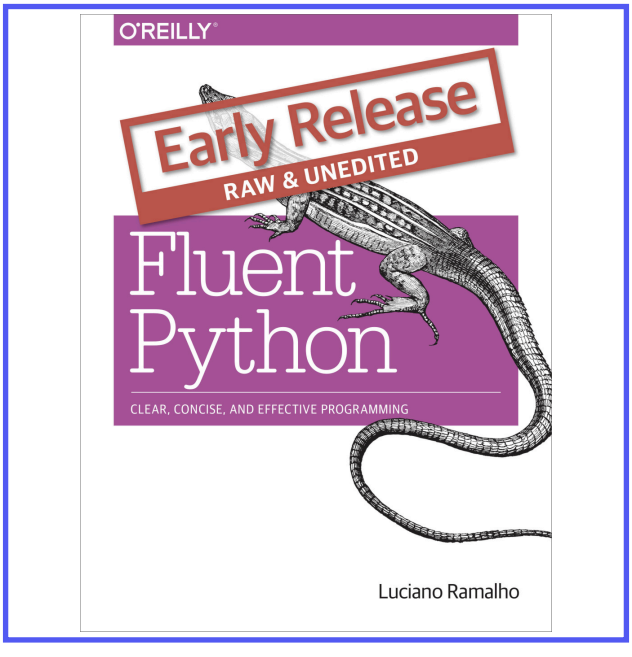


ebookbig.com

or scan the QR code to access the digital study portal.

Table of Contents

Preface xv

Part I. Prologue

1. The Python Data Model 3
A Pythonic Card Deck 4
How special methods are used 8
Emulating numeric types 9
String representation 11
Arithmetic operators 11
Boolean value of a custom type 12
Overview of special methods 12
Why `len` is not a method 14
Chapter summary 14
Further reading 15

Part II. Data structures

2. An array of sequences 19
Overview of built-in sequences 20
List comprehensions and generator expressions 21
List comprehensions and readability 21
Listcomps versus `map` and `filter` 23
Cartesian products 23
Generator expressions 25
Tuples are not just immutable lists 26
Tuples as records 26
Tuple unpacking 27

Nested tuple unpacking 29
Named tuples 30
Tuples as immutable lists 32
Slicing 33
Why slices and range exclude the last item 33
Slice objects 34
Multi-dimensional slicing and ellipsis 35
Assigning to slices 36
Using + and * with sequences 36
Building lists of lists 37
Augmented assignment with sequences 38
A += assignment puzzler 40
`list.sort` and the `sorted` built-in function 42
Managing ordered sequences with `bisect` 44
Searching with `bisect` 44
Inserting with `bisect.insort` 46
When a list is not the answer 47
Arrays 48
Memory views 51
NumPy and SciPy 52
Deques and other queues 54
Chapter summary 57
Further reading 58

3. Dictionaries and sets. 63
Generic mapping types 64
`dict` comprehensions 66
Overview of common mapping methods 66
Handling missing keys with `setdefault` 68
Mappings with flexible key lookup 70
`defaultdict`: another take on missing keys 71
The `__missing__` method 72
Variations of `dict` 75
Subclassing `UserDict`. 76
Immutable mappings 77
Set theory 79
`set` literals 80
`set` comprehensions 81
Set operations 82
`dict` and `set` under the hood 85
A performance experiment 85
Hash tables in dictionaries 87

Practical consequences of how `dict` works 90
How sets work — practical consequences 93
Chapter summary 93
Further reading 94

4. Text versus bytes. 97
Character issues 98
Byte essentials 99
Structs and memory views 102
Basic encoders/decoders 103
Understanding encode/decode problems 105
Coping with `UnicodeEncodeError` 105
Coping with `UnicodeDecodeError` 106
`SyntaxError` when loading modules with unexpected encoding 107
How to discover the encoding of a byte sequence 108
BOM: a useful gremlin 109
Handling text files 110
Encoding defaults: a madhouse 113
Normalizing Unicode for saner comparisons 116
Case folding 119
Utility functions for normalized text matching 120
Extreme "normalization": taking out diacritics 121
Sorting Unicode text 124
Sorting with the Unicode Collation Algorithm 126
The Unicode database 126
Dual mode `str` and `bytes` APIs 128
`str` versus `bytes` in regular expressions 129
`str` versus `bytes` on `os` functions 130
Chapter summary 132
Further reading 133

Part III. Functions as objects

5. First-class functions. 139
Treating a function like an object 140
Higher-order functions 141
Modern replacements for `map`, `filter` and `reduce` 142
Anonymous functions 143
The seven flavors of callable objects 144
User defined callable types 145
Function introspection 147

From positional to keyword-only parameters 148
Retrieving information about parameters 150
Function annotations 154
Packages for functional programming 156
The `operator` module 156
Freezing arguments with `functools.partial` 159
Chapter summary 161
Further reading 162

6. Design patterns with first-class functions. 167
Case study: refactoring Strategy 168
Classic Strategy 168
Function-oriented Strategy 172
Choosing the best strategy: simple approach 175
Finding strategies in a module 176
Command 177
Chapter summary 179
Further reading 180

7. Function decorators and closures. 183
Decorators 101 184
When Python executes decorators 185
Decorator-enhanced Strategy pattern 187
Variable scope rules 189
Closures 192
The `nonlocal` declaration 195
Implementing a simple decorator 197
How it works 198
Decorators in the standard library 200
Memoization with `functools.lru_cache` 200
Generic functions with single dispatch 202
Stacked decorators 205
Parametrized Decorators 206
A parametrized registration decorator 206
The parametrized `clock` decorator 209
Chapter summary 211
Further reading 212

Part IV. Object Oriented Idioms

8. Object references, mutability and recycling. 219
Variables are not boxes 220
Identity, equality and aliases 221
Choosing between == and `is` 223
The relative immutability of tuples 224
Copies are shallow by default 225
Deep and shallow copies of arbitrary objects 227
Function parameters as references 229
Mutable types as parameter defaults: bad idea 230
Defensive programming with mutable parameters 232
`del` and garbage collection 234
Weak references 236
The `WeakValueDictionary` skit 237
Limitations of weak references 239
Tricks Python plays with immutables 240
Chapter summary 242
Further reading 243

9. A Pythonic object. 247
Object representations 248
`Vector` class redux 248
An alternative constructor 251
`classmethod` versus `staticmethod` 252
Formatted displays 253
A hashable `Vector2d` 257
Private and "protected" attributes in Python 263
Saving space with the `__slots__` class attribute 265
The problems with `__slots__` 267
Overriding class attributes 268
Chapter summary 270
Further reading 271

10. Sequence hacking, hashing and slicing. 277
`Vector`: a user-defined sequence type 278
`Vector` take #1: `Vector2d` compatible 278
Protocols and duck typing 281
`Vector` take #2: a sliceable sequence 282
How slicing works 283
A slice-aware `__getitem__` 285

`Vector` take #3: dynamic attribute access 286
`Vector` take #4: hashing and a faster == 290
`Vector` take #5: formatting 296
Chapter summary 303
Further reading 304

11. Interfaces: from protocols to ABCs. 309
Interfaces and protocols in Python culture 310
Python digs sequences 312
Monkey-patching to implement a protocol at run time 314
Waterfowl and ABCs 316
Subclassing an ABC 321
ABCs in the standard library. 323
ABCs in `collections.abc` 323
The `numbers` tower of ABCs 324
Defining and using an ABC 325
ABC syntax details 330
Subclassing the `Tombola` ABC 331
A virtual subclass of `Tombola` 333
How the `Tombola` subclasses were tested 336
Usage of `register` in practice 339
Geese can behave as ducks 340
Chapter summary 341
Further reading 343

12. Inheritance: for good or for worse. 349
Subclassing built-in types is tricky 350
Multiple inheritance and method resolution order 353
Multiple inheritance in the real world 358
Coping with multiple inheritance 360
1. Distinguish interface inheritance from implementation inheritance 361
2. Make interfaces explicit with ABCs 361
3. Use mixins for code reuse 361
4. Make mixins explicit by naming 361
5. An ABC may also be a mixin; the reverse is not true 361
6. Don't subclass from more than one concrete class 362
7. Provide aggregate classes to users 362
8. "Favor object composition over class inheritance." 363
Tkinter: the good, the bad and the ugly 363
A modern example: mixins in Django generic views 364
Chapter summary 368
Further reading 369

13. Operator overloading: doing it right. 373
Operator overloading 101 374
Unary operators 374
Overloading + for vector addition 377
Overloading * for scalar multiplication 382
Rich comparison operators 386
Augmented assignment operators 390
Chapter summary 394
Further reading 395

Part V. Control flow

14. Iterables, iterators and generators. 403
`Sentence` take #1: a sequence of words 404
Why sequences are iterable: the `iter` function 406
Iterables versus iterators 407
`Sentence` take #2: a classic iterator 411
Making `Sentence` an iterator: bad idea 413
`Sentence` take #3: a generator function 414
How a generator function works 415
`Sentence` take #4: a lazy implementation 418
`Sentence` take #5: a generator expression 419
Generator expressions: when to use them 421
Another example: arithmetic progression generator 422
Arithmetic progression with `itertools` 425
Generator functions in the standard library 426
New syntax in Python 3.3: `yield from` 435
Iterable reducing functions 436
A closer look at the `iter` function 438
Case study: generators in a database conversion utility 439
Generators as coroutines 441
Chapter summary 442
Further reading 442

15. Context managers and `else` blocks. 449
Do this, then that: `else` blocks beyond `if` 450
Context managers and `with` blocks 452
The `contextlib` utilities 456
Using `@contextmanager` 457
Chapter summary 461
Further reading 461

16. Coroutines. 465
How coroutines evolved from generators 466
Basic behavior of a generator used as a coroutine 466
Example: coroutine to compute a running average 470
Decorators for coroutine priming 471
Coroutine termination and exception handling 473
Returning a value from a coroutine 477
Using `yield from` 479
The meaning of `yield from` 485
Use case: coroutines for discrete event simulation 491
About discrete event simulations 491
The taxi fleet simulation 492
Chapter summary 500
Further reading 502

17. Concurrency with futures. 507
Example: Web downloads in three styles 507
A sequential download script 509
Downloading with `concurrent.futures` 511
Where are the futures? 513
Blocking I/O and the GIL 517
Launching processes with `concurrent.futures` 517
Experimenting with `Executor.map` 519
Downloads with progress display and error handling 522
Error handling in the `flags2` examples 527
Using `futures.as_completed` 529
Threading and multiprocessing alternatives. 532
Chapter Summary 532
Further reading 533

18. Concurrency with `asyncio`. 539
Thread versus coroutine: a comparison 541
`asyncio.Future`: non-blocking by design 547
Yielding from futures, tasks and coroutines 548
Downloading with `asyncio` and `aiohttp` 550
Running circles around blocking calls 554
Enhancing the `asyncio` downloader script 556
Using `asyncio.as_completed` 557
Using an executor to avoid blocking the event loop 562
From callbacks to futures and coroutines 564
Doing multiple requests for each download 566
Writing `asyncio` servers 569

An asyncio TCP server 570
An aiohttp Web server 575
Smarter clients for better concurrency 578
Chapter Summary 579
Further reading 580

Part VI. Metaprogramming

19. Dynamic attributes and properties. 587
Data wrangling with dynamic attributes 588
Exploring JSON-like data with dynamic attributes 590
The invalid attribute name problem 593
Flexible object creation with __new__ 594
Restructuring the OSCON feed with shelve 596
Linked record retrieval with properties 600
Using a property for attribute validation 606
LineItem take #1: class for an item in an order 606
LineItem take #2: a validating property 607
A proper look at properties 609
Properties override instance attributes 610
Property documentation 612
Coding a property factory 613
Handling attribute deletion 616
Essential attributes and functions for attribute handling 618
Special attributes that affect attribute handling 618
Built-in functions for attribute handling 618
Special methods for attribute handling 619
Chapter summary 621
Further reading 621

20. Attribute descriptors. 627
Descriptor example: attribute validation 627
LineItem take #3: a simple descriptor 628
LineItem take #4: automatic storage attribute names 633
LineItem take #5: a new descriptor type 639
Overriding versus non-overriding descriptors 642
Overriding descriptor 644
Overriding descriptor without __get__ 645
Non-overriding descriptor 646
Overwriting a descriptor in the class 647
Methods are descriptors 648