148
- collections module, 217–219, 382
 - Callable ABC, 383, 391
 - classes, table of, 383
 - Container ABC, 383
 - defaultdict type, 135–136, 153, 183, 450
 - deque type, 218, 383
 - Hashable ABC, 383
 - Iterable ABC, 383
 - Iterator ABC, 383
 - Mapping ABC, 383
 - MutableMapping ABC, 269, 383
 - MutableSequence ABC, 269, 383
 - MutableSet ABC, 383
 - namedtuple type, 111–113, 234, 365, 523
 - OrderedDict type, 136–138, 218
 - Sequence ABC, 383
 - Set ABC, 383
 - Sized ABC, 383
- combining functions, 395–397, 403–407
- command-line arguments; *see* sys.argv list
- comment character (#), 10
- commit() (connection object), 481, 483
- comparing files and directories, 223

- comparing objects, 23, 242
- comparing strings, 68–69
- comparisons; *see* <, <=, ==, !=, >, and >= operators
- compile()
 - built-in, 349
 - re module, 310, 400, 500, 501, 502, 521, 524
- `__complex__()`, 253
- `complex()` (built-in), 253
- Complex ABC (numbers module), 381
- complex type, 62–63, 381
 - `complex()` (built-in), 63, 253
 - `conjugate()`, 62
 - imag attribute, 62
 - real attribute, 62
- composing functions, 395–397, 403–407
- composition, 269
- comprehensions; *see under* dict, list, and set types
- compressing files, 219
- concatenation
 - of lists, 114
 - of strings, 71
 - of tuples, 108
- concepts, object-oriented, 235
- conditional branching; *see if* statement
- conditional expression, 160, 176, 189
- configparser module, 220, 519
- configuration files, 220
- `conjugate()` (complex type), 62
- `connect()` (sqlite3 module), 481
- connection object
 - `close()`, 481
 - `commit()`, 481, 483
 - `cursor()`, 481, 483
 - methods, table of, 481
 - `rollback()`, 481
 - see also* cursor object
- constant set; *see* frozenset type
- constants, 149, 180, 364–365
- Container ABC (collections module), 383
 - `__contains__()`, 265, 274
- context managers, 369–372, 452, 464, 466
- contextlib module, 370, 466
- continue (statement), 161, 162
- conversions, 57
 - date and time, 217
 - float to int, 61
 - int to character, 67
 - int to float, 61
 - to bool, 58
 - to complex, 63
 - to dict, 127
 - to float, 59, 154
 - to int, 15, 55
 - to list, 113, 139
 - to set, 122
 - to str, 15, 65
 - to tuple, 108, 139
- `convert-incident.py` (example), 289–323
- Coordinated Universal Time (UTC), 216
 - `__copy__()`, 275
- `copy()`
 - copy module, 147, 275, 282, 469
 - dict type, 129, 147
 - frozenset type, 123
 - set type, 123, 147
- copy module, 245
 - `copy()`, 147, 275, 282, 469
 - `deepcopy()`, 148
- copying collections, 146–148
- copying objects, 245
- `copysign()` (math module), 60
- coroutines, 399–407
 - `close()`, 399, 401, 402
 - decorator, 401
 - `send()`, 401, 402, 405, 406
- `cos()` (math module), 60
- `cosh()` (math module), 60
- `count()`
 - bytearray type, 299

- count() (*cont.*)
 - bytes type, 299
 - list type, 115
 - str type, 73, 75
 - tuple type, 108
 - cProfile module, 360, 432, 434–437
 - CREATE TABLE (SQL statement), 481
 - creation, of objects, 240
 - .csv (extension), 220
 - csv module, 220
 - csv2html.py (example), 97–102
 - csv2html2_opt.py (example), 215
 - ctypes module, 229
 - currying; *see* partial function application
 - cursor() (connection object), 481, 483
 - cursor object
 - arraysize attribute, 482
 - close(), 482
 - description attribute, 482
 - execute(), 481, 482, 483, 484, 485, 486, 487
 - executemany(), 482
 - fetchall(), 482, 485
 - fetchmany(), 482
 - fetchone(), 482, 484, 486
 - methods, table of, 482
 - rowcount attribute, 482
 - see also* connection object
 - custom exceptions, 168–171, 208
 - custom functions; *see* functions
 - custom modules and packages, 195–202
- D**
- daemon threads, 447, 448, 451
 - data persistence, 220
 - data structures; *see* dict, list, set, and tuple types
 - data type conversion; *see* conversions
 - database connection; *see* connection object
 - database cursor; *see* cursor object
 - datetime.date type (datetime module), 306
 - fromordinal(), 301, 304
 - today(), 187, 477
 - toordinal(), 301
 - datetime.datetime type (datetime module)
 - now(), 217
 - strptime(), 309
 - utcnow(), 217
 - datetime module, 186, 216
 - date type, 301, 309
 - datetime type, 309
 - DB-API; *see* connection object and cursor object
 - deadlock, 445
 - __debug__ constant, 360
 - debug (normal) mode; *see* PYTHONOPTIMIZE
 - debuggers; *see* IDLE and pdb module
 - decimal module, 63–65
 - Decimal(), 64
 - Decimal type, 63–65, 381
 - decode()
 - bytearray type, 93, 94, 299, 326, 336, 443
 - bytes type, 93, 94, 226, 228, 299, 302, 326, 336, 418, 443
 - Decorate, Sort, Undecorate (DSU), 140, 145
 - decorating methods and functions, 356–360
 - decorator
 - class, 378–380, 407–409
 - @classmethod(), 257, 278
 - @functools.wraps(), 357
 - @property(), 246–248, 376, 385, 394
 - @staticmethod(), 255
 - dedent() (textwrap module), 307

- deep copying; *see* copying collections
- deepcopy() (copy module), 148
- def (statement), 37, 173–176, 209, 238
- default arguments, 173, 174, 175
- defaultdict type (collections module), 135–136, 153, 183, 450
- degrees() (math module), 60
- del (statement), 116, 117, 127, 250, 265, 273, 365
- __del__(), 250
- __delattr__(), 364, 365
- delattr() (built-in), 349
- delegation, 378
- DELETE (SQL statement), 487
- __delitem__() ([]), 265, 266, 273, 279, 329, 334
- deque type (collections module), 218, 383
- description attribute (cursor object), 482
- descriptors, 372–377, 407–409
- detach() (stdin file object), 443
- development environment (IDLE), 13–14, 364, 424–425
- dialogs, modal, 584, 587, 592
- __dict__ (attribute), 348, 363, 364
- dict type, 126–135, 383
 - changing, 128
 - clear(), 129
 - comparing, 126
 - comprehensions, 134–135
 - copy(), 129, 147
 - dict() (built-in), 127, 147
 - fromkeys(), 129
 - get(), 129, 130, 264, 351, 374, 469
 - inverting, 134
 - items(), 128, 129
 - keys(), 128, 129, 277
 - methods, table of, 129
 - pop(), 127, 129, 265
 - popitem(), 129
 - setdefault(), 129, 133, 374
- dict type (*cont.*)
 - update(), 129, 188, 276, 295
 - updating, 128
 - values(), 128, 129
 - view, 129
 - see also* collections.defaultdict, collections.OrderedDict, and SortedDict.py
- dictionary, inverting, 134
- dictionary branching, 340–341
- dictionary comprehensions, 134–135, 278
- dictionary keys, 135
- difference_update() (set type), 123
- difference()
 - frozenset type, 123
 - set type, 122, 123
- difflib module, 213
- digit_names.py (example), 180
- __dir__(), 365
- dir() (built-in), 52, 172, 349, 365
- directories, comparing, 223
- directories, temporary, 222
- directory handling, 222–225
- dirname() (os.path module), 223, 348
- discard() (set type), 123, 124
- __divmod__(), 253
- divmod() (built-in), 55, 253
- __doc__ (attribute), 357
- docstrings, 176–177, 202, 204, 210, 211, 247
 - see also* doctest module
- doctest module, 206–207, 211, 228, 426–428
- documentation, 172
- DOM (Document Object Model); *see* xml.dom module
- Domain-Specific Language (DSL), 513
- DoubleVar type (tkinter module), 574
- DSL (Domain-Specific Language), 513
- DSU (Decorate, Sort, Undecorate), 140, 145

duck typing; *see* dynamic typing
 dump() (pickle module), 267, 294
 dumps() (pickle module), 462
 duplicates, eliminating, 122
 dvds-dbm.py (example), 476–479
 dvds-sql.py (example), 480–487
 dynamic code execution, 260,
 344–346
 dynamic functions, 209
 dynamic imports, 346–351
 dynamic typing, 17, 237, 382

E

e (constant) (math module), 60
 editor (IDLE), 13–14, 364, 424–425
 element trees; *see* xml.etree package
 elif (statement); *see* if statement
 else (statement); *see* for loop, if
 statement, and while loop
 email module, 226
 encode() (str type), 73, 92, 93, 296,
 336, 419, 441
 encoding attribute (file object), 325
 encoding errors, 167
 encodings, 91–94
 encodings, XML, 314
 end() (match object), 507
 END constant (tkinter module), 583,
 587, 588
 endianness, 297
 endpos attribute (match object), 507
 endswith()
 bytearray type, 299
 bytes type, 299
 str type, 73, 75, 76
 __enter__(), 369, 371, 372
 entities, HTML, 504
 Entry type (tkinter module), 591
 enumerate() (built-in), 139–141, 398,
 524
 enums; *see* namedtuple type
 environ mapping (os module), 223

environment variable
 LANG, 87
 PATH, 12, 13
 PYTHONDONTWRITEBYTECODE, 199
 PYTHONOPTIMIZE, 185, 199, 359,
 362
 PYTHONPATH, 197, 205
 EnvironmentError (exception), 167
 EOFError (exception), 100
 epsilon; *see* sys.float_info.epsilon
 attribute
 __eq__() (==), 241, 242, 244, 252, 254,
 259, 379
 error handling; *see* exception handling
 error-handling policy, 208
 escape()
 re module, 502
 xml.sax.saxutils module, 186,
 226, 320
 escapes, HTML and XML, 186, 316
 escapes, string, 66, 67
 escaping, newlines, 67
 eval() (built-in), 242, 243, 258, 266,
 275, 344, 349, 379
 event bindings, 576
 event loop, 572, 578, 590
 example
 Abstract.py, 386
 bigdigits.py, 39–42
 BikeStock.py, 332–336
 BinaryRecordFile.py, 324–332
 blocks.py, 525–534, 543–547,
 559–562
 bookmarks-tk.pyw, 578–593
 car_registration_server.py,
 464–471
 car_registration.py, 458–464
 CharGrid.py, 207–212
 checktags.py, 169
 convert-incident.py, 289–323
 csv2html.py, 97–102
 csv2html2_opt.py, 215
 digit_names.py, 180
 dvds-dbm.py, 476–479

example (*cont.*)

dvds-sql.py, 480–487
 external_sites.py, 132
 ExternalStorage.py, 375
 finddup.py, 224
 findduplicates-t.py, 449–453
 first-order-logic.py, 548–553,
 562–566
 FuzzyBool.py, 249–255
 FuzzyBoolAlt.py, 256–261
 generate_grid.py, 42–44
 generate_test_names1.py, 142
 generate_test_names2.py, 143
 generate_usernames.py, 149–152
 grepword-m.py, 448
 grepword-p.py, 440–442
 grepword.py, 139
 grepword-t.py, 446–448
 html2text.py, 503
 Image.py, 261–269
 IndentedList.py, 352–356
 interest-tk.pyw, 572–578
 magic-numbers.py, 346–351
 make_html_skeleton.py, 185–191
 noblanks.py, 166
 playlists.py, 519–525, 539–543,
 555–559
 print_unicode.py, 88–91
 Property.py, 376
 quadratic.py, 94–96
 Shape.py, 238–245
 ShapeAlt.py, 246–248
 SortedDict.py, 276–283
 SortedList.py, 270–275
 SortKey.py, 368
 statistics.py, 152–156
 TextFilter.py, 385
 TextUtil.py, 202–207
 uniquewords1.py, 130
 uniquewords2.py, 136
 untar.py, 221
 Valid.py, 407–409
 XmlShadow.py, 373
 except (statement); *see* try state-
 ment

exception

AssertionError, 184
 AttributeError, 240, 241, 275,
 350, 364, 366
 custom, 168–171, 208
 EnvironmentError, 167
 EOFError, 100
 Exception, 164, 165, 360, 418
 ImportError, 198, 221, 350
 IndexError, 69, 211, 273
 IOError, 167
 KeyboardInterrupt, 190, 418, 442
 KeyError, 135, 164, 279
 LookupError, 164
 NameError, 116
 NotImplementedError, 258, 381,
 385
 OSError, 167
 StopIteration, 138, 279
 SyntaxError, 54, 348, 414–415
 TypeError, 57, 135, 138, 146, 167,
 173, 179, 197, 242, 258, 259, 274,
 364, 380
 UnicodeDecodeError, 167
 UnicodeEncodeError, 93
 ValueError, 57, 272, 279
 ZeroDivisionError, 165, 416
 Exception (exception), 164, 165, 360
 exception handling, 163–171, 312
 see also try statement
 exceptions, chaining, 419–420
 exceptions, custom, 168–171, 208
 exceptions, propagating, 370
 exec() (built-in), 260, 345–346, 348,
 349, 351
 executable attribute (sys module),
 441
 execute() (cursor object), 481, 482,
 483, 484, 485, 486, 487
 executemany() (cursor object), 482
 exists() (os.path module), 224, 327,
 481
`__exit__()`, 369, 371, 372
 exit() (sys module), 141, 215
 exp() (math module), 60

`expand()` (match object), 507
`expandtabs()`
 bytearray type, 299
 bytes type, 299
 str type, 73
 expat XML parser, 315, 317, 318
 expression, conditional, 160, 176, 189
 expressions, Boolean, 54
`extend()`
 bytearray type, 299, 301, 462
 list type, 115, 116
 extending lists, 114
 extension
 .bz2, 219
 .csv, 220
 .gz, 219, 228
 .ini, 220, 519
 .m3u, 522, 541, 557
 .pls, 519, 539, 555
 .py, 9, 195, 571
 .pyc and .pyo, 199
 .pyw, 9, 571
 .svg, 525
 .tar, .tar.gz, .tar.bz2, 219, 221
 .tgz, 219, 221
 .wav, 219
 .xpm, 268
 .zip, 219
 external_sites.py (example), 132
 ExternalStorage.py (example), 375

F

`fabs()` (math module), 60
`factorial()` (math module), 60
 factory functions, 136
 False (built-in constant); *see* bool type
`fetchall()` (cursor object), 482, 485
`fetchmany()` (cursor object), 482
`fetchone()` (cursor object), 482, 484, 486
`__file__` (attribute), 441

File associations, Windows, 11
 file extension; *see* extension
 file globbing, 343
 file handling, 222–225
 file object, 370
 `close()`, 131, 167, 325
 closed attribute, 325
 encoding attribute, 325
 `fileno()`, 325
 `flush()`, 325, 327
 `isatty()`, 325
 methods, table of, 325, 326
 mode attribute, 325
 name attribute, 325
 newlines attribute, 325
 `__next__()`, 325
 `open()`, 131, 141, 167, 174, 267, 268, 327, 347, 369, 398, 443
 `peek()`, 325
 `read()`, 131, 295, 302, 325, 347, 443
 `readable()`, 325
 `readinto()`, 325
 `readline()`, 325
 `readlines()`, 131, 325
 `seek()`, 295, 325, 327, 329
 `seekable()`, 326
 `stderr` (sys module), 184, 214
 `stdin` (sys module), 214
 `stdin.detach()`, 443
 `stdout` (sys module), 181, 214
 `tell()`, 326, 329
 `truncate()`, 326, 331
 `writable()`, 326
 `write()`, 131, 214, 301, 326, 327
 `writelines()`, 326
 file suffix; *see* extension
 file system interaction, 222–225
 File Transfer Protocol (FTP), 226
 filecmp module, 223
 fileinput module, 214
`fileno()` (file object), 325
 files; *see* file object and `open()`
 files, archive, 219
 files, binary, 295–304, 324–336

- files, comparing, 223
- files, compressing and uncompressing, 219
- files, format comparison, 288–289
- files, random access; *see* binary files
- files, temporary, 222
- files, text, 305–312
- files, XML, 312–323
- filter() (built-in), 395, 397
- filtering, 395, 403–407
- finally (statement); *see* try statement
- find()
 - bytearray type, 299
 - bytes type, 299
 - str type, 72–75, 133, 532
- findall()
 - re module, 502
 - regex object, 503
- finddup.py (example), 224
- findduplicates-t.py (example), 449–453
- finditer()
 - re module, 311, 502
 - regex object, 401, 500, 501, 503
- first-order-logic.py (example), 548–553, 562–566
- flags attribute (regex object), 503
- `__float__()`, 252, 253
- float_info.epsilon attribute (sys module), 61, 96, 343
- float() (built-in), 253
- float type, 59–62, 381
 - as_integer_ratio(), 61
 - float() (built-in), 61, 154, 253
 - fromhex(), 61
 - hex(), 61
 - is_integer(), 61
- floor() (math module), 60
- `__floordiv__()` (//), 55, 253
- flush() (file object), 325, 327
- fmod() (math module), 60
- focus, keyboard, 574, 576, 577, 589, 592
- for loop, 120, 138, 141, 143, 162–163
- foreign functions, 229
- `__format__()`, 250, 254
- format()
 - built-in, 250, 254
 - str type, 73, 78–88, 152, 156, 186, 189, 249, 306, 531
- format specifications, for strings, 83–88
- formatting strings; *see* str.format()
- Fraction type (fractions module), 381
- Frame type (tkinter module), 573, 581, 591
- frexp() (math module), 60
- from (statement); *see* chaining exceptions and import statement
- fromhex()
 - bytearray type, 293, 299
 - bytes type, 293, 299
 - float type, 61
- fromkeys() (dict type), 129
- fromordinal() (datetime.date type), 301, 304
- frozenset type, 125–126, 383
 - copy(), 123
 - difference(), 123
 - frozenset() (built-in), 125
 - intersection(), 123
 - isdisjoint(), 123
 - issubset(), 123
 - issuperset(), 123
 - methods, table of, 123
 - symmetric_difference(), 123
- fsum() (math module), 60
- FTP (File Transfer Protocol), 226
- ftplib module, 226
- functions, 171–185
 - annotations, 360–363
 - anonymous; *see* lambda statement
 - composing, 395–397, 403–407
 - decorating, 246–248, 356–360
 - dynamic, 209
 - factory, 136
 - foreign, 229

functions (*cont.*)
 lambda; *see* lambda statement
 local, 296, 319, 351–356
 module, 256
 object reference to, 136, 270, 341
 parameters; *see* arguments, function
 recursive, 351–356
see also functors
 functions, introspection-related, table of, 349
 functions, iterator, table of, 140
 functions, nested; *see* local functions
 functions, table of (math module), 60, 61
 functions, table of (re module), 502
 functools module
 partial(), 398
 reduce(), 396, 397
 @wraps(), 357
 functors, 367–369, 385
 FuzzyBool.py (example), 249–255
 FuzzyBoolAlt.py (example), 256–261

G

garbage collection, 17, 116, 218, 576, 581, 593
 __ge__() (>=), 242, 259, 379
 generate_grid.py (example), 42–44
 generate_test_names1.py (example), 142
 generate_test_names2.py (example), 143
 generate_usernames.py (example), 149–152
 generator object
 send(), 343
 generators, 279, 342–344, 395, 396, 401
 __get__(), 374, 375, 376, 377
 get() (dict type), 129, 130, 264, 351, 374, 469

__getattr__(), 365, 366
 getattr() (built-in), 349, 350, 364, 368, 374, 391, 409
 __getattribute__(), 365, 366
 getcwd() (os module), 223
 __getitem__() ([]), 264, 265, 273, 328, 334
 getmtime() (os.path module), 224
 getopt module; *see* optparse module
 getrecursionlimit() (sys module), 352
 getsize() (os.path module), 134, 224, 407
 gettempdir() (tempfile module), 360
 GIL (Global Interpreter Lock), 449
 glob module, 344
 global (statement), 210
 global functions; *see* functions
 Global Interpreter Lock (GIL), 449
 global variables, 180
 globals() (built-in), 345, 349
 globbing, 343
 GMT; *see* Coordinated Universal Time
 grammar, 515
 greedy regexes, 493
 grepword-m.py (example), 448
 grepword-p.py (example), 440–442
 grepword.py (example), 139
 grepword-t.py (example), 446–448
 grid layout, 573, 575, 591
 group() (match object), 311, 500, 501, 504, 507, 508, 521, 524
 groupdict() (match object), 402, 507
 groupindex attribute (regex object), 503
 groups() (match object), 507
 groups, regex, 494–495, 506
 __gt__() (>), 242, 259, 379
 .gz (extension), 219, 228
 gzip module, 219
 open(), 228, 294
 write(), 301

H

hasattr() (built-in), 270, 349, 350, 391
 __hash__(), 250, 254
 hash() (built-in), 241, 250, 254
 Hashable ABC (collections module), 383
 hashable objects, 121, 126, 130, 135, 241, 254
 heapq module, 217, 218–219
 help() (built-in), 61, 172
 hex()
 built-in, 55, 253
 float type, 61
 hexadecimal numbers, 56
 html.entities module, 504, 505
 HTML escapes, 186
 html.parser module, 226
 html2text.py (example), 503
 http package, 225
 hypot() (math module), 60

I

__iadd__() (+=), 253
 __iand__() (&=), 251, 253, 257
 id() (built-in), 254
 identifiers, 51–54, 127
 identity testing; *see* is identity operator
 IDLE (programming environment), 13–14, 364, 424–425
 if (statement), 159–161
 __ifloordiv__() (//=), 253
 __ilshift__() (<<=), 253
 Image.py (example), 261–269
 IMAP4 (Internet Message Access Protocol), 226
 imaplib module, 226
 immutable arguments, 175
 immutable attributes, 264
 immutable classes, 256, 261
 immutable objects, 15, 16, 108, 113, 126

__imod__() (%=), 253
 import (statement), 196–202, 348
 __import__() (built-in), 349, 350
 import order policy, 196
 ImportError (exception), 198, 221, 350
 imports, dynamic, 346–351
 imports, relative, 202
 __imul__() (*=), 253
 in (membership operator), 114, 118, 122, 140, 265, 274
 indentation, for block structure, 27
 IndentedList.py (example), 352–356
 __index__(), 253
 index()
 bytearray type, 299
 bytes type, 299
 list type, 115, 118
 str type, 72–75
 tuple type, 108
 IndexError (exception), 69, 211, 273
 indexing operator ([]), 273, 274
 infinite loop, 399, 406
 inheritance, 243–245
 inheritance, multiple, 388–390, 466
 .ini (extension), 220, 519
 __init__(), 241, 244, 249, 250, 270, 276
 type type, 391, 392
 __init__.py package file, 199, 200
 initialization, of objects, 240
 input() (built-in), 34, 96
 INSERT (SQL statement), 483
 insert()
 bytearray type, 293, 299
 list type, 115, 117, 271
 inspect module, 362
 installing Python, 4–6
 instance variables, 241
 __int__(), 252, 253, 258
 int() (built-in), 253
 int type, 54–57, 381
 bit_length(), 57
 bitwise operators, table of, 57
 conversions, table of, 55

- int type (*cont.*)
 - int() (built-in), 55, 61, 136, 253, 309
- Integral ABC (numbers module), 381
- interest-tk.pyw (example), 572–578
- internationalization, 86
- Internet Message Access Protocol (IMAP4), 226
- interpreter options, 185, 198, 199
- intersection_update() (set type), 123
- intersection()
 - frozenset type, 123
 - set type, 122, 123
- introspection, 350, 357, 360, 362
- IntVar type (tkinter module), 574
- __invert__() (~), 57, 250, 253, 257
- inverting, a dictionary, 134
- io module
 - StringIO type, 213–214, 228
 - see also* file object and open()
- IOError (exception), 167
- __ior__() (|=), 253
- IP address, 457, 458, 464
- __ipow__() (**=), 253
- __irshift__() (>>=), 253
- is_integer() (float type), 61
- is (identity operator), 22, 254
- isalnum()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73
- isalpha()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73
- isatty() (file object), 325
- isdecimal() (str type), 73
- isdigit()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73, 76
- isdir() (os.path module), 224
- isdisjoint()
 - frozenset type, 123
- isdisjoint() (*cont.*)
 - set type, 123
- isfile() (os.path module), 134, 224, 344, 406
- isidentifier() (str type), 73, 348
- isinf() (math module), 60
- isinstance() (built-in), 170, 216, 242, 270, 382, 390, 391
- islower()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73
- isnan() (math module), 60
- isnumeric() (str type), 74
- isprintable() (str type), 74
- isspace()
 - bytearray type, 299
 - bytes type, 299
 - str type, 74, 531
- issubclass() (built-in), 390
- issubset()
 - frozenset type, 123
 - set type, 123
- issuperset()
 - frozenset type, 123
 - set type, 123
- istitle()
 - bytearray type, 300
 - bytes type, 300
 - str type, 74
- __isub__() (-=), 253
- isupper()
 - bytearray type, 300
 - bytes type, 300
 - str type, 74
- item access operator ([]), 262, 264, 273, 274, 278, 279, 293
- itemgetter() (operator module), 397
- items() (dict type), 128, 129
- __iter__(), 265, 274, 281, 335
- iter() (built-in), 138, 274, 281
- iterable; *see* iterators
- Iterable ABC (collections module), 383

Iterator ABC (collections module), 383
iterators, 138–146
 functions and operators, table of, 140
itertools module, 397
__ixor__() (^=), 253

J

join()
 bytearray type, 300
 bytes type, 300
 os.path module, 223, 224
 str type, 71, 72, 189
json module, 226

K

key bindings, 576
keyboard accelerators, 574, 580, 592
keyboard focus, 574, 576, 577, 589, 592
keyboard shortcuts, 577, 580
KeyboardInterrupt (exception), 190, 418, 442
KeyError (exception), 135, 164, 279
keys() (dict type), 128, 129, 277
keyword arguments, 174–175, 178, 179, 188, 189, 362
keywords, table of, 52

L

Label type (tkinter module), 574, 582, 583, 591
lambda (statement), 182–183, 379, 380, 388, 396, 467, 504
LANG (environment variable), 87
lastgroup attribute (match object), 507
lastindex attribute (match object), 507, 508

Latin 1 encoding, 91, 93
layouts, 573, 575, 591
lazy evaluation, 342
ldexp() (math module), 60
__le__() (<=), 242, 259, 379
__len__(), 265, 330
len() (built-in), 71, 114, 122, 140, 265, 275
lexical analysis, 514
library, standard, 212–229
LifoQueue type (queue module), 446
linear search, 272
list comprehensions, 118–120, 189, 210, 396
list type, 113–120, 383
 append(), 115, 117, 118, 271
 changing, 115
 comparing, 113, 114
 comprehensions, 118–120, 396
 count(), 115
 extend(), 115, 116
 index(), 115, 118
 insert(), 115, 117, 271
 list() (built-in), 113, 147
 methods, table of, 115
 pop(), 115, 117, 118
 remove(), 115, 117, 118
 replication (*, *), 114, 118
 reverse(), 115, 118
 slicing, 113, 114, 116–118
 sort(), 115, 118, 182, 368, 397
 updating, 115
 see also SortedList.py
Listbox type (tkinter module), 582, 583, 587, 588, 589
listdir() (os module), 134, 223, 224, 348
ljust()
 bytearray type, 300
 bytes type, 300
 str type, 74
load() (pickle module), 268, 295
loads() (pickle module), 462
local functions, 296, 319, 351–356
local variables, 163

locale module, 86
 setlocale(), 86, 87
 localization, 86
 locals() (built-in), 81, 82, 97, 154,
 188, 189, 190, 345, 349, 422, 423,
 484
 localtime() (time module), 217
 Lock type (threading module), 452,
 467
 log() (math module), 60
 log10() (math module), 60
 log1p() (math module), 60
 logging module, 229, 360
 logic, short-circuit, 25, 58
 logical operators; *see* and, or, and
 not
 LookupError (exception), 164
 looping, *see* for loop and while loop,
 161
 lower()
 bytearray type, 300
 bytes type, 300
 str type, 74, 76
 __lshift__ () (<<), 57, 253
 lstrip()
 bytearray type, 300
 bytes type, 300
 str type, 75, 76
 __lt__ () (<), 242, 252, 259, 379

M

.m3u (extension), 522, 541, 557
 magic number, 294
 magic-numbers.py (example),
 346–351
 mailbox module, 226
 make_html_skeleton.py (example),
 185–191
 makedirs() (os module), 223
 maketrans() (str type), 74, 77–78
 mandatory parameters, 174
 map() (built-in), 395, 397, 539
 mapping, 395

Mapping ABC (collections module),
 383
 mapping types; *see* dict and collec-
 tions.defaultdict
 mapping unpacking (**), 179, 187,
 304
 match()
 re module, 502, 521, 524
 regex object, 503
 match object
 end(), 507
 endpos attribute, 507
 expand(), 507
 group(), 311, 500, 501, 504, 507,
 508, 521, 524
 groupdict(), 402, 507
 groups(), 507
 lastgroup attribute, 507
 lastindex attribute, 507, 508
 methods, table of, 507
 pos attribute, 507
 re attribute, 507
 span(), 507
 start(), 507
 string attribute, 507
 see also re module and regex ob-
 ject
 math module, 62
 acos(), 60
 acosh(), 60
 asin(), 60
 asinh(), 60
 atan(), 60
 atan2(), 60
 atanh(), 60
 ceil(), 60
 copysign(), 60
 cos(), 60
 cosh(), 60
 degrees(), 60
 e (constant), 60
 exp(), 60
 fabs(), 60
 factorial(), 60
 floor(), 60

math module (cont.)

- fmod(), 60
- frexp(), 60
- fsum(), 60
- functions, table of, 60, 61
- hypot(), 60
- isinf(), 60
- isnan(), 60
- ldexp(), 60
- log(), 60
- log10(), 60
- log1p(), 60
- modf(), 60
- pi (constant), 61
- pow(), 61
- radians(), 61
- sin(), 61
- sinh(), 61
- sqrt(), 61, 96
- tan(), 61
- tanh(), 61
- trunc(), 61

max() (built-in), 140, 154, 396, 397

maxunicode attribute (sys module), 90, 92

MD5 (Message Digest algorithm), 449, 452

membership testing; *see* in operator

memoizing, 351

memory management; *see* garbage collection

Menu type (tkinter module), 579, 580

Message Digest algorithm (MD5), 449, 452

metaclasses, 381, 384, 390–395

methods

- attribute access, table of, 365
- bytearray type, table of, 299, 300, 301
- bytes type, table of, 299, 300, 301
- class, 257

methods (cont.)

- connection object, table of, 481
- cursor object, table of, 482
- decorating, 246–248, 356–360
- dict type, table of, 129
- file object, table of, 325, 326
- frozenset type, table of, 123
- list type, table of, 115
- match object, table of, 507
- object reference to, 377
- regex object, table of, 503
- set type, table of, 123
- static, 257
- str type, table of, 73, 74, 75
- unimplementing, 258–261
- see also* special method

mimetypes module, 224

min() (built-in), 140, 396, 397

minimal regexes, 493, 504

missing dictionary keys, 135

mixin class, 466

mkdir() (os module), 223

__mod__() (%), 55, 253

modal dialogs, 584, 587, 592

mode attribute (file object), 325

modf() (math module), 60

__module__ (attribute), 243

module functions, 256

modules, 195–202, 348

modules attribute (sys module), 348

__mul__() (*), 55, 253

multiple inheritance, 388–390, 466

multiprocessing module, 448, 453

mutable arguments, 175

mutable attributes, policy, 264

mutable objects; *see* immutable objects

MutableMapping ABC (collections module), 269, 383

MutableSequence ABC (collections module), 269, 383

MutableSet ABC (collections module), 383

N

`__name__` (attribute), 206, 252, 357, 362, 377
`name()` (unicodedata module), 90
name attribute (file object), 325
name conflicts, avoiding, 198, 200
name mangling, 366, 379
namedtuple type (collections module), 111–113, 234, 365, 523
NameError (exception), 116
names, qualified, 196
namespace, 236
naming policy, 176–177
`__ne__()` (!=), 241, 242, 259, 379
`__neg__()` (-), 55, 253
nested collections; *see* dict, list, set, and tuple types
nested functions; *see* local functions
Network News Transfer Protocol (NNTP), 226
`__new__()`, 250
 object type, 256
 type type, 392, 394
newline escaping, 67
newlines attribute (file object), 325
`__next__()`, 325, 343
`next()` (built-in), 138, 343, 401
NNTP (Network News Transfer Protocol), 226
nntplib module, 226
noblanks.py (example), 166
None object, 22, 23, 26, 173
nongreedy regexes, 493, 504
nonlocal (statement), 355, 379
nonterminal, 515
normal (debug) mode; *see* PYTHONOPTIMIZE
normalize() (unicodedata module), 68
not (logical operator), 58
NotImplemented object, 242, 258, 259
NotImplementedError (exception), 258, 381, 385
now() (datetime.datetime type), 217

Number ABC (numbers module), 381
numbers module, 216, 382
 classes, table of, 381
 Complex ABC, 381
 Integral ABC, 381
 Number ABC, 381
 Rational ABC, 381
 Real ABC, 381
numeric operators and functions, table of, 55

O

-0 option, interpreter, 185, 199, 359, 362
object creation and initialization, 240
object-oriented concepts and terminology, 235
object references, 16–18, 19, 110, 116, 126, 136, 142, 146, 250, 254, 281, 340, 345, 356, 367, 377, 576
object type, 380
 `__new__()`, 256
 `__repr__()`, 266
objects, comparing, 23, 242
obtaining Python, 4–6
oct() (built-in), 55, 253
octal numbers, 56
open()
 file object, 131, 141, 167, 174, 267, 268, 327, 347, 369, 398, 443
 gzip module, 228, 294
 shelve module, 476
operator module, 396
 attrgetter(), 369, 397
 itemgetter(), 397
operators, iterator, table of, 140
optimized mode; *see* PYTHONOPTIMIZE
optional parameters, 174
options, for interpreter, 185, 198, 199, 359, 362
optparse module, 215
`__or__()` (|), 57, 253

or (logical operator), 58
ord() (built-in), 67, 90, 364
ordered collections; *see* list and tuple types
OrderedDict type (collections module), 136–138, 218
os module, 223, 224–225
 chdir(), 223
 environ mapping, 223
 getcwd(), 223
 listdir(), 134, 223, 224, 348
 makedirs(), 223
 mkdir(), 223
 remove(), 223, 332
 removedirs(), 223
 rename(), 223, 332
 rmdir(), 223
 sep attribute, 142
 stat(), 223, 407
 system(), 444
 walk(), 223, 224, 406
os.path module, 197, 223, 224–225
 abspath(), 223, 406
 basename(), 223
 dirname(), 223, 348
 exists(), 224, 327, 481
 getmtime(), 224
 getsize(), 134, 224, 407
 isdir(), 224
 isfile(), 134, 224, 344, 406
 join(), 223, 224
 split(), 223
 splitext(), 223, 268, 348
OSError (exception), 167

P

pack() (struct module), 296, 297, 301, 336
package directories, 205
packages, 195–202
packrat parsing, 549
parameters; *see* arguments, function

parameters, unpacking, 177–180
parent–child relationships, 572, 576
parsing
 command-line arguments, 215
 dates and times, 216
 text files, 307–310
 with PLY, 553–566
 with PyParsing, 534–553
 with regexes, 310–312, 519–525
 XML (with DOM), 317–319
 XML (with SAX), 321–323
 XML (with xml.etree), 315–316
partial() (functools module), 398
partial function application, 398–399
partition()
 bytearray type, 300
 bytes type, 300
 str type, 74, 76
pass (statement), 26, 160, 381, 385
PATH (environment variable), 12, 13
path attribute (sys module), 197
paths, Unix-style, 142
pattern attribute (regex object), 503
pdb module, 423–424
peek() (file object), 325
PEP 249 (Python Database API Specification v2.0), 480
PEP 3107 (Function Annotations), 363
PEP 3119 (Introducing Abstract Base Classes), 380
PEP 3131 (Supporting Non-ASCII Identifiers), 52
PEP 3134 (Exception Chaining and Embedded Tracebacks), 420
persistence, of data, 220
PhotoImage type (tkinter module), 581
pi (constant) (math module), 61
pickle module, 292–295
 dump(), 267, 294
 dumps(), 462
 load(), 268, 295

- `pickle` module (*cont.*)
 - `loads()`, 462
- `pickles`, 266, 292–295, 476
- `pipelines`, 403–407
- `pipes`; *see* `subprocess` module
- `placeholders`, SQL, 483, 484
- `platform` attribute (`sys` module), 160, 209, 344
- `playlists.py` (example), 519–525, 539–543, 555–559
- `.pls` (extension), 519, 539, 555
- PLY
 - `p_error()`, 555
 - precedence variable, 555, 565
 - states variable, 557–558
 - `t_error()`, 554, 556
 - `t_ignore` variable, 559
 - `t_newline()`, 556
 - tokens variable, 554, 555, 557
- `pointers`; *see* `object` references
- `policy`, error handling, 208
- `policy`, import order, 196
- `policy`, mutable attributes, 264
- `policy`, naming, 176–177
- `polymorphism`, 243–245
- `pop()`
 - bytearray type, 293, 300
 - dict type, 127, 129, 265
 - list type, 115, 117, 118
 - set type, 123
- POP3 (Post Office Protocol), 226
- `Popen()` (`subprocess` module), 441
- `popitem()` (dict type), 129
- `poplib` module, 226
- `__pos__()` (+), 55, 253
- `pos` attribute (`match` object), 507
- positional arguments, 173–175, 178, 179, 189, 362
- Post Office Protocol (POP3), 226
- `__pow__()` (**), 55, 253
- `pow()`
 - built-in, 55
 - `math` module, 61
- `pprint` module, 229, 355
- precedence, 517–518, 551, 565
- `print_unicode.py` (example), 88–91
- `print()` (built-in), 11, 180, 181, 214, 422
- `PriorityQueue` type (`queue` module), 446, 450
- private attributes, 238, 249, 270, 271, 366
- processing pipelines, 403–407
- processor endianness, 297
- `profile` module, 432, 434–437
- propagating exceptions, 370
- properties, 246–248
 - `@property()`, 246–248, 376, 385, 394
- `Property.py` (example), 376
- `.py` (extension), 9, 195, 571
- `.pyc` and `.pyo` (extension), 199
- PyGtk, 570, 593
- PyParsing
 - + (concatenation operator), 536, 539, 541, 543, 544, 545, 550
 - (concatenation operator), 544, 545
 - << (append operator), 538, 544, 550
 - | (OR operator), 536, 539, 541, 543, 544, 550
 - alphanums, 535
 - alphas, 535
 - `CaselessLiteral()`, 535
 - `CharsNotIn()`, 536, 539, 543
 - `Combine()`, 541
 - `delimitedList()`, 536, 538, 550
 - `Empty()`, 537
 - `Forward()`, 538, 544, 550
 - `Group()`, 544, 550, 551
 - `Keyword()`, 535, 550
 - `LineEnd()`, 541, 542
 - `Literal()`, 535, 540, 550
 - `makeHTMLTags()`, 536
 - nums, 541
 - `OneOrMore()`, 536, 539, 541, 544
 - `operatorPrecedence()`, 550–551
 - `Optional()`, 536, 537, 541, 544
 - `pythonStyleComment`, 536
 - `quotedString`, 536

PyParsing (*cont.*)

Regex(), 536
 restOfLine, 536, 539, 541
 SkipTo(), 536
 Suppress(), 535, 536, 539, 541
 Word(), 535, 539, 541, 543
 ZeroOrMore(), 536, 538, 544

PyQt, 570, 593

PYTHONDONTWRITEBYTECODE (environment variable), 199

Python enhancement proposals; *see* PEPs

Python Shell (IDLE or interpreter), 13

PYTHONOPTIMIZE (environment variable), 185, 199, 359, 362

PYTHONPATH (environment variable), 197, 205

.pyw (extension), 9, 571

Q

quadratic.py (example), 94–96

qualified names, 196

quantifiers, regex, 491–494

queue module

LifoQueue type, 446
 PriorityQueue type, 446, 450
 Queue type, 446, 447, 450

Queue type (queue module), 446, 447, 450

quopri module, 219

quoteattr() (xml.sax.saxutils module), 226, 320

R

__radd__() (+), 253

radians() (math module), 61

raise (statement), 167, 211, 350, 360

see also try statement

__rand__() (&), 253

random access files; *see* binary files

random module

choice(), 142

sample(), 143

range() (built-in), 115, 118, 119, 140, 141–142, 365

Rational ABC (numbers module), 381

raw binary data; *see* binary files

raw strings, 67, 204, 310, 500, 556

__rdivmod__(), 253

re attribute (match object), 507

re module, 499–509

compile(), 310, 400, 500, 501, 502, 521, 524

escape(), 502

findall(), 502

finditer(), 311, 502

functions, table of, 502

match(), 502, 521, 524

search(), 500, 502, 508

split(), 502, 509

sub(), 502, 504, 505

subn(), 502

see also match object and regex object

read() (file object), 131, 295, 302, 325, 347, 443

readable() (file object), 325

readinto() (file object), 325

readline() (file object), 325

readlines() (file object), 131, 325

Real ABC (numbers module), 381

records; *see* struct module

recursive descent parser, 529

recursive functions, 351–356

recv() (socket module), 462, 463

reduce() (functools module), 396, 397

reducing, 395

references; *see* object references

regex

alternation, 494–495

assertions, 496–499

backreferences, 495

captures, 494–495, 506

character classes, 491

- regex (*cont.*)
 - flags, 400, 499, 500
 - greedy, 493, 504
 - groups, 494–495, 506
 - match; *see* match object
 - nongreedy, 493, 504
 - quantifiers, 491–494
 - special characters, 491
- regex object
 - findall(), 503
 - finditer(), 401, 500, 501, 503
 - flags attribute, 503
 - groupindex attribute, 503
 - match(), 503
 - methods, table of, 503
 - pattern attribute, 503
 - search(), 500, 503
 - split(), 503, 509
 - sub(), 503
 - subn(), 503
 - see also* re module and match object
- relational integrity, 481
- relative imports, 202
- remove()
 - bytearray type, 300
 - list type, 115, 117, 118
 - os module, 223, 332
 - set type, 123
- removedirs() (os module), 223
- rename() (os module), 223, 332
- replace()
 - bytearray type, 293, 300
 - bytes type, 293, 300
 - str type, 74, 77, 101
- replication (*, *)
 - of lists, 114, 118
 - of strings, 72, 90
 - of tuples, 108
- __repr__(), 242, 244, 250, 252, 258, 281
 - object type, 266
- repr() (built-in), 242, 250
- representational form, 82–83
- resizable windows, 582–583, 591
- return (statement), 161, 162, 173
- reverse()
 - bytearray type, 300
 - list type, 115, 118
 - __reversed__(), 265, 274
- reversed() (built-in), 72, 140, 144, 265
- reversing strings, 71, 72
- rfind()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73, 75, 76
- __rfloordiv__() (//), 253
- rindex()
 - bytearray type, 299
 - bytes type, 299
 - str type, 73, 75
- rjust()
 - bytearray type, 300
 - bytes type, 300
 - str type, 74
- __rlshift__() (<<), 253
- rmdir() (os module), 223
- __rmod__() (%), 253
- __rmul__() (*), 253
- rollback() (connection object), 481
- __ror__() (|), 253
- __round__(), 253
- round() (built-in), 55, 56, 61, 252, 253, 258
- rowcount attribute (cursor object), 482
- rpartition()
 - bytearray type, 300
 - bytes type, 300
 - str type, 74, 76
- __rpow__() (**), 253
- __rrshift__() (>>), 253
- __rshift__() (>>), 57, 253
- rsplit()
 - bytearray type, 300
 - bytes type, 300
 - str type, 74
- rstrip()
 - bytearray type, 300

- `rstrip()` (*cont.*)
 - bytes type, 300
 - str type, 75, 76
- `__rsub__()` (-), 253
- `__rtruediv__()` (/), 253
- `run()` (Thread type), 445, 448
- `__rxor__()` (^), 253

- S**
- `sample()` (random module), 143
- SAX (Simple API for XML); *see*
 - `xml.sax` module
- Scalable Vector Graphics (SVG), 525
- Scale type (tkinter module), 574, 575
- scanning, 514
- Scrollbar type (tkinter module), 582
- `search()`
 - `re` module, 500, 502, 508
 - regex object, 500, 503
- searching, 272
- `seek()` (file object), 295, 325, 327, 329
- `seekable()` (file object), 326
- SELECT (SQL statement), 484, 485, 486
- self object, 239, 257, 469
- `send()`
 - coroutines, 401, 402, 405, 406
 - generator object, 343
 - socket module, 463
- `sendall()` (socket module), 462, 463
- sep attribute (os module), 142
- Sequence ABC (collections module), 383
- sequence types; *see* bytearray, bytes, list, str, and tuple types
- sequence unpacking (*), 110, 114–115, 141, 162, 178, 336, 460
- serialized data access, for threads, 446
- serializing; *see* pickles
- `__set__()`, 375, 377
- Set ABC (collections module), 383
- set comprehensions, 125
- set type, 121–125, 130, 383
 - `add()`, 123
 - `clear()`, 123
 - comprehensions, 125
 - `copy()`, 123, 147
 - `difference_update()`, 123
 - `difference()`, 122, 123
 - `discard()`, 123, 124
 - `intersection_update()`, 123
 - `intersection()`, 122, 123
 - `isdisjoint()`, 123
 - `issubset()`, 123
 - `issuperset()`, 123
 - methods, table of, 123
 - `pop()`, 123
 - `remove()`, 123
 - `set()` (built-in), 122, 147
 - `symmetric_difference_update()`, 123
 - `symmetric_difference()`, 122, 123
 - `union()`, 122, 123
 - `update()`, 123
- set types; *see* frozenset and set types
- `__setattr__()`, 364, 365
- `setattr()` (built-in), 349, 379, 409
- `setdefault()` (dict type), 129, 133, 374
- `__setitem__()` ([]), 265, 274, 278, 327
- `setlocale()` (locale module), 86, 87
- `setrecursionlimit()` (sys module), 352
- shallow copying; *see* copying collections
- Shape.py (example), 238–245
- ShapeAlt.py (example), 246–248
- shebang* (shell execute), 12
- Shell, Python (IDLE or interpreter), 13
- shell execute (#!), 12

- shelve module, 220, 476
 - open(), 476
 - sync(), 477
- short-circuit logic, 25, 58
- shortcut, keyboard, 577, 580
- showwarning() (tkinter.messagebox module), 585, 587
- shutil module, 222
- Simple API for XML (SAX); *see* xml.sax module
- Simple Mail Transfer Protocol (SMTP), 226
- sin() (math module), 61
- single shot timer, 582, 586
- sinh() (math module), 61
- site-packages directory, 205
- Sized ABC (collections module), 383
- slicing ([])
 - bytes, 293
 - lists, 113, 114, 116–118
 - operator, 69, 110, 116, 273, 274, 397
 - strings, 69–71, 151
 - tuples, 108
- __slots__ (attribute), 363, 373, 375, 394
- SMTP (Simple Mail Transfer Protocol), 226
- smtpd module, 226
- smtplib module, 226
- sndhdr module, 219
- socket module, 225, 457
 - recv(), 462, 463
 - send(), 463
 - sendall(), 462, 463
 - socket(), 464
- socketserver module, 225, 464, 466
- sort() (list type), 115, 118, 182, 368, 397
- sort algorithm, 145, 282
- sorted() (built-in), 118, 133, 140, 144–146, 270
- SortedDict.py (example), 276–283
- SortedList.py (example), 270–275
- SortKey.py (example), 368
- sound-related modules, 219
- span() (match object), 507
- special characters, regex, 491
- special method, 235, 239
 - __abs__(), 253
 - __add__() (+), 55, 253
 - __and__() (&), 57, 251, 253, 257
- bitwise and numeric methods, table of, 253
- __bool__(), 250, 252, 258
- __call__(), 367, 368
- collection methods, table of, 265
- comparison methods, table of, 242
 - __complex__(), 253
 - __contains__(), 265, 274
 - __copy__(), 275
 - __del__(), 250
 - __delattr__(), 364, 365
 - __delitem__() ([]), 265, 266, 273, 279, 329, 334
 - __dir__(), 365
 - __divmod__(), 253
 - __enter__(), 369, 371, 372
 - __eq__() (==), 241, 242, 244, 252, 254, 259, 379
 - __exit__(), 369, 371, 372
 - __float__(), 252, 253
 - __floordiv__() (/), 55, 253
 - __format__(), 250, 254
- fundamental methods, table of, 250
 - __ge__() (>=), 242, 259, 379
 - __get__(), 374, 375, 376, 377
 - __getattr__(), 365, 366
 - __getattribute__(), 365, 366
 - __getitem__() ([]), 264, 265, 273, 328, 334
 - __gt__() (>), 242, 259, 379
 - __hash__(), 250, 254
 - __iadd__() (+=), 253
 - __iand__() (&=), 251, 253, 257
 - __ifloordiv__() (/!=), 253
 - __ilshift__() (<<=), 253

special method (*cont.*)

__imod__ () (%=), 253
 __imul__ () (*=), 253
 __index__ (), 253
 __init__ (), 241, 244, 249, 250, 270, 276, 391, 392
 __int__ (), 252, 253, 258
 __invert__ () (~), 57, 250, 253, 257
 __ior__ () (|=), 253
 __ipow__ () (**=), 253
 __irshift__ () (>>=), 253
 __isub__ () (-=), 253
 __iter__ (), 265, 274, 281, 335
 __ixor__ () (^=), 253
 __le__ () (<=), 242, 259, 379
 __len__ (), 265, 330
 __lshift__ () (<<), 57, 253
 __lt__ () (<), 242, 252, 259, 379
 __mod__ () (%), 55, 253
 __mul__ () (*), 55, 253
 __ne__ () (!=), 241, 242, 259, 379
 __neg__ () (-), 55, 253
 __new__ (), 250, 256, 392
 __next__ (), 325, 343
 __or__ () (|), 57, 253
 __pos__ () (+), 55, 253
 __pow__ () (**), 55, 253
 __radd__ () (+), 253
 __rand__ () (&), 253
 __rdivmod__ (), 253
 __repr__ (), 242, 244, 250, 252, 258, 281
 __reversed__ (), 265, 274
 __rfloordiv__ () (//), 253
 __rlshift__ () (<<), 253
 __rmod__ () (%), 253
 __rmul__ () (*), 253
 __ror__ () (|), 253
 __round__ (), 253
 __rpow__ () (**), 253
 __rrshift__ () (>>), 253
 __rshift__ () (>>), 57, 253
 __rsub__ () (-), 253
 __rtruediv__ () (/), 253
 __rxor__ () (^), 253

special method (*cont.*)

__set__ (), 375, 377
 __setattr__ (), 364, 365
 __setitem__ () ([]), 265, 274, 278, 327
 __str__ (), 243, 244, 250, 252
 __sub__ () (-), 55, 253
 __truediv__ () (/), 31, 55, 253
 __xor__ () (^), 57, 253

split()
 bytearray type, 300
 bytes type, 300
 os.path module, 223
 re module, 502, 509
 regex object, 503, 509
 str type, 74, 77, 509

splitext() (os.path module), 223, 268, 348

splitlines()
 bytearray type, 300
 bytes type, 300
 str type, 74

SQL databases, 475, 480
 SQL placeholders, 483, 484
 SQL statement
 CREATE TABLE, 481
 DELETE, 487
 INSERT, 483
 SELECT, 484, 485, 486
 UPDATE, 484

sqlite3 module, 480, 481
 connect(), 481

sqrt() (math module), 61, 96

ssl module, 225

standard library, 212–229

starred arguments, 114, 460

starred expressions; *see* sequence unpacking

start()
 match object, 507
 Thread type, 445

start symbol, 516

startswith()
 bytearray type, 300
 bytes type, 300

- startswith() (*cont.*)
 - str type, 74, 75, 76
- stat() (os module), 223, 407
- statement
 - assert, 184–185, 205, 208, 247
 - break, 161, 162
 - class, 238, 244, 378, 407
 - continue, 161, 162
 - def, 37, 173–176, 209, 238
 - del, 116, 117, 127, 250, 265, 273, 365
 - global, 210
 - if, 159–161
 - import, 196–202, 348
 - lambda, 182–183, 379, 380, 388, 396, 467, 504
 - nonlocal, 355, 379
 - pass, 26, 160, 381, 385
 - raise, 167, 211, 350, 360
 - return, 161, 162, 173
 - try, 163–171, 360
 - with, 369–372, 389
 - yield, 279, 281, 342–344, 399–407

see also for loop and while loop
- statement terminator (\n), 66
- static methods, 257
- static variables, 255
- @staticmethod(), 255
- statistics.py (example), 152–156
- stderr file object (sys module), 184, 214
- stdin file object (sys module), 214
- __stdout__ file object (sys module), 214
- stdout file object (sys module), 181, 214
- StopIteration (exception), 138, 279
- __str__(), 243, 244, 250, 252
- str type, 65–94, 383, 418–419
 - capitalize(), 73
 - center(), 73
 - comparing, 68–69
 - count(), 73, 75
- str type (*cont.*)
 - encode(), 73, 92, 93, 296, 336, 419, 441
 - endswith(), 73, 75, 76
 - escapes, 66, 67
 - expandtabs(), 73
 - find(), 72–75, 133, 532
 - format(), 73, 78–88, 152, 156, 186, 189, 249, 306, 531
 - format specifications, 83–88
 - index(), 72–75
 - isalnum(), 73
 - isalpha(), 73
 - isdecimal(), 73
 - isdigit(), 73, 76
 - isidentifier(), 73, 348
 - islower(), 73
 - isnumeric(), 74
 - isprintable(), 74
 - isspace(), 74, 531
 - istitle(), 74
 - isupper(), 74
 - join(), 71, 72, 74, 189
 - literal concatenation, 78
 - ljust(), 74
 - lower(), 74, 76
 - lstrip(), 75, 76
 - maketrans(), 74, 77–78
 - methods, table of, 73, 74, 75
 - partition(), 74, 76
 - raw strings, 67, 204, 310, 500, 556
 - replace(), 74, 77, 101
 - replication (*, *), 72, 90
 - reversing, 71, 72
 - rfind(), 73, 75, 76
 - rindex(), 73, 75
 - rjust(), 74
 - rpartition(), 74, 76
 - rsplit(), 74
 - rstrip(), 75, 76
 - slicing, 69–71
 - slicing operator ([]), 69
 - split(), 74, 77, 509
 - splitlines(), 74

- str type (*cont.*)
 - startswith(), 74, 75, 76
 - strip(), 75, 76
 - str() (built-in), 65, 136, 243, 250
 - swapcase(), 75
 - title(), 75, 90
 - translate(), 75, 77–78
 - triple quoted, 65, 156, 204
 - upper(), 75
 - zfill(), 75
- striding; *see* slicing
- string attribute (match object), 507
- string form, 82–83
- string handling, 213–214
- string literal concatenation, 78
- string module, 130, 213
- StringIO type (io module), 213–214, 228
- strings; *see* str type
- StringVar type (tkinter module), 574, 590, 592
- strip()
 - bytearray type, 300
 - bytes type, 300
 - str type, 75, 76
- strong typing, 17
- strptime() (datetime.datetime type), 309
- struct module, 213, 296–298
 - calcsize(), 297
 - pack(), 296, 297, 301, 336
 - Struct type, 297, 302, 324, 336, 462
 - unpack(), 297, 302, 336
- __sub__ () (-), 55, 253
- sub()
 - re module, 502, 504, 505
 - regex object, 503
- subn()
 - re module, 502
 - regex object, 503
- subprocess module, 440–442
 - call(), 209
 - Popen(), 441
- suffix; *see* extension
- sum() (built-in), 140, 396, 397
- super() (built-in), 241, 244, 256, 276, 282, 381, 385
- .svg (extension), 525
- SVG (Scalable Vector Graphics), 525
- swapcase()
 - bytearray type, 300
 - bytes type, 300
 - str type, 75
- switch statement; *see* dictionary branching
- symmetric_difference_update() (set type), 123
- symmetric_difference()
 - frozenset type, 123
 - set type, 122, 123
- sync() (shelve module), 477
- syntactic analysis, 514
- syntax rules, 515
- SyntaxError (exception), 54, 348, 414–415
- sys module
 - argv list, 41, 343
 - executable attribute, 441
 - exit(), 141, 215
 - float_info.epsilon attribute, 61, 96, 343
 - getrecursionlimit(), 352
 - maxunicode attribute, 90, 92
 - modules attribute, 348
 - path attribute, 197
 - platform attribute, 160, 209, 344
 - setrecursionlimit(), 352
 - stderr file object, 184, 214
 - stdin file object, 214
 - __stdout__ file object, 214
 - stdout file object, 181, 214
 - system() (os module), 444

T

- tan() (math module), 61
- tanh() (math module), 61
- tarfile module, 219, 221–222
 - .tar, .tar.gz, .tar.bz2 (extension), 219, 221
- Tcl/Tk, 569
- TCP (Transmission Control Protocol), 225, 457
- TDD (Test Driven Development), 426
- tell() (file object), 326, 329
- telnetlib module, 226
- tempfile module, 222
 - gettempdir(), 360
- temporary files and directories, 222
- terminal, 515
- terminology, object-oriented, 235
- Test Driven Development (TDD), 426
- testmod() (doctest module), 206
- text files, 131, 305–312
- TextFilter.py (example), 385
- TextUtil.py (example), 202–207
- textwrap module, 213
 - dedent(), 307
 - TextWrapper type, 306
 - wrap(), 306, 320
- .tgz (extension), 219, 221
- this*; *see* self object
- Thread type (threading module), 445, 448, 450, 451, 452
 - run(), 445, 448
 - start(), 445
- threading module, 445–453
 - Lock type, 452, 467
 - Thread type, 445, 448, 450, 451, 452
- time module, 216
 - localtime(), 217
 - time(), 217
- timeit module, 432–434
- timer, single shot, 582, 586
- title()
 - bytearray type, 300
 - bytes type, 300
 - str type, 75, 90
- Tk type (tkinter module), 572, 578, 589
- tkinter.filedialog module
 - askopenfilename(), 586
 - asksaveasfilename(), 585
- tkinter.messagebox module
 - askyesno(), 589
 - askyesnocancel(), 584
 - showwarning(), 585, 587
- tkinter module, 569
 - Button type, 581, 591
 - DoubleVar type, 574
 - END constant, 583, 587, 588
 - Entry type, 591
 - Frame type, 573, 581, 591
 - IntVar type, 574
 - Label type, 574, 582, 583, 591
 - Listbox type, 582, 583, 587, 588, 589
 - Menu type, 579, 580
 - PhotoImage type, 581
 - Scale type, 574, 575
 - Scrollbar type, 582
 - StringVar type, 574, 590, 592
 - Tk type, 572, 578, 589
 - TopLevel type, 590
- today() (datetime.date type), 187, 477
- tokenizing, 514
- toordinal() (datetime.date type), 301
- TopLevel type (tkinter module), 590
- trace module, 360
- traceback, 415–420
- translate()
 - bytearray type, 300
 - bytes type, 300
 - str type, 75, 77–78
- Transmission Control Protocol (TCP), 225, 457
- triple quoted strings, 65, 156, 204

True (built-in constant); *see* bool type
`__truediv__()` (/), 31, 55, 253
`trunc()` (math module), 61
`truncate()` (file object), 326, 331
 truth values; *see* bool type
`try` (statement), 163–171, 360
 see also exceptions and exception handling
 tuple type, 108–111, 383
 comparing, 108
 `count()`, 108
 `index()`, 108
 parentheses policy, 109
 replication (*, * =), 108
 slicing, 108
 `tuple()` (built-in), 108
`type()` (built-in), 18
 type checking, 361
 type conversion; *see* conversions
 type type, 391
 `__init__()`, 391, 392
 `__new__()`, 392, 394
 `type()` (built-in), 348, 349
 TypeError (exception), 57, 135, 138, 146, 167, 173, 179, 197, 242, 258, 259, 274, 364, 380
 typing; *see* dynamic typing

U

UCS-2/4 encoding (Unicode), 92
 UDP (User Datagram Protocol), 225, 457
 uncompressing files, 219
 underscore (`_`), 53
`unescape()` (xml.sax.saxutils module), 226
 unhandled exception; *see* traceback
 Unicode, 9, 91–94, 505
 collation order, 68–69
 identifiers, 53
 strings; *see* str type, 65–94
 UCS-2/4 encoding, 92

Unicode (*cont.*)
 UTF-8/16/32 encoding, 92, 94, 228
 see also character encodings
 unicodedata module, 68
 `category()`, 361
 `name()`, 90
 `normalize()`, 68
 UnicodeDecodeError (exception), 167
 UnicodeEncodeError (exception), 93
 unimplementing methods, 258–261
`union()` (set type), 122, 123
`uniquewords1.py` (example), 130
`uniquewords2.py` (example), 136
 unittest module, 228, 426–432
 Unix-style paths, 142
 unordered collections; *see* dict, frozenset, and set types
`unpack()` (struct module), 297, 302, 336
 unpacking (* and **), 110, 114–115, 162, 177–180, 187, 268, 304, 336
`untar.py` (example), 221
 UPDATE (SQL statement), 484
`update()`
 dict type, 129, 188, 276, 295
 set type, 123
 updating dictionaries, 128
 updating lists, 115
`upper()`
 bytearray type, 293, 301
 bytes type, 293, 301
 str type, 75
 urllib package, 226
 User Datagram Protocol (UDP), 225, 457
 UTC (Coordinated Universal Time), 216
`utcnow()` (datetime.datetime type), 217
 UTF-8/16/32 encoding (Unicode), 92, 94, 228
 uu module, 219

V

Valid.py (example), 407–409
 ValueError (exception), 57, 272, 279
 values() (dict type), 128, 129
 variables; *see* object references
 variables, callable; *see* functions and methods
 variables, class, 255, 465
 variables, global, 180
 variables, instance, 241
 variables, local, 163
 variables, names; *see* identifiers
 variables, static, 255
 vars() (built-in), 349
 version control, 414
 view (dict type), 129
 virtual subclasses, 391

W

walk() (os module), 223, 224, 406
 .wav (extension), 219
 wave module, 219
 weak reference, 581
 weakref module, 218
 Web Server Gateway Interface (WSGI), 225
 webbrowser module, 589
 while loop, 141, 161–162
 wildcard expansion, 343
 Windows, file association, 11
 windows, resizable, 582–583, 591
 with (statement), 369–372, 389
 wrap() (textwrap module), 306, 320
 @wraps() (functools module), 357
 writable() (file object), 326
 write()
 file object, 131, 214, 301, 326, 327
 gzip module, 301
 writelines() (file object), 326
 WSGI (Web Server Gateway Interface), 225
 wsgiref package, 225
 wxPython, 570, 593

X

xdrllib module, 219
 xml.dom.minidom module, 226
 xml.dom module, 226, 316–319
 XML encoding, 314
 XML escapes, 186, 316
 xml.etree.ElementTree module, 227, 227–228
 xml.etree package, 313–316
 XML file format, 94
 XML files, 312–323
 XML parsers, expat, 315, 317, 318
 xml.parsers.expat module, 227
 xml.sax module, 226, 321–323
 xml.sax.saxutils module, 186, 226
 escape(), 186, 226, 320
 quoteattr(), 226, 320
 unescape(), 226
 xmlrpc package, 226
 XmlShadow.py (example), 373
 __xor__ (^), 57, 253
 .xpm (extension), 268

Y

yield (statement), 279, 281, 342–344, 399–407

Z

ZeroDivisionError (exception), 165, 416
 zfill()
 bytearray type, 301
 bytes type, 301
 str type, 75
 .zip (extension), 219
 zip() (built-in), 127, 140, 143–144, 205, 389
 zipfile module, 219