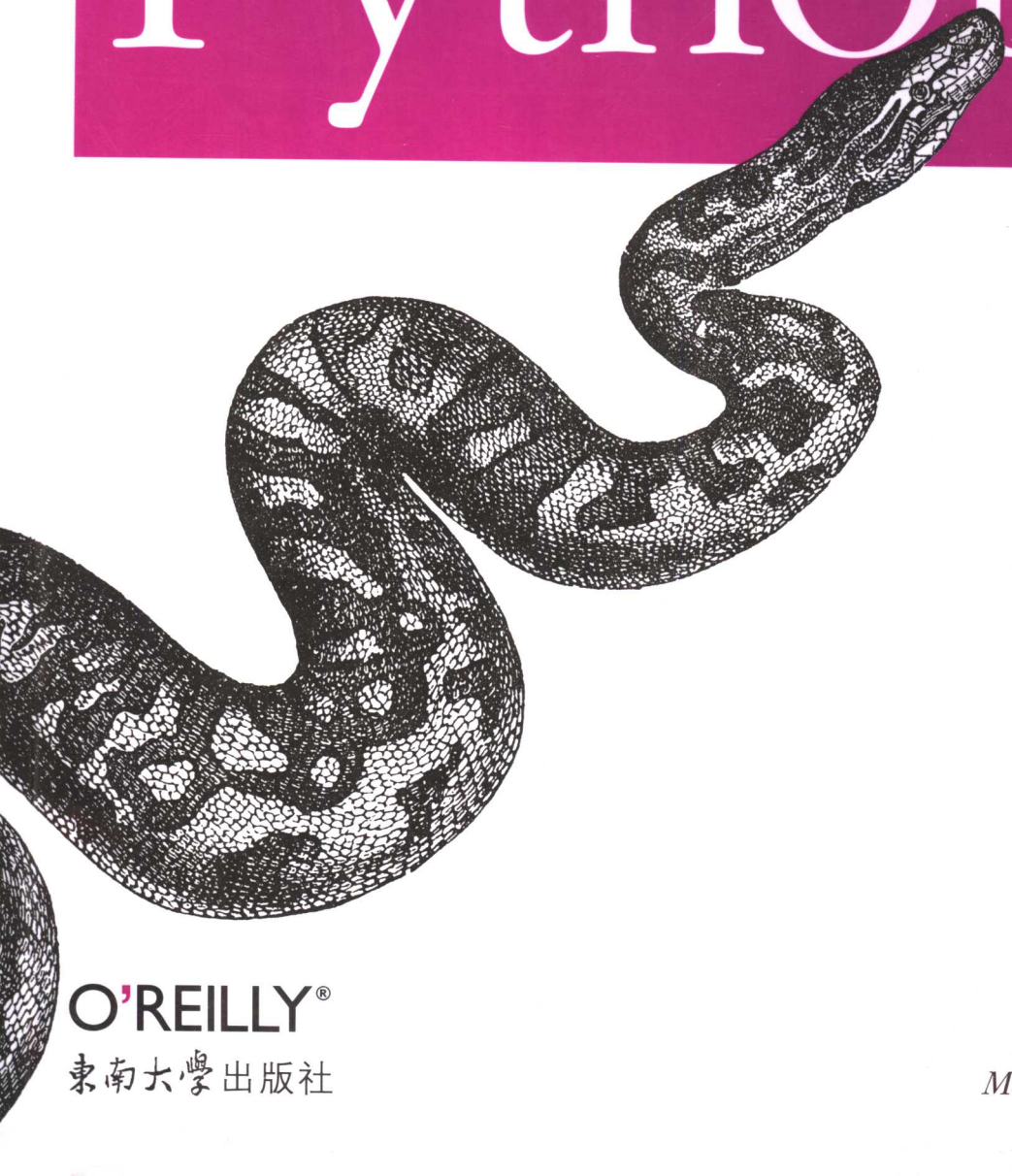
Python 编程（影印版）

3rd Edition
上册

Programming

Python



O'REILLY®
東南大學出版社

Mark Lutz 著

第三版

Python 编程(影印版)

Programming Python

上册

Mark Lutz

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

Python编程：第3版 /（美）卢茨（Lutz, M.）著.—影印本.—南京：东南大学出版社，2006.11

书名原文：Programming Python, Third Edition

ISBN 7-5641-0570-4

Ⅰ.P... Ⅱ.卢 Ⅲ.软件工具－程序设计－英文 Ⅳ.TP311.56

中国版本图书馆CIP数据核字（2006）第115484号

江苏省版权局著作权合同登记

图字：10-2006-257号

书　　名/ Python编程 第三版（影印版）

书　　号/ ISBN 7-5641-0570-4

责任编辑/ 张烨

封面设计/ Edie Freedman，张健

出版发行/ 东南大学出版社（press.seu.edu.cn）

地　　址/ 南京四牌楼2号（邮政编码210096）

印　　刷/ 扬中市印刷有限公司

开　　本/ 787毫米×980毫米　16开本　100.75印张

版　　次/ 2006年11月第1版　2006年11月第1次印刷

印　　数/ 0001-2500册

全套定价/ 138.00元（上下册）

O'Reilly Media, Inc.介绍

O'Reilly Media, Inc.是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。

出版说明

随着计算机技术的成熟和广泛应用，人类正在步入一个技术迅猛发展的新时期。计算机技术的发展给人们的工业生产、商业活动和日常生活都带来了巨大的影响。然而，计算机领域的技术更新速度之快也是众所周知的，为了帮助国内技术人员在第一时间了解国外最新的技术，东南大学出版社和美国 O'Reilly Meida, Inc.达成协议，将陆续引进该公司的代表前沿技术或者在某专项领域享有盛名的著作，以影印版或者简体中文版的形式呈献给读者。其中，影印版书籍力求与国外图书“同步”出版，并且“原汁原味”展现给读者。

我们真诚地希望，所引进的书籍能对国内相关行业的技术人员、科研机构的研究人员和高校师生的学习和工作有所帮助，对国内计算机技术的发展有所促进。也衷心期望读者提出宝贵的意见和建议。

最新出版的一批影印版图书，包括：

- 《深入浅出 Ajax》（影印版）
- 《Ajax Hacks》（影印版）
- 《深入理解 Linux 网络内幕》（影印版）
- 《Web 设计技术手册 第三版》（影印版）
- 《软件预构艺术》（影印版）
- 《Ruby on Rails: Up and Running》（影印版）
- 《Ruby Cookbook》（影印版）
- 《Python 编程 第三版》（影印版）
- 《Python 技术手册 第二版》（影印版）
- 《Ajax 设计模式》（影印版）
- 《实用软件项目管理》（影印版）
- 《用户界面设计模式》（影印版）

Foreword

How Time Flies!

Ten years ago I completed the foreword for the first edition of this book. Python 1.3 was current then, and 1.4 was in beta. I wrote about Python's origins and philosophy, and about how its first six years changed my life. Python was still mostly a one-man show at the time, and I only mentioned other contributors and the Python community in one paragraph near the end.

Five years later the second edition came out, much improved and quite a bit heftier, and I wrote a new foreword. Python 2.0 was hot then, and the main topic of the foreword was evolution of the language. Python 2.0 added a lot of new features, and many were concerned that the pace of change would be unsustainable for the users of the language. I addressed this by promising feature-by-feature backward compatibility for several releases and by regulating change through a community process using Python Enhancement Proposals (PEPs).

By then, Python's development had become truly community-driven, with many developers (besides myself) having commit privileges into the source tree. This move toward community responsibility has continued ever since. My own role has become more limited over time, though have not yet been reduced to playing a purely ceremonial function like that of the Dutch Queen.

Perhaps the biggest change in recent years is the establishment of the Python Software Foundation (PSF), a non-profit organization that formally owns and manages the rights to the Python source code and owns the Python trademark. Its board and members (helped by many nonmember volunteers) also offer many services to the Python community, from the Python.org web site and mailing lists to the yearly Python Conference. Membership in the PSF is by invitation only, but donations are always welcome (and tax-deductible, at least in the U.S.).

The PSF does not directly control Python's development; however, the developers don't have to obey any rules set by the PSF. Rather, it's the other way around: active

Python developers make up the majority of the PSF's membership. This arrangement, together with the open source nature of Python's source code license, ensures that Python will continue to serve the goals of its users and developers.

Coming Attractions

What developments can Python users expect to see in the coming years? Python 3000, which is referred to in the foreword to the second edition as "intentionally vaporware," will see the light of day after all as Python 3.0. After half a decade of talk, it's finally time to start doing something about it. I've created a branch of the 2.5 source tree, and, along with a handful of developers, I'm working on transforming the code base into my vision for Python 3000. At the same time, I'm working with the community on a detailed definition of Python 3000; there's a new mailing dedicated to Python 3000 and a series of PEPs, starting with PEP 3000.

This work is still in the early stages. Some changes, such as removing classic classes and string exceptions, adopting Unicode as the only character type, and changing integer division so that 1/2 returns 0.5 instead of truncating toward zero, have been planned for years. But many other changes are still being hotly debated, and new features are being proposed almost daily.

I see my own role in this debate as a force of moderation: there are many more good ideas than could possibly be implemented in the given time, and, taken together, they would change the language so much that it would be unrecognizable. My goal for Python 3000 is to fix some of my oldest design mistakes, especially the ones that can't be fixed without breaking backward compatibility. That alone will be a huge task. For example, a consequence of the choice to use Unicode everywhere is the need for a total rewrite of the standard I/O library and a new data type to represent binary ("noncharacter") data, dubbed "bytes."

The biggest potential danger for Python 3000 is that of an "accidental paradigm shift": a change, or perhaps a small set of changes that weren't considered together, that would unintentionally cause a huge change to the way people program in Python. For example, adding optional static type checking to the language could easily have the effect of turning Python into "Java without braces"—which is definitely not what most users would like to see happen! For this reason, I am making it my personal responsibility to guide the Python 3000 development process. The new language should continue to represent my own esthetics for language design, not a design-by-committee compromise or a radical departure from today's Python. And if we don't get everything right, well, there's always Python 4000....

The timeline for 3.0 is roughly as follows: I expect the first alpha release in about a year and the first production release a year later. I expect that it will then take another year to shake out various usability issues and get major third-party packages ported, and, finally, another year to gain widespread user acceptance. So, Mark

should have about three to four years before he'll have to start the next revision of this book.

To learn more about Python 3000 and how we plan to help users convert their code, start by reading PEP 3000. (To find PEP 3000 online, search for it in Google.)

In the meantime, Python 2.x is not dead yet. Python 2.5 will be released around the same time as this book (it's in late alpha as I am writing this). Python's normal release cycle produces a new release every 12–18 months. I fully expect version 2.6 to see the light of day while Python 3000 is still in alpha, and it's likely that 2.7 will be released around the same time as 3.0 (and that more users will download 2.7 than 3.0). A 2.8 release is quite likely; such a release might back-port certain Python 3.0 features (while maintaining backward compatibility with 2.7) in order to help users migrate code. A 2.9 release might happen, depending on demand. But in any case, 2.10 will be right out!

(If you're not familiar with Python's release culture, releases like 2.4 and 2.5 are referred to as "major releases." There are also "bug-fix releases," such as 2.4.3. Bug-fix releases are just that: they fix bugs and, otherwise, maintain strict backward and forward compatibility within the same major release. Major releases introduce new features and maintain backward compatibility with at least one or two previous major releases, and, in most cases, many more than that. There's no specific name for "earth-shattering" releases like 3.0, since they happen so rarely.)

Concluding Remarks

Programming Python was the first or second book on Python ever published, and it's the only one of the early batch to endure to this day. I thank its author, Mark Lutz, for his unceasing efforts in keeping the book up-to-date, and its publisher, O'Reilly, for keeping the page count constant for this edition.

Some of my fondest memories are of the book's first editor, the late Frank Willison. Without Frank's inspiration and support, the first two editions would never have been. He would be proud of this third edition.

I must end in a fine tradition, with one of my favorite Monty Python quotes: "Take it away, Eric the orchestra leader!"

—Guido van Rossum
Belmont, California, May 2006

Foreword to the Second Edition (2001)

Less than five years ago, I wrote the Foreword for the first edition of Programming Python. Since then, the book has changed about as much as the language and the Python community! I no longer feel the need to defend Python: the statistics and developments listed in Mark's Preface speak for themselves.

In the past year, Python has made great strides. We released Python 2.0, a big step forward, with new standard library features such as Unicode and XML support, and several new syntactic constructs, including augmented assignment: you can now write `x += 1` instead of `x = x+1`. A few people wondered what the big deal was (answer: instead of `x`, imagine `dict[key]` or `list[index]`), but overall this was a big hit with those users who were already used to augmented assignment in other languages.

Less warm was the welcome for the extended print statement, `print>>file`, a shortcut for printing to a different file object than standard output. Personally, it's the Python 2.0 feature I use most frequently, but most people who opened their mouths about it found it an abomination. The discussion thread on the newsgroup berating this simple language extension was one of the longest ever—apart from the never-ending Python versus Perl thread.

Which brings me to the next topic. (No, not Python versus Perl. There are better places to pick a fight than a Foreword.) I mean the speed of Python's evolution, a topic dear to the heart of the author of this book. Every time I add a feature to Python, another patch of Mark's hair turns gray—there goes another chapter out of date! Especially the slew of new features added to Python 2.0, which appeared just as he was working on this second edition, made him worry: what if Python 2.1 added as many new things? The book would be out of date as soon as it was published!

Relax, Mark. Python will continue to evolve, but I promise that I won't remove things that are in active use! For example, there was a lot of worry about the string module. Now that string objects have methods, the string module is mostly redundant. I wish I could declare it obsolete (or deprecated) to encourage Python programmers to start using string methods instead. But given that a large majority of existing Python code—even many standard library modules—imports the string module, this change is obviously not going to happen overnight. The first likely opportunity to remove the string module will be when we introduce Python 3000; and even at that point, there will probably be a string module in the backwards compatibility library for use with old code.

Python 3000?! Yes, that's the nickname for the next generation of the Python interpreter. The name may be considered a pun on Windows 2000, or a reference to Mystery Science Theater 3000, a suitably Pythonesque TV show with a cult following. When will Python 3000 be released? Not for a looooooong time—although you won't quite have to wait until the year 3000.

Originally, Python 3000 was intended to be a complete rewrite and redesign of the language. It would allow me to make incompatible changes in order to fix problems with the language design that weren't solvable in a backwards compatible way. The current plan, however, is that the necessary changes will be introduced gradually into the current Python 2.x line of development, with a clear transition path that includes a period of backwards compatibility support.

Take, for example, integer division. In line with C, Python currently defines x/y with two integer arguments to have an integer result. In other words, 1/2 yields 0! While most dyed-in-the-wool programmers expect this, it's a continuing source of confusion for newbies, who make up an ever-larger fraction of the (exponentially growing) Python user population. From a numerical perspective, it really makes more sense for the / operator to yield the same value regardless of the type of the operands: after all, that's what all other numeric operators do. But we can't simply change Python so that 1/2 yields 0.5, because (like removing the string module) it would break too much existing code. What to do?

The solution, too complex to describe here in detail, will have to span several Python releases, and involves gradually increasing pressure on Python programmers (first through documentation, then through deprecation warnings, and eventually through errors) to change their code. By the way, a framework for issuing warnings will be introduced as part of Python 2.1. Sorry, Mark!

So don't expect the announcement of the release of Python 3000 any time soon. Instead, one day you may find that you are *already* using Python 3000—only it won't be called that, but rather something like Python 2.8.7. And most of what you've learned in this book will still apply! Still, in the meantime, references to Python 3000 will abound; just know that this is intentionally vaporware in the purest sense of the word. Rather than worry about Python 3000, continue to use and learn more about the Python version that you do have.

I'd like to say a few words about Python's current development model. Until early 2000, there were hundreds of contributors to Python, but essentially all contributions had to go through my inbox. To propose a change to Python, you would mail me a context diff, which I would apply to my work version of Python, and if I liked it, I would check it into my CVS source tree. (CVS is a source code version management system, and the subject of several books.) Bug reports followed the same path, except I also ended up having to come up with the patch. Clearly, with the increasing number of contributions, my inbox became a bottleneck. What to do?

Fortunately, Python wasn't the only open source project with this problem, and a few smart people at VA Linux came up with a solution: SourceForge! This is a dynamic web site with a complete set of distributed project management tools available: a public CVS repository, mailing lists (using Mailman, a very popular Python application!), discussion forums, bug and patch managers, and a download area, all made available to any open source project for the asking.

We currently have a development group of 30 volunteers with SourceForge checkin privileges, and a development mailing list comprising twice as many folks. The privileged volunteers have all sworn their allegiance to the BDFL (Benevolent Dictator For Life—that's me :-). Introduction of major new features is regulated via a lightweight system of proposals and feedback called Python Enhancement Proposals (PEPs). Our PEP system proved so successful that it was copied almost verbatim by the Tcl community when they made a similar transition from Cathedral to Bazaar.

So, it is with confidence in Python's future that I give the floor to Mark Lutz. Excellent job, Mark. And to finish with my favorite Monty Python quote: Take it away, Eric, the orchestra leader!

—Guido van Rossum
Reston, Virginia, January 2001

Foreword from the First Edition (1996)

As Python's creator, I'd like to say a few words about its origins, adding a bit of personal philosophy.

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (a government-run research lab in Amsterdam) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to UNIX/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of *Monty Python's Flying Circus*).

Today, I can safely say that Python has changed my life. I have moved to a different continent. I spend my working days developing large systems in Python, when I'm not hacking on Python or answering Python-related email. There are Python T-shirts, workshops, mailing lists, a newsgroup, and now a book. Frankly, my only unfulfilled wish right now is to have my picture on the front page of the *New York Times*. But before I get carried away daydreaming, here are a few tidbits from Python's past.

It all started with ABC, a wonderful teaching language that I had helped create in the early eighties. It was an incredibly elegant and powerful language aimed at nonprofessional programmers. Despite all its elegance and power and the availability of a free implementation, ABC never became popular in the UNIX/C world. I can only speculate about the reasons, but here's a likely one: the difficulty of adding new "primitive" operations to ABC. It was a monolithic closed system, with only the most basic I/O operations: read a string from the console, write a string to the console. I decided not to repeat this mistake in Python.

Besides this intention, I had a number of other ideas for a language that improved upon ABC, and was eager to try them out. For instance, ABC's powerful data types turned out to be less efficient than we hoped. There was too much emphasis on theoretically optimal algorithms, and not enough tuning for common cases. I also felt that some of ABC's features, aimed at novice programmers, were less desirable for the (then!) intended audience of experienced UNIX/C programmers. For instance: ABC's idiosyncratic syntax (all uppercase keywords!), some terminology (for example, "how-to" instead of "procedure"); and the integrated structured editor, which its users almost universally hated. Python would rely more on the UNIX infrastructure and conventions, without being UNIX-bound. And in fact, the first implementation was done on a Macintosh.

As it turned out, Python is remarkably free from many of the hang-ups of conventional programming languages. This is perhaps due to my choice of examples: besides ABC, my main influence was Modula-3. This is another language with remarkable elegance and power, designed by a small, strong-willed team (most of whom I had met during a summer internship at DEC's Systems Research Center in Palo Alto). Imagine what Python would have looked like if I had modeled it after the UNIX shell and C instead! (Yes, I borrowed from C too, but only its least controversial features, in my desire to please the UNIX/C audience.)

Any individual creation has its idiosyncracies, and occasionally its creator has to justify them. Perhaps Python's most controversial feature is its use of indentation for statement grouping, which derives directly from ABC. It is one of the language's features that is dearest to my heart. It makes Python code more readable in two ways. First, the use of indentation reduces visual clutter and makes programs shorter, thus reducing the attention span needed to take in a basic unit of code. Second, it allows the programmer less freedom in formatting, thereby enabling a more uniform style, which makes it easier to read someone else's code. (Compare, for instance, the three or four different conventions for the placement of braces in C, each with strong proponents.)

This emphasis on readability is no accident. As an object-oriented language, Python aims to encourage the creation of reusable code. Even if we all wrote perfect documentation all of the time, code can hardly be considered reusable if it's not readable. Many of Python's features, in addition to its use of indentation, conspire to make Python code highly readable. This reflects the philosophy of ABC, which was intended to teach programming in its purest form, and therefore placed a high value on clarity.

Readability is often enhanced by reducing unnecessary variability. When possible, there's a single, obvious way to code a particular construct. This reduces the number of choices facing the programmer who is writing the code, and increases the chance that it will appear familiar to a second programmer reading it. Yet another

contribution to Python's readability is the choice to use punctuation mostly in a conservative, conventional manner. Most operator symbols are familiar to anyone with even a vague recollection of high school math, and no new meanings have to be learned for comic strip curse characters like @&$!.

I will gladly admit that Python is not the fastest running scripting language. It is a good runner-up, though. With ever-increasing hardware speed, the accumulated running time of a program during its lifetime is often negligible compared to the programmer time needed to write and debug it. This, of course, is where the real time savings can be made. While this is hard to assess objectively, Python is considered a winner in coding time by most programmers who have tried it. In addition, many consider using Python a pleasure—a better recommendation is hard to imagine.

I am solely responsible for Python's strengths and shortcomings, even when some of the code has been written by others. However, its success is the product of a community, starting with Python's early adopters who picked it up when I first published it on the Net, and who spread the word about it in their own environment. They sent me their praise, criticism, feature requests, code contributions, and personal revelations via email. They were willing to discuss every aspect of Python in the mailing list that I soon set up, and to educate me or nudge me in the right direction where my initial intuition failed me. There have been too many contributors to thank individually. I'll make one exception, however: this book's author was one of Python's early adopters and evangelists. With this book's publication, his longstanding wish (and mine!) of having a more accessible description of Python than the standard set of manuals, has been fulfilled.

But enough rambling. I highly recommend this book to anyone interested in learning Python, whether for personal improvement or as a career enhancement. Take it away, Eric, the orchestra leader! (If you don't understand this last sentence, you haven't watched enough Monty Python reruns.)

—Guido van Rossum
Reston, Virginia, May 1996

Preface

"And Now for Something Completely Different . . . Again"

This book teaches *application-level programming* with Python. That is, it is about what you can *do* with the language once you've mastered its fundamentals.

By reading this book, you will learn to use Python in some of its most common roles: to build GUIs, web sites, networked tools, scripting interfaces, system administration programs, database and text processing utilities, and more.

Along the way, you will also learn how to use the Python language in realistically scaled programs—concepts such as object-oriented programming (OOP) and code reuse are recurring side themes throughout this text. And you will gain enough information to further explore the application domains introduced in the book, as well as to explore others.

About This Book

Now that I've told you what this book is, I should tell you what it is not. First of all, this book is not a reference manual. Although the index can be used to hunt for information, this text is not a dry collection of facts; it is designed to be read. And while many larger examples are presented along the way, this book is also not just a collection of minimally documented code samples.

Rather, this book is a *tutorial* that teaches the most common Python application domains from the ground up. It covers each of Python's target domains gradually, beginning with in-depth discussions of core concepts in each domain, before progressing toward complete programs. Large examples do appear, but only after you've learned enough to understand their techniques and code.

For example, network scripting begins with coverage of network basics and protocols and progresses through sockets, client-side tools, HTML and CGI fundamentals, and web frameworks. GUI programming gets a similarly gentle presentation, with one introductory and two tutorial chapters, before reaching larger, complete programs. And system interfaces are explored carefully before being applied in real and useful scripts.

In a sense, this book is to application-level programming what the book *Learning Python* is to the core Python language—a learning resource that makes no assumptions about your prior experience in the domains it covers. Because of this focus, this book is designed to be a natural follow-up to the core language material in *Learning Python* and a next step on the way to mastering the many facets of Python programming.

In deference to all the topic suggestions I have received over the years, I should also point out that this book is not intended to be an in-depth look at specific systems or tools. With perhaps one million Python users in the world today, it would be impossible to cover in a useful way every Python-related system that is of interest to users.

Instead, this book is designed as a tutorial for readers new to the application domains covered. The web chapters, for instance, focus on core web scripting ideas, such as server-side scripts and state retention options, not on specific systems, such as SOAP, Twisted, and Plone. By reading this book, you will gain the groundwork necessary to move on to more specific tools such as these in the domains that interest you.

About This Edition

To some extent, this edition's structure is a result of this book's history. The first edition of this book, written in 1995 and 1996, was the first book project to present the Python language. Its focus was broad. It covered the core Python language, and it briefly introduced selected application domains. Over time, the core language and reference material in the first edition evolved into more focused books *Learning Python* and *Python Pocket Reference*.

Given that evolution, the second edition of this book, written from 1999 to 2000, was an almost completely new book on advanced Python topics. Its content was an expanded and more complete version of the first edition's application domain material, designed to be an application-level follow-up to the core language material in *Learning Python*, and supplemented by the reference material in *Python Pocket Reference*. The second edition focused on application libraries and tools rather than on the Python language itself, and it was oriented toward the practical needs of real developers and real tasks—GUIs, web sites, databases, text processing, and so on.

This third edition, which I wrote in 2005 and 2006, is exactly like the second in its scope and focus, but it has been updated to reflect Python version 2.4, and to be compatible with the upcoming Python 2.5. It is a minor update, and it retains the second edition's design and scope as well as much of its original material. However, its code and descriptions have been updated to incorporate both recent changes in the Python language, as well as current best practices in Python programming.

Python Changes

You'll find that new language features such as string methods, enclosing-function scope references, list comprehensions, and new standard library tools, such as the `email` package, have been integrated throughout this edition. Smaller code changes—for instance, replacing `apply` calls and `exc_type` usage with the newer `func(*args)` and `exc_info()`—have been applied globally as well (and show up surprisingly often, because this book is concerned with building general tools).

All string-based, user-defined exceptions are now class-based, too; string exceptions appeared half a dozen times in the book's examples, but are documented as deprecated today. This is usually just a matter of changing to `class MyExc(Exception): pass`, though, in one case, exception constructor arguments must be extracted manually with the instance's `args` attribute. `` `X` `` also became `repr(X)` across all examples, and I've replaced some appearances of `while 1:` with the newer and more mnemonic `while True:`, though either form works as advertised and C programmers often find the former a natural pattern. Hopefully, these changes will future-proof the examples for as long as possible; be sure to watch the updates page described later for future Python changes.

One futurisms note: some purists might notice that I have not made all classes in this book derive from `object` to turn on new-style class features (e.g., `class MyClass(object)`). This is partly because the programs here don't employ the new-style model's slightly modified search pattern or advanced extensions. This is also because Python's creator, Guido van Rossum, told me that he believes this derivation will not be required in Python 3.0—standalone classes will simply be new-style too, automatically (in fact, the new-style class distinction is really just a temporary regression due to its incompatible search order in particular rare, multiple-inheritance trees). This is impossible to predict with certainty, of course, and Python 3.0 might abandon compatibility in other ways that break some examples in this book. Be sure to both watch for 3.0 release notes and keep an eye on this book's updates page over time.

Example Changes

You'll also notice that many of the second edition's larger examples have been upgraded substantially, especially the two larger GUI and CGI email-based examples (which are arguably the implicit goals of much of the book). For instance:

- The PyMailGUI email client is a complete rewrite and now supports sending and receiving attachments, offline viewing from mail save files, true transfer thread overlap, header-only fetches and mail caches, auto-open of attachments, detection of server inbox message number synchronization errors, and more.
- The PyMailCGI email web site was also augmented to support sending and receiving mail attachments, locate an email's main text intelligently, minimize mail fetches to run more efficiently, and use the PyCrypto extension for password encryption.
- The PyEdit text editor has grown a font dialog; unlimited undo and redo; a configuration module for fonts, colors, and sizes; intelligent modified tests on quit, open, new, and run; and case-insensitive searches.
- PyPhoto, a new, major example in Chapter 12, implements an image viewer GUI with Tkinter and the optional PIL extension. It supports cached image thumbnails, image resizing, saving images to files, and a variety of image formats thanks to PIL.
- PyClock has incorporated a countdown timer and a custom window icon; PyCalc has various cosmetic and functionality upgrades; and PyDemos now automatically pops up examples' source files.

In addition to the enhanced and new, major examples, you'll also find many other examples that demonstrate new and advanced topics such as thread queues.

Topic Changes

In addition to example changes, new topics have been added throughout. Among these are the following:

- Part II, *System Programming*, looks at the `struct`, `mimetools`, and `StringIO` modules and has been updated for newer tools such as file iterators.
- Part III, *GUI Programming*, has fresh coverage of threading and queues, the PIL imaging library, and techniques for linking a separately spawned GUI with pipes and sockets.
- Part IV, *Internet Programming*, now uses the new `email` package; covers running a web server on your local machine for CGI scripts; has substantially more on cookies, Zope, and XML parsing; and uses the PyCrypto encryption toolkit.