

*Powerful Object-Oriented Programming*

5th Edition  
Updated for 3.3 and 2.7

*Learning*

# Python



Free Sampler

O'REILLY®

Mark Lutz

# Want to read more?

You can [buy this book](#) at [oreilly.com](#)  
in print and ebook format.

**Buy 2 books, get the 3rd FREE!**

Use discount code: OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

---

It's also available at your favorite book retailer,  
including the iBookstore, the [Android Marketplace](#),  
and [Amazon.com](#).



O'REILLY®

Spreading the knowledge of innovators

[oreilly.com](#)

FIFTH EDITION

---

# Learning Python

*Mark Lutz*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

## **Learning Python, Fifth Edition**

by Mark Lutz

Copyright © 2013 Mark Lutz. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editor:** Rachel Roumeliotis

**Production Editor:** Christopher Hearse

**Copyeditor:** Rachel Monaghan

**Proofreader:** Julie Van Keuren

**Indexer:** Lucie Haskins

**Cover Designer:** Randy Comer

**Interior Designer:** David Futato

**Illustrator:** Rebecca Demarest

June 2013: Fifth Edition.

### **Revision History for the Fifth Edition:**

2013-06-07 First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449355739> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Learning Python*, 5th Edition, the image of a wood rat, and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-35573-9

[QG]

1370970520



---

# Table of Contents

Preface .....	xxxiii
---------------	--------

---

## Part I. Getting Started

1. A Python Q&A Session .....	3
Why Do People Use Python?	3
Software Quality	4
Developer Productivity	5
Is Python a “Scripting Language”?	5
OK, but What’s the Downside?	7
Who Uses Python Today?	9
What Can I Do with Python?	10
Systems Programming	11
GUIs	11
Internet Scripting	11
Component Integration	12
Database Programming	12
Rapid Prototyping	13
Numeric and Scientific Programming	13
And More: Gaming, Images, Data Mining, Robots, Excel...	14
How Is Python Developed and Supported?	15
Open Source Tradeoffs	15
What Are Python’s Technical Strengths?	16
It’s Object-Oriented and Functional	16
It’s Free	17
It’s Portable	17
It’s Powerful	18
It’s Mixable	19
It’s Relatively Easy to Use	19
It’s Relatively Easy to Learn	20
It’s Named After Monty Python	20

---

How Does Python Stack Up to Language X?	21
Chapter Summary	22
Test Your Knowledge: Quiz	23
Test Your Knowledge: Answers	23
<b>2. How Python Runs Programs .....</b>	<b>27</b>
Introducing the Python Interpreter	27
Program Execution	28
The Programmer's View	28
Python's View	30
Execution Model Variations	33
Python Implementation Alternatives	33
Execution Optimization Tools	37
Frozen Binaries	39
Future Possibilities?	40
Chapter Summary	40
Test Your Knowledge: Quiz	41
Test Your Knowledge: Answers	41
<b>3. How You Run Programs .....</b>	<b>43</b>
The Interactive Prompt	43
Starting an Interactive Session	44
The System Path	45
New Windows Options in 3.3: PATH, Launcher	46
Where to Run: Code Directories	47
What Not to Type: Prompts and Comments	48
Running Code Interactively	49
Why the Interactive Prompt?	50
Usage Notes: The Interactive Prompt	52
System Command Lines and Files	54
A First Script	55
Running Files with Command Lines	56
Command-Line Usage Variations	57
Usage Notes: Command Lines and Files	58
Unix-Style Executable Scripts: #!	59
Unix Script Basics	59
The Unix env Lookup Trick	60
The Python 3.3 Windows Launcher: #! Comes to Windows	60
Clicking File Icons	62
Icon-Click Basics	62
Clicking Icons on Windows	63
The input Trick on Windows	63
Other Icon-Click Limitations	66

Module Imports and Reloads	66
Import and Reload Basics	66
The Grander Module Story: Attributes	68
Usage Notes: import and reload	71
Using exec to Run Module Files	72
The IDLE User Interface	73
IDLE Startup Details	74
IDLE Basic Usage	75
IDLE Usability Features	76
Advanced IDLE Tools	77
Usage Notes: IDLE	78
Other IDEs	79
Other Launch Options	81
Embedding Calls	81
Frozen Binary Executables	82
Text Editor Launch Options	82
Still Other Launch Options	82
Future Possibilities?	83
Which Option Should I Use?	83
Chapter Summary	85
Test Your Knowledge: Quiz	85
Test Your Knowledge: Answers	86
Test Your Knowledge: Part I Exercises	87

---

## Part II. Types and Operations

<b>4. Introducing Python Object Types .....</b>	<b>93</b>
The Python Conceptual Hierarchy	93
Why Use Built-in Types?	94
Python's Core Data Types	95
Numbers	97
Strings	99
Sequence Operations	99
Immutability	101
Type-Specific Methods	102
Getting Help	104
Other Ways to Code Strings	105
Unicode Strings	106
Pattern Matching	108
Lists	109
Sequence Operations	109
Type-Specific Operations	109

Bounds Checking	110
Nesting	110
Comprehensions	111
Dictionaries	113
Mapping Operations	114
Nesting Revisited	115
Missing Keys: if Tests	116
Sorting Keys: for Loops	118
Iteration and Optimization	120
Tuples	121
Why Tuples?	122
Files	122
Binary Bytes Files	123
Unicode Text Files	124
Other File-Like Tools	126
Other Core Types	126
How to Break Your Code's Flexibility	128
User-Defined Classes	129
And Everything Else	130
Chapter Summary	130
Test Your Knowledge: Quiz	131
Test Your Knowledge: Answers	131
<b>5. Numeric Types .....</b>	<b>133</b>
Numeric Type Basics	133
Numeric Literals	134
Built-in Numeric Tools	136
Python Expression Operators	136
Numbers in Action	141
Variables and Basic Expressions	141
Numeric Display Formats	143
Comparisons: Normal and Chained	144
Division: Classic, Floor, and True	146
Integer Precision	150
Complex Numbers	151
Hex, Octal, Binary: Literals and Conversions	151
Bitwise Operations	153
Other Built-in Numeric Tools	155
Other Numeric Types	157
Decimal Type	157
Fraction Type	160
Sets	163
Booleans	171

Numeric Extensions	172
Chapter Summary	172
Test Your Knowledge: Quiz	173
Test Your Knowledge: Answers	173
<b>6. The Dynamic Typing Interlude .....</b>	<b>175</b>
The Case of the Missing Declaration Statements	175
Variables, Objects, and References	176
Types Live with Objects, Not Variables	177
Objects Are Garbage-Collected	178
Shared References	180
Shared References and In-Place Changes	181
Shared References and Equality	183
Dynamic Typing Is Everywhere	185
Chapter Summary	186
Test Your Knowledge: Quiz	186
Test Your Knowledge: Answers	186
<b>7. String Fundamentals .....</b>	<b>189</b>
This Chapter's Scope	189
Unicode: The Short Story	189
String Basics	190
String Literals	192
Single- and Double-Quoted Strings Are the Same	193
Escape Sequences Represent Special Characters	193
Raw Strings Suppress Escapes	196
Triple Quotes Code Multiline Block Strings	198
Strings in Action	200
Basic Operations	200
Indexing and Slicing	201
String Conversion Tools	205
Changing Strings I	208
String Methods	209
Method Call Syntax	209
Methods of Strings	210
String Method Examples: Changing Strings II	211
String Method Examples: Parsing Text	213
Other Common String Methods in Action	214
The Original string Module's Functions (Gone in 3.X)	215
String Formatting Expressions	216
Formatting Expression Basics	217
Advanced Formatting Expression Syntax	218
Advanced Formatting Expression Examples	220

Dictionary-Based Formatting Expressions	221
String Formatting Method Calls	222
Formatting Method Basics	222
Adding Keys, Attributes, and Offsets	223
Advanced Formatting Method Syntax	224
Advanced Formatting Method Examples	225
Comparison to the % Formatting Expression	227
Why the Format Method?	230
General Type Categories	235
Types Share Operation Sets by Categories	235
Mutable Types Can Be Changed in Place	236
Chapter Summary	237
Test Your Knowledge: Quiz	237
Test Your Knowledge: Answers	237
<b>8. Lists and Dictionaries .....</b>	<b>239</b>
Lists	239
Lists in Action	242
Basic List Operations	242
List Iteration and Comprehensions	242
Indexing, Slicing, and Matrixes	243
Changing Lists in Place	244
Dictionaries	250
Dictionaries in Action	252
Basic Dictionary Operations	253
Changing Dictionaries in Place	254
More Dictionary Methods	254
Example: Movie Database	256
Dictionary Usage Notes	258
Other Ways to Make Dictionaries	262
Dictionary Changes in Python 3.X and 2.7	264
Chapter Summary	271
Test Your Knowledge: Quiz	272
Test Your Knowledge: Answers	272
<b>9. Tuples, Files, and Everything Else .....</b>	<b>275</b>
Tuples	276
Tuples in Action	277
Why Lists and Tuples?	279
Records Revisited: Named Tuples	280
Files	282
Opening Files	283
Using Files	284

Files in Action	285
Text and Binary Files: The Short Story	287
Storing Python Objects in Files: Conversions	288
Storing Native Python Objects: pickle	290
Storing Python Objects in JSON Format	291
Storing Packed Binary Data: struct	293
File Context Managers	294
Other File Tools	294
Core Types Review and Summary	295
Object Flexibility	297
References Versus Copies	297
Comparisons, Equality, and Truth	300
The Meaning of True and False in Python	304
Python’s Type Hierarchies	306
Type Objects	306
Other Types in Python	308
Built-in Type Gotchas	308
Assignment Creates References, Not Copies	308
Repetition Adds One Level Deep	309
Beware of Cyclic Data Structures	310
Immutable Types Can’t Be Changed in Place	311
Chapter Summary	311
Test Your Knowledge: Quiz	311
Test Your Knowledge: Answers	312
Test Your Knowledge: Part II Exercises	313

---

## Part III. Statements and Syntax

<b>10. Introducing Python Statements .....</b>	<b>319</b>
The Python Conceptual Hierarchy Revisited	319
Python’s Statements	320
A Tale of Two ifs	322
What Python Adds	322
What Python Removes	323
Why Indentation Syntax?	324
A Few Special Cases	327
A Quick Example: Interactive Loops	329
A Simple Interactive Loop	329
Doing Math on User Inputs	331
Handling Errors by Testing Inputs	332
Handling Errors with try Statements	333
Nesting Code Three Levels Deep	335

Chapter Summary	336
Test Your Knowledge: Quiz	336
Test Your Knowledge: Answers	336
<b>11. Assignments, Expressions, and Prints .....</b>	<b>339</b>
Assignment Statements	339
Assignment Statement Forms	340
Sequence Assignments	341
Extended Sequence Unpacking in Python 3.X	344
Multiple-Target Assignments	348
Augmented Assignments	350
Variable Name Rules	352
Expression Statements	356
Expression Statements and In-Place Changes	357
Print Operations	358
The Python 3.X print Function	359
The Python 2.X print Statement	361
Print Stream Redirection	363
Version-Neutral Printing	366
Chapter Summary	369
Test Your Knowledge: Quiz	370
Test Your Knowledge: Answers	370
<b>12. if Tests and Syntax Rules .....</b>	<b>371</b>
if Statements	371
General Format	371
Basic Examples	372
Multiway Branching	372
Python Syntax Revisited	375
Block Delimiters: Indentation Rules	376
Statement Delimiters: Lines and Continuations	378
A Few Special Cases	379
Truth Values and Boolean Tests	380
The if/else Ternary Expression	382
Chapter Summary	385
Test Your Knowledge: Quiz	385
Test Your Knowledge: Answers	386
<b>13. while and for Loops .....</b>	<b>387</b>
while Loops	387
General Format	388
Examples	388
break, continue, pass, and the Loop else	389

General Loop Format	389
pass	390
continue	391
break	391
Loop else	392
for Loops	395
General Format	395
Examples	395
Loop Coding Techniques	402
Counter Loops: range	402
Sequence Scans: while and range Versus for	403
Sequence Shufflers: range and len	404
Nonexhaustive Traversals: range Versus Slices	405
Changing Lists: range Versus Comprehensions	406
Parallel Traversals: zip and map	407
Generating Both Offsets and Items: enumerate	410
Chapter Summary	413
Test Your Knowledge: Quiz	414
Test Your Knowledge: Answers	414
<b>14. Iterations and Comprehensions .....</b>	<b>415</b>
Iterations: A First Look	416
The Iteration Protocol: File Iterators	416
Manual Iteration: iter and next	419
Other Built-in Type Iterables	422
List Comprehensions: A First Detailed Look	424
List Comprehension Basics	425
Using List Comprehensions on Files	426
Extended List Comprehension Syntax	427
Other Iteration Contexts	429
New Iterables in Python 3.X	434
Impacts on 2.X Code: Pros and Cons	434
The range Iterable	435
The map, zip, and filter Iterables	436
Multiple Versus Single Pass Iterators	438
Dictionary View Iterables	439
Other Iteration Topics	440
Chapter Summary	441
Test Your Knowledge: Quiz	441
Test Your Knowledge: Answers	441
<b>15. The Documentation Interlude .....</b>	<b>443</b>
Python Documentation Sources	443

# Comments	444
The dir Function	444
Docstrings: <code>__doc__</code>	446
PyDoc: The help Function	449
PyDoc: HTML Reports	452
Beyond docstrings: Sphinx	461
The Standard Manual Set	461
Web Resources	462
Published Books	463
Common Coding Gotchas	463
Chapter Summary	465
Test Your Knowledge: Quiz	466
Test Your Knowledge: Answers	466
Test Your Knowledge: Part III Exercises	467

---

## Part IV. Functions and Generators

<b>16. Function Basics .....</b>	<b>473</b>
Why Use Functions?	474
Coding Functions	475
def Statements	476
def Executes at Runtime	477
A First Example: Definitions and Calls	478
Definition	478
Calls	478
Polymorphism in Python	479
A Second Example: Intersecting Sequences	480
Definition	481
Calls	481
Polymorphism Revisited	482
Local Variables	483
Chapter Summary	483
Test Your Knowledge: Quiz	483
Test Your Knowledge: Answers	484
<b>17. Scopes .....</b>	<b>485</b>
Python Scope Basics	485
Scope Details	486
Name Resolution: The LEGB Rule	488
Scope Example	490
The Built-in Scope	491
The <code>global</code> Statement	494

---

Program Design: Minimize Global Variables	495
Program Design: Minimize Cross-File Changes	497
Other Ways to Access Globals	498
Scopes and Nested Functions	499
Nested Scope Details	500
Nested Scope Examples	500
Factory Functions: Closures	501
Retaining Enclosing Scope State with Defaults	504
The nonlocal Statement in 3.X	508
nonlocal Basics	508
nonlocal in Action	509
Why nonlocal? State Retention Options	512
State with nonlocal: 3.X only	512
State with Globals: A Single Copy Only	513
State with Classes: Explicit Attributes (Preview)	513
State with Function Attributes: 3.X and 2.X	515
Chapter Summary	519
Test Your Knowledge: Quiz	519
Test Your Knowledge: Answers	520
<b>18. Arguments .....</b>	<b>523</b>
Argument-Passing Basics	523
Arguments and Shared References	524
Avoiding Mutable Argument Changes	526
Simulating Output Parameters and Multiple Results	527
Special Argument-Matching Modes	528
Argument Matching Basics	529
Argument Matching Syntax	530
The Gritty Details	531
Keyword and Default Examples	532
Arbitrary Arguments Examples	534
Python 3.X Keyword-Only Arguments	539
The min Wakeup Call!	542
Full Credit	542
Bonus Points	544
The Punch Line...	544
Generalized Set Functions	545
Emulating the Python 3.X print Function	547
Using Keyword-Only Arguments	548
Chapter Summary	550
Test Your Knowledge: Quiz	551
Test Your Knowledge: Answers	552

<b>19. Advanced Function Topics .....</b>	<b>553</b>
Function Design Concepts	553
Recursive Functions	555
Summation with Recursion	555
Coding Alternatives	556
Loop Statements Versus Recursion	557
Handling Arbitrary Structures	558
Function Objects: Attributes and Annotations	562
Indirect Function Calls: “First Class” Objects	562
Function Introspection	563
Function Attributes	564
Function Annotations in 3.X	565
Anonymous Functions: lambda	567
lambda Basics	568
Why Use lambda?	569
How (Not) to Obfuscate Your Python Code	571
Scopes: lambdas Can Be Nested Too	572
Functional Programming Tools	574
Mapping Functions over Iterables: map	574
Selecting Items in Iterables: filter	576
Combining Items in Iterables: reduce	576
Chapter Summary	578
Test Your Knowledge: Quiz	578
Test Your Knowledge: Answers	578
<b>20. Comprehensions and Generations .....</b>	<b>581</b>
List Comprehensions and Functional Tools	581
List Comprehensions Versus map	582
Adding Tests and Nested Loops: filter	583
Example: List Comprehensions and Matrixes	586
Don’t Abuse List Comprehensions: KISS	588
Generator Functions and Expressions	591
Generator Functions: yield Versus return	592
Generator Expressions: Iterables Meet Comprehensions	597
Generator Functions Versus Generator Expressions	602
Generators Are Single-Iteration Objects	604
Generation in Built-in Types, Tools, and Classes	606
Example: Generating Scrambled Sequences	609
Don’t Abuse Generators: EIBTI	614
Example: Emulating zip and map with Iteration Tools	617
Comprehension Syntax Summary	622
Scopes and Comprehension Variables	623
Comprehending Set and Dictionary Comprehensions	624

Extended Comprehension Syntax for Sets and Dictionaries	625
Chapter Summary	626
Test Your Knowledge: Quiz	626
Test Your Knowledge: Answers	626
<b>21. The Benchmarking Interlude .....</b>	<b>629</b>
Timing Iteration Alternatives	629
Timing Module: Homegrown	630
Timing Script	634
Timing Results	635
Timing Module Alternatives	638
Other Suggestions	642
Timing Iterations and Pythons with timeit	642
Basic timeit Usage	643
Benchmark Module and Script: timeit	647
Benchmark Script Results	649
More Fun with Benchmarks	651
Other Benchmarking Topics: pystones	656
Function Gotchas	656
Local Names Are Detected Statically	657
Defaults and Mutable Objects	658
Functions Without returns	660
Miscellaneous Function Gotchas	661
Chapter Summary	661
Test Your Knowledge: Quiz	662
Test Your Knowledge: Answers	662
Test Your Knowledge: Part IV Exercises	663

---

## Part V. Modules and Packages

<b>22. Modules: The Big Picture .....</b>	<b>669</b>
Why Use Modules?	669
Python Program Architecture	670
How to Structure a Program	671
Imports and Attributes	671
Standard Library Modules	673
How Imports Work	674
1. Find It	674
2. Compile It (Maybe)	675
3. Run It	675
Byte Code Files: <code>__pycache__</code> in Python 3.2+	676
Byte Code File Models in Action	677

The Module Search Path	678
Configuring the Search Path	681
Search Path Variations	681
The <code>sys.path</code> List	681
Module File Selection	682
Chapter Summary	685
Test Your Knowledge: Quiz	685
Test Your Knowledge: Answers	685
<b>23. Module Coding Basics .....</b>	<b>687</b>
Module Creation	687
Module Filenames	687
Other Kinds of Modules	688
Module Usage	688
The <code>import</code> Statement	689
The <code>from</code> Statement	689
The <code>from *</code> Statement	689
Imports Happen Only Once	690
<code>import</code> and <code>from</code> Are Assignments	691
<code>import</code> and <code>from</code> Equivalence	692
Potential Pitfalls of the <code>from</code> Statement	693
Module Namespaces	694
Files Generate Namespaces	695
Namespace Dictionaries: <code>__dict__</code>	696
Attribute Name Qualification	697
Imports Versus Scopes	698
Namespace Nesting	699
Reloading Modules	700
<code>reload</code> Basics	701
<code>reload</code> Example	702
Chapter Summary	703
Test Your Knowledge: Quiz	704
Test Your Knowledge: Answers	704
<b>24. Module Packages .....</b>	<b>707</b>
Package Import Basics	708
Packages and Search Path Settings	708
Package <code>__init__.py</code> Files	709
Package Import Example	711
<code>from</code> Versus <code>import</code> with Packages	713
Why Use Package Imports?	713
A Tale of Three Systems	714
Package Relative Imports	717

Changes in Python 3.X	718
Relative Import Basics	718
Why Relative Imports?	720
The Scope of Relative Imports	722
Module Lookup Rules Summary	723
Relative Imports in Action	723
Pitfalls of Package-Relative Imports: Mixed Use	729
Python 3.3 Namespace Packages	734
Namespace Package Semantics	735
Impacts on Regular Packages: Optional <code>__init__.py</code>	736
Namespace Packages in Action	737
Namespace Package Nesting	738
Files Still Have Precedence over Directories	740
Chapter Summary	742
Test Your Knowledge: Quiz	742
Test Your Knowledge: Answers	742
<b>25. Advanced Module Topics .....</b>	<b>745</b>
Module Design Concepts	745
Data Hiding in Modules	747
Minimizing from * Damage: <code>_X</code> and <code>__all__</code>	747
Enabling Future Language Features: <code>__future__</code>	748
Mixed Usage Modes: <code>__name__</code> and <code>__main__</code>	749
Unit Tests with <code>__name__</code>	750
Example: Dual Mode Code	751
Currency Symbols: Unicode in Action	754
Docstrings: Module Documentation at Work	756
Changing the Module Search Path	756
The <code>as</code> Extension for <code>import</code> and <code>from</code>	758
Example: Modules Are Objects	759
Importing Modules by Name String	761
Running Code Strings	762
Direct Calls: Two Options	762
Example: Transitive Module Reloads	763
A Recursive Reloader	764
Alternative Codings	767
Module Gotchas	770
Module Name Clashes: Package and Package-Relative Imports	771
Statement Order Matters in Top-Level Code	771
<code>from</code> Copies Names but Doesn't Link	772
<code>from *</code> Can Obscure the Meaning of Variables	773
<code>reload</code> May Not Impact from Imports	773
<code>reload</code> , <code>from</code> , and Interactive Testing	774

Recursive from Imports May Not Work	775
Chapter Summary	776
Test Your Knowledge: Quiz	777
Test Your Knowledge: Answers	777
Test Your Knowledge: Part V Exercises	778
<hr/>	
<b>Part VI. Classes and OOP</b>	
<b>26. OOP: The Big Picture .....</b>	<b>783</b>
Why Use Classes?	784
OOP from 30,000 Feet	785
Attribute Inheritance Search	785
Classes and Instances	788
Method Calls	788
Coding Class Trees	789
Operator Overloading	791
OOP Is About Code Reuse	792
Chapter Summary	795
Test Your Knowledge: Quiz	795
Test Your Knowledge: Answers	795
<b>27. Class Coding Basics .....</b>	<b>797</b>
Classes Generate Multiple Instance Objects	797
Class Objects Provide Default Behavior	798
Instance Objects Are Concrete Items	798
A First Example	799
Classes Are Customized by Inheritance	801
A Second Example	802
Classes Are Attributes in Modules	804
Classes Can Intercept Python Operators	805
A Third Example	806
Why Use Operator Overloading?	808
The World's Simplest Python Class	809
Records Revisited: Classes Versus Dictionaries	812
Chapter Summary	814
Test Your Knowledge: Quiz	815
Test Your Knowledge: Answers	815
<b>28. A More Realistic Example .....</b>	<b>817</b>
Step 1: Making Instances	818
Coding Constructors	818
Testing As You Go	819

Using Code Two Ways	820
Step 2: Adding Behavior Methods	822
Coding Methods	824
Step 3: Operator Overloading	826
Providing Print Displays	826
Step 4: Customizing Behavior by Subclassing	828
Coding Subclasses	828
Augmenting Methods: The Bad Way	829
Augmenting Methods: The Good Way	829
Polymorphism in Action	832
Inherit, Customize, and Extend	833
OOP: The Big Idea	833
Step 5: Customizing Constructors, Too	834
OOP Is Simpler Than You May Think	836
Other Ways to Combine Classes	836
Step 6: Using Introspection Tools	840
Special Class Attributes	840
A Generic Display Tool	842
Instance Versus Class Attributes	843
Name Considerations in Tool Classes	844
Our Classes' Final Form	845
Step 7 (Final): Storing Objects in a Database	847
Pickles and Shelves	847
Storing Objects on a Shelve Database	848
Exploring Shelves Interactively	849
Updating Objects on a Shelve	851
Future Directions	853
Chapter Summary	855
Test Your Knowledge: Quiz	855
Test Your Knowledge: Answers	856
<b>29. Class Coding Details .....</b>	<b>859</b>
The class Statement	859
General Form	860
Example	860
Methods	862
Method Example	863
Calling Superclass Constructors	864
Other Method Call Possibilities	864
Inheritance	865
Attribute Tree Construction	865
Specializing Inherited Methods	866
Class Interface Techniques	867

Abstract Superclasses	869
Namespaces: The Conclusion	872
Simple Names: Global Unless Assigned	872
Attribute Names: Object Namespaces	872
The “Zen” of Namespaces: Assignments Classify Names	873
Nested Classes: The LEGB Scopes Rule Revisited	875
Namespace Dictionaries: Review	878
Namespace Links: A Tree Climber	880
Documentation Strings Revisited	882
Classes Versus Modules	884
Chapter Summary	884
Test Your Knowledge: Quiz	884
Test Your Knowledge: Answers	885
<b>30. Operator Overloading .....</b>	<b>887</b>
The Basics	887
Constructors and Expressions: <code>__init__</code> and <code>__sub__</code>	888
Common Operator Overloading Methods	888
Indexing and Slicing: <code>__getitem__</code> and <code>__setitem__</code>	890
Intercepting Slices	891
Slicing and Indexing in Python 2.X	893
But 3.X’s <code>__index__</code> Is Not Indexing!	894
Index Iteration: <code>__getitem__</code>	894
Iterable Objects: <code>__iter__</code> and <code>__next__</code>	895
User-Defined Iterables	896
Multiple Iterators on One Object	899
Coding Alternative: <code>__iter__</code> plus <code>yield</code>	902
Membership: <code>__contains__</code> , <code>__iter__</code> , and <code>__getitem__</code>	906
Attribute Access: <code>__getattr__</code> and <code>__setattr__</code>	909
Attribute Reference	909
Attribute Assignment and Deletion	910
Other Attribute Management Tools	912
Emulating Privacy for Instance Attributes: Part 1	912
String Representation: <code>__repr__</code> and <code>__str__</code>	913
Why Two Display Methods?	914
Display Usage Notes	916
Right-Side and In-Place Uses: <code>__radd__</code> and <code>__iadd__</code>	917
Right-Side Addition	917
In-Place Addition	920
Call Expressions: <code>__call__</code>	921
Function Interfaces and Callback-Based Code	923
Comparisons: <code>__lt__</code> , <code>__gt__</code> , and Others	925
The <code>__cmp__</code> Method in Python 2.X	926

Boolean Tests: <code>__bool__</code> and <code>__len__</code>	927
Boolean Methods in Python 2.X	928
Object Destruction: <code>__del__</code>	929
Destructor Usage Notes	930
Chapter Summary	931
Test Your Knowledge: Quiz	931
Test Your Knowledge: Answers	931
<b>31. Designing with Classes .....</b>	<b>933</b>
Python and OOP	933
Polymorphism Means Interfaces, Not Call Signatures	934
OOP and Inheritance: “Is-a” Relationships	935
OOP and Composition: “Has-a” Relationships	937
Stream Processors Revisited	938
OOP and Delegation: “Wrapper” Proxy Objects	942
Pseudoprivate Class Attributes	944
Name Mangling Overview	945
Why Use Pseudoprivate Attributes?	945
Methods Are Objects: Bound or Unbound	948
Unbound Methods Are Functions in 3.X	950
Bound Methods and Other Callable Objects	951
Classes Are Objects: Generic Object Factories	954
Why Factories?	955
Multiple Inheritance: “Mix-in” Classes	956
Coding Mix-in Display Classes	957
Other Design-Related Topics	977
Chapter Summary	977
Test Your Knowledge: Quiz	978
Test Your Knowledge: Answers	978
<b>32. Advanced Class Topics .....</b>	<b>979</b>
Extending Built-in Types	980
Extending Types by Embedding	980
Extending Types by Subclassing	981
The “New Style” Class Model	983
Just How New Is New-Style?	984
New-Style Class Changes	985
Attribute Fetch for Built-ins Skips Instances	987
Type Model Changes	992
All Classes Derive from “object”	995
Diamond Inheritance Change	997
More on the MRO: Method Resolution Order	1001
Example: Mapping Attributes to Inheritance Sources	1004

New-Style Class Extensions	1010
Slots: Attribute Declarations	1010
Properties: Attribute Accessors	1020
<code>__getattribute__</code> and Descriptors: Attribute Tools	1023
Other Class Changes and Extensions	1023
Static and Class Methods	1024
Why the Special Methods?	1024
Static Methods in 2.X and 3.X	1025
Static Method Alternatives	1027
Using Static and Class Methods	1028
Counting Instances with Static Methods	1030
Counting Instances with Class Methods	1031
Decorators and Metaclasses: Part 1	1034
Function Decorator Basics	1035
A First Look at User-Defined Function Decorators	1037
A First Look at Class Decorators and Metaclasses	1038
For More Details	1040
The super Built-in Function: For Better or Worse?	1041
The Great super Debate	1041
Traditional Superclass Call Form: Portable, General	1042
Basic super Usage and Its Tradeoffs	1043
The super Upsides: Tree Changes and Dispatch	1049
Runtime Class Changes and super	1049
Cooperative Multiple Inheritance Method Dispatch	1050
The super Summary	1062
Class Gotchas	1064
Changing Class Attributes Can Have Side Effects	1064
Changing Mutable Class Attributes Can Have Side Effects, Too	1065
Multiple Inheritance: Order Matters	1066
Scopes in Methods and Classes	1068
Miscellaneous Class Gotchas	1069
KISS Revisited: “Overwrapping-itis”	1070
Chapter Summary	1070
Test Your Knowledge: Quiz	1071
Test Your Knowledge: Answers	1071
Test Your Knowledge: Part VI Exercises	1072

---

## Part VII. Exceptions and Tools

<b>33. Exception Basics .....</b>	<b>1081</b>
Why Use Exceptions?	1081
Exception Roles	1082

Exceptions: The Short Story	1083
Default Exception Handler	1083
Catching Exceptions	1084
Raising Exceptions	1085
User-Defined Exceptions	1086
Termination Actions	1087
Chapter Summary	1089
Test Your Knowledge: Quiz	1090
Test Your Knowledge: Answers	1090
<b>34. Exception Coding Details . . . . .</b>	<b>1093</b>
The try/except/else Statement	1093
How try Statements Work	1094
try Statement Clauses	1095
The try else Clause	1098
Example: Default Behavior	1098
Example: Catching Built-in Exceptions	1100
The try/finally Statement	1100
Example: Coding Termination Actions with try/finally	1101
Unified try/except/finally	1102
Unified try Statement Syntax	1104
Combining finally and except by Nesting	1104
Unified try Example	1105
The raise Statement	1106
Raising Exceptions	1107
Scopes and try except Variables	1108
Propagating Exceptions with raise	1110
Python 3.X Exception Chaining: raise from	1110
The assert Statement	1112
Example: Trapping Constraints (but Not Errors!)	1113
with/as Context Managers	1114
Basic Usage	1114
The Context Management Protocol	1116
Multiple Context Managers in 3.1, 2.7, and Later	1118
Chapter Summary	1119
Test Your Knowledge: Quiz	1120
Test Your Knowledge: Answers	1120
<b>35. Exception Objects . . . . .</b>	<b>1123</b>
Exceptions: Back to the Future	1124
String Exceptions Are Right Out!	1124
Class-Based Exceptions	1125
Coding Exceptions Classes	1126

Why Exception Hierarchies?	1128
Built-in Exception Classes	1131
Built-in Exception Categories	1132
Default Printing and State	1133
Custom Print Displays	1135
Custom Data and Behavior	1136
Providing Exception Details	1136
Providing Exception Methods	1137
Chapter Summary	1139
Test Your Knowledge: Quiz	1139
Test Your Knowledge: Answers	1139
<b>36. Designing with Exceptions .....</b>	<b>1141</b>
Nesting Exception Handlers	1141
Example: Control-Flow Nesting	1143
Example: Syntactic Nesting	1143
Exception Idioms	1145
Breaking Out of Multiple Nested Loops: “go to”	1145
Exceptions Aren’t Always Errors	1146
Functions Can Signal Conditions with raise	1147
Closing Files and Server Connections	1148
Debugging with Outer try Statements	1149
Running In-Process Tests	1149
More on sys.exc_info	1150
Displaying Errors and Tracebacks	1151
Exception Design Tips and Gotchas	1152
What Should Be Wrapped	1152
Catching Too Much: Avoid Empty except and Exception	1153
Catching Too Little: Use Class-Based Categories	1155
Core Language Summary	1155
The Python Toolset	1156
Development Tools for Larger Projects	1157
Chapter Summary	1160
Test Your Knowledge: Quiz	1161
Test Your Knowledge: Answers	1161
Test Your Knowledge: Part VII Exercises	1161

---

## Part VIII. Advanced Topics

<b>37. Unicode and Byte Strings .....</b>	<b>1165</b>
String Changes in 3.X	1166
String Basics	1167

Character Encoding Schemes	1167
How Python Stores Strings in Memory	1170
Python's String Types	1171
Text and Binary Files	1173
Coding Basic Strings	1174
Python 3.X String Literals	1175
Python 2.X String Literals	1176
String Type Conversions	1177
Coding Unicode Strings	1178
Coding ASCII Text	1178
Coding Non-ASCII Text	1179
Encoding and Decoding Non-ASCII text	1180
Other Encoding Schemes	1181
Byte String Literals: Encoded Text	1183
Converting Encodings	1184
Coding Unicode Strings in Python 2.X	1185
Source File Character Set Encoding Declarations	1187
Using 3.X bytes Objects	1189
Method Calls	1189
Sequence Operations	1190
Other Ways to Make bytes Objects	1191
Mixing String Types	1192
Using 3.X/2.6+ bytearray Objects	1192
byterarrays in Action	1193
Python 3.X String Types Summary	1195
Using Text and Binary Files	1195
Text File Basics	1196
Text and Binary Modes in 2.X and 3.X	1197
Type and Content Mismatches in 3.X	1198
Using Unicode Files	1199
Reading and Writing Unicode in 3.X	1199
Handling the BOM in 3.X	1201
Unicode Files in 2.X	1204
Unicode Filenames and Streams	1205
Other String Tool Changes in 3.X	1206
The re Pattern-Matching Module	1206
The struct Binary Data Module	1207
The pickle Object Serialization Module	1209
XML Parsing Tools	1211
Chapter Summary	1215
Test Your Knowledge: Quiz	1215
Test Your Knowledge: Answers	1216

<b>38. Managed Attributes .....</b>	<b>1219</b>
Why Manage Attributes?	1219
Inserting Code to Run on Attribute Access	1220
Properties	1221
The Basics	1222
A First Example	1222
Computed Attributes	1224
Coding Properties with Decorators	1224
Descriptors	1226
The Basics	1227
A First Example	1229
Computed Attributes	1231
Using State Information in Descriptors	1232
How Properties and Descriptors Relate	1236
__getattr__ and __getattribute__	1237
The Basics	1238
A First Example	1241
Computed Attributes	1243
__getattr__ and __getattribute__ Compared	1245
Management Techniques Compared	1246
Intercepting Built-in Operation Attributes	1249
Example: Attribute Validations	1256
Using Properties to Validate	1256
Using Descriptors to Validate	1259
Using __getattr__ to Validate	1263
Using __getattribute__ to Validate	1265
Chapter Summary	1266
Test Your Knowledge: Quiz	1266
Test Your Knowledge: Answers	1267
<b>39. Decorators .....</b>	<b>1269</b>
What's a Decorator?	1269
Managing Calls and Instances	1270
Managing Functions and Classes	1270
Using and Defining Decorators	1271
Why Decorators?	1271
The Basics	1273
Function Decorators	1273
Class Decorators	1277
Decorator Nesting	1279
Decorator Arguments	1281
Decorators Manage Functions and Classes, Too	1282
Coding Function Decorators	1283

Tracing Calls	1283
Decorator State Retention Options	1285
Class Blunders I: Decorating Methods	1289
Timing Calls	1295
Adding Decorator Arguments	1298
Coding Class Decorators	1301
Singleton Classes	1301
Tracing Object Interfaces	1303
Class Blunders II: Retaining Multiple Instances	1308
Decorators Versus Manager Functions	1309
Why Decorators? (Revisited)	1310
Managing Functions and Classes Directly	1312
Example: “Private” and “Public” Attributes	1314
Implementing Private Attributes	1314
Implementation Details I	1317
Generalizing for Public Declarations, Too	1318
Implementation Details II	1320
Open Issues	1321
Python Isn’t About Control	1329
Example: Validating Function Arguments	1330
The Goal	1330
A Basic Range-Testing Decorator for Positional Arguments	1331
Generalizing for Keywords and Defaults, Too	1333
Implementation Details	1336
Open Issues	1338
Decorator Arguments Versus Function Annotations	1340
Other Applications: Type Testing (If You Insist!)	1342
Chapter Summary	1343
Test Your Knowledge: Quiz	1344
Test Your Knowledge: Answers	1345
<b>40. Metaclasses .....</b>	<b>1355</b>
To Metaclass or Not to Metaclass	1356
Increasing Levels of “Magic”	1357
A Language of Hooks	1358
The Downside of “Helper” Functions	1359
Metaclasses Versus Class Decorators: Round 1	1361
The Metaclass Model	1364
Classes Are Instances of type	1364
Metaclasses Are Subclasses of Type	1366
Class Statement Protocol	1367
Declaring Metaclasses	1368
Declaration in 3.X	1369

Declaration in 2.X	1369
Metaclass Dispatch in Both 3.X and 2.X	1370
Coding Metaclasses	1370
A Basic Metaclass	1371
Customizing Construction and Initialization	1372
Other Metaclass Coding Techniques	1373
Inheritance and Instance	1378
Metaclass Versus Superclass	1381
Inheritance: The Full Story	1382
Metaclass Methods	1388
Metaclass Methods Versus Class Methods	1389
Operator Overloading in Metaclass Methods	1390
Example: Adding Methods to Classes	1391
Manual Augmentation	1391
Metaclass-Based Augmentation	1393
Metaclasses Versus Class Decorators: Round 2	1394
Example: Applying Decorators to Methods	1400
Tracing with Decoration Manually	1400
Tracing with Metaclasses and Decorators	1401
Applying Any Decorator to Methods	1403
Metaclasses Versus Class Decorators: Round 3 (and Last)	1404
Chapter Summary	1407
Test Your Knowledge: Quiz	1407
Test Your Knowledge: Answers	1408
<b>41. All Good Things .....</b>	<b>1409</b>
The Python Paradox	1409
On “Optional” Language Features	1410
Against Disquieting Improvements	1411
Complexity Versus Power	1412
Simplicity Versus Elitism	1412
Closing Thoughts	1413
Where to Go From Here	1414
Encore: Print Your Own Completion Certificate!	1414

---

## Part IX. Appendixes

<b>A. Installation and Configuration .....</b>	<b>1421</b>
Installing the Python Interpreter	1421
Is Python Already Present?	1421
Where to Get Python	1422
Installation Steps	1423

Configuring Python	1427
Python Environment Variables	1427
How to Set Configuration Options	1429
Python Command-Line Arguments	1432
Python 3.3 Windows Launcher Command Lines	1435
For More Help	1436
<b>B. The Python 3.3 Windows Launcher .....</b>	<b>1437</b>
The Unix Legacy	1437
The Windows Legacy	1438
Introducing the New Windows Launcher	1439
A Windows Launcher Tutorial	1441
Step 1: Using Version Directives in Files	1441
Step 2: Using Command-Line Version Switches	1444
Step 3: Using and Changing Defaults	1445
Pitfalls of the New Windows Launcher	1447
Pitfall 1: Unrecognized Unix !# Lines Fail	1447
Pitfall 2: The Launcher Defaults to 2.X	1448
Pitfall 3: The New PATH Extension Option	1449
Conclusions: A Net Win for Windows	1450
<b>C. Python Changes and This Book .....</b>	<b>1451</b>
Major 2.X/3.X Differences	1451
3.X Differences	1452
3.X-Only Extensions	1453
General Remarks: 3.X Changes	1454
Changes in Libraries and Tools	1454
Migrating to 3.X	1455
Fifth Edition Python Changes: 2.7, 3.2, 3.3	1456
Changes in Python 2.7	1456
Changes in Python 3.3	1457
Changes in Python 3.2	1458
Fourth Edition Python Changes: 2.6, 3.0, 3.1	1458
Changes in Python 3.1	1458
Changes in Python 3.0 and 2.6	1459
Specific Language Removals in 3.0	1460
Third Edition Python Changes: 2.3, 2.4, 2.5	1462
Earlier and Later Python Changes	1463
<b>D. Solutions to End-of-Part Exercises .....</b>	<b>1465</b>
Part I, Getting Started	1465
Part II, Types and Operations	1467
Part III, Statements and Syntax	1473

Part IV, Functions and Generators	1475
Part V, Modules and Packages	1485
Part VI, Classes and OOP	1489
Part VII, Exceptions and Tools	1497
<b>Index .....</b>	<b>1507</b>

PART I

---

# Getting Started



# A Python Q&A Session

If you've bought this book, you may already know what Python is and why it's an important tool to learn. If you don't, you probably won't be sold on Python until you've learned the language by reading the rest of this book and have done a project or two. But before we jump into details, this first chapter of this book will briefly introduce some of the main reasons behind Python's popularity. To begin sculpting a definition of Python, this chapter takes the form of a question-and-answer session, which poses some of the most common questions asked by beginners.

## Why Do People Use Python?

Because there are many programming languages available today, this is the usual first question of newcomers. Given that there are roughly 1 million Python users out there at the moment, there really is no way to answer this question with complete accuracy; the choice of development tools is sometimes based on unique constraints or personal preference.

But after teaching Python to roughly 260 groups and over 4,000 students during the last 16 years, I have seen some common themes emerge. The primary factors cited by Python users seem to be these:

### *Software quality*

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the scripting world. Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and function programming.

### *Developer productivity*

Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically *one-third* to

*one-fifth* the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain after the fact. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

#### *Program portability*

Most Python programs run unchanged on *all major computer platforms*. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script’s code between machines. Moreover, Python offers multiple options for coding portable graphical user interfaces, database access programs, web-based systems, and more. Even operating system interfaces, including program launches and directory processing, are as portable in Python as they can possibly be.

#### *Support libraries*

Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both homegrown libraries and a vast collection of third-party application support software. Python’s *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling). The NumPy extension, for instance, has been described as a free and more powerful equivalent to the Matlab numeric programming system.

#### *Component integration*

Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product *customization and extension* tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

#### *Enjoyment*

Because of Python’s ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this may be an intangible benefit, its effect on productivity is an important asset.

Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users, and merit a fuller description.

## **Software Quality**

By design, Python implements a deliberately simple and readable syntax and a highly coherent programming model. As a slogan at a past Python conference attests, the net result is that Python seems to “fit your brain”—that is, features of the language interact

in consistent and limited ways and follow naturally from a small set of core concepts. This makes the language easier to learn, understand, and remember. In practice, Python programmers do not need to constantly refer to manuals when reading or writing code; it's a consistently designed system that many find yields surprisingly uniform code.

By philosophy, Python adopts a somewhat minimalist approach. This means that although there are usually multiple ways to accomplish a coding task, there is usually just one obvious way, a few less obvious alternatives, and a small set of coherent interactions everywhere in the language. Moreover, Python doesn't make arbitrary decisions for you; when interactions are ambiguous, explicit intervention is preferred over "magic." In the Python way of thinking, explicit is better than implicit, and simple is better than complex.<sup>1</sup>

Beyond such design themes, Python includes tools such as modules and OOP that naturally promote code reusability. And because Python is focused on quality, so too, naturally, are Python programmers.

## Developer Productivity

During the great Internet boom of the mid-to-late 1990s, it was difficult to find enough programmers to implement software projects; developers were asked to implement systems as fast as the Internet evolved. In later eras of layoffs and economic recession, the picture shifted. Programming staffs were often asked to accomplish the same tasks with even fewer people.

In both of these scenarios, Python has shined as a tool that allows programmers to get more done with less effort. It is deliberately optimized for *speed of development*—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools. The net effect is that Python typically boosts developer productivity many times beyond the levels supported by traditional languages. That's good news in both boom and bust times, and everywhere the software industry goes in between.

## Is Python a “Scripting Language”?

Python is a general-purpose programming language that is often applied in scripting roles. It is commonly defined as an *object-oriented scripting language*—a definition that blends support for OOP with an overall orientation toward scripting roles. If pressed for a one-liner, I'd say that Python is probably better known as a *general-purpose pro-*

1. For a more complete look at the Python philosophy, type the command `import this` at any Python interactive prompt (you'll see how in [Chapter 3](#)). This invokes an “Easter egg” hidden in Python—a collection of design principles underlying Python that permeate both the language and its user community. Among them, the acronym EIBTI is now fashionable jargon for the “explicit is better than implicit” rule. These principles are not religion, but are close enough to qualify as a Python motto and creed, which we'll be quoting from often in this book.

*gramming language that blends procedural, functional, and object-oriented paradigms*—a statement that captures the richness and scope of today’s Python.

Still, the term “scripting” seems to have stuck to Python like glue, perhaps as a contrast with larger programming effort required by some other tools. For example, people often use the word “script” instead of “program” to describe a Python code file. In keeping with this tradition, this book uses the terms “script” and “program” interchangeably, with a slight preference for “script” to describe a simpler top-level file and “program” to refer to a more sophisticated multifile application.

Because the term “scripting language” has so many different meanings to different observers, though, some would prefer that it not be applied to Python at all. In fact, people tend to make three very different associations, some of which are more useful than others, when they hear Python labeled as such:

#### *Shell tools*

Sometimes when people hear Python described as a scripting language, they think it means that Python is a tool for coding operating-system-oriented scripts. Such programs are often launched from console command lines and perform tasks such as processing text files and launching other programs.

Python programs can and do serve such roles, but this is just one of dozens of common Python application domains. It is not just a better shell-script language.

#### *Control language*

To others, scripting refers to a “glue” layer used to control and direct (i.e., script) other application components. Python programs are indeed often deployed in the context of larger applications. For instance, to test hardware devices, Python programs may call out to components that give low-level access to a device. Similarly, programs may run bits of Python code at strategic points to support end-user product customization without the need to ship and recompile the entire system’s source code.

Python’s simplicity makes it a naturally flexible control tool. Technically, though, this is also just a common Python role; many (perhaps most) Python programmers code standalone scripts without ever using or knowing about any integrated components. It is not just a control language.

#### *Ease of use*

Probably the best way to think of the term “scripting language” is that it refers to a simple language used for quickly coding tasks. This is especially true when the term is applied to Python, which allows much faster program development than compiled languages like C++. Its rapid development cycle fosters an exploratory, incremental mode of programming that has to be experienced to be appreciated.

Don’t be fooled, though—Python is not just for simple tasks. Rather, it makes tasks simple by its ease of use and flexibility. Python has a simple feature set, but it allows programs to scale up in sophistication as needed. Because of that, it is commonly used for quick tactical tasks and longer-term strategic development.

So, is Python a scripting language or not? It depends on whom you ask. In general, the term “scripting” is probably best used to describe the rapid and flexible mode of development that Python supports, rather than a particular application domain.

## OK, but What’s the Downside?

After using it for 21 years, writing about it for 18, and teaching it for 16, I’ve found that the only significant universal downside to Python is that, as currently implemented, its *execution speed* may not always be as fast as that of fully compiled and lower-level languages such as C and C++. Though relatively rare today, for some tasks you may still occasionally need to get “closer to the iron” by using lower-level languages such as these that are more directly mapped to the underlying hardware architecture.

We’ll talk about implementation concepts in detail later in this book. In short, the standard implementations of Python today compile (i.e., translate) source code statements to an intermediate format known as *byte code* and then interpret the byte code. Byte code provides portability, as it is a platform-independent format. However, because Python is not normally compiled all the way down to binary machine code (e.g., instructions for an Intel chip), some programs will run more slowly in Python than in a fully compiled language like C. The PyPy system discussed in the next chapter can achieve a 10X to 100X speedup on some code by compiling further as your program runs, but it’s a separate, alternative implementation.

Whether you will ever *care* about the execution speed difference depends on what kinds of programs you write. Python has been optimized numerous times, and Python code runs fast enough by itself in most application domains. Furthermore, whenever you do something “real” in a Python script, like processing a file or constructing a graphical user interface (GUI), your program will actually run at C speed, since such tasks are immediately dispatched to compiled C code inside the Python interpreter. More fundamentally, Python’s speed-of-development gain is often far more important than any speed-of-execution loss, especially given modern computer speeds.

Even at today’s CPU speeds, though, there still are some domains that do require optimal execution speeds. Numeric programming and animation, for example, often need at least their core number-crunching components to run at C speed (or better). If you work in such a domain, you can still use Python—simply split off the parts of the application that require optimal speed into *compiled extensions*, and link those into your system for use in Python scripts.

We won’t talk about extensions much in this text, but this is really just an instance of the Python-as-control-language role we discussed earlier. A prime example of this dual language strategy is the *NumPy* numeric programming extension for Python; by combining compiled and optimized numeric extension libraries with the Python language, NumPy turns Python into a numeric programming tool that is simultaneously efficient and easy to use. When needed, such extensions provide a powerful optimization tool.

## Other Python Tradeoffs: The Intangible Bits

I mentioned that execution speed is the only major downside to Python. That's indeed the case for most Python users, and especially for newcomers. Most people find Python to be easy to learn and fun to use, especially when compared with its contemporaries like Java, C#, and C++. In the interest of full disclosure, though, I should also note up front some more abstract tradeoffs I've observed in my two decades in the Python world—both as an educator and developer.

*As an educator*, I've sometimes found the *rate of change* in Python and its libraries to be a negative, and have on occasion lamented its *growth* over the years. This is partly because trainers and book authors live on the front lines of such things—it's been my job to teach the language despite its constant change, a task at times akin to chronicling the herding of cats! Still, it's a broadly shared concern. As we'll see in this book, Python's original "keep it simple" motif is today often subsumed by a trend toward more sophisticated solutions at the expense of the learning curve of newcomers. This book's size is indirect evidence of this trend.

On the other hand, by most measures Python is still much simpler than its alternatives, and perhaps only as complex as it needs to be given the many roles it serves today. Its overall coherence and open nature remain compelling features to most. Moreover, not everyone needs to stay up to date with the cutting edge—as Python 2.X's ongoing popularity clearly shows.

*As a developer*, I also at times question the tradeoffs inherent in Python's "*batteries included*" approach to development. Its emphasis on prebuilt tools can add dependencies (what if a battery you use is changed, broken, or deprecated?), and encourage special-case solutions over general principles that may serve users better in the long run (how can you evaluate or use a tool well if you don't understand its purpose?). We'll see examples of both of these concerns in this book.

For typical users, and especially for hobbyists and beginners, Python's toolset approach is a major asset. But you shouldn't be surprised when you outgrow precoded tools, and can benefit from the sorts of skills this book aims to impart. Or, to paraphrase a proverb: give people a tool, and they'll code for a day; teach them how to build tools, and they'll code for a lifetime. This book's job is more the latter than the former.

As mentioned elsewhere in this chapter, both Python and its toolbox model are also susceptible to downsides common to *open source* projects in general—the potential triumph of the *personal preference* of the few over common usage of the many, and the occasional appearance of *anarchy* and even *elitism*—though these tend to be most grievous on the leading edge of new releases.

We'll return to some of these tradeoffs at the end of the book, after you've learned Python well enough to draw your own conclusions. As an open source system, what Python "is" is up to its users to define. In the end, Python is more popular today than ever, and its growth shows no signs of abating. To some, that may be a more telling metric than individual opinions, both pro and con.

## Who Uses Python Today?

At this writing, the best estimate anyone can seem to make of the size of the Python user base is that there are roughly 1 million Python users around the world today (plus or minus a few). This estimate is based on various statistics, like download rates, web statistics, and developer surveys. Because Python is open source, a more exact count is difficult—there are no license registrations to tally. Moreover, Python is automatically included with Linux distributions, Macintosh computers, and a wide range of products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a large user base and a very active developer community. It is generally considered to be in *the top 5 or top 10* most widely used programming languages in the world today (its exact ranking varies per source and date). Because Python has been around for *over two decades* and has been widely used, it is also very stable and robust.

Besides being leveraged by individual users, Python is also being applied in real revenue-generating products by real companies. For instance, among the generally known Python user base:

- *Google* makes extensive use of Python in its web search systems.
- The popular *YouTube* video sharing service is largely written in Python.
- The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
- The *Raspberry Pi* single-board computer promotes Python as its educational language.
- *EVE Online*, a massively multiplayer online game (MMOG) by CCP Games, uses Python broadly.
- The widespread *BitTorrent* peer-to-peer file sharing system began its life as a Python program.
- *Industrial Light & Magic*, *Pixar*, and others use Python in the production of animated movies.
- *ESRI* uses Python as an end-user customization tool for its popular GIS mapping products.
- Google's *App Engine* web development framework uses Python as an application language.
- The *IronPort* email server product uses more than 1 million lines of Python code to do its job.
- *Maya*, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
- The NSA uses Python for cryptography and intelligence analysis.
- *iRobot* uses Python to develop commercial and military robotic devices.

- The *Civilization IV* game’s customizable scripted events are written entirely in Python.
- The One Laptop Per Child (*OLPC*) project built its user interface and activity model in Python.
- *Netflix* and *Yelp* have both documented the role of Python in their software infrastructures.
- *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm*, and *IBM* use Python for hardware testing.
- *JPMorgan Chase*, *UBS*, *Getco*, and *Citadel* apply Python to financial market forecasting.
- *NASA*, *Los Alamos*, *Fermilab*, *JPL*, and others use Python for scientific programming tasks.

And so on—though this list is representative, a full accounting is beyond this book’s scope, and is almost guaranteed to change over time. For an up-to-date sampling of additional Python users, applications, and software, try the following pages currently at Python’s site and Wikipedia, as well as a search in your favorite web browser:

- Success stories: <http://www.python.org/about/success>
- Application domains: <http://www.python.org/about/apps>
- User quotes: <http://www.python.org/about/quotes>
- Wikipedia page: [http://en.wikipedia.org/wiki/List\\_of\\_Python\\_software](http://en.wikipedia.org/wiki/List_of_Python_software)

Probably the only common thread among the companies using Python today is that Python is used all over the map, in terms of application domains. Its general-purpose nature makes it applicable to almost all fields, not just one. In fact, it’s safe to say that virtually every substantial organization writing software is using Python, whether for short-term tactical tasks, such as testing and administration, or for long-term strategic product development. Python has proven to work well in both modes.

## What Can I Do with Python?

In addition to being a well-designed programming language, Python is useful for accomplishing real-world tasks—the sorts of things developers do day in and day out. It’s commonly used in a variety of domains, as a tool for scripting other components and implementing standalone programs. In fact, as a general-purpose language, Python’s roles are virtually unlimited: you can use it for everything from website development and gaming to robotics and spacecraft control.

However, the most common Python roles currently seem to fall into a few broad categories. The next few sections describe some of Python’s most common applications today, as well as tools used in each domain. We won’t be able to explore the tools

mentioned here in any depth—if you are interested in any of these topics, see the Python website or other resources for more details.

## Systems Programming

Python’s built-in interfaces to operating-system services make it ideal for writing portable, maintainable system-administration tools and utilities (sometimes called *shell tools*). Python programs can search files and directory trees, launch other programs, do parallel processing with processes and threads, and so on.

Python’s standard library comes with POSIX bindings and support for all the usual OS tools: environment variables, files, sockets, pipes, processes, multiple threads, regular expression pattern matching, command-line arguments, standard stream interfaces, shell-command launchers, filename expansion, zip file utilities, XML and JSON parsers, CSV file handlers, and more. In addition, the bulk of Python’s system interfaces are designed to be portable; for example, a script that copies directory trees typically runs unchanged on all major Python platforms. The *Stackless* Python implementation, described in [Chapter 2](#) and used by *EVE Online*, also offers advanced solutions to multiprocessing requirements.

## GUIs

Python’s simplicity and rapid turnaround also make it a good match for graphical user interface programming on the desktop. Python comes with a standard object-oriented interface to the Tk GUI API called *tkinter* (*Tkinter* in 2.X) that allows Python programs to implement portable GUIs with a native look and feel. Python/tkinter GUIs run unchanged on Microsoft Windows, X Windows (on Unix and Linux), and the Mac OS (both Classic and OS X). A free extension package, *PMW*, adds advanced widgets to the *tkinter* toolkit. In addition, the *wxPython* GUI API, based on a C++ library, offers an alternative toolkit for constructing portable GUIs in Python.

Higher-level toolkits such as *Dabo* are built on top of base APIs such as *wxPython* and *tkinter*. With the proper library, you can also use GUI support in other toolkits in Python, such as *Qt* with *PyQt*, *GTK* with *PyGTK*, *MFC* with *PyWin32*, *.NET* with *IronPython*, and *Swing* with *Jython* (the Java version of Python, described in [Chapter 2](#)) or *JPyte*. For applications that run in web browsers or have simple interface requirements, both *Jython* and Python web frameworks and server-side CGI scripts, described in the next section, provide additional user interface options.

## Internet Scripting

Python comes with standard Internet modules that allow Python programs to perform a wide variety of networking tasks, in client and server modes. Scripts can communicate over sockets; extract form information sent to server-side CGI scripts; transfer files by FTP; parse and generate XML and JSON documents; send, receive, compose, and parse

email; fetch web pages by URLs; parse the HTML of fetched web pages; communicate over XML-RPC, SOAP, and Telnet; and more. Python's libraries make these tasks remarkably simple.

In addition, a large collection of third-party tools are available on the Web for doing Internet programming in Python. For instance, the *HTMLGen* system generates HTML files from Python class-based descriptions, the *mod\_python* package runs Python efficiently within the Apache web server and supports server-side templating with its Python Server Pages, and the *Jython* system provides for seamless Python/Java integration and supports coding of server-side applets that run on clients.

In addition, full-blown web development framework packages for Python, such as *Django*, *TurboGears*, *web2py*, *Pylons*, *Zope*, and *WebWare*, support quick construction of full-featured and production-quality websites with Python. Many of these include features such as object-relational mappers, a Model/View/Controller architecture, server-side scripting and templating, and AJAX support, to provide complete and enterprise-level web development solutions.

More recently, Python has expanded into rich Internet applications (RIAs), with tools such as *Silverlight* in *IronPython*, and *pyjs* (a.k.a. *pyjamas*) and its Python-to-JavaScript compiler, AJAX framework, and widget set. Python also has moved into cloud computing, with *App Engine*, and others described in the database section ahead. Where the Web leads, Python quickly follows.

## Component Integration

We discussed the component integration role earlier when describing Python as a control language. Python's ability to be extended by and embedded in C and C++ systems makes it useful as a flexible glue language for scripting the behavior of other systems and components. For instance, integrating a C library into Python enables Python to test and launch the library's components, and embedding Python in a product enables onsite customizations to be coded without having to recompile the entire product (or ship its source code at all).

Tools such as the *SWIG* and *SIP* code generators can automate much of the work needed to link compiled components into Python for use in scripts, and the *Cython* system allows coders to mix Python and C-like code. Larger frameworks, such as Python's *COM* support on Windows, the *Jython* Java-based implementation, and the *IronPython* .NET-based implementation provide alternative ways to script components. On Windows, for example, Python scripts can use frameworks to script Word and Excel, access *Silverlight*, and much more.

## Database Programming

For traditional database demands, there are Python interfaces to all commonly used relational database systems—Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL,

SQLite, and more. The Python world has also defined a *portable database API* for accessing SQL database systems from Python scripts, which looks the same on a variety of underlying database systems. For instance, because the vendor interfaces implement the portable API, a script written to work with the free MySQL system will work largely unchanged on other systems (such as Oracle); all you generally have to do is replace the underlying vendor interface. The in-process *SQLite* embedded SQL database engine is a standard part of Python itself since 2.5, supporting both prototyping and basic program storage needs.

In the non-SQL department, Python's standard `pickle` module provides a simple object persistence system—it allows programs to easily save and restore entire Python objects to files and file-like objects. On the Web, you'll also find third-party open source systems named *ZODB* and *Durus* that provide complete object-oriented database systems for Python scripts; others, such as *SQLObject* and *SQLAlchemy*, that implement object relational mappers (ORMs), which graft Python's class model onto relational tables; and *PyMongo*, an interface to *MongoDB*, a high-performance, non-SQL, open source JSON-style document database, which stores data in structures very similar to Python's own lists and dictionaries, and whose text may be parsed and created with Python's own standard library `json` module.

Still other systems offer more specialized ways to store data, including the datastore in Google's *App Engine*, which models data with Python classes and provides extensive scalability, as well as additional emerging cloud storage options such as *Azure*, *Pi-Cloud*, *OpenStack*, and *Stackato*.

## Rapid Prototyping

To Python programs, components written in Python and C look the same. Because of this, it's possible to prototype systems in Python initially, and then move selected components to a compiled language such as C or C++ for delivery. Unlike some prototyping tools, Python doesn't require a complete rewrite once the prototype has solidified. Parts of the system that don't require the efficiency of a language such as C++ can remain coded in Python for ease of maintenance and use.

## Numeric and Scientific Programming

Python is also heavily used in numeric programming—a domain that would not traditionally have been considered to be in the scope of scripting languages, but has grown to become one of Python's most compelling use cases. Prominent here, the *NumPy* high-performance numeric programming extension for Python mentioned earlier includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more. By integrating Python with numeric routines coded in a compiled language for speed, NumPy turns Python into a sophisticated yet easy-to-use numeric programming tool that can often replace existing code written in traditional compiled languages such as FORTRAN or C++.

Additional numeric tools for Python support animation, 3D visualization, parallel processing, and so on. The popular *SciPy* and *ScientificPython* extensions, for example, provide additional libraries of scientific programming tools and use NumPy as a core component. The *PyPy* implementation of Python (discussed in [Chapter 2](#)) has also gained traction in the numeric domain, in part because heavily algorithmic code of the sort that's common in this domain can run dramatically faster in PyPy—often 10X to 100X quicker.

## And More: Gaming, Images, Data Mining, Robots, Excel...

Python is commonly applied in more domains than can be covered here. For example, you'll find tools that allow you to use Python to do:

- Game programming and multimedia with *pygame*, *cokit*, *pyglet*, *PySoy*, *Panda3D*, and others
- Serial port communication on Windows, Linux, and more with the *PySerial* extension
- Image processing with *PIL* and its newer *Pillow* fork, *PyOpenGL*, *Blender*, *Maya*, and more
- Robot control programming with the *PyRo* toolkit
- Natural language analysis with the *NLTK* package
- Instrumentation on the *Raspberry Pi* and *Arduino* boards
- Mobile computing with ports of Python to the Google *Android* and Apple *iOS* platforms
- Excel spreadsheet function and macro programming with the *PyXLL* or *DataNitro* add-ins
- Media file content and metadata tag processing with *PyMedia*, *ID3*, *PIL/Pillow*, and more
- Artificial intelligence with the *PyBrain* neural net library and the *Milk* machine learning toolkit
- Expert system programming with *PyCLIPS*, *Pyke*, *Pyrolog*, and *pyDatalog*
- Network monitoring with *zenoss*, written in and customized with Python
- Python-scripted design and modeling with *PythonCAD*, *PythonOCC*, *FreeCAD*, and others
- Document processing and generation with *ReportLab*, *Sphinx*, *Cheetah*, *PyPDF*, and so on
- Data visualization with *Mayavi*, *matplotlib*, *VTK*, *VPython*, and more
- XML parsing with the *xml* library package, the *xmlrpclib* module, and third-party extensions
- JSON and CSV file processing with the *json* and *csv* modules

- Data mining with the *Orange* framework, the *Pattern* bundle, *Scrapy*, and custom code

You can even play solitaire with the *PySolFC* program. And of course, you can always code custom Python scripts in less buzzword-laden domains to perform day-to-day system administration, process your email, manage your document and media libraries, and so on. You'll find links to the support in many fields at the PyPI website, and via web searches (search Google or <http://www.python.org> for links).

Though of broad practical use, many of these specific domains are largely just instances of Python's component integration role in action again. Adding it as a frontend to libraries of components written in a compiled language such as C makes Python useful for scripting in a wide variety of domains. As a general-purpose language that supports integration, Python is widely applicable.

## How Is Python Developed and Supported?

As a popular open source system, Python enjoys a large and active development community that responds to issues and develops enhancements with a speed that many commercial software developers might find remarkable. Python developers coordinate work online with a source-control system. Changes are developed per a formal protocol, which includes writing a *PEP* (Python Enhancement Proposal) or other document, and extensions to Python's regression testing system. In fact, modifying Python today is roughly as involved as changing commercial software—a far cry from Python's early days, when an email to its creator would suffice, but a good thing given its large user base today.

The *PSF* (Python Software Foundation), a formal nonprofit group, organizes conferences and deals with intellectual property issues. Numerous Python conferences are held around the world; O'Reilly's *OSCON* and the *PSF's PyCon* are the largest. The former of these addresses multiple open source projects, and the latter is a Python-only event that has experienced strong growth in recent years. *PyCon* 2012 and 2013 reached 2,500 *attendees* each; in fact, *PyCon* 2013 had to cap its limit at this level after a surprise sell-out in 2012 (and managed to grab wide attention on both technical and nontechnical grounds that I won't chronicle here). Earlier years often saw attendance double—from 586 attendees in 2007 to over 1,000 in 2008, for example—indicative of Python's growth in general, and impressive to those who remember early conferences whose attendees could largely be served around a single restaurant table.

## Open Source Tradeoffs

Having said that, it's important to note that while Python enjoys a vigorous development community, this comes with inherent tradeoffs. Open source software can also appear chaotic and even resemble *anarchy* at times, and may not always be as smoothly implemented as the prior paragraphs might imply. Some changes may still manage to

defy official protocols, and as in all human endeavors, mistakes still happen despite the process controls (Python 3.2.0, for instance, came with a broken console `input` function on Windows).

Moreover, open source projects exchange commercial interests for the *personal preferences* of a current set of developers, which may or may not be the same as yours—you are not held hostage by a company, but you are at the mercy of those with spare time to change the system. The net effect is that open source software evolution is often driven by the few, but imposed on the many.

In practice, though, these tradeoffs impact those on the “bleeding” edge of new releases much more than those using established versions of the system, including prior releases in both Python 3.X and 2.X. If you kept using classic classes in Python 2.X, for example, you were largely immune to the *explosion* of class functionality and change in new-style classes that occurred in the early-to-mid 2000s. Though these become mandatory in 3.X (along with much more), many 2.X users today still happily skirt the issue.

## What Are Python’s Technical Strengths?

Naturally, this is a developer’s question. If you don’t already have a programming background, the language in the next few sections may be a bit baffling—don’t worry, we’ll explore all of these terms in more detail as we proceed through this book. For developers, though, here is a quick introduction to some of Python’s top technical features.

### It’s Object-Oriented and Functional

Python is an object-oriented language, from the ground up. Its *class model* supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet, in the context of Python’s simple syntax and typing, OOP is remarkably easy to apply. In fact, if you don’t understand these terms, you’ll find they are much easier to learn with Python than with just about any other OOP language available.

Besides serving as a powerful code structuring and reuse device, Python’s OOP nature makes it ideal as a *scripting tool* for other object-oriented systems languages. For example, with the appropriate glue code, Python programs can subclass (specialize) classes implemented in C++, Java, and C#.

Of equal significance, OOP is an *option* in Python; you can go far without having to become an object guru all at once. Much like C++, Python supports both procedural and object-oriented programming modes. Its object-oriented tools can be applied if and when constraints allow. This is especially useful in tactical development modes, which preclude design phases.

In addition to its original *procedural* (statement-based) and *object-oriented* (class-based) paradigms, Python in recent years has acquired built-in support for *functional*

*programming*—a set that by most measures includes generators, comprehensions, closures, maps, decorators, anonymous function lambdas, and first-class function objects. These can serve as both complement and alternative to its OOP tools.

## It's Free

Python is completely free to use and distribute. As with other open source software, such as Tcl, Perl, Linux, and Apache, you can fetch the entire Python system's source code for free on the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can even sell Python's source code, if you are so inclined.

But don't get the wrong idea: "free" doesn't mean "unsupported." On the contrary, the Python online community responds to user queries with a speed that most commercial software help desks would do well to try to emulate. Moreover, because Python comes with complete source code, it empowers developers, leading to the creation of a large team of implementation experts. Although studying or changing a programming language's implementation isn't everyone's idea of fun, it's comforting to know that you can do so if you need to. You're not dependent on the whims of a commercial vendor, because the ultimate documentation—*source code*—is at your disposal as a last resort.

As mentioned earlier, Python development is performed by a community that largely coordinates its efforts over the Internet. It consists of Python's original creator—*Guido van Rossum*, the officially anointed Benevolent Dictator for Life (BDFL) of Python—plus a supporting cast of thousands. Language changes must follow a formal enhancement procedure and be scrutinized by both other developers and the BDFL. This tends to make Python more conservative with changes than some other languages and systems. While the Python 3.X/2.X split broke with this tradition soundly and deliberately, it still holds generally true within each Python line.

## It's Portable

The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from PDAs to supercomputers. As a partial list, Python is available on:

- Linux and Unix systems
- Microsoft Windows (all modern flavors)
- Mac OS (both OS X and Classic)
- BeOS, OS/2, VMS, and QNX
- Real-time systems such as VxWorks
- Cray supercomputers and IBM mainframes

- PDAs running Palm OS, PocketPC, and Linux
- Cell phones running Symbian OS, and Windows Mobile
- Gaming consoles and iPods
- Tablets and smartphones running Google's Android and Apple's iOS
- And more

Like the language interpreter itself, the standard library modules that ship with Python are implemented to be as portable across platform boundaries as possible. Further, Python programs are automatically compiled to portable byte code, which runs the same on any platform with a compatible version of Python installed (more on this in the next chapter).

What that means is that Python programs using the core language and standard libraries run the same on Linux, Windows, and most other systems with a Python interpreter. Most Python ports also contain platform-specific extensions (e.g., COM support on Windows), but the core Python language and libraries work the same everywhere. As mentioned earlier, Python also includes an interface to the Tk GUI toolkit called `tkinter` (`Tkinter` in 2.X), which allows Python programs to implement full-featured graphical user interfaces that run on all major GUI desktop platforms without program changes.

## It's Powerful

From a features perspective, Python is something of a hybrid. Its toolset places it between traditional scripting languages (such as Tcl, Scheme, and Perl) and systems development languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced software-engineering tools typically found in compiled languages. Unlike some scripting languages, this combination makes Python useful for large-scale development projects. As a preview, here are some of the main things you'll find in Python's toolbox:

### *Dynamic typing*

Python keeps track of the kinds of objects your program uses when it runs; it doesn't require complicated type and size declarations in your code. In fact, as you'll see in [Chapter 6](#), there is no such thing as a type or variable declaration anywhere in Python. Because Python code does not constrain data types, it is also usually automatically applicable to a whole range of objects.

### *Automatic memory management*

Python automatically allocates objects and reclaims ("garbage collects") them when they are no longer used, and most can grow and shrink on demand. As you'll learn, Python keeps track of low-level memory details so you don't have to.

### *Programming-in-the-large support*

For building larger systems, Python includes tools such as modules, classes, and exceptions. These tools allow you to organize systems into components, use OOP

to reuse and customize code, and handle events and errors gracefully. Python’s functional programming tools, described earlier, provide additional ways to meet many of the same goals.

#### *Built-in object types*

Python provides commonly used data structures such as lists, dictionaries, and strings as intrinsic parts of the language; as you’ll see, they’re both flexible and easy to use. For instance, built-in objects can grow and shrink on demand, can be arbitrarily nested to represent complex information, and more.

#### *Built-in tools*

To process all those object types, Python comes with powerful and standard operations, including concatenation (joining collections), slicing (extracting sections), sorting, mapping, and more.

#### *Library utilities*

For more specific tasks, Python also comes with a large collection of pre-coded library tools that support everything from regular expression matching to networking. Once you learn the language itself, Python’s library tools are where much of the application-level action occurs.

#### *Third-party utilities*

Because Python is open source, developers are encouraged to contribute pre-coded tools that support tasks beyond those supported by its built-ins; on the Web, you’ll find free support for COM, imaging, numeric programming, XML, database access, and much more.

Despite the array of tools in Python, it retains a remarkably simple syntax and design. The result is a powerful programming tool with all the usability of a scripting language.

## **It’s Mixable**

Python programs can easily be “glued” to components written in other languages in a variety of ways. For example, Python’s C API lets C programs call and be called by Python programs flexibly. That means you can add functionality to the Python system as needed, and use Python programs within other environments or systems.

Mixing Python with libraries coded in languages such as C or C++, for instance, makes it an easy-to-use frontend language and customization tool. As mentioned earlier, this also makes Python good at rapid prototyping—systems may be implemented in Python first, to leverage its speed of development, and later moved to C for delivery, one piece at a time, according to performance demands.

## **It’s Relatively Easy to Use**

Compared to alternatives like C++, Java, and C#, Python programming seems astonishingly simple to most observers. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps, like there are for languages

such as C or C++. Python executes programs immediately, which makes for an interactive programming experience and *rapid turnaround* after program changes—in many cases, you can witness the effect of a program change nearly as fast as you can type it.

Of course, development cycle turnaround is only one aspect of Python’s ease of use. It also provides a deliberately simple syntax and powerful built-in tools. In fact, some have gone so far as to call Python *executable pseudocode*. Because it eliminates much of the complexity in other tools, Python programs are simpler, smaller, and more flexible than equivalent programs in other popular languages.

## It’s Relatively Easy to Learn

This brings us to the point of this book: especially when compared to other widely used programming languages, the core Python language is remarkably easy to learn. In fact, if you’re an experienced programmer, you can expect to be coding small-scale Python programs in a matter of days, and may be able to pick up some limited portions of the language in just hours—though you shouldn’t expect to become an expert quite that fast (despite what you may have heard from marketing departments!).

Naturally, mastering any topic as substantial as today’s Python is not trivial, and we’ll devote the rest of this book to this task. But the true investment required to master Python is worthwhile—in the end, you’ll gain programming skills that apply to nearly every computer application domain. Moreover, most find Python’s learning curve to be much gentler than that of other programming tools.

That’s good news for professional developers seeking to learn the language to use on the job, as well as for end users of systems that expose a Python layer for customization or control. Today, many systems rely on the fact that end users can learn enough Python to tailor their Python customization code onsite, with little or no support. Moreover, Python has spawned a large group of users who program for fun instead of career, and may never need full-scale software development skills. Although Python does have advanced programming tools, its core language essentials will still seem relatively simple to beginners and gurus alike.

## It’s Named After Monty Python

OK, this isn’t quite a technical strength, but it does seem to be a surprisingly well-kept secret in the Python world that I wish to expose up front. Despite all the reptiles on Python books and icons, the truth is that Python is named after the British comedy group *Monty Python*—makers of the 1970s BBC comedy series *Monty Python’s Flying Circus* and a handful of later full-length films, including *Monty Python and the Holy Grail*, that are still widely popular today. Python’s original creator was a fan of Monty Python, as are many software developers (indeed, there seems to be a sort of symmetry between the two fields...).

This legacy inevitably adds a humorous quality to Python code examples. For instance, the traditional “foo” and “bar” for generic variable names become “spam” and “eggs” in the Python world. The occasional “Brian,” “ni,” and “shrubbery” likewise owe their appearances to this namesake. It even impacts the Python community at large: some events at Python conferences are regularly billed as “The Spanish Inquisition.”

All of this is, of course, very funny if you are familiar with the shows, but less so otherwise. You don’t need to be familiar with Monty Python’s work to make sense of examples that borrow references from it, including many you will see in this book, but at least you now know their root. (Hey—I’ve warned you.)

## How Does Python Stack Up to Language X?

Finally, to place it in the context of what you may already know, people sometimes compare Python to languages such as Perl, Tcl, and Java. This section summarizes common consensus in this department.

I want to note up front that I’m not a fan of winning by disparaging the competition—it doesn’t work in the long run, and that’s not the goal here. Moreover, this is not a zero sum game—most programmers will use many languages over their careers. Nevertheless, programming tools present choices and tradeoffs that merit consideration. After all, if Python didn’t offer something over its alternatives, it would never have been used in the first place.

We talked about performance tradeoffs earlier, so here we’ll focus on functionality. While other languages are also useful tools to know and use, many people find that Python:

- Is more powerful than *Tcl*. Python’s strong support for “programming in the large” makes it applicable to the development of larger systems, and its library of application tools is broader.
- Is more readable than *Perl*. Python has a clear syntax and a simple, coherent design. This in turn makes Python more reusable and maintainable, and helps reduce program bugs.
- Is simpler and easier to use than *Java* and *C#*. Python is a scripting language, but Java and C# both inherit much of the complexity and syntax of larger OOP systems languages like C++.
- Is simpler and easier to use than C++. Python code is simpler than the equivalent C++ and often one-third to one-fifth as large, though as a scripting language, Python sometimes serves different roles.
- Is simpler and higher-level than C. Python’s detachment from underlying hardware architecture makes code less complex, better structured, and more approachable than C, C++’s progenitor.

- Is more powerful, general-purpose, and cross-platform than *Visual Basic*. Python is a richer language that is used more widely, and its open source nature means it is not controlled by a single company.
- Is more readable and general-purpose than *PHP*. Python is used to construct websites too, but it is also applied to nearly every other computer domain, from robotics to movie animation and gaming.
- Is more powerful and general-purpose than *JavaScript*. Python has a larger toolset, and is not as tightly bound to web development. It's also used for scientific modeling, instrumentation, and more.
- Is more readable and established than *Ruby*. Python syntax is less cluttered, especially in nontrivial code, and its OOP is fully optional for users and projects to which it may not apply.
- Is more mature and broadly focused than *Lua*. Python's larger feature set and more extensive library support give it a wider scope than Lua, an embedded "glue" language like Tcl.
- Is less esoteric than *Smalltalk*, *Lisp*, and *Prolog*. Python has the dynamic flavor of languages like these, but also has a traditional syntax accessible to both developers and end users of customizable systems.

Especially for programs that do more than scan text files, and that might have to be read in the future by others (or by you!), many people find that Python fits the bill better than any other scripting or programming language available today. Furthermore, unless your application requires peak performance, Python is often a viable alternative to systems development languages such as C, C++, and Java: Python code can often achieve the same goals, but will be much less difficult to write, debug, and maintain.

Of course, your author has been a card-carrying Python evangelist since 1992, so take these comments as you may (and other languages' advocates' mileage may vary arbitrarily). They do, however, reflect the common experience of many developers who have taken time to explore what Python has to offer.

## Chapter Summary

And that concludes the "hype" portion of this book. In this chapter, we've explored some of the reasons that people pick Python for their programming tasks. We've also seen how it is applied and looked at a representative sample of who is using it today. My goal is to teach Python, though, not to sell it. The best way to judge a language is to see it in action, so the rest of this book focuses entirely on the language details we've glossed over here.

The next two chapters begin our technical introduction to the language. In them, we'll explore ways to run Python programs, peek at Python's byte code execution model, and introduce the basics of module files for saving code. The goal will be to give you

just enough information to run the examples and exercises in the rest of the book. You won't really start programming per se until [Chapter 4](#), but make sure you have a handle on the startup details before moving on.

## Test Your Knowledge: Quiz

In this edition of the book, we will be closing each chapter with a quick open-book quiz about the material presented herein to help you review the key concepts. The answers for these quizzes appear immediately after the questions, and you are encouraged to read the answers once you've taken a crack at the questions yourself, as they sometimes give useful context.

In addition to these end-of-chapter quizzes, you'll find lab *exercises* at the end of each part of the book, designed to help you start coding Python on your own. For now, here's your first quiz. Good luck, and be sure to refer back to this chapter's material as needed.

1. What are the six main reasons that people choose to use Python?
2. Name four notable companies or organizations using Python today.
3. Why might you *not* want to use Python in an application?
4. What can you do with Python?
5. What's the significance of the Python `import this` statement?
6. Why does "spam" show up in so many Python examples in books and on the Web?
7. What is your favorite color?

## Test Your Knowledge: Answers

How did you do? Here are the answers I came up with, though there may be multiple solutions to some quiz questions. Again, even if you're sure of your answer, I encourage you to look at mine for additional context. See the chapter's text for more details if any of these responses don't make sense to you.

1. Software quality, developer productivity, program portability, support libraries, component integration, and simple enjoyment. Of these, the quality and productivity themes seem to be the main reasons that people choose to use Python.
2. Google, Industrial Light & Magic, CCP Games, Jet Propulsion Labs, Maya, ESRI, and many more. Almost every organization doing software development uses Python in some fashion, whether for long-term strategic product development or for short-term tactical tasks such as testing and system administration.
3. Python's main downside is performance: it won't run as quickly as fully compiled languages like C and C++. On the other hand, it's quick enough for most applications, and typical Python code runs at close to C speed anyhow because it invokes

linked-in C code in the interpreter. If speed is critical, compiled extensions are available for number-crunching parts of an application.

4. You can use Python for nearly anything you can do with a computer, from website development and gaming to robotics and spacecraft control.
5. This was mentioned in a footnote: `import this` triggers an Easter egg inside Python that displays some of the design philosophies underlying the language. You'll learn how to run this statement in the next chapter.
6. "Spam" is a reference from a famous Monty Python skit in which people trying to order food in a cafeteria are drowned out by a chorus of Vikings singing about spam. Oh, and it's also a common variable name in Python scripts...
7. Blue. No, yellow! (See the prior answer.)

## Python Is Engineering, Not Art

When Python first emerged on the software scene in the early 1990s, it spawned what is now something of a classic conflict between its proponents and those of another popular scripting language, Perl. Personally, I think the debate is tired and unwarranted today—developers are smart enough to draw their own conclusions. Still, this is one of the most common topics I'm asked about on the training road, and underscores one of the main reasons people choose to use Python; it seems fitting to say a few brief words about it here.

The short story is this: *you can do everything in Python that you can in Perl, but you can read your code after you do it.* That's it—their domains largely overlap, but Python is more focused on producing readable code. For many, the enhanced readability of Python translates to better code reusability and maintainability, making Python a better choice for programs that will not be written once and thrown away. Perl code is easy to write, but can be difficult to read. Given that most software has a lifespan much longer than its initial creation, many see Python as the more effective tool.

The somewhat longer story reflects the backgrounds of the designers of the two languages. *Python* originated with a mathematician by training, who seems to have naturally produced an orthogonal language with a high degree of uniformity and coherence. *Perl* was spawned by a linguist, who created a programming tool closer to natural language, with its context sensitivities and wide variability. As a well-known Perl motto states, *there's more than one way to do it*. Given this mindset, both the Perl language and its user community have historically encouraged untethered freedom of expression when writing code. One person's Perl code can be radically different from another's. In fact, writing unique, tricky code is often a source of pride among Perl users.

But as anyone who has done any substantial code maintenance should be able to attest, *freedom of expression is great for art, but lousy for engineering*. In engineering, we need a minimal feature set and predictability. In engineering, freedom of expression can lead to maintenance nightmares. As more than one Perl user has confided to me, the result of too much freedom is often code that is much easier to rewrite from scratch than to modify. This is clearly less than ideal.

Consider this: when people create a painting or a sculpture, they do so largely for themselves; the prospect of someone else changing their work later doesn't enter into it. This is a critical difference between art and engineering. When people write *software*, they are not writing it for themselves. In fact, they are not even writing primarily for the computer. Rather, good programmers know that code is written for the next human being who has to read it in order to maintain or reuse it. If that person cannot understand the code, it's all but useless in a realistic development scenario. In other words, programming is not about being clever and obscure—it's about how clearly your program communicates its purpose.

This readability focus is where many people find that Python most clearly differentiates itself from other scripting languages. Because Python's syntax model almost *forces* the creation of readable code, Python programs lend themselves more directly to the full software development cycle. And because Python emphasizes ideas such as limited interactions, code uniformity, and feature consistency, it more directly fosters code that can be used long after it is first written.

In the long run, Python's focus on *code quality* in itself boosts programmer productivity, as well as programmer satisfaction. Python programmers can be wildly creative, too, of course, and as we'll see, the language does offer multiple solutions for some tasks—sometimes even more than it should today, an issue we'll confront head-on in this book too. In fact, this sidebar can also be read as a *cautionary tale*: quality turns out to be a fragile state, one that depends as much on *people* as on technology. Python has historically encouraged good engineering in ways that other scripting languages often did not, but the rest of the quality story is up to you.

At least, that's some of the common consensus among many people who have adopted Python. You should judge such claims for yourself, of course, by learning what Python has to offer. To help you get started, let's move on to the next chapter.

# O'Reilly Ebooks—Your bookshelf on your devices!



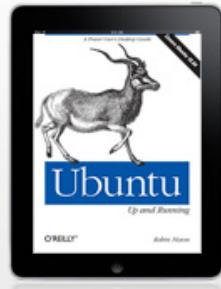
PDF



ePub



Mobi



APK



DAISY

When you buy an ebook through [oreilly.com](http://oreilly.com) you get lifetime access to the book, and whenever possible we provide it to you in five, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, Android .apk, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at [ebooks.oreilly.com](http://ebooks.oreilly.com)

You can also purchase O'Reilly ebooks through the iBookstore, the [Android Marketplace](#), and [Amazon.com](#).

**O'REILLY®**

Spreading the knowledge of innovators

[oreilly.com](http://oreilly.com)