ssssssssssssssssssssssssssssssssssAbout This eBook ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer’s Web site. Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app. Effective Python 59 SPECIFIC WAYS TO WRITE BETTER PYTHON Brett Slatkin Upper Saddle River, NJ • Boston • Indianapolis • San Francisco New York • Toronto • Montreal • London • Munich • Paris • Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City 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. For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419. For government sales inquiries, please contact governmentsales@pearsoned.com. For questions about sales outside the United States, please contact international@pearsoned.com. Visit us on the Web: informit.com/aw Library of Congress Cataloging-in-Publication Data Slatkin, Brett, author. Effective Python : 59 specific ways to write better Python / Brett Slatkin. pages cm Includes index. ISBN 978-0-13-403428-7 (pbk. : alk. paper)—ISBN 0-13-403428-7 (pbk. : alk. paper) 1. Python (Computer program language) 2. Computer programming. I. Title. QA76.73.P98S57 2015 005.13’3—dc23 2014048305 Copyright © 2015 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290. ISBN-13: 978-0-13-403428-7 ISBN-10: 0-13-403428-7 Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana. First printing, March 2015 Editor-in-Chief Mark L. Taub Senior Acquisitions Editor Trina MacDonald Managing Editor John Fuller Full-Service Production Manager Julie B. Nahil Copy Editor Stephanie Geels Indexer Jack Lewis Proofreader Melissa Panagos Technical Reviewers Brett Cannon Tavis Rudd Mike Taylor Editorial Assistant Olivia Basegio Cover Designer Chuti Prasertsith Compositor LaurelTech Praise for Effective Python “Each item in Slatkin’s Effective Python teaches a self-contained lesson with its own source code. This makes the book random-access: Items are easy to browse and study in whatever order the reader needs. I will be recommending Effective Python to students as an admirably compact source of mainstream advice on a very broad range of topics for the intermediate Python programmer.” —Brandon Rhodes, software engineer at Dropbox and chair of PyCon 2016-2017 “I’ve been programming in Python for years and thought I knew it pretty well. Thanks to this treasure trove of tips and techniques, I realize there’s so much more I could be doing with my Python code to make it faster (e.g., using built-in data structures), easier to read (e.g., enforcing keyword-only arguments), and much more Pythonic (e.g., using zip to iterate over lists in parallel).” —Pamela Fox, educationeer, Khan Academy “If I had this book when I first switched from Java to Python, it would have saved me many months of repeated code rewrites, which happened each time I realized I was doing particular things ‘non-Pythonically.’ This book collects the vast majority of basic Python ‘must-knows’ into one place, eliminating the need to stumble upon them one-by-one over the course of months or years. The scope of the book is impressive, starting with the importance of PEP8 as well as that of major Python idioms, then reaching through function, method and class design, effective standard library use, quality API design, testing, and performance measurement—this book really has it all. A fantastic introduction to what it really means to be a Python programmer for both the novice and the experienced developer.” —Mike Bayer, creator of SQLAlchemy “Effective Python will take your Python skills to the next level with clear guidelines for improving Python code style and function.” —Leah Culver, developer advocate, Dropbox “This book is an exceptionally great resource for seasoned developers in other languages who are looking to quickly pick up Python and move beyond the basic language constructs into more Pythonic code. The organization of the book is clear, concise, and easy to digest, and each item and chapter can stand on its own as a meditation on a particular topic. The book covers the breadth of language constructs in pure Python without confusing the reader with the complexities of the broader Python ecosystem. For more seasoned developers the book provides in-depth examples of language constructs they may not have previously encountered, and provides examples of less commonly used language features. It is clear that the author is exceptionally facile with Python, and he uses his professional experience to alert the reader to common subtle bugs and common failure modes. Furthermore, the book does an excellent job of pointing out subtleties between Python 2.X and Python 3.X and could serve as a refresher course as one transitions between variants of Python.” —Katherine Scott, software lead, Tempo Automation “This is a great book for both novice and experienced programmers. The code examples and explanations are well thought out and explained concisely and thoroughly.” —C. Titus Brown, associate professor, UC Davis “This is an immensely useful resource for advanced Python usage and building cleaner, more maintainable software. Anyone looking to take their Python skills to the next level would benefit from putting the book’s advice into practice.” —Wes McKinney, creator of pandas; author of Python for Data Analysis; and software engineer at Cloudera To our family, loved and lost Contents Preface Acknowledgments About the Author Chapter 1: Pythonic Thinking Item 1: Know Which Version of Python You’re Using Item 2: Follow the PEP 8 Style Guide Item 3: Know the Differences Between bytes, str, and unicode Item 4: Write Helper Functions Instead of Complex Expressions Item 5: Know How to Slice Sequences Item 6: Avoid Using start, end, and stride in a Single Slice Item 7: Use List Comprehensions Instead of map and filter Item 8: Avoid More Than Two Expressions in List Comprehensions Item 9: Consider Generator Expressions for Large Comprehensions Item 10: Prefer enumerate Over range Item 11: Use zip to Process Iterators in Parallel Item 12: Avoid else Blocks After for and while Loops Item 13: Take Advantage of Each Block in try/except/else/finally Chapter 2: Functions Item 14: Prefer Exceptions to Returning None Item 15: Know How Closures Interact with Variable Scope Item 16: Consider Generators Instead of Returning Lists Item 17: Be Defensive When Iterating Over Arguments Item 18: Reduce Visual Noise with Variable Positional Arguments Item 19: Provide Optional Behavior with Keyword Arguments Item 20: Use None and Docstrings to Specify Dynamic Default Arguments Item 21: Enforce Clarity with Keyword-Only Arguments Chapter 3: Classes and Inheritance Item 22: Prefer Helper Classes Over Bookkeeping with Dictionaries and Tuples Item 23: Accept Functions for Simple Interfaces Instead of Classes Item 24: Use @classmethod Polymorphism to Construct Objects Generically Item 25: Initialize Parent Classes with super Item 26: Use Multiple Inheritance Only for Mix-in Utility Classes Item 27: Prefer Public Attributes Over Private Ones Item 28: Inherit from collections.abc for Custom Container Types Chapter 4: Metaclasses and Attributes Item 29: Use Plain Attributes Instead of Get and Set Methods Item 30: Consider @property Instead of Refactoring Attributes Item 31: Use Descriptors for Reusable @property Methods Item 32: Use \_\_getattr\_\_, \_\_getattribute\_\_, and \_\_setattr\_\_ for Lazy Attributes Item 33: Validate Subclasses with Metaclasses Item 34: Register Class Existence with Metaclasses Item 35: Annotate Class Attributes with Metaclasses Chapter 5: Concurrency and Parallelism Item 36: Use subprocess to Manage Child Processes Item 37: Use Threads for Blocking I/O, Avoid for Parallelism Item 38: Use Lock to Prevent Data Races in Threads Item 39: Use Queue to Coordinate Work Between Threads Item 40: Consider Coroutines to Run Many Functions Concurrently Item 41: Consider concurrent.futures for True Parallelism Chapter 6: Built-in Modules Item 42: Define Function Decorators with functools.wraps Item 43: Consider contextlib and with Statements for Reusable try/finally Behavior Item 44: Make pickle Reliable with copyreg Item 45: Use datetime Instead of time for Local Clocks Item 46: Use Built-in Algorithms and Data Structures Item 47: Use decimal When Precision Is Paramount Item 48: Know Where to Find Community-Built Modules Chapter 7: Collaboration Item 49: Write Docstrings for Every Function, Class, and Module Item 50: Use Packages to Organize Modules and Provide Stable APIs Item 51: Define a Root Exception to Insulate Callers from APIs Item 52: Know How to Break Circular Dependencies Item 53: Use Virtual Environments for Isolated and Reproducible Dependencies Chapter 8: Production Item 54: Consider Module-Scoped Code to Configure Deployment Environments Item 55: Use repr Strings for Debugging Output Item 56: Test Everything with unittest Item 57: Consider Interactive Debugging with pdb Item 58: Profile Before Optimizing Item 59: Use tracemalloc to Understand Memory Usage and Leaks Index Preface The Python programming language has unique strengths and charms that can be hard to grasp. Many programmers familiar with other languages often approach Python from a limited mindset instead of embracing its full expressivity. Some programmers go too far in the other direction, overusing Python features that can cause big problems later. This book provides insight into the Pythonic way of writing programs: the best way to use Python. It builds on a fundamental understanding of the language that I assume you already have. Novice programmers will learn the best practices of Python’s capabilities. Experienced programmers will learn how to embrace the strangeness of a new tool with confidence. My goal is to prepare you to make a big impact with Python. What This Book Covers Each chapter in this book contains a broad but related set of items. Feel free to jump between items and follow your interest. Each item contains concise and specific guidance explaining how you can write Python programs more effectively. Items include advice on what to do, what to avoid, how to strike the right balance, and why this is the best choice. The items in this book are for Python 3 and Python 2 programmers alike (see Item 1: “Know Which Version of Python You’re Using”). Programmers using alternative runtimes like Jython, IronPython, or PyPy should also find the majority of items to be applicable. Chapter 1: Pythonic Thinking The Python community has come to use the adjective Pythonic to describe code that follows a particular style. The idioms of Python have emerged over time through experience using the language and working with others. This chapter covers the best way to do the most common things in Python. Chapter 2: Functions Functions in Python have a variety of extra features that make a programmer’s life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. This chapter covers how to use functions to clarify intention, promote reuse, and reduce bugs. Chapter 3: Classes and Inheritance Python is an object-oriented language. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies. This chapter covers how to use classes and inheritance to express your intended behaviors with objects. Chapter 4: Metaclasses and Attributes Metaclasses and dynamic attributes are powerful Python features. However, they also enable you to implement extremely bizarre and unexpected behaviors. This chapter covers the common idioms for using these mechanisms to ensure that you follow the rule of least surprise. Chapter 5: Concurrency and Parallelism Python makes it easy to write concurrent programs that do many different things seemingly at the same time. Python can also be used to do parallel work through system calls, subprocesses, and C-extensions. This chapter covers how to best utilize Python in these subtly different situations. Chapter 6: Built-in Modules Python is installed with many of the important modules that you’ll need to write programs. These standard packages are so closely intertwined with idiomatic Python that they may as well be part of the language specification. This chapter covers the essential built-in modules. Chapter 7: Collaboration Collaborating on Python programs requires you to be deliberate about how you write your code. Even if you’re working alone, you’ll want to understand how to use modules written by others. This chapter covers the standard tools and best practices that enable people to work together on Python programs. Chapter 8: Production Python has facilities for adapting to multiple deployment environments. It also has built-in modules that aid in hardening your programs and making them bulletproof. This chapter covers how to use Python to debug, optimize, and test your programs to maximize quality and performance at runtime. Conventions Used in This Book Python code snippets in this book are in monospace font and have syntax highlighting. I take some artistic license with the Python style guide to make the code examples better fit the format of a book or to highlight the most important parts. When lines are long, I use characters to indicate that they wrap. I truncate snippets with ellipses comments (#…) to indicate regions where code exists that isn’t essential for expressing the point. I’ve also left out embedded documentation to reduce the size of code examples. I strongly suggest that you don’t do this in your projects; instead, you should follow the style guide (see Item 2: “Follow the PEP 8 Style Guide”) and write documentation (see Item 49: “Write Docstrings for Every Function, Class, and Module”). Most code snippets in this book are accompanied by the corresponding output from running the code. When I say “output,” I mean console or terminal output: what you see when running the Python program in an interactive interpreter. Output sections are in monospace font and are preceded by a >>> line (the Python interactive prompt). The idea is that you could type the code snippets into a Python shell and reproduce the expected output. Finally, there are some other sections in monospace font that are not preceded by a >>> line. These represent the output of running programs besides the Python interpreter. These examples often begin with $ characters to indicate that I’m running programs from a command-line shell like Bash. Where to Get the Code and Errata It’s useful to view some of the examples in this book as whole programs without interleaved prose. This also gives you a chance to tinker with the code yourself and understand why the program works as described. You can find the source code for all code snippets in this book on the book’s website (http://www.effectivepython.com). Any errors found in the book will have corrections posted on the website. Acknowledgments This book would not have been possible without the guidance, support, and encouragement from many people in my life. Thanks to Scott Meyers for the Effective Software Development series. I first read Effective C++ when I was 15 years old and fell in love with the language. There’s no doubt that Scott’s books led to my academic experience and first job at Google. I’m thrilled to have had the opportunity to write this book. Thanks to my core technical reviewers for the depth and thoroughness of their feedback: Brett Cannon, Tavis Rudd, and Mike Taylor. Thanks to Leah Culver and Adrian Holovaty for thinking this book would be a good idea. Thanks to my friends who patiently read earlier versions of this book: Michael Levine, Marzia Niccolai, Ade Oshineye, and Katrina Sostek. Thanks to my colleagues at Google for their review. Without all of your help, this book would have been inscrutable. Thanks to everyone involved in making this book a reality. Thanks to my editor Trina MacDonald for kicking off the process and being supportive throughout. Thanks to the team who were instrumental: development editors Tom Cirtin and Chris Zahn, editorial assistant Olivia Basegio, marketing manager Stephane Nakib, copy editor Stephanie Geels, and production editor Julie Nahil. Thanks to the wonderful Python programmers I’ve known and worked with: Anthony Baxter, Brett Cannon, Wesley Chun, Jeremy Hylton, Alex Martelli, Neal Norwitz, Guido van Rossum, Andy Smith, Greg Stein, and Ka-Ping Yee. I appreciate your tutelage and leadership. Python has an excellent community and I feel lucky to be a part of it. Thanks to my teammates over the years for letting me be the worst player in the band. Thanks to Kevin Gibbs for helping me take risks. Thanks to Ken Ashcraft, Ryan Barrett, and Jon McAlister for showing me how it’s done. Thanks to Brad Fitzpatrick for taking it to the next level. Thanks to Paul McDonald for co-founding our crazy project. Thanks to Jeremy Ginsberg and Jack Hebert for making it a reality. Thanks to the inspiring programming teachers I’ve had: Ben Chelf, Vince Hugo, Russ Lewin, Jon Stemmle, Derek Thomson, and Daniel Wang. Without your instruction, I would never have pursued our craft or gained the perspective required to teach others. Thanks to my mother for giving me a sense of purpose and encouraging me to become a programmer. Thanks to my brother, my grandparents, and the rest of my family and childhood friends for being role models as I grew up and found my passion. Finally, thanks to my wife, Colleen, for her love, support, and laughter through the journey of life. About the Author Brett Slatkin is a senior staff software engineer at Google. He is the engineering lead and co-founder of Google Consumer Surveys. He formerly worked on Google App Engine’s Python infrastructure. He is the co-creator of the PubSubHubbub protocol. Nine years ago he cut his teeth using Python to manage Google’s enormous fleet of servers. Outside of his day job, he works on open source tools and writes about software, bicycles, and other topics on his personal website (http://onebigfluke.com). He earned his B.S. in computer engineering from Columbia University in the City of New York. He lives in San Francisco. 1. Pythonic Thinking The idioms of a programming language are defined by its users. Over the years, the Python community has come to use the adjective Pythonic to describe code that follows a particular style. The Pythonic style isn’t regimented or enforced by the compiler. It has emerged over time through experience using the language and working with others. Python programmers prefer to be explicit, to choose simple over complex, and to maximize readability (type import this). Programmers familiar with other languages may try to write Python as if it’s C++, Java, or whatever they know best. New programmers may still be getting comfortable with the vast range of concepts expressible in Python. It’s important for everyone to know the best—the Pythonic—way to do the most common things in Python. These patterns will affect every program you write. Item 1: Know Which Version of Python You’re Using Throughout this book, the majority of example code is in the syntax of Python 3.4 (released March 17, 2014). This book also provides some examples in the syntax of Python 2.7 (released July 3, 2010) to highlight important differences. Most of my advice applies to all of the popular Python runtimes: CPython, Jython, IronPython, PyPy, etc. Many computers come with multiple versions of the standard CPython runtime preinstalled. However, the default meaning of python on the command-line may not be clear. python is usually an alias for python2.7, but it can sometimes be an alias for older versions like python2.6 or python2.5. To find out exactly which version of Python you’re using, you can use the --version flag. $ python —version Python 2.7.8 Python 3 is usually available under the name python3. $ python3 —version Python 3.4.2 You can also figure out the version of Python you’re using at runtime by inspecting values in the sys built-in module. Click here to view code image import sys print(sys.version\_info) print(sys.version) >>> sys.version\_info(major=3, minor=4, micro=2, releaselevel=‘final’, serial=0) 3.4.2 (default, Oct 19 2014, 17:52:17) [GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)] Python 2 and Python 3 are both actively maintained by the Python community. Development on Python 2 is frozen beyond bug fixes, security improvements, and backports to ease the transition from Python 2 to Python 3. Helpful tools like the 2to3 and six exist to make it easier to adopt Python 3 going forward. Python 3 is constantly getting new features and improvements that will never be added to Python 2. As of the writing of this book, the majority of Python’s most common open source libraries are compatible with Python 3. I strongly encourage you to use Python 3 for your next Python project. Things to Remember There are two major versions of Python still in active use: Python 2 and Python 3. There are multiple popular runtimes for Python: CPython, Jython, IronPython, PyPy, etc. Be sure that the command-line for running Python on your system is the version you expect it to be. Prefer Python 3 for your next project because that is the primary focus of the Python community. Item 2: Follow the PEP 8 Style Guide Python Enhancement Proposal #8, otherwise known as PEP 8, is the style guide for how to format Python code. You are welcome to write Python code however you want, as long as it has valid syntax. However, using a consistent style makes your code more approachable and easier to read. Sharing a common style with other Python programmers in the larger community facilitates collaboration on projects. But even if you are the only one who will ever read your code, following the style guide will make it easier to change things later. PEP 8 has a wealth of details about how to write clear Python code. It continues to be updated as the Python language evolves. It’s worth reading the whole guide online (http://www.python.org/dev/peps/pep-0008/). Here are a few rules you should be sure to follow: Whitespace: In Python, whitespace is syntactically significant. Python programmers are especially sensitive to the effects of whitespace on code clarity. • Use spaces instead of tabs for indentation. • Use four spaces for each level of syntactically significant indenting. • Lines should be 79 characters in length or less. • Continuations of long expressions onto additional lines should be indented by four extra spaces from their normal indentation level. • In a file, functions and classes should be separated by two blank lines. • In a class, methods should be separated by one blank line. • Don’t put spaces around list indexes, function calls, or keyword argument assignments. • Put one—and only one—space before and after variable assignments. Naming: PEP 8 suggests unique styles of naming for different parts in the language. This makes it easy to distinguish which type corresponds to each name when reading code. • Functions, variables, and attributes should be in lowercase\_underscore format. • Protected instance attributes should be in \_leading\_underscore format. • Private instance attributes should be in \_\_double\_leading\_underscore format. • Classes and exceptions should be in CapitalizedWord format. • Module-level constants should be in ALL\_CAPS format. • Instance methods in classes should use self as the name of the first parameter (which refers to the object). • Class methods should use cls as the name of the first parameter (which refers to the class). Expressions and Statements: The Zen of Python states: “There should be one—and preferably only one—obvious way to do it.” PEP 8 attempts to codify this style in its guidance for expressions and statements. • Use inline negation (if a is not b) instead of negation of positive expressions (if not a is b). • Don’t check for empty values (like [] or '') by checking the length (if len(somelist) == 0). Use if not somelist and assume empty values implicitly evaluate to False. • The same thing goes for non-empty values (like [1] or 'hi'). The statement if somelist is implicitly True for non-empty values. • Avoid single-line if statements, for and while loops, and except compound statements. Spread these over multiple lines for clarity. • Always put import statements at the top of a file. • Always use absolute names for modules when importing them, not names relative to the current module’s own path. For example, to import the foo module from the bar package, you should do from bar import foo, not just import foo. • If you must do relative imports, use the explicit syntax from . import foo. • Imports should be in sections in the following order: standard library modules, thirdparty modules, your own modules. Each subsection should have imports in alphabetical order. Note The Pylint tool (http://www.pylint.org/) is a popular static analyzer for Python source code. Pylint provides automated enforcement of the PEP 8 style guide and detects many other types of common errors in Python programs. Things to Remember Always follow the PEP 8 style guide when writing Python code. Sharing a common style with the larger Python community facilitates collaboration with others. Using a consistent style makes it easier to modify your own code later. Item 3: Know the Differences Between bytes, str, and unicode In Python 3, there are two types that represent sequences of characters: bytes and str. Instances of bytes contain raw 8-bit values. Instances of str contain Unicode characters. In Python 2, there are two types that represent sequences of characters: str and unicode. In contrast to Python 3, instances of str contain raw 8-bit values. Instances of unicode contain Unicode characters. There are many ways to represent Unicode characters as binary data (raw 8-bit values). The most common encoding is UTF-8. Importantly, str instances in Python 3 and unicode instances in Python 2 do not have an associated binary encoding. To convert Unicode characters to binary data, you must use the encode method. To convert binary data to Unicode characters, you must use the decode method. When you’re writing Python programs, it’s important to do encoding and decoding of Unicode at the furthest boundary of your interfaces. The core of your program should use Unicode character types (str in Python 3, unicode in Python 2) and should not assume anything about character encodings. This approach allows you to be very accepting of alternative text encodings (such as Latin-1, Shift JIS, and Big5) while being strict about your output text encoding (ideally, UTF-8). The split between character types leads to two common situations in Python code: You want to operate on raw 8-bit values that are UTF-8-encoded characters (or some other encoding). You want to operate on Unicode characters that have no specific encoding. You’ll often need two helper functions to convert between these two cases and to ensure that the type of input values matches your code’s expectations. In Python 3, you’ll need one method that takes a str or bytes and always returns a str. Click here to view code image def to\_str(bytes\_or\_str): if isinstance(bytes\_or\_str, bytes): value = bytes\_or\_str.decode(‘utf-8’) else: value = bytes\_or\_str return value # Instance of str You’ll need another method that takes a str or bytes and always returns a bytes. Click here to view code image def to\_bytes(bytes\_or\_str): if isinstance(bytes\_or\_str, str): value = bytes\_or\_str.encode(‘utf-8’) else: value = bytes\_or\_str return value # Instance of bytes In Python 2, you’ll need one method that takes a str or unicode and always returns a unicode. Click here to view code image # Python 2 def to\_unicode(unicode\_or\_str): if isinstance(unicode\_or\_str, str): value = unicode\_or\_str.decode(‘utf-8’) else: value = unicode\_or\_str return value # Instance of unicode You’ll need another method that takes str or unicode and always returns a str. Click here to view code image # Python 2 def to\_str(unicode\_or\_str): if isinstance(unicode\_or\_str, unicode): value = unicode\_or\_str.encode(‘utf-8’) else: value = unicode\_or\_str return value # Instance of str There are two big gotchas when dealing with raw 8-bit values and Unicode characters in Python. The first issue is that in Python 2, unicode and str instances seem to be the same type when a str only contains 7-bit ASCII characters. You can combine such a str and unicode together using the + operator. You can compare such str and unicode instances using equality and inequality operators. You can use unicode instances for format strings like '%s'. All of this behavior means that you can often pass a str or unicode instance to a function expecting one or the other and things will just work (as long as you’re only dealing with 7-bit ASCII). In Python 3, bytes and str instances are never equivalent— not even the empty string—so you must be more deliberate about the types of character sequences that you’re passing around. The second issue is that in Python 3, operations involving file handles (returned by the open built-in function) default to UTF-8 encoding. In Python 2, file operations default to binary encoding. This causes surprising failures, especially for programmers accustomed to Python 2. For example, say you want to write some random binary data to a file. In Python 2, this works. In Python 3, this breaks. Click here to view code image with open(‘/tmp/random.bin’, ‘w’) as f: f.write(os.urandom(10)) >>> TypeError: must be str, not bytes The cause of this exception is the new encoding argument for open that was added in Python 3. This parameter defaults to 'utf-8'. That makes read and write operations on file handles expect str instances containing Unicode characters instead of bytes instances containing binary data. To make this work properly, you must indicate that the data is being opened in write binary mode ('wb') instead of write character mode ('w'). Here, I use open in a way that works correctly in Python 2 and Python 3: Click here to view code image with open(‘/tmp/random.bin’, ‘wb’) as f: f.write(os.urandom(10)) This problem also exists for reading data from files. The solution is the same: Indicate binary mode by using 'rb' instead of 'r' when opening a file. Things to Remember In Python 3, bytes contains sequences of 8-bit values, str contains sequences of Unicode characters. bytes and str instances can’t be used together with operators (like > or +). In Python 2, str contains sequences of 8-bit values, unicode contains sequences of Unicode characters. str and unicode can be used together with operators if the str only contains 7-bit ASCII characters. Use helper functions to ensure that the inputs you operate on are the type of character sequence you expect (8-bit values, UTF-8 encoded characters, Unicode characters, etc.). If you want to read or write binary data to/from a file, always open the file using a binary mode (like 'rb' or 'wb'). Item 4: Write Helper Functions Instead of Complex Expressions Python’s pithy syntax makes it easy to write single-line expressions that implement a lot of logic. For example, say you want to decode the query string from a URL. Here, each query string parameter represents an integer value: Click here to view code image from urllib.parse import parse\_qs my\_values = parse\_qs(‘red=5&blue=0&green=’, keep\_blank\_values=True) print(repr(my\_values)) >>> {‘red’: [‘5’], ‘green’: [”], ‘blue’: [‘0’]} Some query string parameters may have multiple values, some may have single values, some may be present but have blank values, and some may be missing entirely. Using the get method on the result dictionary will return different values in each circumstance. Click here to view code image print(‘Red: ’, my\_values.get(‘red’)) print(‘Green: ’, my\_values.get(‘green’)) print(‘Opacity: ‘, my\_values.get(‘opacity’)) >>> Red: [‘5’] Green: [”] Opacity: None It’d be nice if a default value of 0 was assigned when a parameter isn’t supplied or is blank. You might choose to do this with Boolean expressions because it feels like this logic doesn’t merit a whole if statement or helper function quite yet. Python’s syntax makes this choice all too easy. The trick here is that the empty string, the empty list, and zero all evaluate to False implicitly. Thus, the expressions below will evaluate to the subexpression after the or operator when the first subexpression is False. Click here to view code image # For query string ‘red=5&blue=0&green=’ red = my\_values.get(‘red’, [”])[0] or 0 green = my\_values.get(‘green’, [”])[0] or 0 opacity = my\_values.get(‘opacity’, [”])[0] or 0 print(‘Red: %r’ % red) print(‘Green: %r’ % green) print(‘Opacity: %r’ % opacity) >>> Red: ‘5’ Green: 0 Opacity: 0 The red case works because the key is present in the my\_values dictionary. The value is a list with one member: the string '5'. This string implicitly evaluates to True, so red is assigned to the first part of the or expression. The green case works because the value in the my\_values dictionary is a list with one member: an empty string. The empty string implicitly evaluates to False, causing the or expression to evaluate to 0. The opacity case works because the value in the my\_values dictionary is missing altogether. The behavior of the get method is to return its second argument if the key doesn’t exist in the dictionary. The default value in this case is a list with one member, an empty string. When opacity isn’t found in the dictionary, this code does exactly the same thing as the green case. However, this expression is difficult to read and it still doesn’t do everything you need. You’d also want to ensure that all the parameter values are integers so you can use them in mathematical expressions. To do that, you’d wrap each expression with the int built-in function to parse the string as an integer. Click here to view code image red = int(my\_values.get(‘red’, [”])[0] or 0) This is now extremely hard to read. There’s so much visual noise. The code isn’t approachable. A new reader of the code would have to spend too much time picking apart the expression to figure out what it actually does. Even though it’s nice to keep things short, it’s not worth trying to fit this all on one line. Python 2.5 added if/else conditional—or ternary—expressions to make cases like this clearer while keeping the code short. Click here to view code image red = my\_values.get(‘red’, [”]) red = int(red[0]) if red[0] else 0 This is better. For less complicated situations, if/else conditional expressions can make things very clear. But the example above is still not as clear as the alternative of a full if/else statement over multiple lines. Seeing all of the logic spread out like this makes the dense version seem even more complex. Click here to view code image green = my\_values.get(‘green’, [”]) if green[0]: green = int(green[0]) else: green = 0 Writing a helper function is the way to go, especially if you need to use this logic repeatedly. Click here to view code image def get\_first\_int(values, key, default=0): found = values.get(key, [”]) if found[0]: found = int(found[0]) else: found = default return found The calling code is much clearer than the complex expression using or and the two-line version using the if/else expression. Click here to view code image green = get\_first\_int(my\_values, ‘green’) As soon as your expressions get complicated, it’s time to consider splitting them into smaller pieces and moving logic into helper functions. What you gain in readability always outweighs what brevity may have afforded you. Don’t let Python’s pithy syntax for complex expressions get you into a mess like this. Things to Remember Python’s syntax makes it all too easy to write single-line expressions that are overly complicated and difficult to read. Move complex expressions into helper functions, especially if you need to use the same logic repeatedly. The if/else expression provides a more readable alternative to using Boolean operators like or and and in expressions. Item 5: Know How to Slice Sequences Python includes syntax for slicing sequences into pieces. Slicing lets you access a subset of a sequence’s items with minimal effort. The simplest uses for slicing are the built-in types list, str, and bytes. Slicing can be extended to any Python class that implements the \_\_getitem\_\_ and \_\_setitem\_\_ special methods (see Item 28: “Inherit from collections.abc for Custom Container Types”). The basic form of the slicing syntax is somelist[start:end], where start is inclusive and end is exclusive. Click here to view code image a = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’] print(‘First four:’, a[:4]) print(‘Last four: ‘, a[-4:]) print(‘Middle two:’, a[3:-3]) >>> First four: [‘a’, ‘b’, ‘c’, ‘d’] Last four: [‘e’, ‘f’, ‘g’, ‘h’] Middle two: [‘d’, ‘e’] When slicing from the start of a list, you should leave out the zero index to reduce visual noise. assert a[:5] == a[0:5] When slicing to the end of a list, you should leave out the final index because it’s redundant. assert a[5:] == a[5:len(a)] Using negative numbers for slicing is helpful for doing offsets relative to the end of a list. All of these forms of slicing would be clear to a new reader of your code. There are no surprises, and I encourage you to use these variations. Click here to view code image a[:] # [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’] a[:5] # [‘a’, ‘b’, ‘c’, ‘d’, ‘e’] a[:-1] # [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’] a[4:] # [‘e’, ‘f’, ‘g’, ‘h’] a[-3:] # [‘f’, ‘g’, ‘h’] a[2:5] # [‘c’, ‘d’, ‘e’] a[2:-1] # [‘c’, ‘d’, ‘e’, ‘f’, ‘g’] a[-3:-1] # [‘f’, ‘g’] Slicing deals properly with start and end indexes that are beyond the boundaries of the list. That makes it easy for your code to establish a maximum length to consider for an input sequence. first\_twenty\_items = a[:20] last\_twenty\_items = a[-20:] In contrast, accessing the same index directly causes an exception. Click here to view code image a[20] >>> IndexError: list index out of range Note Beware that indexing a list by a negative variable is one of the few situations in which you can get surprising results from slicing. For example, the expression somelist[-n:] will work fine when n is greater than one (e.g., somelist[-3:]). However, when n is zero, the expression somelist[-0:] will result in a copy of the original list. The result of slicing a list is a whole new list. References to the objects from the original list are maintained. Modifying the result of slicing won’t affect the original list. Click here to view code image b = a[4:] print(‘Before: ’, b) b[1] = 99 print(‘After: ’, b) print(‘No change:’, a) >>> Before: [‘e’, ‘f’, ‘g’, ‘h’] After: [‘e’, 99, ‘g’, ‘h’] No change: [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’] When used in assignments, slices will replace the specified range in the original list. Unlike tuple assignments (like a, b = c[:2]), the length of slice assignments don’t need to be the same. The values before and after the assigned slice will be preserved. The list will grow or shrink to accommodate the new values. Click here to view code image print(‘Before ‘, a) a[2:7] = [99, 22, 14] print(‘After ’, a) >>> Before [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’] After [‘a’, ‘b’, 99, 22, 14, ‘h’] If you leave out both the start and the end indexes when slicing, you’ll end up with a copy of the original list. Click here to view code image b = a[:] assert b == a and b is not a If you assign a slice with no start or end indexes, you’ll replace its entire contents with a copy of what’s referenced (instead of allocating a new list). Click here to view code image b = a print(‘Before’, a) a[:] = [101, 102, 103] assert a is b # Still the same list object print(‘After ‘, a) # Now has different contents >>> Before [‘a’, ‘b’, 99, 22, 14, ‘h’] After [101, 102, 103] Things to Remember Avoid being verbose: Don’t supply 0 for the start index or the length of the sequence for the end index. Slicing is forgiving of start or end indexes that are out of bounds, making it easy to express slices on the front or back boundaries of a sequence (like a[:20] or a[-20:]). Assigning to a list slice will replace that range in the original sequence with what’s referenced even if their lengths are different. Item 6: Avoid Using start, end, and stride in a Single Slice In addition to basic slicing (see Item 5: “Know How to Slice Sequences”), Python has special syntax for the stride of a slice in the form somelist[start:end:stride]. This lets you take every nth item when slicing a sequence. For example, the stride makes it easy to group by even and odd indexes in a list. Click here to view code image a = [‘red’, ‘orange’, ‘yellow’, ‘green’, ‘blue’, ‘purple’] odds = a[::2] evens = a[1::2] print(odds) print(evens) >>> [‘red’, ‘yellow’, ‘blue’] [‘orange’, ‘green’, ‘purple’] The problem is that the stride syntax often causes unexpected behavior that can introduce bugs. For example, a common Python trick for reversing a byte string is to slice the string with a stride of -1. x = b’mongoose’ y = x[::-1] print(y) >>> b’esoognom’ That works well for byte strings and ASCII characters, but it will break for Unicode characters encoded as UTF-8 byte strings. Click here to view code image w = ‘ ’ x = w.encode(‘utf-8’) y = x[::-1] z = y.decode(‘utf-8’) >>> UnicodeDecodeError: ‘utf-8’ codec can’t decode byte 0x9d in position 0: invalid start byte Are negative strides besides -1 useful? Consider the following examples. Click here to view code image a = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’] a[::2] # [‘a’, ‘c’, ‘e’, ‘g’] a[::-2] # [‘h’, ‘f’, ‘d’, ‘b’] Here, ::2 means select every second item starting at the beginning. Trickier, ::-2 means select every second item starting at the end and moving backwards. What do you think 2::2 means? What about -2::-2 vs. -2:2:-2 vs. 2:2:-2? Click here to view code image a[2::2] # [‘c’, ‘e’, ‘g’] a[-2::-2] # [‘g’, ‘e’, ‘c’, ‘a’] a[-2:2:-2] # [‘g’, ‘e’] a[2:2:-2] # [] The point is that the stride part of the slicing syntax can be extremely confusing. Having three numbers within the brackets is hard enough to read because of its density. Then it’s not obvious when the start and end indexes come into effect relative to the stride value, especially when stride is negative. To prevent problems, avoid using stride along with start and end indexes. If you must use a stride, prefer making it a positive value and omit start and end indexes. If you must use stride with start or end indexes, consider using one assignment to stride and another to slice. Click here to view code image b = a[::2] # [‘a’, ‘c’, ‘e’, ‘g’] c = b[1:-1] # [‘c’, ‘e’] Slicing and then striding will create an extra shallow copy of the data. The first operation should try to reduce the size of the resulting slice by as much as possible. If your program can’t afford the time or memory required for two steps, consider using the itertools built-in module’s islice method (see Item 46: “Use Built-in Algorithms and Data Structures”), which doesn’t permit negative values for start, end, or stride. Things to Remember Specifying start, end, and stride in a slice can be extremely confusing. Prefer using positive stride values in slices without start or end indexes. Avoid negative stride values if possible. Avoid using start, end, and stride together in a single slice. If you need all three parameters, consider doing two assignments (one to slice, another to stride) or using islice from the itertools built-in module. Item 7: Use List Comprehensions Instead of map and filter Python provides compact syntax for deriving one list from another. These expressions are called list comprehensions. For example, say you want to compute the square of each number in a list. You can do this by providing the expression for your computation and the input sequence to loop over. Click here to view code image a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] squares = [x\*\*2 for x in a] print(squares) >>> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] Unless you’re applying a single-argument function, list comprehensions are clearer than the map built-in function for simple cases. map requires creating a lambda function for the computation, which is visually noisy. Click here to view code image squares = map(lambda x: x \*\* 2, a) Unlike map, list comprehensions let you easily filter items from the input list, removing corresponding outputs from the result. For example, say you only want to compute the squares of the numbers that are divisible by 2. Here, I do this by adding a conditional expression to the list comprehension after the loop: Click here to view code image even\_squares = [x\*\*2 for x in a if x % 2 == 0] print(even\_squares) >>> [4, 16, 36, 64, 100] The filter built-in function can be used along with map to achieve the same outcome, but it is much harder to read. Click here to view code image alt = map(lambda x: x\*\*2, filter(lambda x: x % 2 == 0, a)) assert even\_squares == list(alt) Dictionaries and sets have their own equivalents of list comprehensions. These make it easy to create derivative data structures when writing algorithms. Click here to view code image chile\_ranks = {‘ghost’: 1, ‘habanero’: 2, ‘cayenne’: 3} rank\_dict = {rank: name for name, rank in chile\_ranks.items()} chile\_len\_set = {len(name) for name in rank\_dict.values()} print(rank\_dict) print(chile\_len\_set) >>> {1: ‘ghost’, 2: ‘habanero’, 3: ‘cayenne’} {8, 5, 7} Things to Remember List comprehensions are clearer than the map and filter built-in functions because they don’t require extra lambda expressions. List comprehensions allow you to easily skip items from the input list, a behavior map doesn’t support without help from filter. Dictionaries and sets also support comprehension expressions. Item 8: Avoid More Than Two Expressions in List Comprehensions Beyond basic usage (see Item 7: “Use List Comprehensions Instead of map and filter”), list comprehensions also support multiple levels of looping. For example, say you want to simplify a matrix (a list containing other lists) into one flat list of all cells. Here, I do this with a list comprehension by including two for expressions. These expressions run in the order provided from left to right. Click here to view code image matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] flat = [x for row in matrix for x in row] print(flat) >>> [1, 2, 3, 4, 5, 6, 7, 8, 9] The example above is simple, readable, and a reasonable usage of multiple loops. Another reasonable usage of multiple loops is replicating the two-level deep layout of the input list. For example, say you want to square the value in each cell of a two-dimensional matrix. This expression is noisier because of the extra [] characters, but it’s still easy to read. Click here to view code image squared = [[x\*\*2 for x in row] for row in matrix] print(squared) >>> [[1, 4, 9], [16, 25, 36], [49, 64, 81]] If this expression included another loop, the list comprehension would get so long that you’d have to split it over multiple lines. Click here to view code image my\_lists = [ [[1, 2, 3], [4, 5, 6]], # … ] flat = [x for sublist1 in my\_lists for sublist2 in sublist1 for x in sublist2] At this point, the multiline comprehension isn’t much shorter than the alternative. Here, I produce the same result using normal loop statements. The indentation of this version makes the looping clearer than the list comprehension. flat = [] for sublist1 in my\_lists: for sublist2 in sublist1: flat.extend(sublist2) List comprehensions also support multiple if conditions. Multiple conditions at the same loop level are an implicit and expression. For example, say you want to filter a list of numbers to only even values greater than four. These two list comprehensions are equivalent. Click here to view code image a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] b = [x for x in a if x > 4 if x % 2 == 0] c = [x for x in a if x > 4 and x % 2 == 0] Conditions can be specified at each level of looping after the for expression. For example, say you want to filter a matrix so the only cells remaining are those divisible by 3 in rows that sum to 10 or higher. Expressing this with list comprehensions is short, but extremely difficult to read. Click here to view code image matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] filtered = [[x for x in row if x % 3 == 0] for row in matrix if sum(row) >= 10] print(filtered) >>> [[6], [9]] Though this example is a bit convoluted, in practice you’ll see situations arise where such expressions seem like a good fit. I strongly encourage you to avoid using list comprehensions that look like this. The resulting code is very difficult for others to comprehend. What you save in the number of lines doesn’t outweigh the difficulties it could cause later. The rule of thumb is to avoid using more than two expressions in a list comprehension. This could be two conditions, two loops, or one condition and one loop. As soon as it gets more complicated than that, you should use normal if and for statements and write a helper function (see Item 16: “Consider Generators Instead of Returning Lists”). Things to Remember List comprehensions support multiple levels of loops and multiple conditions per loop level. List comprehensions with more than two expressions are very difficult to read and should be avoided. Item 9: Consider Generator Expressions for Large Comprehensions The problem with list comprehensions (see Item 7: “Use List Comprehensions Instead of map and filter”) is that they may create a whole new list containing one item for each value in the input sequence. This is fine for small inputs, but for large inputs this could consume significant amounts of memory and cause your program to crash. For example, say you want to read a file and return the number of characters on each line. Doing this with a list comprehension would require holding the length of every line of the file in memory. If the file is absolutely enormous or perhaps a never-ending network socket, list comprehensions are problematic. Here, I use a list comprehension in a way that can only handle small input values. Click here to view code image value = [len(x) for x in open(‘/tmp/my\_file.txt’)] print(value) >>> [100, 57, 15, 1, 12, 75, 5, 86, 89, 11] To solve this, Python provides generator expressions, a generalization of list comprehensions and generators. Generator expressions don’t materialize the whole output sequence when they’re run. Instead, generator expressions evaluate to an iterator that yields one item at a time from the expression. A generator expression is created by putting list-comprehension-like syntax between () characters. Here, I use a generator expression that is equivalent to the code above. However, the generator expression immediately evaluates to an iterator and doesn’t make any forward progress. Click here to view code image it = (len(x) for x in open(‘/tmp/my\_file.txt’)) print(it) >>>

# [Tproger](https://tproger.ru/)

* [Новости](https://tproger.ru/category/news/)
* [Задачи с IT-собеседований](https://tproger.ru/category/problems/)
* [Эксперты отвечают читателям](https://tproger.ru/category/experts/)
* [Викторины](https://tproger.ru/category/quiz/)
* [О проекте](https://tproger.ru/about/)
* [Реклама](https://tproger.ru/ad/)





* [Алгоритмы](https://tproger.ru/tag/algorithms/)
* [Для новичков](https://tproger.ru/tag/for-beginners/)
* [Gamedev](https://tproger.ru/tag/gamedev/)
* [Mobile](https://tproger.ru/tag/mobiledev/)
* [Web](https://tproger.ru/tag/web/)
* [Linux](https://tproger.ru/tag/linux/)
* [Android](https://tproger.ru/tag/android/)
* [С++](https://tproger.ru/tag/cpp/)
* [Язык Си](https://tproger.ru/tag/c-language/)
* [C#](https://tproger.ru/tag/c-sharp/)
* [Java](https://tproger.ru/tag/java/)
* [JS](https://tproger.ru/tag/javascript/)
* [Python](https://tproger.ru/tag/python/)

# На замену гуглу: сборник полезных для программиста ссылок

* , [Подборки](https://tproger.ru/category/digest/)
* 1 минута
* 13 945



Tproger подготовил для вас сборник полезных ссылок по наиболее распространенным технологиям и языкам программирования — чтобы не гуглить каждый раз.

### Python

[Объяснение глобальной и локальной областей видимости](https://automatetheboringstuff.com/chapter3/)

[Регулярные выражения Python для начинающих](https://developers.google.com/edu/python/regular-expressions)

[Делаем JSON REST API на Python 3 и Flask](http://techarena51.com/index.php/buidling-a-database-driven-restful-json-api-in-python-3-with-flask-flask-restful-and-sqlalchemy/)

[Кросс-платформенная библиотека для получения системной информации](https://pypi.python.org/pypi/psutil)

[Автоматизируем тестирование с помощью Selenium](http://techarena51.com/index.php/install-selenium-linux-automate-web-tests/)

[Потоки и глобальная блокировка интерпретатора Python](http://jessenoller.com/blog/2009/02/01/python-threads-and-the-global-interpreter-lock)

### C

[Файловый ввод/вывод](http://gribblelab.org/CBootcamp/10_Input_and_Output.html)

[Небольшой учебник по С — наиболее важные темы в сжатом виде](http://gribblelab.org/CBootcamp/)

[Книга «Сетевое программирование с Биджем», в электронном виде распространяется бесплатно, в том числе на русском языке](https://beej.us/guide/bgnet/)

### Perl

Шпаргалки по регулярным выражениям в Perl: [первая](http://www.erudil.com/preqr.pdf), [вторая](http://www.rexegg.com/regex-quickstart.html)

[Обширный туториал, охватывающий все основы языка](http://perlmaven.com/perl-tutorial)

### Git

[Простое введение в Git, выполненное в виде удобной шпаргалки, есть русская версия](https://rogerdudler.github.io/git-guide/)

[Объемный урок, охватывает множество команд и схему работы Git](http://www.vogella.com/tutorials/Git/article.html" \l "gitdefintion_tools1)

### Vim

[Шпаргалка по действиям, назначенным на каждую кнопку клавиатуры](http://michael.peopleofhonoronly.com/vim)

### Bash

[Шпаргалка с полезными однострочными командами Bash](https://github.com/stephenturner/oneliners)

Пособие по написанию Bash-скриптов для новичков: [часть 1](http://techarena51.com/index.php/a-beginners-guide-to-bash-scripting/), [часть 2](http://techarena51.com/index.php/bash-scripting-tutorial-part-2/)

[Список используемых в Bash специальных символов](http://mywiki.wooledge.org/BashGuide/SpecialCharacters)

[Большое пособие по программированию на Bash](http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html)

### Редакторы кода

[Vim](http://www.vim.org/about.php)

[Atom](https://atom.io/docs/v0.196.0/getting-started-why-atom)

[Brackets](http://brackets.io/)

[Sublime Text](https://www.sublimetext.com/)

[GNU Emacs](https://www.gnu.org/software/emacs/)

[Notepad++](https://notepad-plus-plus.org/)

### Разное

[Системные вызовы Linux](http://www.digilife.be/quickreferences/qrc/linux system call quick reference.pdf)

[Добиваемся загрузки Linux в 1 секунду, презентация](https://events.linuxfoundation.org/sites/events/files/slides/praesentation.pdf)

[Установка VPN сервера на Linux](http://techarena51.com/index.php/how-to-install-an-opensource-vpn-server-on-linux/)

[Список вопросов, которые могут задать на собеседовании разработчику](https://github.com/Leo-G/DevopsWiki/wiki/Devops-Interview-Questions)

[Кроссплатформенная система разработки десктопных приложений](http://electron.atom.io/)

[Сайт с видеоуроками по системному администрированию](https://sysadmincasts.com/)

[Структуры данных: стек или куча?](http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html)

Полный список доступен на [GitHub](https://github.com/Leo-G/DevopsWiki)

* [Инструменты](https://tproger.ru/tag/tools/)

#### Другим программистам нравится:

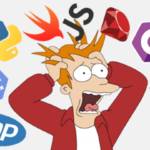
Пишу новую прогу. Пока компилирую старую.



### [Ответы экспертов](https://tproger.ru/category/experts/)

* 

##### [Какой смысл имеют олимпиады по программированию? Не поздно ли начать участвовать в них 11-класснику? — Отвечают эксперты Tproger](https://tproger.ru/experts/23/)

* 

##### [Какие языки программирования сейчас наиболее перспективны для изучения? — Отвечают эксперты Tproger](https://tproger.ru/experts/best-prog-lang-for-novice/)

* 

##### [Заняться теорией и олимпиадами или пойти на работу в IT-компанию? — Эксперты дают советы начинающим программистам](https://tproger.ru/experts/20/)

* 

##### [Как лучше действовать, если вы хотите научиться программировать, но не знаете, как встать на истинный путь — готовые инструкции для начинающих от экспертов Tproger](https://tproger.ru/experts/19/)

* 

##### [С какой платформы лучше начинать мобильную разработку? Обязательно ли сразу выпускаться под все платформы?](https://tproger.ru/experts/18/)

### Рубрики

[/dev/null](https://tproger.ru/category/devnull/)[Видео](https://tproger.ru/category/video/)[Викторины](https://tproger.ru/category/quiz/)[Еженедельная подборка](https://tproger.ru/category/weekly/)[Задачки](https://tproger.ru/category/problems/)[Инструменты](https://tproger.ru/category/tools/)[Интервью](https://tproger.ru/category/interview/)[Интересные ссылки](https://tproger.ru/category/links/)[Ищем программиста](https://tproger.ru/category/looking-for-programmer/)[Книги](https://tproger.ru/category/books/)[Новости](https://tproger.ru/category/news/)[Ответы экспертов](https://tproger.ru/category/experts/)[Переводы](https://tproger.ru/category/translations/)[Подборки](https://tproger.ru/category/digest/)[Рассказы о своих проектах](https://tproger.ru/category/projects/)[События](https://tproger.ru/category/events/)[Спонсорское](https://tproger.ru/category/sponsored/)[Статьи](https://tproger.ru/category/articles/)

#### Уведомления о свежих постах (Chrome, Firefox, Android, Telegram).

[Подписаться](https://pushall.ru/tproger)

### Темы

[Android](https://tproger.ru/tag/android/)[C#](https://tproger.ru/tag/c-sharp/)[C++](https://tproger.ru/tag/cpp/)[Git](https://tproger.ru/tag/git/)[Google](https://tproger.ru/tag/google/)[Hardware](https://tproger.ru/tag/hardware/)[Java](https://tproger.ru/tag/java/)[JavaScript](https://tproger.ru/tag/javascript/)[Linux](https://tproger.ru/tag/linux/)[Microsoft](https://tproger.ru/tag/microsoft/)[Open Source](https://tproger.ru/tag/open-source/)[PHP](https://tproger.ru/tag/php/)[Python](https://tproger.ru/tag/python/)[StackOverflow](https://tproger.ru/tag/stackoverflow/)[Unity](https://tproger.ru/tag/unity/)[Windows](https://tproger.ru/tag/windows/)[Алгоритмы](https://tproger.ru/tag/algorithms/)[Безопасность](https://tproger.ru/tag/security/)[Безопасный код](https://tproger.ru/tag/safe-code/)[Браузеры](https://tproger.ru/tag/browsers/)[Веб-разработка](https://tproger.ru/tag/web/)[Виртуальная реальность](https://tproger.ru/tag/virtual-reality/)[Головоломки](https://tproger.ru/tag/brain-teasers/)[Для мотивации](https://tproger.ru/tag/for-motivation/)[Для начинающих](https://tproger.ru/tag/for-beginners/)[Задачи повышенной сложности](https://tproger.ru/tag/expert-problems/)[Задачи умеренной сложности](https://tproger.ru/tag/middle-problems/)[Инструменты](https://tproger.ru/tag/tools/)[Интернет](https://tproger.ru/tag/internet/)[Искусственный интеллект](https://tproger.ru/tag/ai/)[История успеха](https://tproger.ru/tag/success-story/)[Лучшая практика](https://tproger.ru/tag/best-practice/)[Машинное обучение](https://tproger.ru/tag/machine-learning/)[Мобильная разработка](https://tproger.ru/tag/mobiledev/)[Нейронные сети](https://tproger.ru/tag/neural-network/)[Низкоуровневое программирование](https://tproger.ru/tag/low-level-coding/)[Обучение программированию](https://tproger.ru/tag/learn-programming/)[Оптимизация](https://tproger.ru/tag/optimization/)[Разработка игр](https://tproger.ru/tag/gamedev/)[Рекомендуем](https://tproger.ru/tag/featured/)[Ретро](https://tproger.ru/tag/retro/)[Собеседование](https://tproger.ru/tag/job-interview/)[Советы](https://tproger.ru/tag/advices/)[Язык Си](https://tproger.ru/tag/c-language/)[Языки программирования](https://tproger.ru/tag/programming-languages/)

[О проекте](https://tproger.ru/about)[Реклама](https://tproger.ru/ad)[admin@tproger.ru](mailto:admin@tproger.ru)

Нашли опечатку? Выделите фрагмент и отправьте нажатием Ctrl+Enter.

Спасибо за внимательность.  
Опечатка уже отправлена нашим редакторам.

# [Tproger](https://tproger.ru/)

* [Новости](https://tproger.ru/category/news/)
* [Задачи с IT-собеседований](https://tproger.ru/category/problems/)
* [Эксперты отвечают читателям](https://tproger.ru/category/experts/)
* [Викторины](https://tproger.ru/category/quiz/)
* [О проекте](https://tproger.ru/about/)
* [Реклама](https://tproger.ru/ad/)





* [Алгоритмы](https://tproger.ru/tag/algorithms/)
* [Для новичков](https://tproger.ru/tag/for-beginners/)
* [Gamedev](https://tproger.ru/tag/gamedev/)
* [Mobile](https://tproger.ru/tag/mobiledev/)
* [Web](https://tproger.ru/tag/web/)
* [Linux](https://tproger.ru/tag/linux/)
* [Android](https://tproger.ru/tag/android/)
* [С++](https://tproger.ru/tag/cpp/)
* [Язык Си](https://tproger.ru/tag/c-language/)
* [C#](https://tproger.ru/tag/c-sharp/)
* [Java](https://tproger.ru/tag/java/)
* [JS](https://tproger.ru/tag/javascript/)
* [Python](https://tproger.ru/tag/python/)

# На замену гуглу: сборник полезных для программиста ссылок

* , [Подборки](https://tproger.ru/category/digest/)
* 1 минута
* 13 945



Tproger подготовил для вас сборник полезных ссылок по наиболее распространенным технологиям и языкам программирования — чтобы не гуглить каждый раз.

### Python

[Объяснение глобальной и локальной областей видимости](https://automatetheboringstuff.com/chapter3/)

[Регулярные выражения Python для начинающих](https://developers.google.com/edu/python/regular-expressions)

[Делаем JSON REST API на Python 3 и Flask](http://techarena51.com/index.php/buidling-a-database-driven-restful-json-api-in-python-3-with-flask-flask-restful-and-sqlalchemy/)

[Кросс-платформенная библиотека для получения системной информации](https://pypi.python.org/pypi/psutil)

[Автоматизируем тестирование с помощью Selenium](http://techarena51.com/index.php/install-selenium-linux-automate-web-tests/)

[Потоки и глобальная блокировка интерпретатора Python](http://jessenoller.com/blog/2009/02/01/python-threads-and-the-global-interpreter-lock)

### C

[Файловый ввод/вывод](http://gribblelab.org/CBootcamp/10_Input_and_Output.html)

[Небольшой учебник по С — наиболее важные темы в сжатом виде](http://gribblelab.org/CBootcamp/)

[Книга «Сетевое программирование с Биджем», в электронном виде распространяется бесплатно, в том числе на русском языке](https://beej.us/guide/bgnet/)

### Perl

Шпаргалки по регулярным выражениям в Perl: [первая](http://www.erudil.com/preqr.pdf), [вторая](http://www.rexegg.com/regex-quickstart.html)

[Обширный туториал, охватывающий все основы языка](http://perlmaven.com/perl-tutorial)

### Git

[Простое введение в Git, выполненное в виде удобной шпаргалки, есть русская версия](https://rogerdudler.github.io/git-guide/)

[Объемный урок, охватывает множество команд и схему работы Git](http://www.vogella.com/tutorials/Git/article.html" \l "gitdefintion_tools1)

### Vim

[Шпаргалка по действиям, назначенным на каждую кнопку клавиатуры](http://michael.peopleofhonoronly.com/vim)

### Bash

[Шпаргалка с полезными однострочными командами Bash](https://github.com/stephenturner/oneliners)

Пособие по написанию Bash-скриптов для новичков: [часть 1](http://techarena51.com/index.php/a-beginners-guide-to-bash-scripting/), [часть 2](http://techarena51.com/index.php/bash-scripting-tutorial-part-2/)

[Список используемых в Bash специальных символов](http://mywiki.wooledge.org/BashGuide/SpecialCharacters)

[Большое пособие по программированию на Bash](http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html)

### Редакторы кода

[Vim](http://www.vim.org/about.php)

[Atom](https://atom.io/docs/v0.196.0/getting-started-why-atom)

[Brackets](http://brackets.io/)

[Sublime Text](https://www.sublimetext.com/)

[GNU Emacs](https://www.gnu.org/software/emacs/)

[Notepad++](https://notepad-plus-plus.org/)

### Разное

[Системные вызовы Linux](http://www.digilife.be/quickreferences/qrc/linux system call quick reference.pdf)

[Добиваемся загрузки Linux в 1 секунду, презентация](https://events.linuxfoundation.org/sites/events/files/slides/praesentation.pdf)

[Установка VPN сервера на Linux](http://techarena51.com/index.php/how-to-install-an-opensource-vpn-server-on-linux/)

[Список вопросов, которые могут задать на собеседовании разработчику](https://github.com/Leo-G/DevopsWiki/wiki/Devops-Interview-Questions)

[Кроссплатформенная система разработки десктопных приложений](http://electron.atom.io/)

[Сайт с видеоуроками по системному администрированию](https://sysadmincasts.com/)

[Структуры данных: стек или куча?](http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html)

Полный список доступен на [GitHub](https://github.com/Leo-G/DevopsWiki)

* [Инструменты](https://tproger.ru/tag/tools/)

#### Другим программистам нравится:

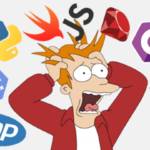
Пишу новую прогу. Пока компилирую старую.



### [Ответы экспертов](https://tproger.ru/category/experts/)

* 

##### [Какой смысл имеют олимпиады по программированию? Не поздно ли начать участвовать в них 11-класснику? — Отвечают эксперты Tproger](https://tproger.ru/experts/23/)

* 

##### [Какие языки программирования сейчас наиболее перспективны для изучения? — Отвечают эксперты Tproger](https://tproger.ru/experts/best-prog-lang-for-novice/)

* 

##### [Заняться теорией и олимпиадами или пойти на работу в IT-компанию? — Эксперты дают советы начинающим программистам](https://tproger.ru/experts/20/)

* 

##### [Как лучше действовать, если вы хотите научиться программировать, но не знаете, как встать на истинный путь — готовые инструкции для начинающих от экспертов Tproger](https://tproger.ru/experts/19/)

* 

##### [С какой платформы лучше начинать мобильную разработку? Обязательно ли сразу выпускаться под все платформы?](https://tproger.ru/experts/18/)

### Рубрики

[/dev/null](https://tproger.ru/category/devnull/)[Видео](https://tproger.ru/category/video/)[Викторины](https://tproger.ru/category/quiz/)[Еженедельная подборка](https://tproger.ru/category/weekly/)[Задачки](https://tproger.ru/category/problems/)[Инструменты](https://tproger.ru/category/tools/)[Интервью](https://tproger.ru/category/interview/)[Интересные ссылки](https://tproger.ru/category/links/)[Ищем программиста](https://tproger.ru/category/looking-for-programmer/)[Книги](https://tproger.ru/category/books/)[Новости](https://tproger.ru/category/news/)[Ответы экспертов](https://tproger.ru/category/experts/)[Переводы](https://tproger.ru/category/translations/)[Подборки](https://tproger.ru/category/digest/)[Рассказы о своих проектах](https://tproger.ru/category/projects/)[События](https://tproger.ru/category/events/)[Спонсорское](https://tproger.ru/category/sponsored/)[Статьи](https://tproger.ru/category/articles/)

#### Уведомления о свежих постах (Chrome, Firefox, Android, Telegram).

[Подписаться](https://pushall.ru/tproger)

### Темы

[Android](https://tproger.ru/tag/android/)[C#](https://tproger.ru/tag/c-sharp/)[C++](https://tproger.ru/tag/cpp/)[Git](https://tproger.ru/tag/git/)[Google](https://tproger.ru/tag/google/)[Hardware](https://tproger.ru/tag/hardware/)[Java](https://tproger.ru/tag/java/)[JavaScript](https://tproger.ru/tag/javascript/)[Linux](https://tproger.ru/tag/linux/)[Microsoft](https://tproger.ru/tag/microsoft/)[Open Source](https://tproger.ru/tag/open-source/)[PHP](https://tproger.ru/tag/php/)[Python](https://tproger.ru/tag/python/)[StackOverflow](https://tproger.ru/tag/stackoverflow/)[Unity](https://tproger.ru/tag/unity/)[Windows](https://tproger.ru/tag/windows/)[Алгоритмы](https://tproger.ru/tag/algorithms/)[Безопасность](https://tproger.ru/tag/security/)[Безопасный код](https://tproger.ru/tag/safe-code/)[Браузеры](https://tproger.ru/tag/browsers/)[Веб-разработка](https://tproger.ru/tag/web/)[Виртуальная реальность](https://tproger.ru/tag/virtual-reality/)[Головоломки](https://tproger.ru/tag/brain-teasers/)[Для мотивации](https://tproger.ru/tag/for-motivation/)[Для начинающих](https://tproger.ru/tag/for-beginners/)[Задачи повышенной сложности](https://tproger.ru/tag/expert-problems/)[Задачи умеренной сложности](https://tproger.ru/tag/middle-problems/)[Инструменты](https://tproger.ru/tag/tools/)[Интернет](https://tproger.ru/tag/internet/)[Искусственный интеллект](https://tproger.ru/tag/ai/)[История успеха](https://tproger.ru/tag/success-story/)[Лучшая практика](https://tproger.ru/tag/best-practice/)[Машинное обучение](https://tproger.ru/tag/machine-learning/)[Мобильная разработка](https://tproger.ru/tag/mobiledev/)[Нейронные сети](https://tproger.ru/tag/neural-network/)[Низкоуровневое программирование](https://tproger.ru/tag/low-level-coding/)[Обучение программированию](https://tproger.ru/tag/learn-programming/)[Оптимизация](https://tproger.ru/tag/optimization/)[Разработка игр](https://tproger.ru/tag/gamedev/)[Рекомендуем](https://tproger.ru/tag/featured/)[Ретро](https://tproger.ru/tag/retro/)[Собеседование](https://tproger.ru/tag/job-interview/)[Советы](https://tproger.ru/tag/advices/)[Язык Си](https://tproger.ru/tag/c-language/)[Языки программирования](https://tproger.ru/tag/programming-languages/)

[О проекте](https://tproger.ru/about)[Реклама](https://tproger.ru/ad)[admin@tproger.ru](mailto:admin@tproger.ru)

Нашли опечатку? Выделите фрагмент и отправьте нажатием Ctrl+Enter.

Спасибо за внимательность.  
Опечатка уже отправлена нашим редакторам.