Python: Zero to Hero

Abhilash Sahoo

2026

Python: Zero to Hero

Contents

Copyright 1

Dedication 2

Epigraph 3

Preface 4

Acknowledgments 5

Introduction 6
- 0.1 What you'll learn . . . 6
- 0.2 How this book is organized . . . 6
- 0.3 How to read this book . . . 6

1 Chapter 1: Your First Python Program 8
- 1.1 What is Python? . . . 8
- 1.2 How Python Runs Your Code . . . 9
- 1.3 Three Ways to Run Python . . . 9
  - 1.3.1 Option A: Install Python on Your Computer . . . 9
  - 1.3.2 Option B: Run Python in Your Browser (Google Colab or Kaggle) . . . 10
  - 1.3.3 Option C: VS Code with Jupyter . . . 11
- 1.4 Your First Program . . . 12
- 1.5 Reading Error Messages Without Panic . . . 12
- 1.6 How to Use This Book . . . 14
- 1.7 What You Learned in This Chapter . . . 14
- 1.8 What's Next? . . . 14

2 Chapter 2: Variables, Numbers, and Text 16
- 2.1 What is a Variable? . . . 16
- 2.2 Naming Rules and Conventions . . . 17
- 2.3 Storing Numbers: Integers and Floats . . . 18
- 2.4 Storing Text: Strings . . . 19
- 2.5 Escape Sequences and Special Characters . . . 20
- 2.6 Multiline Strings . . . 21

2.7 Checking the Type of a Variable . . . 22
2.8 Putting It All Together . . . 23
2.9 What You Learned in This Chapter . . . 23
2.10 What's Next? . . . 24

3 Chapter 3: Calculations and Operators 25
3.1 The Seven Arithmetic Operators . . . 25
3.1.1 Addition, Subtraction, Multiplication, Division . . . 25
3.1.2 Floor Division (`//`) . . . 26
3.1.3 Modulo (`%`) — The Remainder . . . 26
3.1.4 Exponentiation (`**`) — Powers . . . 26
3.2 Order of Operations . . . 27
3.3 Using Variables in Calculations . . . 27
3.4 Augmented Assignment Operators . . . 28
3.5 Integer vs Float in Calculations . . . 29
3.6 Built-in Math Functions . . . 29
3.6.1 `abs()` — Absolute Value . . . 29
3.6.2 `round()` — Rounding . . . 30
3.6.3 `min()` and `max()` — Smallest and Largest . . . 30
3.6.4 `pow()` — Power (alternative to `**`) . . . 30
3.7 The `math` Module . . . 30
3.7.1 Square Roots . . . 31
3.7.2 Ceiling and Floor . . . 31
3.7.3 Pi and Euler's Number . . . 31
3.7.4 Logarithms and Trigonometry . . . 32
3.8 A Real-World Example: Shopping Calculator . . . 32
3.9 What You Learned in This Chapter . . . 33
3.10 What's Next? . . . 33

4 Chapter 4: Making Decisions — `if`, `elif`, `else` 34
4.1 Boolean Values: True and False . . . 34
4.2 Comparison Operators . . . 35
4.3 Getting Input from the User . . . 36
4.4 `if` — One Branch . . . 36
4.5 `if` / `else` — Two Branches . . . 37
4.6 `if` / `elif` / `else` — Many Branches . . . 38
4.7 Logical Operators: `and`, `or`, `not` . . . 39
4.7.1 `and` — Both must be True . . . 39
4.7.2 `or` — At least one must be True . . . 40
4.7.3 `not` — Flips True to False and vice versa . . . 40
4.7.4 Combining them . . . 40
4.8 Nested `if` Statements . . . 41
4.9 Putting It All Together: A Number Classifier . . . 41
4.10 What You Learned in This Chapter . . . 42
4.11 What's Next? . . . 43

5 Chapter 5: Repeating Things — Loops 44
5.1 Why Loops Exist . . . 44

5.2 `for` Loops — Repeating a Known Number of Times 45
5.3 `range()` — Generating Sequences of Numbers 46
5.3.1 `range(stop)` — Start at 0, count up to (but not including) stop 46
5.3.2 `range(start, stop)` — Control where you start 46
5.3.3 `range(start, stop, step)` — Control the step size . . 46
5.4 Doing Real Work Inside a Loop 47
5.4.1 Summing numbers 47
5.4.2 Multiplication table 48
5.4.3 Looping over a string 48
5.5 `while` Loops — Repeating Until a Condition Changes 49
5.5.1 Waiting for valid input 49
5.6 Infinite Loops and How to Avoid Them 50
5.7 `break` — Stop the Loop Early 51
5.8 `continue` — Skip One Iteration 52
5.9 `else` on a Loop — The Secret Clause 52
5.10 `for` vs `while` — When to Use Each 53
5.11 Nested Loops 54
5.12 Putting It All Together: The Guessing Game 55
5.13 What You Learned in This Chapter 56
5.14 What's Next? 56

6 Chapter 6: Lists — Your First Collection **57**
6.1 What is a List? 57
6.2 Indexing — Getting One Item 58
6.3 Slicing — Getting Multiple Items 59
6.4 Changing Items 59
6.5 Adding Items 60
6.5.1 `append()` — Add one item to the end 60
6.5.2 `insert()` — Add one item at a specific position 60
6.5.3 `extend()` — Add all items from another list 60
6.6 Removing Items 60
6.6.1 `remove()` — Remove by value 60
6.6.2 `pop()` — Remove by index (and return the value) 61
6.6.3 `del` — Remove by index or slice 61
6.6.4 `clear()` — Remove everything 61
6.7 Useful Functions for Lists 61
6.7.1 `len()` — How many items? 61
6.7.2 `sum()`, `min()`, `max()` — Math on a list of numbers 61
6.7.3 `sorted()` — Sort without changing the original 62
6.7.4 `.sort()` — Sort in place (changes the original) 62
6.7.5 `reversed()` — Iterate in reverse order 62
6.8 More List Methods 62
6.8.1 `count()` — How many times does a value appear? 62
6.8.2 `index()` — What position is this value at? 62
6.8.3 `copy()` — Make a shallow copy 63
6.9 Checking if an Item is in a List 63
6.10 Looping Over a List 64

6.10.1 Loop over values directly . . . 64
6.10.2 Loop with index using `enumerate()` . . . 64
6.11 Nested Lists — Lists Inside Lists . . . 64
6.12 List Comprehensions — A Powerful Shortcut . . . 65
6.13 Putting It All Together: Student Grade Tracker . . . 66
6.14 What You Learned in This Chapter . . . 67
6.15 What's Next? . . . 67

7 Chapter 7: Functions — Reusable Blocks of Code 69
7.1 What is a Function? . . . 69
7.2 Defining Your First Function . . . 70
7.3 Parameters — Passing Information In . . . 71
7.4 Return Values — Getting Information Back . . . 72
7.5 Default Parameter Values . . . 73
7.6 Keyword Arguments . . . 74
7.7 `*args` — Any Number of Arguments . . . 74
7.8 `**kwargs` — Any Number of Keyword Arguments . . . 75
7.9 Scope — Where Variables Live . . . 76
7.9.1 Local scope . . . 76
7.9.2 Global scope . . . 76
7.9.3 The `global` keyword . . . 77
7.10 Docstrings — Documenting Your Functions . . . 77
7.11 Calling Functions from Functions . . . 78
7.12 Returning Multiple Values . . . 78
7.13 Putting It All Together: A BMI Calculator . . . 79
7.14 What You Learned in This Chapter . . . 80
7.15 What's Next? . . . 81

8 Chapter 8: Working with Files 82
8.1 How Files Work in Python . . . 82
8.2 File Modes . . . 83
8.3 Reading a File . . . 83
8.3.1 `read()` — Read the whole file at once . . . 83
8.3.2 `readlines()` — Read into a list of lines . . . 84
8.3.3 Looping line by line — The memory-efficient way . . . 84
8.4 Writing to a File . . . 85
8.5 Appending to a File . . . 86
8.6 File Paths: Absolute vs Relative . . . 86
8.7 Handling Files That Don't Exist . . . 87
8.8 Working with CSV Files . . . 88
8.8.1 Reading a CSV file . . . 88
8.8.2 `DictReader` — Rows as dictionaries . . . 89
8.8.3 Writing a CSV file . . . 89
8.9 Putting It All Together: A Persistent To-Do List . . . 90
8.10 What You Learned in This Chapter . . . 92
8.11 What's Next? . . . 92

9 Chapter 9: A Beginner Project — Score Keeper 93

9.1 What We're Building . . . 93
9.2 Step 1: Plan Before You Code . . . 94
9.3 Step 2: Get the Team Names . . . 94
9.4 Step 3: Set Up the Scores and the Main Loop . . . 95
9.5 Step 4: Announce the Winner . . . 96
9.6 Step 5: Save the Result to a File . . . 97
9.7 Step 6: The Complete Program . . . 98
9.8 What the Code Teaches . . . 100
9.9 How to Extend This Project . . . 100
9.10 Ideas for Your Own Project . . . 101
9.11 What You Learned in This Chapter . . . 102
9.12 What's Next? . . . 102

10 Chapter 10: Strings in Depth 103
10.1 Strings Are Sequences . . . 103
10.1.1 Slicing strings . . . 104
10.1.2 Looping over a string . . . 104
10.1.3 Strings are immutable . . . 104
10.2 String Methods . . . 104
10.2.1 Case methods . . . 105
10.2.2 Stripping whitespace . . . 105
10.2.3 `split()` — Break a string into a list . . . 105
10.2.4 `join()` — Combine a list into a string . . . 106
10.2.5 `replace()` — Swap text . . . 106
10.2.6 `find()` and `index()` — Search for a substring . . . 106
10.2.7 `startswith()` and `endswith()` . . . 107
10.2.8 `count()` — How many times does a substring appear? . . . 107
10.2.9 `in` — Check if a substring exists . . . 107
10.3 String Formatting . . . 108
10.3.1 1. `%` formatting (old style — avoid in new code) . . . 108
10.3.2 2. `.format()` (better — still common) . . . 108
10.3.3 3. F-strings (best — use these) . . . 108
10.3.4 F-string formatting cheat sheet . . . 109
10.4 Checking String Content . . . 109
10.5 Multiline Strings and Raw Strings . . . 110
10.5.1 Triple-quoted strings . . . 110
10.5.2 Raw strings . . . 110
10.6 Unicode and Encoding . . . 111
10.6.1 `ord()` and `chr()` . . . 111
10.6.2 Encoding . . . 112
10.7 String Methods Reference . . . 112
10.8 Putting It All Together: A Text Analyzer . . . 113
10.9 What You Learned in This Chapter . . . 114
10.10What's Next? . . . 115

11 Chapter 11: Tuples and Sets 116
11.1 Tuples — Immutable Sequences . . . 116
11.1.1 Creating a tuple . . . 116

11.1.2 Indexing and slicing — same as lists . . . 117
11.1.3 Tuples are immutable . . . 117
11.1.4 Looping over a tuple . . . 117
11.1.5 Tuple methods . . . 118
11.1.6 Tuple packing and unpacking . . . 118
11.2 When to Use a Tuple Instead of a List . . . 119
11.3 Sets — Unique, Unordered Collections . . . 120
11.3.1 Creating a set . . . 120
11.3.2 Duplicates are automatically removed . . . 120
11.3.3 Adding and removing . . . 121
11.3.4 Membership testing — this is where sets shine . . . 121
11.4 Set Operations — The Real Power . . . 122
11.4.1 Union — everything in either set . . . 122
11.4.2 Intersection — only what's in both . . . 122
11.4.3 Difference — in the first but not the second . . . 122
11.4.4 Symmetric difference — in one or the other, but not both 122
11.4.5 Checking subset and superset . . . 123
11.4.6 A real example: comparing survey responses . . . 123
11.5 Updating Sets In Place . . . 124
11.6 `frozenset` — An Immutable Set . . . 124
11.7 Choosing Between List, Tuple, and Set . . . 124
11.8 Putting It All Together: Analyzing Word Frequency . . . 125
11.9 What You Learned in This Chapter . . . 126
11.10 What's Next? . . . 127

12 Chapter 12: Dictionaries — Key-Value Data 128
12.1 What is a Dictionary? . . . 128
12.2 Creating a Dictionary . . . 129
12.3 Accessing Values . . . 130
12.3.1 `.get()` — Safe access with a default . . . 130
12.4 Adding and Updating Values . . . 130
12.4.1 `update()` — Add or overwrite multiple pairs at once . . 131
12.5 Deleting Keys . . . 131
12.6 Checking if a Key Exists . . . 131
12.7 Looping Over a Dictionary . . . 132
12.7.1 `.keys()` — Loop over keys . . . 132
12.7.2 `.values()` — Loop over values . . . 132
12.7.3 `.items()` — Loop over key-value pairs (most common) . 133
12.8 Dictionary Methods Reference . . . 133
12.9 Nested Dictionaries . . . 133
12.10 Dictionary Comprehensions . . . 134
12.11 The Counting Pattern . . . 135
12.12 `defaultdict` and `Counter` from `collections` . . . 135
12.12.1 `Counter` — Count things instantly . . . 136
12.12.2 `defaultdict` — Never get a KeyError again . . . 136
12.13 Merging Dictionaries . . . 137
12.13.1 Python 3.9+: the `|` operator . . . 137
12.13.2 `{**dict1, **dict2}` — Works in all Python 3 versions . 137

12.14 Putting It All Together: A Word Frequency Analyzer 138
12.15 What You Learned in This Chapter 139
12.16 What's Next? 140

13 Chapter 13: Error Handling — `try`, `except`, `finally` 141
13.1 Why Programs Crash 141
13.2 `try` and `except` — The Basic Pattern 142
13.3 Common Exception Types 143
13.4 Catching Specific Exceptions 144
13.4.1 Catching multiple exceptions in one block 144
13.4.2 Catching all exceptions with `Exception` 144
13.5 The `as` Keyword — Accessing the Exception Object 145
13.6 `else` — Runs Only When No Exception Occurred 145
13.7 `finally` — Always Runs 146
13.8 `raise` — Raising Your Own Exceptions 147
13.8.1 Re-raising an exception 148
13.9 Creating Custom Exceptions 148
13.10 Exception Hierarchy 150
13.11 Practical Patterns 150
13.11.1 Pattern 1: Retry loop 150
13.11.2 Pattern 2: Try multiple approaches 151
13.11.3 Pattern 3: Context manager for cleanup 151
13.12 Best Practices 152
13.13 Putting It All Together: A Robust CSV Loader 153
13.14 What You Learned in This Chapter 155
13.15 What's Next? 155

14 Chapter 14: Comprehensions — Elegant, Fast Code 157
14.1 List Comprehensions — Deep Dive 157
14.1.1 The pattern 157
14.1.2 The expression can be anything 158
14.1.3 Filtering with a condition 158
14.1.4 `if`/`else` inside the expression (not the filter) 159
14.1.5 Nested comprehensions 159
14.2 Dictionary Comprehensions 160
14.3 Set Comprehensions 161
14.4 Generator Expressions — Lazy Evaluation 161
14.4.1 When to use generators vs list comprehensions 162
14.5 Comprehensions vs Loops — When to Use Which 163
14.6 `zip()`, `enumerate()`, and `map()` in Comprehensions 164
14.6.1 `zip()` — Pair up two or more iterables 164
14.6.2 `enumerate()` — Index + value 164
14.6.3 `map()` — Apply a function to every item 164
14.7 Putting It All Together: A Data Pipeline 165
14.8 Quick Reference 166
14.9 What You Learned in This Chapter 167
14.10 What's Next? 168

15 Chapter 15: Modules and Packages **169**
15.1 What is a Module? . . . 169
15.2 Three Ways to Import . . . 170
15.2.1 `import module` — Import the whole module . . . 170
15.2.2 `from module import name` — Import specific names . . . 170
15.2.3 `import module as alias` — Import with a shorter name 170
15.2.4 `from module import *` — Import everything (avoid this) 171
15.3 How Python Finds Modules . . . 171
15.4 Writing Your Own Module . . . 171
15.5 `if __name__ == "__main__":` . . . 172
15.6 What is a Package? . . . 173
15.6.1 Controlling what `import utils` exports . . . 174
15.7 The Standard Library . . . 175
15.8 Installing Third-Party Packages with `pip` . . . 176
15.8.1 Installing a package . . . 176
15.8.2 Installing a specific version . . . 176
15.8.3 Upgrading a package . . . 176
15.8.4 Uninstalling a package . . . 176
15.8.5 Listing installed packages . . . 177
15.8.6 Seeing where a package is installed . . . 177
15.9 `requirements.txt` — Sharing Your Environment . . . 177
15.9.1 Creating it . . . 177
15.9.2 Installing from it . . . 177
15.10 Virtual Environments — Isolating Projects . . . 177
15.10.1 Creating a virtual environment . . . 178
15.10.2 Activating it . . . 178
15.10.3 Using it . . . 178
15.10.4 Deactivating it . . . 178
15.10.5 The standard project layout . . . 178
15.11 Relative Imports . . . 179
15.12 `__init__.py` in Depth . . . 179
15.13 Putting It All Together: A Utility Package . . . 180
15.14 What You Learned in This Chapter . . . 181
15.15 What's Next? . . . 182

16 Chapter 16: Useful Standard Library Modules **183**
16.1 `os` and `os.path` — The File System . . . 183
16.1.1 Environment variables . . . 185
16.2 `pathlib` — Modern File Paths . . . 185
16.3 `sys` — The Python Interpreter . . . 186
16.4 `datetime` — Dates and Times . . . 187
16.4.1 Common `strftime` format codes . . . 189
16.5 `random` — Randomness . . . 189
16.6 `json` — Reading and Writing JSON . . . 190
16.6.1 JSON □ Python type mapping . . . 191
16.7 `re` — Regular Expressions . . . 191
16.7.1 Compile for repeated use . . . 192
16.8 `itertools` — Powerful Iteration Tools . . . 193

16.9 `functools` — Higher-Order Functions . . . 194
16.9.1 `lru_cache` — Memoization (cache function results) . . . 194
16.9.2 `partial` — Pre-fill function arguments . . . 195
16.9.3 `wraps` — Preserve function metadata in decorators . . . 195
16.10 Putting It All Together: A File Report Tool . . . 196
16.11 What You Learned in This Chapter . . . 198
16.12 What's Next? . . . 198

17 Chapter 17: Classes and Objects — Introduction to OOP 199
17.1 What is Object-Oriented Programming? . . . 199
17.2 Your First Class . . . 200
17.3 `__init__` — The Constructor . . . 200
17.4 `self` — What It Means and Why It's There . . . 201
17.5 Instance Attributes vs Class Attributes . . . 201
17.6 Methods — What Objects Can Do . . . 202
17.7 `__str__` and `__repr__` — Representing Objects as Text . . . 203
17.8 A Complete, Realistic Class . . . 204
17.9 Multiple Objects, Same Class . . . 206
17.10 Objects in Collections . . . 207
17.11 Comparing Objects . . . 207
17.12 Putting It All Together: A Student Grade Tracker . . . 208
17.13 What You Learned in This Chapter . . . 211
17.14 What's Next? . . . 211

18 Chapter 18: Inheritance and Polymorphism 213
18.1 What is Inheritance? . . . 213
18.2 Creating a Child Class . . . 214
18.3 `super()` — Calling the Parent's Method . . . 216
18.4 Overriding Methods . . . 217
18.5 `isinstance()` and `issubclass()` . . . 218
18.6 Inheritance Chains — Multiple Levels . . . 219
18.7 Multiple Inheritance . . . 220
18.8 Abstract Base Classes — Enforcing a Contract . . . 221
18.9 A Real-World Inheritance Example: Payment System . . . 223
18.10 What You Learned in This Chapter . . . 227
18.11 What's Next? . . . 227

19 Chapter 19: OOP in Depth — Encapsulation and Special Methods 228
19.1 Encapsulation — Protecting Data . . . 228
19.1.1 Single underscore `_name` — "private by convention" . . . 229
19.1.2 Double underscore `__name` — name mangling . . . 229
19.2 Properties — Controlled Attribute Access . . . 230
19.2.1 `@property` — A readable "attribute" . . . 230
19.2.2 `@property.setter` — Allow controlled writing . . . 230
19.2.3 `@property.deleter` — Handle deletion . . . 231
19.2.4 Computed properties . . . 232
19.3 Dunder (Magic) Methods — Making Objects Feel Native . . . 233

19.3.1 `__len__` — `len()` support . . . 233
19.3.2 `__getitem__` and `__setitem__` — Index access `[]` . . . 233
19.3.3 `__contains__` — `in` operator . . . 234
19.3.4 `__iter__` and `__next__` — Making objects iterable . . . 234
19.3.5 Arithmetic operators . . . 236
19.3.6 Comparison operators . . . 237
19.3.7 `__call__` — Making objects callable . . . 238
19.3.8 `__enter__` and `__exit__` — Context managers . . . 238
19.4 Class Methods and Static Methods . . . 239
19.4.1 `@classmethod` — Works on the class, not an instance . . . 239
19.4.2 `@staticmethod` — Belongs to the class, needs neither `self` nor `cls` . . . 240
19.5 Dataclasses — The Modern Shortcut . . . 241
19.5.1 Default values and field options . . . 241
19.5.2 Frozen dataclasses — immutable objects . . . 242
19.5.3 Ordering with dataclasses . . . 243
19.6 Putting It All Together: A Full Playlist Class . . . 243
19.7 Dunder Methods Quick Reference . . . 246
19.8 What You Learned in This Chapter . . . 247
19.9 What's Next? . . . 248

20 Chapter 20: Intermediate OOP Project — Build a Mini Library System 249
20.1 What We're Building . . . 249
20.2 Step 1: Design the Classes First . . . 250
20.3 Step 2: The `Book` Class . . . 251
20.4 Step 3: The `DigitalBook` Subclass . . . 253
20.5 Step 4: The `Member` Class . . . 254
20.6 Step 5: The `Library` Class . . . 256
20.7 Step 6: The Menu Interface . . . 260
20.8 Step 7: Run It . . . 262
20.9 What This Project Demonstrates . . . 262
20.10 Extensions to Try . . . 263
20.11 What You Learned in This Chapter . . . 264
20.12 What's Next? . . . 264

21 Chapter 21: First-Class Functions and Closures 265
21.1 Functions Are Objects . . . 265
21.2 Passing Functions as Arguments . . . 266
21.2.1 A practical example: applying transformations to a list . . . 267
21.2.2 Sorting with a key function . . . 267
21.3 Storing Functions in Data Structures . . . 268
21.4 Returning Functions from Functions . . . 270
21.5 Closures — Functions That Remember . . . 270
21.5.1 Inspecting a closure . . . 271
21.6 Factory Functions . . . 272
21.7 The `nonlocal` Keyword . . . 273
21.8 Closures vs Classes . . . 273

21.9 Partial Application with Closures . . . 274
21.10 Memoization with Closures . . . 275
21.11 Putting It All Together: A Pipeline Builder . . . 276
21.12 What You Learned in This Chapter . . . 277
21.13 What's Next? . . . 278

22 Chapter 22: Decorators 279
22.1 The Long Way First . . . 279
22.2 The `@` Syntax . . . 280
22.3 Preserving the Wrapped Function's Identity . . . 281
22.4 Useful Decorator Patterns . . . 282
22.4.1 Timing . . . 282
22.4.2 Logging . . . 282
22.4.3 Retry on failure . . . 283
22.4.4 Input validation . . . 285
22.4.5 Caching (your own `lru_cache`) . . . 285
22.5 Decorator Factories (Decorators with Arguments) . . . 286
22.5.1 Rate limiter . . . 287
22.6 Stacking Decorators . . . 288
22.7 Class-Based Decorators . . . 289
22.8 Decorators in the Real World . . . 290
22.9 Project: A Decorator Toolkit . . . 291
22.10 What You Learned in This Chapter . . . 295
22.11 What's Next? . . . 295

23 Chapter 23: Generators and Iterators 296
23.1 How `for` Loops Actually Work . . . 296
23.2 Writing an Iterator Class . . . 297
23.3 Generator Functions: `yield` . . . 298
23.4 Memory Efficiency: The Big Win . . . 299
23.5 Common Generator Patterns . . . 300
23.5.1 Infinite sequences . . . 300
23.5.2 Reading large files line by line . . . 301
23.5.3 Filtering and transforming . . . 301
23.5.4 Batching . . . 302
23.6 Generator Expressions . . . 302
23.7 `yield from` — Delegating to Sub-generators . . . 303
23.7.1 Recursive flattening with `yield from` . . . 304
23.8 Sending Values into a Generator . . . 304
23.9 `itertools` — Your Generator Toolkit . . . 305
23.10 Building a Lazy Data Pipeline . . . 307
23.11 Project: Build Your Own `range` . . . 308
23.12 What You Learned in This Chapter . . . 309
23.13 What's Next? . . . 310

24 Chapter 24: Context Managers 311
24.1 The Problem They Solve . . . 311
24.2 How `with` Works . . . 312

24.3 Writing a Context Manager Class . . . 313
24.3.1 Handling exceptions in `__exit__` . . . 313
24.4 Writing a Context Manager with `@contextlib.contextmanager` 314
24.5 Practical Context Manager Patterns . . . 315
24.5.1 Timer . . . 315
24.5.2 Temporary directory . . . 315
24.5.3 Changing directory temporarily . . . 316
24.5.4 Redirecting stdout . . . 316
24.5.5 Database transaction . . . 317
24.5.6 Lock management . . . 318
24.6 `contextlib` Utilities You Should Know . . . 318
24.7 Nested and Multiple Context Managers . . . 319
24.8 Project: A Robust Database Connection Manager . . . 320
24.9 What You Learned in This Chapter . . . 324
24.10 What's Next? . . . 324

25 Chapter 25: Functional Programming 325
25.1 Pure Functions . . . 325
25.2 Immutability . . . 327
25.3 `map` — Transform Every Element . . . 328
25.4 `filter` — Keep Only Matching Elements . . . 329
25.5 `reduce` — Fold a Sequence into One Value . . . 330
25.6 `functools` — Your Functional Toolkit . . . 330
25.6.1 `partial` — Fix some arguments . . . 330
25.6.2 `lru_cache` — Memoize with a size limit . . . 331
25.6.3 `cache` — Unbounded memoization (Python 3.9+) . . . 331
25.6.4 `total_ordering` — Fill in comparison methods . . . 332
25.6.5 `reduce` — Already covered above . . . 332
25.6.6 `singledispatch` — Function overloading by type . . . 332
25.7 Lambda Functions . . . 333
25.8 Function Composition . . . 334
25.9 Putting It All Together: A Data Processing Library . . . 335
25.10 What You Learned in This Chapter . . . 338
25.11 What's Next? . . . 338

26 Chapter 26: Type Hints and the `typing` Module 339
26.1 The Basics . . . 339
26.2 Built-in Collection Types (Python 3.9+) . . . 340
26.3 `Optional` — Values That Can Be `None` . . . 341
26.4 `Union` — Multiple Possible Types . . . 342
26.5 `Any` — Opt Out of Type Checking . . . 342
26.6 `Callable` — Typing Functions . . . 342
26.7 `Sequence`, `Iterable`, `Mapping` — Accept More Types . . . 343
26.8 `TypeVar` — Generic Functions . . . 344
26.9 Generic Classes . . . 344
26.10 `Protocol` — Structural Subtyping (Duck Typing, but Typed) . . . 346
26.11 `NamedTuple` — Typed Tuples . . . 347
26.12 `TypedDict` — Typed Dictionaries . . . 348

26.13 Literal Types . . . 349
26.14 `Final` — Constants . . . 350
26.15 Running `mypy` . . . 350
26.16 Type Aliases . . . 351
26.17 Project: A Typed Data Pipeline . . . 352
26.18 Type Hint Cheat Sheet . . . 355
26.19 What You Learned in This Chapter . . . 355
26.20 What's Next? . . . 356

27 Chapter 27: Concurrency — Threading, Multiprocessing, and asyncio 357
27.1 The Two Types of Bottleneck . . . 357
27.2 The GIL — Why Threads Don't Parallelize CPU Work . . . 358
27.3 Threading . . . 358
27.3.1 `ThreadPoolExecutor` — the cleaner way . . . 360
27.3.2 Thread safety and locks . . . 361
27.4 Multiprocessing . . . 362
27.4.1 `ProcessPoolExecutor` — mirrors `ThreadPoolExecutor` 363
27.4.2 Sharing data between processes . . . 364
27.5 `asyncio` — Async/Await . . . 364
27.5.1 `async` and `await` — the rules . . . 365
27.5.2 `asyncio.gather` — run tasks concurrently . . . 366
27.5.3 `asyncio.create_task` — fire and forget . . . 366
27.5.4 Async context managers and iterators . . . 366
27.5.5 Async with `aiohttp` — real HTTP requests . . . 367
27.6 Choosing the Right Tool . . . 368
27.7 Project: Async Web Scraper with Rate Limiting . . . 368
27.8 What You Learned in This Chapter . . . 372
27.9 What's Next? . . . 372

28 Chapter 28: Testing with pytest 373
28.1 Why Test? . . . 373
28.2 Installing and Running pytest . . . 374
28.3 Your First Tests . . . 374
28.3.1 Reading a failure . . . 376
28.4 `assert` — The Only Assertion You Need . . . 376
28.5 `pytest.raises` — Testing Exceptions . . . 377
28.6 Parametrize — Test Many Inputs at Once . . . 378
28.7 Fixtures — Shared Setup and Teardown . . . 379
28.7.1 Fixture scope — how long it lives . . . 380
28.7.2 Fixtures with teardown . . . 380
28.7.3 Built-in fixtures . . . 381
28.8 Mocking — Replace Real Dependencies . . . 381
28.8.1 `Mock` — a fake object . . . 381
28.8.2 `patch` — replace real objects temporarily . . . 382
28.8.3 `patch` as a decorator . . . 383
28.8.4 Mocking time and random . . . 383
28.9 Testing Classes . . . 384

28.10 Test Coverage . . . 387
28.11 Test Organization . . . 387
28.12 Marks — Categorizing Tests . . . 388
28.13 Test-Driven Development (TDD) . . . 389
28.14 Project: Test Suite for a Shopping Cart . . . 391
28.15 What You Learned in This Chapter . . . 395
28.16 What's Next? . . . 396

29 Chapter 29: Debugging 397
29.1 The Debugging Mindset . . . 397
29.2 Print Debugging — Simple but Effective . . . 398
29.3 `breakpoint()` and `pdb` . . . 399
29.3.1 Post-mortem debugging . . . 400
29.3.2 Conditional breakpoints . . . 400
29.4 `logging` — Structured Debug Output . . . 401
29.4.1 Log levels . . . 401
29.4.2 Named loggers . . . 402
29.4.3 Logging to a file . . . 403
29.4.4 Structured logging with `extra` . . . 403
29.5 Reading Tracebacks . . . 403
29.5.1 Common exceptions and their usual causes . . . 404
29.6 Common Bug Patterns and How to Find Them . . . 405
29.6.1 Off-by-one errors . . . 405
29.6.2 Mutating a list while iterating it . . . 405
29.6.3 Mutable default arguments . . . 405
29.6.4 `is` vs `==` . . . 406
29.6.5 Variable shadowing . . . 406
29.6.6 Integer division returning float, or vice versa . . . 407
29.7 The `traceback` Module . . . 407
29.8 `assert` for Invariants . . . 408
29.9 Introspection Tools . . . 408
29.10 IDE Debugging . . . 409
29.11 `icecream` — A Better Debug Print . . . 409
29.12 Systematic Debugging: The Bisect Method . . . 410
29.13 Project: A Debug-Friendly Logger Class . . . 410
29.14 Debugging Checklist . . . 414
29.15 What You Learned in This Chapter . . . 414
29.16 What's Next? . . . 415

30 Chapter 30: Performance and Profiling 416
30.1 The Golden Rule: Measure Before You Optimize . . . 416
30.2 `timeit` — Measuring Small Code Snippets . . . 417
30.2.1 `timeit` from the command line . . . 418
30.2.2 `%%timeit` in Jupyter notebooks . . . 418
30.3 `cProfile` — Profiling Entire Programs . . . 418
30.3.1 From the command line . . . 419
30.3.2 Visualizing with `snakeviz` . . . 419
30.4 `line_profiler` — Line-by-Line Timing . . . 419

30.5 `memory_profiler` — Measuring Memory Usage 420
30.6 Algorithmic Complexity — The Real Performance Win 421
30.6.1 Seeing it in practice 421
30.7 Choosing the Right Data Structure 422
30.7.1 Data structure complexity cheat sheet 423
30.8 Practical Optimization Techniques 424
30.8.1 1. Avoid repeated work — cache results 424
30.8.2 2. Move work out of inner loops 425
30.8.3 3. Use built-ins — they're implemented in C 425
30.8.4 4. List comprehensions over explicit loops 425
30.8.5 5. Use generators for large data 426
30.8.6 6. Local variable lookup is faster than global 426
30.8.7 7. String formatting — f-strings are fastest 426
30.8.8 8. `functools.lru_cache` for expensive pure functions . 427
30.9 `numpy` for Numerical Work 427
30.10 `__slots__` — Reducing Memory for Many Objects 428
30.11 Profiling in Practice: A Case Study 429
30.12 Performance Checklist 430
30.13 What You Learned in This Chapter 431
30.14 What's Next? 432

31 Chapter 31: Working with Databases 433
31.1 Why Databases? 433
31.2 SQL Basics 434
31.3 SQLite — The Built-in Database 435
31.3.1 Connecting and creating tables 435
31.3.2 Inserting data safely 435
31.3.3 Querying data 436
31.3.4 Transactions 437
31.3.5 A full SQLite example 437
31.4 PostgreSQL with psycopg2 440
31.4.1 Connection pooling with `psycopg2.pool` 441
31.5 SQLAlchemy — The Python ORM 442
31.5.1 Defining models 442
31.5.2 Creating the database and session 443
31.5.3 CRUD with SQLAlchemy 444
31.5.4 Querying with joins and aggregates 445
31.5.5 SQLAlchemy 2.0 style (the modern way) 446
31.6 Database Migrations with Alembic 447
31.7 Project: A Complete User + Order System 447
31.8 SQLite vs PostgreSQL — When to Use Which 453
31.9 What You Learned in This Chapter 454
31.10 What's Next? 454

32 Chapter 32: Web Development with Flask and FastAPI 455
32.1 HTTP in 60 Seconds 455
32.2 Flask 456
32.2.1 Hello, Flask 456

32.2.2 Returning JSON 457
32.2.3 Error handlers 458
32.2.4 Request object 458
32.2.5 Blueprints — organizing a larger app 459
32.2.6 A complete Flask API with SQLite 460
32.3 FastAPI 463
32.3.1 Hello, FastAPI 463
32.3.2 Pydantic models for validation 464
32.3.3 Query parameters and filters 466
32.3.4 Path parameters with validation 466
32.3.5 Dependency injection 466
32.3.6 Async routes 467
32.3.7 A complete FastAPI app with SQLAlchemy 467
32.4 Adding Authentication 471
32.5 Flask vs FastAPI — When to Use Which 473
32.6 Project Structure for a Real API 473
32.7 What You Learned in This Chapter 475
32.8 What's Next? 476

33 Chapter 33: Data Science with Python 477
33.1 The Data Science Workflow 477
33.2 numpy — Fast Array Mathematics 478
33.2.1 Creating arrays 478
33.2.2 Vectorized operations 478
33.2.3 Indexing, slicing, and masking 479
33.2.4 Aggregation and statistics 479
33.2.5 Linear algebra 480
33.3 pandas — Tabular Data 480
33.3.1 Creating DataFrames 480
33.3.2 Loading data from files 481
33.3.3 Exploring data 481
33.3.4 Selecting data 482
33.3.5 Cleaning data 482
33.3.6 Transforming data 483
33.3.7 Grouping and aggregation 483
33.3.8 Sorting and ranking 484
33.3.9 Merging DataFrames 484
33.4 matplotlib — Data Visualization 484
33.4.1 Common chart types 485
33.4.2 `seaborn` — statistical visualization 486
33.5 scikit-learn — Machine Learning 487
33.5.1 The scikit-learn API 487
33.5.2 A classification example 488
33.5.3 Pipelines — the right way to combine steps 489
33.6 Project: End-to-End Sales Analysis 490
33.7 The Data Science Toolkit Summary 493
33.8 What You Learned in This Chapter 493
33.9 What's Next? 494

34 Chapter 34: Packaging and Distribution **495**
34.1 What a Package Actually Is 495
34.2 Modern Project Structure 495
34.3 `pyproject.toml` — The Modern Configuration File 496
34.3.1 Choosing a build backend 498
34.4 Building a Real Package 498
34.4.1 Directory structure 498
34.4.2 The package code 499
34.4.3 `pyproject.toml` for texttools 502
34.5 Installing in Development Mode 503
34.6 Writing a Good README 504
34.7 Quick Start 504
34.8 CLI 504
34.9 API Reference 504
34.9.1 `word_count(text: str) -> int` 504
34.9.2 `slug(text: str) -> str` 505
34.10 License 505
34.11 Building Distribution Files 505
34.12 Publishing to PyPI 506
34.13 Automating Releases with GitHub Actions 507
34.14 Code Quality Tools 508
34.14.1 `ruff` — fast linter + formatter 508
34.14.2 `mypy` — static type checking 508
34.14.3 `pre-commit` — run checks before every commit 508
34.15 Dependency Management Best Practices 509
34.15.1 Pinning dependencies 509
34.15.2 `pip-tools` for reproducible installs 509
34.15.3 Using `uv` — the modern, fast alternative 510
34.16 Complete Workflow Summary 510
34.17 What You Learned in This Chapter 511
34.18 What's Next? 511

35 Chapter 35: Professional Python Practices **512**
35.1 Code Style — PEP 8 and Why It Matters 512
35.1.1 Let tools enforce style — don't do it manually 514
35.2 Writing Good Docstrings 514
35.2.1 The Google style (most common in modern Python) 515
35.2.2 Class docstrings 515
35.2.3 Module docstrings 516
35.2.4 `doctest` — runnable examples in docstrings 516
35.3 The Zen of Python Applied 517
35.4 Code Review Practices 518
35.4.1 Writing reviewable code 518
35.4.2 Reviewing others' code 519
35.5 Git Workflow for Python Projects 519
35.5.1 `.gitignore` 519
35.5.2 Commit message convention 520
35.5.3 Branch strategy 521

35.6 Environment Variables and Configuration 522
35.7 Logging in Production 523
35.8 Writing Clean Code — Practical Rules 524
35.8.1 Functions should do one thing 524
35.8.2 Avoid magic numbers 525
35.8.3 Return early, don't nest deeply 525
35.8.4 Name things clearly 526
35.8.5 Avoid comments that explain what — write code that shows what 526
35.9 The Habits of Professional Python Developers 527
35.10 Project: A Professional Python Module 527
35.11 What You Learned in This Chapter 533
35.12 What's Next? 533

36 Chapter 36: Advanced Python Internals 534
36.1 How Python Runs Your Code 534
36.1.1 Inspecting bytecode of any function 536
36.2 The CPython Virtual Machine 536
36.2.1 Code objects 536
36.2.2 Frame objects 537
36.3 The GIL — Global Interpreter Lock in Depth 538
36.3.1 Why the GIL exists 538
36.3.2 The GIL releases during I/O 538
36.3.3 The GIL check interval 539
36.3.4 Python 3.13 — the no-GIL build 539
36.4 Memory Management 539
36.4.1 Reference counting 539
36.4.2 Cyclic garbage collector 540
36.4.3 Object internment and small integer caching 541
36.4.4 Memory pools — `pymalloc` 542
36.5 The Descriptor Protocol 542
36.5.1 Building a typed attribute descriptor 543
36.5.2 How `@property` is implemented 544
36.6 Metaclasses 546
36.6.1 Writing a metaclass 546
36.6.2 Metaclass `__new__` and `__init__` 546
36.6.3 `__init_subclass__` — the modern alternative to metaclasses 547
36.7 `__slots__` Internals 548
36.8 `__getattr__` and `__getattribute__` 549
36.9 AST Manipulation 550
36.10 `sys` Module Internals 551
36.11 Project: A Simple Bytecode Analyzer 552
36.12 What You Learned in This Chapter 554
36.13 What's Next? 555

37 Chapter 37: Metaprogramming 556
37.1 What Metaprogramming Is (and Isn't) 556

37.2 `type()` as a Class Factory . . . 557
37.2.1 Building an animal factory . . . 557
37.2.2 A model factory for database tables . . . 558
37.3 Class Decorators . . . 559
37.4 `__init_subclass__` — Framework Hooks . . . 561
37.5 Dynamic Attribute Generation . . . 562
37.5.1 Auto-generating methods . . . 563
37.6 `exec` and `eval` — Runtime Code Execution . . . 564
37.6.1 Safe use: controlled namespace . . . 565
37.6.2 Generating code at runtime . . . 565
37.7 `__class_getitem__` — Making Classes Generic . . . 567
37.8 `__set_name__` — Descriptor Self-Registration . . . 568
37.9 Building a Mini ORM with Metaprogramming . . . 570
37.10 What You Learned in This Chapter . . . 576
37.11 What's Next? . . . 576

38 Chapter 38: The Capstone Project — Build a Production Task Manager API 577
38.1 Project Structure . . . 577
38.2 Configuration . . . 579
38.3 Logging . . . 580
38.4 Database . . . 581
38.5 Models . . . 582
38.6 Schemas (Pydantic) . . . 583
38.7 Services . . . 585
38.8 Dependencies . . . 589
38.9 Routers . . . 590
38.10 Background Queue . . . 593
38.11 Main Application . . . 595
38.12 Tests . . . 597
38.13 Dockerfile . . . 601
38.14 GitHub Actions CI/CD . . . 602
38.15 `pyproject.toml` . . . 604
38.16 Running the Application . . . 605
38.17 What This Project Used . . . 606
38.18 Where to Go From Here . . . 606
38.19 Final Summary: What You Now Know . . . 607

39 Chapter 39: Web Scraping 609
39.1 How the Web Works (What You Need to Know) . . . 609
39.2 Before You Scrape — Check These Things . . . 610
39.3 `requests` — Fetching Pages . . . 610
39.3.1 Setting headers to look like a browser . . . 610
39.3.2 Handling errors . . . 611
39.4 `BeautifulSoup` — Parsing HTML . . . 611
39.4.1 Finding elements . . . 612
39.4.2 Extracting data . . . 612
39.5 A Real Scraper — Books to Scrape . . . 613

39.6 Handling Pagination 615
39.7 Storing Scraped Data 616
39.8 Scraping with Sessions and Cookies 617
39.9 Dynamic Pages with Selenium 618
39.10 Rate Limiting and Politeness 619
39.11 Project: Job Listing Scraper 620
39.12 What You Learned in This Chapter 622
39.13 What's Next? 622

40 Chapter 40: Web Development with Flask 623
40.1 What is Flask? 623
40.2 Your First Web App 623
40.3 Routes and URL Parameters 624
40.3.1 Static routes 624
40.3.2 Dynamic routes — path parameters 624
40.3.3 HTTP methods 625
40.4 The `request` Object 625
40.5 HTML Templates with Jinja2 626
40.6 Static Files — CSS and Images 628
40.7 Forms — Reading POST Data 628
40.8 Flask with SQLite — A Real Task App 630
40.9 Blueprints — Organizing a Larger App 633
40.10 Building a JSON API with Flask 634
40.11 Error Handlers 636
40.12 Deploying Flask 636
40.12.1 Option 1: Render (free tier) 636
40.12.2 Option 2: PythonAnywhere (free tier) 636
40.12.3 `requirements.txt` for deployment 637
40.13 What You Learned in This Chapter 637
40.14 What's Next? 637

41 Chapter 41: Web Development with FastAPI 638
41.1 What Makes FastAPI Different? 638
41.2 Setup 639
41.3 Your First FastAPI App 639
41.4 Path Parameters 639
41.5 Query Parameters 640
41.6 Request Bodies with Pydantic 640
41.6.1 Field validators 641
41.7 Response Models 642
41.8 A Full CRUD API 643
41.9 Async Endpoints 645
41.10 Dependency Injection 646
41.11 Authentication with JWT 647
41.12 Lifespan Events — Startup and Shutdown 649
41.13 Middleware 650
41.14 Background Tasks 650
41.15 Deploying FastAPI 651

41.15.1 Option 1: Render . . . 651
41.15.2 Option 2: Docker . . . 651
41.15.3 Option 3: Multiple workers with Gunicorn . . . 651
41.16 Flask vs FastAPI — When to Use Which . . . 652
41.17 What You Learned in This Chapter . . . 652
41.18 What's Next? . . . 652

42 Chapter 42: Data Science 653
42.1 Setup . . . 653
42.2 Arrays — The Core Concept . . . 653
42.2.1 Key properties . . . 654
42.3 Creating Arrays . . . 654
42.4 Data Types . . . 655
42.5 Indexing and Slicing . . . 655
42.5.1 1D arrays — just like lists . . . 655
42.5.2 2D arrays . . . 656
42.5.3 Boolean indexing — filter without a loop . . . 656
42.5.4 Fancy indexing — select by index list . . . 656
42.6 Vectorized Operations — No Loops Needed . . . 656
42.6.1 Speed comparison . . . 657
42.7 Broadcasting . . . 657
42.8 Mathematical Functions . . . 658
42.9 Aggregation — Summary Statistics . . . 658
42.10 Reshaping and Combining . . . 659
42.11 Linear Algebra . . . 660
42.12 Working with Real Data . . . 660
42.13 Practical Project — Grade Analysis . . . 661
42.14 Common Pitfalls . . . 664
42.14.1 Views vs copies . . . 664
42.14.2 Integer division . . . 664
42.14.3 Shape mismatches . . . 664
42.15 What You Learned in This Chapter . . . 664
42.16 What's Next? . . . 665

43 Chapter 43: Data Science with Pandas 666
43.1 Setup . . . 666
43.2 The Two Core Objects . . . 666
43.2.1 Series — a labelled 1D array . . . 666
43.2.2 DataFrame — a labelled 2D table . . . 667
43.3 Reading and Writing Data . . . 667
43.3.1 CSV . . . 667
43.3.2 Excel . . . 668
43.3.3 JSON . . . 668
43.3.4 SQL . . . 668
43.4 Exploring a DataFrame . . . 668
43.5 Selecting Data . . . 669
43.5.1 Columns . . . 669
43.5.2 Rows with `.iloc` — by position . . . 669

43.5.3 Rows with `.loc` — by label 669
43.5.4 Boolean filtering 669
43.6 Modifying Data 670
43.6.1 Adding and changing columns 670
43.6.2 Renaming columns 670
43.6.3 Dropping columns and rows 670
43.6.4 The `apply` function 670
43.7 Handling Missing Data 671
43.8 Sorting 671
43.9 Grouping and Aggregation 671
43.10 Merging and Joining 672
43.11 Working with Dates 673
43.12 Practical Project — Sales Analysis 673
43.13 What You Learned in This Chapter 676
43.14 What's Next? 677

44 Chapter 44: Data Visualization 678
44.1 Setup 678
44.2 How matplotlib Works 678
44.3 Line Plot 679
44.4 Bar Chart 679
44.4.1 Horizontal bar chart 680
44.5 Histogram 680
44.6 Scatter Plot 681
44.7 Pie / Donut Chart 682
44.8 Box Plot 682
44.9 Multiple Subplots 683
44.10 Seaborn — Statistical Plots Made Easy 684
44.10.1 Distribution plots 684
44.10.2 Categorical plots 684
44.10.3 Scatter and relationship plots 685
44.10.4 Heatmaps 685
44.11 Plotting Directly from pandas 686
44.12 Saving High-Quality Figures 687
44.13 Project: Complete Sales Dashboard 687
44.14 What You Learned in This Chapter 689
44.15 What's Next? 689

45 Chapter 45: Machine Learning Basics with scikit-learn 690
45.1 Setup 690
45.2 The Two Types of Problems 691
45.3 The scikit-learn API 691
45.4 Step 1 — Preparing Your Data 691
45.4.1 Load a dataset 691
45.4.2 Train/test split 692
45.4.3 Feature scaling 692
45.5 Linear Regression — Predict a Number 692
45.5.1 On real data — California housing prices 693

45.6 Decision Trees and Random Forests — Flexible and Powerful . 694
45.7 Classification — Predict a Category 695
45.7.1 Logistic Regression 695
45.7.2 Understanding the metrics 696
45.7.3 Confusion matrix 696
45.7.4 Random Forest Classifier 696
45.8 Cross-Validation — A More Reliable Score 696
45.9 Pipelines — Prevent Data Leakage 697
45.10 Hyperparameter Tuning with GridSearchCV 698
45.11 Unsupervised Learning — Clustering 699
45.12 Full Project — House Price Predictor 700
45.13 Choosing the Right Algorithm 702
45.14 What You Learned in This Chapter 703
45.15 What's Next? 703

46 Chapter 46: Automation and Scripting 704
46.1 File System Automation with `pathlib` and `shutil` 704
46.1.1 Rename files in bulk 704
46.1.2 Organise files by extension 705
46.1.3 Copy, move, and delete with `shutil` 706
46.1.4 Find files matching a pattern 707
46.2 Scheduling Tasks with `schedule` 707
46.3 Sending Emails with `smtplib` 708
46.4 Working with PDFs 709
46.4.1 Read and extract text 710
46.4.2 Merge PDFs 710
46.4.3 Split a PDF 710
46.4.4 Generate a PDF from scratch with ReportLab 711
46.5 Image Processing with Pillow 712
46.6 Automating Excel with `openpyxl` 714
46.7 Project: Automated Daily Report Generator 716
46.8 What You Learned in This Chapter 719
46.9 What's Next? 720

47 Chapter 47: Build a REST API with FastAPI and a Database 721
47.1 What We're Building 721
47.2 Project Structure 722
47.3 Step 1 — Configuration (`config.py`) 722
47.4 Step 2 — Database (`database.py`) 723
47.5 Step 3 — ORM Models (`models.py`) 724
47.6 Step 4 — Pydantic Schemas (`schemas.py`) 725
47.7 Step 5 — Services 728
47.7.1 Auth service (`services/auth.py`) 728
47.7.2 Product service (`services/products.py`) 729
47.8 Step 6 — Dependencies (`dependencies.py`) 731
47.9 Step 7 — Routers 732
47.9.1 Auth router (`routers/auth.py`) 732
47.9.2 Products router (`routers/products.py`) 732

47.10 Step 8 — Main App (`main.py`) 734
47.11 Step 9 — Tests 735
47.11.1 `tests/conftest.py` 735
47.11.2 `tests/test_auth.py` 737
47.11.3 `tests/test_products.py` 738
47.12 Step 10 — Run It 740
47.13 What You Built 740
47.14 What's Next? 740

48 Chapter 48: Build a Web Scraper and Data Pipeline 741
48.1 The Plan 741
48.2 Step 1 — The Scraper (`scraper.py`) 742
48.3 Step 2 — The Cleaner (`cleaner.py`) 744
48.4 Step 3 — Storage (`storage.py`) 747
48.5 Step 4 — Analysis (`analyser.py`) 749
48.6 Step 5 — Visualisation (`visualiser.py`) 751
48.7 Step 6 — The Runner (`runner.py`) 753
48.8 Running the Pipeline 756
48.9 Sample Output 756
48.10 What You Built 757
48.11 What's Next? 758

49 Chapter 49: Build a Desktop App with Tkinter 759
49.1 How Tkinter Works 759
49.2 Core Widgets 760
49.3 Layout Managers 761
49.3.1 `pack` — stack widgets in a direction 761
49.3.2 `grid` — position in a row/column table 761
49.3.3 `place` — exact pixel coordinates 762
49.4 Variables — Linking Widgets to Data 762
49.5 Dialogs and Messageboxes 762
49.6 Menus 763
49.7 Treeview — Displaying Tabular Data 764
49.8 Embedding matplotlib Charts 765
49.9 Project — Personal Budget Tracker 765
49.10 What You Built 774
49.11 What's Next? 774

50 Chapter 50: Package Your Code and Share It 775
50.1 What is a Package? 775
50.2 Project Structure — The `src` Layout 776
50.3 `pyproject.toml` — The Modern Config File 776
50.4 Write the Package Code 778
50.5 Write Tests 781
50.6 Install in Editable Mode 783
50.7 Semantic Versioning 784
50.8 Write a Good README 784
50.9 Build the Distribution Files 785

50.10 Publish to TestPyPI First . . . 786
50.11 Publish to PyPI . . . 786
50.12 GitHub Actions — Automate Publishing . . . 786
50.13 CI — Test on Every Push . . . 787
50.14 The Full Publishing Workflow . . . 788
50.15 What You Learned in This Chapter . . . 789
50.16 What's Next? . . . 789

51 Chapter 51: Git and Version Control for Python Developers 790
51.1 Why Git Matters . . . 790
51.2 Setup . . . 791
51.3 The Core Concepts . . . 791
51.4 Starting a Repository . . . 791
51.5 The Basic Workflow . . . 792
51.6 `.gitignore` — What NOT to Track . . . 792
51.7 Viewing Changes . . . 794
51.8 Undoing Mistakes . . . 794
51.9 Branches — Work in Isolation . . . 795
51.10 Merge Conflicts — How to Resolve Them . . . 796
51.11 Working with GitHub . . . 796
51.11.1 Push your code . . . 796
51.11.2 Pull changes from GitHub . . . 796
51.11.3 The fetch/pull difference . . . 797
51.12 The Professional Git Workflow . . . 797
51.13 Conventional Commits . . . 798
51.14 Useful Git Commands for Python Developers . . . 799
51.15 `git add -p` — Stage Like a Pro . . . 799
51.16 Git Tags — Mark Releases . . . 800
51.17 A Clean Python Project on GitHub . . . 800
51.18 What You Learned in This Chapter . . . 801
51.19 What's Next? . . . 802

52 Chapter 52: Environment Management and Project Structur803
52.1 The Problem — Why You Need Virtual Environments . . . 803
52.2 `venv` — Python's Built-in Solution . . . 804
52.2.1 Creating `requirements.txt` . . . 804
52.2.2 The problem with `pip freeze` . . . 805
52.3 `pip-tools` — Clean Dependency Management . . . 805
52.4 `uv` — The Modern Fast Alternative . . . 806
52.5 `poetry` — Dependencies + Packaging in One . . . 807
52.6 Choosing a Tool . . . 807
52.7 Professional Project Structure . . . 807
52.8 Environment Variables and Secrets . . . 809
52.8.1 `.env` files . . . 809
52.8.2 `pydantic-settings` — Validated Configuration . . . 809
52.8.3 Never Do This . . . 811
52.8.4 Production Secrets . . . 811
52.9 Managing Multiple Python Versions . . . 811

52.10 A Complete Project Setup Script . . . 812
52.11 The Checklist for Every New Python Project . . . 813
52.12 What You Learned in This Chapter . . . 813
52.13 What's Next? . . . 814

53 Chapter 53: Logging 815
53.1 The Problem with `print()` . . . 815
53.2 The Logging Module — Basics . . . 816
53.3 Log Levels . . . 816
53.4 Named Loggers — The Right Way . . . 817
53.5 Configuring Logging Properly . . . 817
53.6 Logging to Files . . . 818
53.7 Logging Exceptions . . . 820
53.8 Structured Logging — JSON Logs for Production . . . 820
53.9 Adding Context with LoggerAdapter . . . 821
53.10 Complete Production Logging Setup . . . 822
53.11 Logging in FastAPI . . . 825
53.12 What You Learned in This Chapter . . . 826
53.13 What's Next? . . . 827

54 Chapter 54: Design Patterns in Python 828
54.1 Why Design Patterns Matter . . . 828
54.2 Category 1: Creational Patterns . . . 829
54.2.1 Singleton — One Instance Only . . . 829
54.2.2 Factory — Create Objects Without Specifying the Exact Class . . . 830
54.2.3 Builder — Construct Complex Objects Step by Step . . . 831
54.3 Category 2: Structural Patterns . . . 833
54.3.1 Decorator Pattern (vs Python's `@decorator`) . . . 833
54.3.2 Adapter — Make Incompatible Interfaces Work Together 834
54.3.3 Facade — Simplify a Complex System . . . 835
54.4 Category 3: Behavioural Patterns . . . 836
54.4.1 Observer — Notify Multiple Objects of State Changes . . . 836
54.4.2 Strategy — Swap Algorithms at Runtime . . . 838
54.4.3 Command — Encapsulate Requests as Objects . . . 839
54.4.4 Iterator — Traverse Without Exposing Internals . . . 841
54.5 Patterns Python Makes Unnecessary . . . 841
54.6 Pythonic Patterns — The Ones Python Adds . . . 842
54.6.1 Context Manager (already in Chapter 24) . . . 842
54.6.2 Descriptor (already in Chapter 36) . . . 842
54.6.3 Mixin . . . 842
54.7 When to Use Design Patterns . . . 843
54.8 What You Learned in This Chapter . . . 843
54.9 What's Next? . . . 844

55 Chapter 55: Security Best Practices 845
55.1 Mistake 1: Hardcoding Secrets . . . 845
55.2 Mistake 2: SQL Injection . . . 846

55.3 Mistake 3: Storing Passwords Incorrectly 847
55.3.1 Password strength requirements 848
55.4 Mistake 4: Not Validating Input 848
55.4.1 Sanitising HTML output 849
55.5 Mistake 5: Weak or No Authentication 850
55.5.1 Use strong secret keys 850
55.5.2 Secure JWT configuration 850
55.6 Mistake 6: Path Traversal 851
55.7 Mistake 7: Insecure File Uploads 852
55.8 Mistake 8: Dependency Vulnerabilities 853
55.9 Mistake 9: Verbose Error Messages in Production 854
55.10 Mistake 10: HTTPS and Headers 854
55.11 A Security Checklist for Every Python Project 855
55.12 What You Learned in This Chapter 856
55.13 What's Next? 857

56 Chapter 56: Python Internals — How Python Really Works 858
56.1 CPython — The Reference Implementation 858
56.2 How Python Runs Your Code 859
56.2.1 Step 1 — Lexing (Tokenisation) 859
56.2.2 Step 2 — Parsing (AST) 860
56.2.3 Step 3 — Compilation (Bytecode) 860
56.2.4 Step 4 — Execution (CPython VM) 861
56.3 Code Objects — The Compiled Unit 861
56.4 The GIL — Global Interpreter Lock 862
56.4.1 Why does the GIL exist? 862
56.4.2 What the GIL means for you 863
56.4.3 Python 3.13+ — The No-GIL Build 863
56.5 Memory Management 863
56.5.1 Reference Counting 863
56.5.2 Cyclic Garbage Collector 864
56.5.3 Object Interning 865
56.5.4 Memory Profiling 865
56.6 The Import System 866
56.6.1 `__all__` — Controlling What's Exported 866
56.6.2 `if __name__ == "__main__"` 866
56.7 Bytecode Deep Dive 867
56.8 `__slots__` — Memory-Efficient Classes 868
56.9 `__getattr__` vs `__getattribute__` 869
56.10 AST Manipulation — Code That Reads Code 869
56.11 What You Learned in This Chapter 870
56.12 What's Next? 871

57 Chapter 57: The Python Ecosystem Map 872
57.1 Web Development 872
57.1.1 Frameworks 872
57.1.2 HTTP Clients 873
57.1.3 Web Scraping 873

57.2 Data Science and Analytics . . . 874
57.2.1 The Core Stack . . . 874
57.2.2 Databases from Python . . . 874
57.3 Machine Learning and AI . . . 874
57.3.1 Classical ML . . . 874
57.3.2 Deep Learning . . . 875
57.3.3 AI / LLM Tooling . . . 876
57.3.4 Computer Vision . . . 876
57.4 Automation and Scripting . . . 877
57.4.1 Office Automation . . . 877
57.5 Command-Line Tools . . . 877
57.6 Testing and Code Quality . . . 878
57.7 DevOps and Infrastructure . . . 879
57.8 Finance and Quantitative Analysis . . . 880
57.9 Game Development . . . 880
57.10 Embedded and IoT . . . 881
57.11 Scientific and Academic . . . 881
57.12 Async and Concurrency . . . 882
57.13 How to Choose a Library . . . 883
57.14 The 20 Libraries Every Python Developer Should Know . . . 883
57.15 What You Learned in This Chapter . . . 884
57.16 What's Next? . . . 885

58 Chapter 58: Building Your Portfolio and Getting Hired 886
58.1 What Makes a Python Portfolio Stand Out . . . 886
58.2 Five Project Ideas (With Difficulty Levels) . . . 887
58.2.1 1. Personal Finance Tracker API [*][*] . . . 887
58.2.2 2. Web Scraper + Data Dashboard [*][*][*] . . . 887
58.2.3 3. CLI Tool for Developer Productivity [*][*] . . . 887
58.2.4 4. Machine Learning Pipeline [*][*][*] . . . 888
58.2.5 5. Full-Stack Web App [*][*][*][*] . . . 888
58.3 How to Document Your Projects on GitHub . . . 889
58.3.1 1. A compelling README . . . 889
58.4 Architecture . . . 889
58.5 Tech Stack . . . 890
58.5.1 Intermediate . . . 892
58.5.2 Advanced . . . 892
58.6 Coding Challenges — Practice Platforms . . . 893
58.7 Open Source Contributions . . . 894
58.8 The Job Search Strategy . . . 894
58.9 What You Learned in This Chapter . . . 894
58.10 What's Next? . . . 895

59 Chapter 59: Keep Learning — Resources and Community 896
59.1 How to Read the Official Python Docs . . . 896
59.2 Books Worth Reading Next . . . 897
59.2.1 Python-Specific . . . 897
59.2.2 Software Engineering (Language-Agnostic) . . . 897

59.3 Podcasts . . . 898
59.4 YouTube Channels . . . 898
59.5 Newsletters . . . 899
59.6 Communities . . . 899
59.6.1 Online . . . 899
59.6.2 Conferences . . . 900
59.7 How to Read Other People's Code . . . 900
59.8 A Personal Learning Roadmap for the Next 12 Months . . . 901
59.8.1 Months 1-2: Consolidate and Ship . . . 901
59.8.2 Months 3-4: Specialise . . . 901
59.8.3 Months 5-6: Go Deeper . . . 901
59.8.4 Months 7-9: Build in Public . . . 902
59.8.5 Months 10-12: Apply and Iterate . . . 902
59.9 The Most Important Lessons . . . 902
59.10 The Zen of Python — One More Time . . . 903
59.11 A Final Word . . . 903

60 Appendices **905**
60.1 Appendix A: Python Quick Reference . . . 905
60.1.1 Built-in Types . . . 905
60.1.2 Numeric Operations . . . 905
60.1.3 String Methods . . . 906
60.1.4 List Operations . . . 906
60.1.5 Dictionary Operations . . . 906
60.1.6 Set Operations . . . 907
60.1.7 Control Flow . . . 907
60.1.8 Comprehensions . . . 907
60.1.9 Functions . . . 908
60.1.10 Built-in Functions . . . 908
60.2 Appendix B: Standard Library Cheat Sheet . . . 909
60.2.1 `os` — Operating System Interface . . . 909
60.2.2 `pathlib` — File Paths (Modern) . . . 910
60.2.3 `sys` — System Information . . . 910
60.2.4 `datetime` — Dates and Times . . . 911
60.2.5 `collections` — Specialised Containers . . . 911
60.2.6 `itertools` — Iterator Building Blocks . . . 912
60.2.7 `functools` — Higher-Order Functions . . . 912
60.2.8 `json` — JSON Encoding/Decoding . . . 913
60.2.9 `re` — Regular Expressions . . . 913
60.2.10 `subprocess` — Run System Commands . . . 914
60.2.11 `typing` — Type Hints . . . 914
60.3 Appendix C: Virtual Environment and Project Setup Cheat Sheet 915
60.3.1 Quick Start (venv) . . . 915
60.3.2 Quick Start (uv — recommended) . . . 915
60.3.3 Project Structure Template . . . 916
60.3.4 Python `.gitignore` (essential lines) . . . 916
60.4 Appendix D: Git Quick Reference . . . 917
60.4.1 Conventional Commits . . . 918

60.5 Appendix E: pytest Quick Reference 918
60.6 Appendix F: f-String Formatting Reference 920

61 Glossary **921**

62 Index **928**
62.1 A 928
62.2 B 928
62.3 C 929
62.4 D 929
62.5 E 930
62.6 F 930
62.7 G 931
62.8 H 931
62.9 I 931
62.10 J 932
62.11 K 932
62.12 L 932
62.13 M 932
62.14 N 932
62.15 O 933
62.16 P 933
62.17 R 933
62.18 S 934
62.19 T 934
62.20 U 935
62.21 V 935
62.22 W 935
62.23 Y 935
62.24 Z 935

Copyright

(c) 2026 Abhilash Sahoo. All rights reserved.

This book is self-published. No part of this book may be reproduced or used without written permission from the author. Python is a trademark of the Python Software Foundation. This book is for educational purposes. The author is not responsible for any errors or omissions.

Dedication

For everyone who wants to learn—and for my grandma.

Epigraph

“The only way to learn a new programming language is by writing programs in it.”
— Dennis Ritchie

Preface

You're here because you want to learn Python—and you don't want to get lost in jargon or long explanations.

I wrote this book for you. Maybe you're in fifth grade and curious about coding. Maybe you're a grandparent who's always wanted to try. Maybe you're someone who's tried other books and found them too heavy. Here, we keep things simple.

What you'll get

- Short, clear steps. One idea at a time.
- Lots of examples. You'll see code, run it, and change it.
- Plain English. If a word is technical, I'll explain it right away.
- A path from zero (no Python) to hero (building small programs on your own).

How to use this book

- Read in order. Each chapter builds on the last.
- Type the code yourself. Don't just read it—run it.
- Try the "Your turn" bits. They're short and meant to stick.

No prior programming experience is needed. Just a computer and a little time.

Let's go.

Acknowledgments

Thank you to everyone who made this book possible.

To the readers who asked for a simpler way to learn Python—you inspired this.

To the people who read early drafts and said what was clear and what wasn't—your feedback shaped every page.

To my family, for the time and support to write.

And to you, for picking up this book and giving it a try.

Introduction

0.1 What you'll learn

By the end of this book, you will:

- Install Python and run your first program.
- Use variables, numbers, and text.
- Do simple calculations (add, subtract, multiply, divide) with numbers and variables.
- Make decisions and repeat steps (so the computer does the work).
- Store data in lists and use it.
- Define your own steps (functions) and reuse them.
- Read and write files.
- Build a few small projects from scratch.

We go step by step. No big jumps. Each chapter has one clear goal.

0.2 How this book is organized

- **Part 1** gets you set up and running ("Hello, World" and the basics).
- **Part 2** covers the core ideas: variables, conditions, loops, and data.
- **Part 3** adds structure: functions, files, and simple projects.

At the back you'll find appendices (like installing Python and a cheat sheet), a glossary, and an index.

0.3 How to read this book

1. **Start at the beginning.** Don't skip chapters.
2. **Type the code.** Running it yourself makes it stick.
3. **Do the "Your turn" exercises.** They're short and fun.
4. **Use the glossary** if a word is new. We keep language simple, but a quick definition can help.

You're not in a hurry. Take your time. When something is unclear, re-read the example and try changing it a little. That's how you learn.

Ready? Turn the page and let's start from zero.

Chapter 1

Chapter 1: Your First Python Program

This chapter has one job: get Python running and write your first line of code. By the end, you will see output on your screen, understand what just happened, and be ready for everything that follows.

Let's go.

1.1 What is Python?

Python is a language for telling computers what to do.

You write instructions in a file. Python reads them, one line at a time, and carries them out. That's the whole idea.

Here is the world's most famous Python program:

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

One line. One result. That's real Python code — the same language used by Google, Instagram, NASA, and Netflix.

Python is popular for a reason: it reads almost like plain English. Compare it to some other languages and you'll see why beginners love it. But don't let "beginner-friendly" fool you. The same language that teaches you to code is also used to build rockets and train AI models.

In plain English: Python is a set of instructions you write, and the computer follows them. Like a recipe, but for a machine.

1.2 How Python Runs Your Code

Before you write your first program, I want you to understand one simple idea: Python is an *interpreter*.

That means Python reads your code line by line and runs each line immediately, top to bottom.

```
print("Line 1")
print("Line 2")
print("Line 3")
```

Output:

```
Line 1
Line 2
Line 3
```

Python ran line 1, printed it, moved to line 2, printed it, then line 3. In order. Every time.

This is different from languages like C or Java, where you compile everything first into a separate program. With Python, you just run your file and it goes. That's why it's so fast to test ideas.

In plain English: Python reads your code the way you read a book — top to bottom, one line at a time.

1.3 Three Ways to Run Python

You have three options. Pick the one that feels right for you. You can always switch later.

1.3.1 Option A: Install Python on Your Computer

This is the most powerful option. Your code runs on your own machine, you can save files anywhere, and you're not dependent on the internet.

Step 1 — Download Python

Go to python.org in your browser. Click **Downloads**. The site will detect your operating system (Windows, Mac, or Linux) and suggest the right version. Download it.

Step 2 — Install Python

Run the installer you just downloaded.

- **Windows:** Before you click "Install Now," check the box that says **"Add Python to PATH."** This is important. If you skip it, your computer won't know where to find Python. Then click "Install Now."
- **Mac:** Run the `.pkg` file and follow the steps.
- **Linux:** Python is often pre-installed. Open a terminal and type `python3 --version` to check.

Step 3 — Verify the installation

Open a terminal (Windows: search for "Command Prompt" or "PowerShell"; Mac/Linux: open "Terminal") and type:

```
python --version
```

You should see something like:

```
Python 3.12.4
```

The exact numbers will differ — anything 3.10 or higher is perfect for this book.

Your turn: Run the check above. Write down the version number you see. If you see an error instead, jump to the Appendix (page XX) for troubleshooting steps.

1.3.2 Option B: Run Python in Your Browser (Google Colab or Kaggle)

You don't need to install anything. Open a website, type code, run it. Done.

This is the fastest way to start, and it's completely free.

Google Colab

1. Go to colab.research.google.com.
2. Sign in with a Google account (or create one — it's free).
3. Click **New notebook**.
4. Click inside the gray code cell, type `print("Hello, World!")`, and press **Shift+Enter**.

You'll see the output appear right below the cell:

```
Hello, World!
```

That's it. You just ran Python in a browser.

Kaggle

1. Go to kaggle.com and create a free account.
2. Click **Code** in the left menu, then **New Notebook**.
3. In the code cell that appears, type `print("Hello, World!")` and click the **Run** button (or press **Shift+Enter**).

In plain English: Colab and Kaggle are like Python running inside a webpage. You type code, they run it, you see the result. No installation. No setup. Perfect for right now.

Your turn: If you chose this option, run `print("Hello, World!")` in a cell. See the output. You're done with setup — jump ahead to "Your First Program" below.

1.3.3 Option C: VS Code with Jupyter

VS Code is a free code editor used by millions of professional developers. The Jupyter extension lets you run Python in "cells" — small runnable chunks — right inside the editor. This gives you the best of both worlds: a real editor on your computer, with the cell-by-cell style of Colab.

You need Python installed first (Option A), then:

1. Download **VS Code** from code.visualstudio.com and install it.
2. Open VS Code. Click the **Extensions** icon on the left sidebar (or press **Ctrl+Shift+X** on Windows, **Cmd+Shift+X** on Mac).
3. Search for **"Jupyter"** and install the one by Microsoft.
4. Search for **"Python"** and install the one by Microsoft too.
5. Go to **File -> New File**, save it as `hello.ipynb` (the `.ipynb` extension makes it a Jupyter notebook).
6. VS Code will show you a notebook with a code cell. Type `print("Hello, World!")` in the cell and press **Shift+Enter** or click the play button.

Output appears right below the cell:

```
Hello, World!
```

Your turn: Run `print("Hello, World!")` in a notebook cell. If it works, you're set up and ready.

1.4 Your First Program

No matter which option you chose, let's write a real program together — one that does something slightly more interesting than printing a single line.

If you're using Colab, Kaggle, or VS Code with Jupyter, type this into a code cell. If you installed Python and prefer plain files, open a text editor (not Word — Notepad, VS Code, or any plain-text editor), type this, and save it as `hello.py`.

```
print("Hello, World!")
print("My name is Python.")
print("I will teach you everything.")
```

Run it. You should see:

```
Hello, World!
My name is Python.
I will teach you everything.
```

Three lines of code, three lines of output. Python ran each `print` statement top to bottom, exactly as you wrote it.

What `print()` does: It takes whatever you put inside the parentheses and displays it on the screen. The text you want to show must be wrapped in quotes — either single (`'`) or double (`"`). Both work the same way.

```
print("Double quotes work.")
print('Single quotes work too.')
```

Output:

```
Double quotes work.
Single quotes work too.
```

Your turn: Change the messages to say something about yourself. Add a fourth `print` line. Run it and see your custom output.

1.5 Reading Error Messages Without Panic

Here's a truth no other book tells beginners early enough: you *will* see error messages. Every programmer does, every single day. An error message is not a sign you're doing it wrong — it's Python telling you exactly what it needs from you.

Let's make an error on purpose so you know what to expect.

Type this and run it:

```
print("Hello, World!"
```

You'll see something like:

```
  File "hello.py", line 1
    print("Hello, World!"
                         ^
SyntaxError: '(' was never closed
```

That looks scary the first time. Let's read it together.

- **File "hello.py", line 1** — The error is on line 1 of your file. Python tells you exactly where to look.
- **The little arrow ^** — It points to the spot where Python got confused.
- **SyntaxError: '(' was never closed** — "Syntax" means the grammar of the code. Python is saying: "You opened a parenthesis but never closed it." The fix is to add the missing `)`.

Fixed code:

```
print("Hello, World!")
```

Here are the three most common errors you'll see as a beginner:

SyntaxError — You made a typo or forgot something (a closing quote, a parenthesis, a colon). Python couldn't even understand the code.

```
print("Hello)   # missing closing quote --- SyntaxError
```

NameError — You used a word Python doesn't recognize, usually because you mistyped a function name or used a variable before creating it.

```
Print("Hello")  # capital P --- NameError: name 'Print' is not defined
```

Python is case-sensitive. `print` and `Print` are completely different things.

IndentationError — Python uses spacing (indentation) to understand the structure of your code. If the spacing is wrong, Python complains. You'll understand this more in later chapters when we start writing `if` statements and loops.

The habit to build now: When you see an error, read the last line first. It tells you the type of error and a short description. Then look at the line number. Go to that line and look for the issue. Nine times out of ten, the fix takes five seconds once you know where to look.

In plain English: An error message is Python's way of saying "I got confused here, and here's why." It's helpful, not hostile.

1.6 How to Use This Book

Before we move on, a quick word on how to get the most out of every chapter.

Read with your hands. Don't just read the code — type it. Typing code yourself is how it sticks. Copy-pasting is fine sometimes, but typing forces you to read every character.

Do every "Your turn" exercise. These are not optional. They're short, they're focused, and they're where the real learning happens.

Don't skip chapters. Each chapter builds on the one before it. Chapter 3 assumes you know Chapter 2. The structure is deliberate.

Break things on purpose. Once you get an example working, change something and see what happens. Delete a quote. Change a word. This is the fastest way to understand what each part of the code actually does.

Google is not cheating. Every professional developer searches for answers constantly. The skill is knowing what to search for and how to read what comes back. You'll build that skill naturally as you go.

1.7 What You Learned in This Chapter

- Python is an interpreter that reads and runs your code line by line, top to bottom.
- You can run Python three ways: installed locally, in a browser (Colab/Kaggle), or in VS Code with Jupyter.
- `print()` displays text on the screen.
- Text in `print()` must be wrapped in quotes — single or double, both work.
- Error messages tell you what went wrong and where. Read the last line first, then find the line number.
- Python is case-sensitive: `print` and `Print` are not the same.

1.8 What's Next?

You've got Python running and you've written your first program. Now it's time to make programs do more than just print fixed messages.

In Chapter 2, you'll learn about **variables** — a way to store information so your program can remember and use it. Instead of printing `"Hello, World!"` forever, you'll store your name in a variable and print that. A small step that opens up everything else.

Your turn: Before you go, try this. Write a program that prints three things: your name, your city, and one thing you want to learn this year. Save it, run it, and see your output. That's your first real program.

Chapter 2

Chapter 2: Variables, Numbers, and Text

In Chapter 1 you made Python print a message. That's a start, but it's a fixed message — the same every time. Real programs need to *remember* things: a user's name, a price, a score, a date. That's what variables are for.

By the end of this chapter you'll know how to store text and numbers, name them properly, check what type of data you have, and do some basic things with strings. Everything from here on uses these ideas, so let's get them solid.

2.1 What is a Variable?

A variable is a name that holds a value.

Think of it like a labeled box. You write a name on the outside (say, `age`), and you put something inside (say, `25`). Whenever you say `age` in your program, Python opens that box and hands you `25`.

```
age = 25
print(age)
```

Output:

```
25
```

The = sign is called the **assignment operator**. It means "put this value into this name." The name goes on the left, the value on the right.

You can change what's in the box at any time:

```
age = 25
print(age)

age = 26
print(age)
```

Output:

```
25
26
```

Python just overwrites the old value. The variable `age` now holds `26`.

In plain English: A variable is a name you choose. You decide what to store under that name, and you can change it whenever you want.

Your turn: Create a variable called `score`, put the number `100` in it, and print it. Then change `score` to `95` and print it again. You should see both numbers.

2.2 Naming Rules and Conventions

You can name a variable almost anything — but there are rules, and there are habits that make your code easier to read.

The rules (Python enforces these):

- Names can contain letters, numbers, and underscores (_).
- Names cannot start with a number.
- Names cannot have spaces.
- Names are case-sensitive: `score`, `Score`, and `SCORE` are three different variables.

```
first_name = "Maya"    # valid
score2 = 100           # valid --- number at the end is fine
2score = 100           # INVALID --- starts with a number
my score = 100         # INVALID --- spaces not allowed
```

The conventions (good habits):

- Use `snake_case`: all lowercase, words separated by underscores.
  - Good: `user_name`, `total_price`, `number_of_students`
  - Avoid: `UserName`, `totalPrice`, `NumberOfStudents` (those styles exist in Python but are used for other things — you'll see this in the OOP chapters)
- Use descriptive names. `age` is better than `a`. `monthly_salary` is better than `ms`.
- Avoid single letters except in short loops (more on that in Chapter 5).

```
# Hard to read
x = "Ravi"
y = 28
z = 55000.0

# Easy to read
employee_name = "Ravi"
employee_age = 28
monthly_salary = 55000.0
```

Both versions work. The second one tells you what the values mean without any guesswork.

Your turn: Write three variables using snake_case — one for a person's name, one for their city, and one for their age. Print all three on one line using commas inside `print()`.

2.3 Storing Numbers: Integers and Floats

Python has two main types of numbers.

Integers are whole numbers — no decimal point.

```
apples = 5
temperature = -3
year = 2026
```

Floats are numbers with a decimal point.

```
price = 2.99
height = 1.75
pi = 3.14159
```

You don't have to declare which type you want. Python figures it out from the value you assign.

```
whole = 10
decimal = 10.0

print(whole)    # 10
print(decimal)  # 10.0
```

Even though both hold "ten," they are different types. `10` is an integer. `10.0` is a float. This matters sometimes — you'll see why in the next chapter.

Large numbers: Python lets you use underscores as visual separators in big numbers. It doesn't change the value — it just makes them easier to read.

```
population = 1_400_000_000
distance_km = 384_400

print(population)     # 1400000000
print(distance_km)    # 384400
```

Negative numbers: Put a minus sign in front.

```
temperature = -18
bank_balance = -500
```

Your turn: Create a variable for the price of a coffee (a float), the number of coffees you drink per week (an integer), and the year you were born (an integer). Print all three.

2.4 Storing Text: Strings

Text in Python is called a **string**. You create a string by wrapping text in quotes — single or double, both work.

```
first_name = "Priya"
last_name = 'Sharma'

print(first_name)    # Priya
print(last_name)     # Sharma
```

Joining strings together is called **concatenation**. Use the `+` operator.

```
first_name = "Priya"
last_name = "Sharma"
full_name = first_name + " " + last_name

print(full_name)    # Priya Sharma
```

Notice the `" "` in the middle — that's a string with just a space. Without it you'd get `PriyaSharma`.

Printing variables and text together:

The easiest way in modern Python is an **f-string** (short for "formatted string"). Put an `f` before the opening quote, then use `{}` to drop a variable right into the text.

```
name = "Carlos"
age = 31

print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Carlos and I am 31 years old.
```

The `{}` is a placeholder. Python replaces it with the variable's value at the moment the line runs. This is clean, readable, and the way most professional Python code is written today.

You can also use commas in `print()` — Python adds a space between each item:

```
name = "Carlos"
age = 31

print("My name is", name, "and I am", age, "years old.")
```

Output:

```
My name is Carlos and I am 31 years old.
```

Both approaches give the same output here. F-strings are more flexible for complex formatting, so get into the habit of using them.

Your turn: Create variables for your name and your favorite food. Use an f-string to print a sentence like: `My name is ___ and my favorite food is ___`.

2.5 Escape Sequences and Special Characters

Sometimes you need to put special characters inside a string — a newline, a tab, or a quote mark. You do this with **escape sequences**: a backslash `\` followed by a letter.

Escape	What it does	Example
`\n`	New line	`"Line 1\nLine 2"`
`\t`	Tab (indent)	`"Name:\tCarlos"`
`\\`	A literal backslash	`"C:\\Users\\file.txt"`
`\"`	A literal double quote inside double quotes	`"He said \"hello\""`
`\'`	A literal single quote inside single quotes	`'It\'s fine'`

```
print("Line 1\nLine 2\nLine 3")
```

Output:

```
Line 1
Line 2
```

```
Line 3
```

```
print("Name:\tCarlos\nAge:\t31")
```

Output:

```
Name:   Carlos
Age:    31
```

Raw strings: If you don't want escape sequences processed — for example, when writing file paths on Windows — put an `r` before the opening quote.

```
path = r"C:\Users\Carlos\Documents"
print(path)
```

Output:

```
C:\Users\Carlos\Documents
```

Without the `r`, Python would try to interpret `\U`, `\C`, `\D` as escape sequences and you'd get strange results or an error.

2.6 Multiline Strings

Use triple quotes (`"""` or `'''`) to write a string that spans multiple lines.

```
message = """Dear Carlos,

Welcome to Python.
We hope you enjoy the book.

Best,
The Author"""

print(message)
```

Output:

```
Dear Carlos,

Welcome to Python.
We hope you enjoy the book.

Best,
The Author
```

Everything between the triple quotes — including line breaks — is part of the string. This is useful for long messages, documentation, and templates.

2.7 Checking the Type of a Variable

You've stored integers, floats, and strings. Python always knows what type each variable is. You can ask it with the built-in `type()` function.

```
age = 25
price = 9.99
name = "Maya"

print(type(age))     # <class 'int'>
print(type(price))   # <class 'float'>
print(type(name))    # <class 'str'>
```

- `int` — integer (whole number)
- `float` — floating-point number (decimal)
- `str` — string (text)

You'll see more types as the book progresses (`bool`, `list`, `dict`, and others), but these three are the ones you'll use most.

Why does type matter? Because Python behaves differently depending on type. Look at this:

```
print(10 + 5)        # 15  (addition)
print("10" + "5")    # 105 (joining two strings --- not math!)
```

`10` (integer) plus `5` (integer) gives you `15`. But `"10"` (string) plus `"5"` (string) gives you `"105"` — because joining two strings just sticks them together.

This surprises almost every beginner at least once. Now you know why.

Converting between types: You can convert a value from one type to another using `int()`, `float()`, and `str()`.

```
x = "42"
print(type(x))         # <class 'str'>

y = int(x)
print(type(y))         # <class 'int'>
print(y + 8)           # 50

n = 100
text = str(n)
print("The number is " + text)    # The number is 100
```

Without `str(n)`, that last line would crash — you can't join a string and an integer with `+`. The conversion makes it work.

Your turn: Create variables of three different types: one integer, one float, one string. Print the `type()` of each. Then try adding two of them without converting and see the error you get. Read the error message. What does it tell you?

2.8 Putting It All Together

Here is a short program that uses everything from this chapter:

```
# A simple profile program
first_name = "Amara"
last_name = "Osei"
age = 27
height = 1.68
city = "Accra"

full_name = first_name + " " + last_name

print(f"Name:   {full_name}")
print(f"Age:    {age}")
print(f"Height: {height} m")
print(f"City:   {city}")
```

Output:

```
Name:   Amara Osei
Age:    27
Height: 1.68 m
City:   Accra
```

Clean. Readable. Every variable has a descriptive name. The f-strings make the output easy to format.

Your turn: Write your own profile program. At minimum, include: your name (two variables combined), your age, one decimal value (height, weight, shoe size — your choice), and your city. Print it with f-strings.

2.9 What You Learned in This Chapter

- A variable is a name that holds a value. Use = to assign a value.
- Variable names: lowercase, `snake_case`, descriptive, no spaces, can't start with a number.

- Python has two number types: `int` (whole numbers) and `float` (decimals).
- Text is called a string (`str`). Wrap it in single or double quotes.
- F-strings (`f"... {variable} ..."`) are the best way to mix text and variables.
- Escape sequences: `\n` (newline), `\t` (tab), `\\` (backslash), `\"` (quote).
- Triple quotes (`"""`) create multiline strings.
- `type()` tells you what kind of data a variable holds.
- You can convert between types with `int()`, `float()`, and `str()`.
- Python treats `10 + 5` (math) and `"10" + "5"` (joining text) very differently.

2.10 What's Next?

You can now store data. The next step is to *do things* with it.

In Chapter 3, you'll learn how to do calculations — add, subtract, multiply, divide — and use the results in your programs. You'll also meet some useful built-in tools like `abs()`, `round()`, and the `math` module. If you thought variables were useful, wait until you combine them with math.

Your turn: Before moving on, write a program with at least four variables. Use two numbers, one float, and one string. Print a sentence using f-strings that uses all of them. Run it. If it works, you're ready for Chapter 3.

Chapter 3

Chapter 3: Calculations and Operators

In Chapter 2 you stored numbers in variables. Now you'll put them to work.

Python is an extraordinary calculator. It handles basic arithmetic, follows the right order of operations, gives you shortcuts for common patterns, and comes with built-in tools for rounding, finding extremes, and doing advanced math. By the end of this chapter you'll be able to write programs that compute real things — prices, discounts, averages, distances.

3.1 The Seven Arithmetic Operators

Python has seven operators for working with numbers. Let's go through all of them.

3.1.1 Addition, Subtraction, Multiplication, Division

These four work exactly as you'd expect.

```
print(10 + 3)    # 13
print(10 - 3)    # 7
print(10 * 3)    # 30
print(10 / 3)    # 3.3333333333333335
```

One thing to notice: dividing two integers with `/` always gives you a **float**, even if the result is a whole number.

```
print(10 / 2)    # 5.0  (not 5)
```

3.1.2 Floor Division (`//`)

Floor division divides and then rounds *down* to the nearest whole number. It always gives you an integer (if both inputs are integers).

```
print(10 // 3)    # 3   (not 3.33...)
print(17 // 5)    # 3
print(20 // 4)    # 5
```

This is useful when you need a whole number result — for example, figuring out how many full boxes fit in a truck.

```
items = 47
box_size = 10
full_boxes = items // box_size
print(full_boxes)    # 4
```

3.1.3 Modulo (`%`) — The Remainder

The modulo operator gives you the **remainder** after division.

```
print(10 % 3)    # 1   (10 divided by 3 is 3, remainder 1)
print(17 % 5)    # 2   (17 divided by 5 is 3, remainder 2)
print(20 % 4)    # 0   (20 divided by 4 is exactly 5, no remainder)
```

Two classic uses: checking if a number is even or odd, and figuring out leftovers.

```
# Even or odd?
number = 7
print(number % 2)    # 1 means odd, 0 means even

# Leftovers after packing boxes
items = 47
box_size = 10
leftover = items % box_size
print(leftover)    # 7 items left over
```

3.1.4 Exponentiation (`**`) — Powers

`**` raises a number to a power.

```
print(2 ** 3)     # 8   (2 x 2 x 2)
print(5 ** 2)     # 25  (5 squared)
print(9 ** 0.5)   # 3.0 (square root of 9)
```

That last one is a neat trick: raising to the power of `0.5` gives you the square root.

In plain English: Think of `**` as "to the power of." `2 ** 10` means 2 to the power of 10, which is 1024.

Your turn: Calculate how many seconds are in a week using only multiplication. (Hint: 60 seconds x 60 minutes x 24 hours x 7 days.) Store the result in a variable called `seconds_in_a_week` and print it.

3.2 Order of Operations

Python follows the same order of operations you learned in school: exponentiation first, then multiplication and division, then addition and subtraction. If you've seen the acronym PEMDAS or BODMAS, that's what this is.

```
print(2 + 3 * 4)     # 14, not 20 --- multiplication happens first
print((2 + 3) * 4)   # 20 --- parentheses force addition first
```

When in doubt, use parentheses. They make your intention clear and remove any ambiguity.

```
# Ambiguous
result = 100 - 20 / 4 + 5 * 2

# Clear
result = 100 - (20 / 4) + (5 * 2)
print(result)    # 100 - 5.0 + 10 = 105.0
```

A real example: calculating a discounted price with tax.

```
original_price = 200
discount = 30
tax_rate = 0.08

final_price = (original_price - discount) * (1 + tax_rate)
print(final_price)    # 183.6
```

Without the parentheses, the math would be wrong. With them, it's obvious what happens: subtract the discount first, then apply the tax.

Your turn: Write a program that calculates the area of a rectangle (length x width) and the area of a circle (pi x radius^2). Use `3.14159` for pi. Store each result in a variable and print it.

3.3 Using Variables in Calculations

Variables make calculations reusable and readable. Instead of hardcoding numbers, you name them.

```
hourly_rate = 25.0
hours_worked = 40
weekly_pay = hourly_rate * hours_worked

print(f"Weekly pay: ${weekly_pay}")
```

Output:

```
Weekly pay: $1000.0
```

Now if the hourly rate changes, you only update one line. The calculation stays the same.

You can also use a variable's own value to update it:

```
score = 50
score = score + 10   # take the old value, add 10, store the result back
print(score)         # 60
```

This is a very common pattern — you'll use it constantly in loops.

3.4 Augmented Assignment Operators

Because updating a variable with its own value is so common, Python gives you a shorthand.

Long form	Short form	Meaning
`x = x + 5`	`x += 5`	add 5 to x
`x = x - 5`	`x -= 5`	subtract 5 from x
`x = x * 5`	`x *= 5`	multiply x by 5
`x = x / 5`	`x /= 5`	divide x by 5
`x = x // 5`	`x //= 5`	floor-divide x by 5
`x = x % 5`	`x %= 5`	replace x with x mod 5
`x = x ** 5`	`x **= 5`	raise x to the power 5

```
lives = 3
print(lives)   # 3

lives -= 1
print(lives)   # 2

lives -= 1
print(lives)   # 1
```

```
price = 100.0
price *= 1.15   # apply 15% markup
print(price)    # 115.0
```

Your turn: Start with a variable `balance = 1000`. Apply these operations one at a time, printing after each: add 250, subtract 75, multiply by 1.05 (5% interest), floor-divide by 1. What's the final balance?

3.5 Integer vs Float in Calculations

When you mix integers and floats in a calculation, Python always returns a float.

```
print(5 + 2.0)     # 7.0   (not 7)
print(10 * 0.5)    # 5.0
print(9 / 3)       # 3.0   (division always returns float)
print(9 // 3)      # 3     (floor division returns int if both inputs are int)
```

This rarely causes problems, but it's good to know. If you need a guaranteed integer, wrap the result in `int()`.

```
result = int(9 / 3)
print(result)        # 3
print(type(result))  # <class 'int'>
```

3.6 Built-in Math Functions

Python comes with several built-in functions for common math operations — no import needed.

3.6.1 `abs()` — Absolute Value

Returns the positive version of a number.

```
print(abs(-42))    # 42
print(abs(42))     # 42
print(abs(-3.14))  # 3.14
```

Useful for things like finding the distance between two points (distance is always positive).

```
point_a = 10
point_b = 35
distance = abs(point_a - point_b)
print(distance)    # 25
```

3.6.2 `round()` — Rounding

Rounds a float to the nearest integer, or to a specified number of decimal places.

```
print(round(3.7))        # 4
print(round(3.2))        # 3
print(round(3.14159, 2))  # 3.14
print(round(3.14159, 4))  # 3.1416
```

The second argument to `round()` is optional — it tells Python how many decimal places to keep.

```
price = 19.9999
print(f"${round(price, 2)}")    # $20.0
```

3.6.3 `min()` and `max()` — Smallest and Largest

`min()` returns the smallest value, `max()` returns the largest.

```
print(min(4, 7, 2, 9, 1))    # 1
print(max(4, 7, 2, 9, 1))    # 9
```

You can also compare two variables:

```
score_alice = 88
score_bob = 94

winner_score = max(score_alice, score_bob)
print(f"Top score: {winner_score}")    # Top score: 94
```

3.6.4 `pow()` — Power (alternative to `**`)

`pow(base, exponent)` does the same as `base ** exponent`.

```
print(pow(2, 8))    # 256
print(2 ** 8)       # 256  --- same result
```

You'll see both in real code. `**` is more common.

Your turn: You have five test scores: 72, 88, 95, 61, 80. Find the highest score, the lowest score, and the rounded average. Store each in a variable and print them with f-strings.

3.7 The `math` Module

For more advanced math, Python has a built-in module called `math`. You load it with one line at the top of your file.

```
import math
```

Then you can use `math.` followed by the function name.

3.7.1 Square Roots

```
import math

print(math.sqrt(25))     # 5.0
print(math.sqrt(144))    # 12.0
print(math.sqrt(2))      # 1.4142135623730951
```

3.7.2 Ceiling and Floor

`math.ceil()` rounds *up* to the next integer. `math.floor()` rounds *down*.

```
import math

print(math.ceil(3.1))     # 4
print(math.ceil(3.9))     # 4
print(math.floor(3.9))    # 3
print(math.floor(3.1))    # 3
```

This is different from `round()`, which goes to the nearest value. `ceil` always goes up. `floor` always goes down.

Real use: you're packaging 47 items into boxes of 6. How many boxes do you need?

```
import math

items = 47
box_size = 6
boxes_needed = math.ceil(items / box_size)
print(boxes_needed)    # 8  (you can't have a partial box)
```

3.7.3 Pi and Euler's Number

```
import math

print(math.pi)     # 3.141592653589793
print(math.e)      # 2.718281828459045
```

```
import math

radius = 5
area = math.pi * radius ** 2
print(round(area, 2))    # 78.54
```

3.7.4 Logarithms and Trigonometry

```
import math

print(math.log(100, 10))    # 2.0  (log base 10 of 100)
print(math.log(math.e))     # 1.0  (natural log of e)

print(math.sin(math.pi / 2))    # 1.0
print(math.cos(0))               # 1.0
```

You won't need these often as a beginner, but they're there when you do.

Your turn: Use the `math` module to calculate the hypotenuse of a right triangle with sides 3 and 4. (Hint: `math.sqrt(3**2 + 4**2)`). The answer should be `5.0`.

3.8 A Real-World Example: Shopping Calculator

Let's combine everything from this chapter into one small program.

```
import math

item_name = "Wireless Headphones"
original_price = 149.99
discount_percent = 20
tax_rate = 0.08

# Apply discount
discount_amount = original_price * (discount_percent / 100)
discounted_price = original_price - discount_amount

# Apply tax
final_price = discounted_price * (1 + tax_rate)

# Round to 2 decimal places
final_price = round(final_price, 2)

print(f"Item:           {item_name}")
print(f"Original price: ${original_price}")
print(f"Discount:       {discount_percent}% (${round(discount_amount, 2)})")
print(f"After discount: ${round(discounted_price, 2)}")
print(f"Tax ({int(tax_rate * 100)}%):      ${round(final_price - discounted_price,
```

```
print(f"Final price:    ${final_price}")
```

Output:

```
Item:           Wireless Headphones
Original price: $149.99
Discount:       20% ($30.0)
After discount: $119.99
Tax (8%):       $9.6
Final price:    $129.59
```

Real math. Real output. Everything you've learned so far, working together.

Your turn: Modify this program to use your own item, price, and discount. Try changing the tax rate to `0.10` (10%) and see what happens to the final price.

3.9 What You Learned in This Chapter

- Python has 7 arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`
- `/` always returns a float. `//` returns the whole-number part of a division.
- `%` gives the remainder — great for checking if a number is even or odd.
- `**` raises to a power. `9 ** 0.5` gives the square root.
- Python follows PEMDAS order. Use parentheses to be explicit.
- Augmented operators (`+=`, `-=`, `*=`, `/=`, etc.) update a variable using its own value.
- Mixing an int and a float in a calculation gives you a float.
- Built-in functions: `abs()`, `round()`, `min()`, `max()`, `pow()`
- The `math` module: `sqrt()`, `ceil()`, `floor()`, `pi`, `e`, `log()`, trig functions.

3.10 What's Next?

So far your programs run straight through from top to bottom. They don't make decisions — they just calculate.

In Chapter 4, you'll change that. You'll use `if`, `elif`, and `else` to make programs that do different things depending on conditions. Is the score above 50? Print "Pass." Is it below? Print "Fail." Your programs are about to get a lot more interesting.

Your turn: Before moving on, write a program that takes a circle's radius (store it in a variable), then calculates and prints: the area (`pi x r^2`), the circumference (`2 x pi x r`), and the diameter (`2 x r`). Round all results to 2 decimal places. Use the `math` module for `pi`.

Chapter 4

Chapter 4: Making Decisions — `if`, `elif`, `else`

So far your programs have run top to bottom like a straight road — same steps, same result, every time. That's useful, but it's not how real programs work.

Real programs *decide*. A login page checks if your password is correct. A weather app shows "bring an umbrella" or "enjoy the sunshine." A game says "You won!" or "Try again." All of those are decisions — and in Python, decisions are built with `if`.

By the end of this chapter, your programs will react to data and to the people who use them.

4.1 Boolean Values: True and False

Before you can make a decision, you need to understand how Python represents "yes" and "no."

Python has a data type called `bool` (short for Boolean, named after mathematician George Boole). A bool has exactly two possible values: `True` and `False`. Capital T, capital F. That matters — Python is case-sensitive.

```
is_raining = True
is_sunny = False

print(is_raining)    # True
print(is_sunny)      # False
print(type(is_raining))   # <class 'bool'>
```

You'll see booleans constantly. Every comparison you make in Python — "is x greater than y?" "is this string equal to that string?" — gives you back a `True` or a `False`.

4.2 Comparison Operators

A comparison operator looks at two values and answers a yes/no question. The answer is always `True` or `False`.

Operator	Meaning	Example	Result
`==`	equal to	`5 == 5`	`True`
`!=`	not equal to	`5 != 3`	`True`
`<`	less than	`3 < 5`	`True`
`>`	greater than	`5 > 3`	`True`
`<=`	less than or equal to	`5 <= 5`	`True`
`>=`	greater than or equal to	`6 >= 10`	`False`

```
print(10 == 10)    # True
print(10 == 9)     # False
print(10 != 9)     # True
print(10 > 9)      # True
print(10 < 9)      # False
print(10 >= 10)    # True
print(10 <= 9)     # False
```

The most common beginner mistake: using = when you mean ==.

- = is assignment: "put this value into this variable."
- == is comparison: "are these two values equal?"

```
age = 18        # assignment --- storing 18 in age
age == 18       # comparison --- asking if age equals 18 (returns True)
```

Mix them up and Python will either crash or do something unexpected. Watch for it.

You can compare strings too:

```
name = "Alice"
print(name == "Alice")    # True
print(name == "alice")    # False --- case-sensitive
print(name != "Bob")      # True
```

Your turn: Try five comparisons in Python — mix numbers and strings. Predict the result before you run each one, then check.

4.3 Getting Input from the User

To make useful decisions, programs often need information from the person running them. Python's `input()` function handles this.

```
name = input("What is your name? ")
print(f"Hello, {name}!")
```

When this runs, Python prints the message, waits for the user to type something and press Enter, then stores whatever they typed as a string in `name`.

Important: `input()` always returns a string. Even if the user types a number.

```
answer = input("How old are you? ")
print(type(answer))    # <class 'str'>
```

If you need a number, convert it with `int()` or `float()`:

```
age_text = input("How old are you? ")
age = int(age_text)
print(f"Next year you will be {age + 1}.")
```

Or more concisely — wrap `input()` directly inside `int()`:

```
age = int(input("How old are you? "))
print(f"Next year you will be {age + 1}.")
```

Both do the same thing. The second style is what you'll see most often in real code.

What happens if the user types something that isn't a number? Python crashes with a `ValueError`. You'll learn how to handle that gracefully in Chapter 13 (Error Handling). For now, trust that the user will type a number when you ask for one.

Your turn: Ask the user for their favorite number. Convert it to an integer. Print the number, its double, and its square.

4.4 `if` — One Branch

The `if` statement runs a block of code only when a condition is `True`. If the condition is `False`, the block is skipped entirely.

```
score = 75
```

```
if score >= 50:
    print("You passed!")
```

Output:

```
You passed!
```

Change `score` to `30` and run again — nothing prints, because `30 >= 50` is `False`.

The colon and the indent are not optional. The `if` line ends with `:`. Everything that belongs to the `if` block must be indented — pushed to the right by 4 spaces. This is how Python knows where the block starts and ends.

```
if score >= 50:
    print("You passed!")    # inside the if --- indented
    print("Great job!")     # also inside the if --- same indentation

print("Done.")              # outside the if --- not indented, always runs
```

Python doesn't use `{}` curly braces like many other languages. Indentation *is* the structure. Every editor you'll use (VS Code, Colab, Jupyter) handles the indentation automatically — just press Enter after the `:` and it adds the spaces.

In plain English: The `if` statement says "only do this if it's true." The indented lines are the "this."

Your turn: Create a variable `temperature`. If it's above 30, print "It's hot today." Run it with a temperature of 35, then with 20. Make sure you see the right behavior each time.

4.5 `if` / `else` — Two Branches

`else` is the fallback — the code that runs when the `if` condition is `False`.

```
score = 35

if score >= 50:
    print("You passed!")
else:
    print("You didn't pass. Keep practicing.")
```

Output:

```
You didn't pass. Keep practicing.
```

Python checks the condition. It's `False`, so it skips the `if` block and runs the `else` block instead. Exactly one of the two blocks always runs.

A real example with user input:

```
age = int(input("How old are you? "))

if age >= 18:
    print("You can vote.")
else:
    print("You're not old enough to vote yet.")
```

If the user types `21`, they see "You can vote." If they type `16`, they see the other message.

Your turn: Ask the user for a number. If it's even (`number % 2 == 0`), print "Even." Otherwise print "Odd."

4.6 `if` / `elif` / `else` — Many Branches

When you have more than two possible outcomes, use `elif` (short for "else if"). Python checks each condition in order — the moment it finds one that's `True`, it runs that block and skips all the rest.

```
grade = 85

if grade >= 90:
    print("A")
elif grade >= 80:
    print("B")
elif grade >= 70:
    print("C")
elif grade >= 60:
    print("D")
else:
    print("F")
```

Output:

```
B
```

Python checks: is `85 >= 90`? No. Is `85 >= 80`? Yes. It prints "B" and stops — it never looks at the `elif grade >= 70` line.

You can have as many `elif` branches as you need. The `else` at the end is optional, but it's good practice — it catches anything that didn't match any condition.

```
time_of_day = 14    # 2 PM in 24-hour format

if time_of_day < 12:
```

```
    print("Good morning!")
elif time_of_day < 17:
    print("Good afternoon!")
elif time_of_day < 21:
    print("Good evening!")
else:
    print("Good night!")
```

Output:

```
Good afternoon!
```

Your turn: Write a program that asks the user for a temperature (in Celsius) and prints: - “Freezing” if below 0 - “Cold” if 0 to 10 - “Mild” if 11 to 20 - “Warm” if 21 to 30 - “Hot” if above 30

4.7 Logical Operators: `and`, `or`, `not`

Sometimes one condition isn’t enough. You want to check two things at once, or the opposite of a condition. That’s what logical operators are for.

4.7.1 `and` — Both must be True

```
age = 25
has_ticket = True

if age >= 18 and has_ticket:
    print("You can enter the concert.")
else:
    print("You cannot enter.")
```

Output:

```
You can enter the concert.
```

With `and`, the whole condition is `True` only if *both* sides are `True`. If either one is `False`, the whole thing is `False`.

```
print(True and True)     # True
print(True and False)    # False
print(False and True)    # False
print(False and False)   # False
```

4.7.2 `or` — At least one must be True

```
is_weekend = False
is_holiday = True

if is_weekend or is_holiday:
    print("No work today!")
else:
    print("Back to work.")
```

Output:

```
No work today!
```

With `or`, the whole condition is `True` if *at least one* side is `True`.

```
print(True or True)     # True
print(True or False)    # True
print(False or True)    # True
print(False or False)   # False
```

4.7.3 `not` — Flips True to False and vice versa

```
is_logged_in = False

if not is_logged_in:
    print("Please log in first.")
```

Output:

```
Please log in first.
```

`not True` is `False`. `not False` is `True`. It reverses the value.

4.7.4 Combining them

You can chain logical operators together. Use parentheses to make the logic clear.

```
age = 20
has_id = True
is_member = False

if age >= 18 and has_id and (is_member or age >= 21):
    print("Access granted.")
else:
    print("Access denied.")
```

When combining multiple conditions, parentheses work just like in math — they group things and control the order of evaluation.

Your turn: Write a program that asks for a username and password (just use `input()`). If the username is `"admin"` **and** the password is `"1234"`, print "Access granted." Otherwise print "Access denied."

4.8 Nested `if` Statements

You can put an `if` inside another `if`. This is called **nesting**.

```
score = 88
completed_bonus = True

if score >= 80:
    print("You passed!")
    if completed_bonus:
        print("And you earned a bonus star!")
    else:
        print("Complete the bonus task for an extra star.")
else:
    print("You didn't pass. Keep trying.")
```

Output:

```
You passed!
And you earned a bonus star!
```

The inner `if` only runs if the outer `if` already passed. Each level of nesting adds another 4 spaces of indentation.

A word of caution: Nesting more than 2 or 3 levels deep makes code very hard to read. When you find yourself going deeper than that, there's almost always a cleaner way to write it. You'll learn those techniques as you progress through the book.

Your turn: Write a program that asks for a number. First check if it's positive or negative (or zero). If it's positive, also check if it's even or odd. Print the result.

4.9 Putting It All Together: A Number Classifier

Here's a program that uses everything from this chapter.

```
number = int(input("Enter a number: "))

if number > 0:
    print(f"{number} is positive.")
    if number % 2 == 0:
        print("It's also even.")
    else:
        print("It's also odd.")
    if number > 1000:
        print("That's a big number!")
elif number < 0:
    print(f"{number} is negative.")
    if number < -1000:
        print("That's a very small number!")
else:
    print("You entered zero.")

print("Done.")
```

Sample run with `42`:

```
Enter a number: 42
42 is positive.
It's also even.
Done.
```

Sample run with `-2500`:

```
Enter a number: -2500
-2500 is negative.
That's a very small number!
Done.
```

Sample run with `0`:

```
Enter a number: 0
You entered zero.
Done.
```

Notice "Done." always prints — it's outside all the `if` blocks, so it runs no matter what the user enters.

4.10 What You Learned in This Chapter

- `bool` is a data type with two values: `True` and `False`.
- Comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) return `True` or `False`.

- `=` is assignment. `==` is comparison. Never mix them up.
- `input()` always returns a string. Use `int()` or `float()` to convert it to a number.
- `if` runs a block when a condition is `True`.
- `else` runs when the `if` condition is `False`.
- `elif` adds more branches — Python checks each one in order and runs the first `True` one.
- `and` requires both conditions to be `True`.
- `or` requires at least one condition to be `True`.
- `not` flips `True` to `False` and `False` to `True`.
- Nested `if` statements let you check conditions inside conditions.
- Indentation (4 spaces) defines which code belongs to which block. It is not optional.

4.11 What's Next?

Your programs can now decide — but they still do each thing only once. What if you want to do something ten times, or a hundred times, or until the user says stop?

In Chapter 5, you'll learn about **loops**. A loop lets you repeat a block of code without writing it over and over. It's one of the most powerful ideas in all of programming, and once you have it, you can build things you couldn't even attempt before.

Your turn: Build a simple number guessing game. Set a secret number with `secret = 7`. Ask the user to guess a number. If they guess too low, print "Too low!" If they guess too high, print "Too high!" If they get it right, print "You got it!" (Don't worry about letting them guess more than once — that's a loop, and it's coming in the next chapter.)

Chapter 5

Chapter 5: Repeating Things — Loops

At the end of Chapter 4 you built a number guessing game. The user guesses once, gets feedback, and the program ends. But a real guessing game lets you keep trying until you get it right.

That's what loops do. A loop runs a block of code again and again — as many times as you need. Without loops, you'd have to copy and paste the same code dozens of times. With loops, you write it once and let Python repeat it.

This is one of the most important ideas in all of programming. Let's dig in.

5.1 Why Loops Exist

Imagine printing every number from 1 to 100. Without a loop:

```
print(1)
print(2)
print(3)
# ... 97 more lines
print(100)
```

That's 100 lines of nearly identical code. Now with a loop:

```
for i in range(1, 101):
    print(i)
```

Two lines. Same result. That's the power of loops.

But loops aren't just for counting. They're for processing every item in a list, reading every line in a file, repeating a question until you get a valid answer, and a thousand other patterns you'll use every day.

Python has two kinds of loops: `for` and `while`. You'll use both.

5.2 `for` Loops — Repeating a Known Number of Times

A `for` loop goes through a sequence — a range of numbers, a list, a string, anything you can iterate over — and runs the indented block once for each item.

```
for i in [1, 2, 3, 4, 5]:
    print(i)
```

Output:

```
1
2
3
4
5
```

The variable `i` takes each value from the list in turn. First it's `1`, the block runs. Then it's `2`, the block runs. And so on until the list is exhausted.

The variable name `i` is just a convention for "index" or "iteration." You can call it anything.

```
for number in [10, 20, 30]:
    print(f"The number is {number}")
```

Output:

```
The number is 10
The number is 20
The number is 30
```

Your turn: Write a `for` loop that prints these three words, one per line: "Python", "is", "fun".

5.3 `range()` — Generating Sequences of Numbers

Typing out a list like `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` is tedious. `range()` generates a sequence of numbers on demand, without storing them all in memory.

5.3.1 `range(stop)` — Start at 0, count up to (but not including) stop

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

`range(5)` gives you 0, 1, 2, 3, 4. It stops *before* 5. This trips up beginners constantly — remember: the stop value is excluded.

5.3.2 `range(start, stop)` — Control where you start

```
for i in range(1, 6):
    print(i)
```

Output:

```
1
2
3
4
5
```

Now it starts at 1 and stops before 6. Much more natural for counting things humans care about.

5.3.3 `range(start, stop, step)` — Control the step size

```
for i in range(0, 11, 2):
    print(i)
```

Output:

```
0
2
4
6
8
10
```

The step is 2, so we jump by 2 each time. You can use any step, including negative — which lets you count backwards.

```
for i in range(10, 0, -1):
    print(i)
print("Blastoff!")
```

Output:

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

Your turn: Use `range()` to: 1. Print all odd numbers from 1 to 19. 2. Print a countdown from 20 to 0, stepping by 5 (20, 15, 10, 5, 0).

5.4 Doing Real Work Inside a Loop

The real power of loops is doing something *useful* on each iteration — not just printing a counter.

5.4.1 Summing numbers

```
total = 0

for number in range(1, 11):
    total = total + number

print(f"Sum of 1 to 10 is: {total}")
```

Output:

```
Sum of 1 to 10 is: 55
```

Before the loop, `total` is 0. On each pass, we add the current `number` to `total`. After the loop, `total` holds the sum of everything.

5.4.2 Multiplication table

```
n = 7

for i in range(1, 11):
    print(f"{n} x {i} = {n * i}")
```

Output:

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

5.4.3 Looping over a string

Strings are sequences too. You can loop over every character.

```
for letter in "Python":
    print(letter)
```

Output:

```
P
y
t
h
o
n
```

Your turn: Write a loop that calculates and prints the factorial of a number. (Factorial of 5 = 1 x 2 x 3 x 4 x 5 = 120.) Start with `result = 1` and multiply on each iteration.

5.5 `while` Loops — Repeating Until a Condition Changes

A `while` loop keeps running as long as its condition is `True`. You use it when you don't know in advance how many times you'll need to repeat — you just know *when to stop*.

```
count = 0

while count < 5:
    print(f"count is {count}")
    count += 1
```

Output:

```
count is 0
count is 1
count is 2
count is 3
count is 4
```

Python checks the condition before each iteration. If it's `True`, the block runs. If it's `False`, the loop ends. Here, once `count` reaches 5, `count < 5` becomes `False` and we stop.

The critical rule: Something inside the loop *must* eventually make the condition `False`. Here `count += 1` does that. Without it, `count` would never change, the condition would stay `True` forever, and you'd have an infinite loop.

5.5.1 Waiting for valid input

This is the classic use case for `while` — keep asking until the user gives you what you need.

```
age = -1

while age < 0:
    age = int(input("Enter your age (must be 0 or more): "))
    if age < 0:
        print("That's not a valid age. Try again.")

print(f"Your age is {age}.")
```

Sample run:

```
Enter your age (must be 0 or more): -5
That's not a valid age. Try again.
Enter your age (must be 0 or more): -1
```

```
That's not a valid age. Try again.
Enter your age (must be 0 or more): 28
Your age is 28.
```

The program will not move on until the user gives a non-negative number. This pattern — validate input in a loop — is everywhere in real software.

Your turn: Write a `while` loop that keeps asking the user to enter a password until they type the correct one (`"opensesame"`). When they get it right, print "Access granted!"

5.6 Infinite Loops and How to Avoid Them

An infinite loop runs forever because its condition never becomes `False`.

```
# DANGER --- this runs forever
while True:
    print("This never stops!")
```

If you run this by accident, press **Ctrl+C** in the terminal to force-stop it.

But infinite loops aren't always a mistake. Sometimes you *want* to loop forever and use `break` to exit at the right moment (more on that below). Many real programs — servers, games, dashboards — run in a loop intentionally: "keep running until the user quits."

```
while True:
    command = input("Enter command (or 'quit' to exit): ")
    if command == "quit":
        print("Goodbye!")
        break
    print(f"You typed: {command}")
```

Sample run:

```
Enter command (or 'quit' to exit): hello
You typed: hello
Enter command (or 'quit' to exit): test
You typed: test
Enter command (or 'quit' to exit): quit
Goodbye!
```

This pattern — `while True` with a `break` to exit — is one of the most useful structures in all of Python.

5.7 `break` — Stop the Loop Early

`break` exits the loop immediately, no matter where you are in the iteration.

```
for i in range(10):
    if i == 5:
        break
    print(i)

print("Loop ended.")
```

Output:

```
0
1
2
3
4
Loop ended.
```

When `i` reaches 5, `break` fires and the loop ends. The `print("Loop ended.")` runs because it's outside the loop.

A real example: searching for something in a sequence.

```
names = ["Alice", "Bob", "Carlos", "Diana", "Eve"]
target = "Carlos"

for name in names:
    print(f"Checking {name}...")
    if name == target:
        print(f"Found {target}!")
        break
```

Output:

```
Checking Alice...
Checking Bob...
Checking Carlos...
Found Carlos!
```

Once we find the target, there's no reason to keep checking. `break` stops the loop early.

5.8 `continue` — Skip One Iteration

`continue` skips the rest of the current iteration and jumps straight to the next one. The loop doesn't stop — it just skips this pass.

```
for i in range(10):
    if i % 2 == 0:
        continue    # skip even numbers
    print(i)
```

Output:

```
1
3
5
7
9
```

When `i` is even, `continue` fires and we skip the `print`. When `i` is odd, nothing skips the `print`, so it runs.

Another example: printing only the positive numbers from a mixed list.

```
numbers = [4, -2, 7, -1, 0, 9, -5, 3]

for n in numbers:
    if n <= 0:
        continue
    print(n)
```

Output:

```
4
7
9
3
```

5.9 `else` on a Loop — The Secret Clause

Python loops have a feature most beginners never learn: an `else` clause. The `else` block runs *after* the loop finishes normally — meaning it does **not** run if the loop was stopped by a `break`.

```
for i in range(5):
    print(i)
else:
    print("Loop finished normally.")
```

Output:

```
0
1
2
3
4
Loop finished normally.
```

Now with a `break`:

```
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print("Loop finished normally.")    # this will NOT print
```

Output:

```
0
1
2
```

The `else` didn't run because `break` stopped the loop early.

This is incredibly useful for "search and report if not found" patterns:

```
names = ["Alice", "Bob", "Diana", "Eve"]
target = "Carlos"

for name in names:
    if name == target:
        print(f"Found {target}!")
        break
else:
    print(f"{target} was not found in the list.")
```

Output:

```
Carlos was not found in the list.
```

Without the `else` clause, you'd need an extra variable to track whether you found the item. With it, Python handles the logic for you.

5.10 `for` vs `while` — When to Use Each

Situation	Use
You know exactly how many times to repeat	`for`
You're going through a sequence (list, string, range)	`for`
You're repeating until a condition changes	`while`
You're waiting for user input	`while`
You want to loop forever until a break	`while True`

If you can write it cleanly as a `for` loop, prefer `for`. It's more readable and harder to accidentally infinite-loop.

5.11 Nested Loops

You can put a loop inside another loop. The inner loop runs completely for every single iteration of the outer loop.

```
for row in range(1, 4):
    for col in range(1, 4):
        print(f"({row},{col})", end="  ")
    print()    # new line after each row
```

Output:

```
(1,1)  (1,2)  (1,3)
(2,1)  (2,2)  (2,3)
(3,1)  (3,2)  (3,3)
```

The `end="  "` in `print()` tells Python not to add a newline — just a space — after each item. The plain `print()` at the end of the outer loop adds the newline to finish each row.

Nested loops are essential for working with grids, tables, and combinations. But be careful: if the outer loop runs 100 times and the inner loop runs 100 times, that's 10,000 total iterations. Nesting goes deep fast.

Your turn: Use nested loops to print a multiplication table from 1x1 to 5x5. Each row should show all five products for that multiplier.

5.12 Putting It All Together: The Guessing Game

Remember the guessing game from Chapter 4 that only let you guess once? Here it is, completed with a loop.

```
import random

secret = random.randint(1, 100)
attempts = 0
max_attempts = 7

print("I'm thinking of a number between 1 and 100.")
print(f"You have {max_attempts} attempts.")

while attempts < max_attempts:
    guess = int(input("Your guess: "))
    attempts += 1

    if guess < secret:
        print(f"Too low! ({max_attempts - attempts} attempts left)")
    elif guess > secret:
        print(f"Too high! ({max_attempts - attempts} attempts left)")
    else:
        print(f"You got it in {attempts} attempt(s)! The number was {secret}.")
        break
else:
    print(f"Out of attempts! The number was {secret}.")
```

Sample run:

```
I'm thinking of a number between 1 and 100.
You have 7 attempts.
Your guess: 50
Too high! (6 attempts left)
Your guess: 25
Too low! (5 attempts left)
Your guess: 37
Too low! (4 attempts left)
Your guess: 43
You got it in 4 attempt(s)! The number was 43.
```

Look at what's happening here: - `while attempts < max_attempts` keeps the loop going until attempts run out - `break` exits the loop when the player wins - The `else` on the `while` loop runs only if attempts ran out (i.e., `break` was never hit) - `random.randint(1, 100)` picks a random number — you'll learn more

about `random` in Chapter 16

This is a complete, playable game. Built with tools you now know.

5.13 What You Learned in This Chapter

- `for` loops iterate over a sequence — a list, a `range()`, or any iterable.
- `range(stop)` starts at 0. `range(start, stop)` gives you control. `range(start, stop, step)` lets you skip or count backwards.
- `while` loops repeat as long as a condition is `True`. Use them when you don't know the count in advance.
- `while True` with a `break` is the standard pattern for "loop until done."
- `break` exits the loop immediately.
- `continue` skips the rest of the current iteration and moves to the next.
- `else` on a loop runs after the loop finishes *normally* — it's skipped if `break` fired.
- Nested loops let you work with rows and columns, grids, and combinations.
- Always make sure a `while` loop's condition will eventually become `False`.

5.14 What's Next?

Your programs can now repeat. But they're still working with one value at a time — one number, one name, one score.

In Chapter 6 you'll learn about **lists** — a way to store many values in a single variable. You'll combine lists with loops and suddenly be able to process hundreds of items with just a few lines of code. It's where things start to feel truly powerful.

Your turn: Write a program that asks the user to enter numbers one at a time. After each number, add it to a running total and print the current total. Stop when the user types `0`. At the end, print how many numbers they entered and what the total was. (Hint: use `while True` and `break`.)

Chapter 6

Chapter 6: Lists — Your First Collection

So far every variable has held exactly one value — one number, one string, one boolean. That works for simple programs, but real programs deal with collections of things: a shopping cart with many items, a class with many students, a playlist with many songs.

Python's answer is the **list**. A list stores many values in a single variable, keeps them in order, and lets you do powerful things with them — add, remove, sort, search, slice, and loop through every item.

Lists are one of the most used data structures in all of Python. Master this chapter and you'll use what you learn here every single day.

6.1 What is a List?

A list is an ordered collection of items. You write it with square brackets `[]`, with items separated by commas.

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
```

One variable. Three values. That's the idea.

Lists can hold any type of data — numbers, strings, booleans, even other lists. They can also mix types, though in practice you usually keep one type per list.

```
numbers = [10, 20, 30, 40, 50]
names = ["Alice", "Bob", "Carlos"]
mixed = [1, "hello", True, 3.14]
empty = []
```

In plain English: A list is like a numbered to-do list. Each item has a position, and you can look up, change, add, or remove items by their position.

Your turn: Create a list called `favorites` with three of your favorite movies. Print the whole list.

6.2 Indexing — Getting One Item

Every item in a list has a position called its **index**. Python starts counting from `0`.

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]
#           0         1         2        3          4
```

Use square brackets with the index to get one item.

```
print(fruits[0])   # apple
print(fruits[1])   # banana
print(fruits[4])   # elderberry
```

Negative indexing counts from the end. `-1` is the last item, `-2` is second to last, and so on.

```
print(fruits[-1])   # elderberry  (last)
print(fruits[-2])   # date        (second to last)
print(fruits[-5])   # apple       (same as index 0)
```

This is incredibly useful — you can always get the last item of any list with `[-1]` without knowing the length.

What happens if you use an index that doesn't exist?

```
print(fruits[10])   # IndexError: list index out of range
```

Python crashes with an `IndexError`. Always make sure your index is between `0` and `len(list) - 1` (or between `-len(list)` and `-1` for negatives).

Your turn: Create a list of five animals. Print the first animal, the last animal using a negative index, and the middle animal.

6.3 Slicing — Getting Multiple Items

A **slice** extracts a portion of a list and gives you a new list.

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]

print(fruits[1:4])     # ['banana', 'cherry', 'date']
print(fruits[0:3])     # ['apple', 'banana', 'cherry']
print(fruits[:3])      # ['apple', 'banana', 'cherry']  (start defaults to 0)
print(fruits[2:])      # ['cherry', 'date', 'elderberry']  (stop defaults to end
print(fruits[:])       # ['apple', 'banana', 'cherry', 'date', 'elderberry']  (e
```

The syntax is `list[start:stop]`. The `start` index is included. The `stop` index is excluded — same rule as `range()`.

You can also add a step:

```
print(fruits[::2])     # ['apple', 'cherry', 'elderberry']  (every 2nd item)
print(fruits[::-1])    # ['elderberry', 'date', 'cherry', 'banana', 'apple']  (r
```

`[::-1]` reverses a list. It's a common Python idiom.

Your turn: Create a list of ten numbers (0 through 9). Use slicing to print: the first three, the last three, and every other number.

6.4 Changing Items

Lists are **mutable** — you can change their contents after creation. Assign directly to an index.

```
colors = ["red", "green", "blue"]
colors[1] = "yellow"
print(colors)    # ['red', 'yellow', 'blue']
```

You can also replace a slice with new values:

```
numbers = [1, 2, 3, 4, 5]
numbers[1:3] = [20, 30]
print(numbers)    # [1, 20, 30, 4, 5]
```

6.5 Adding Items

6.5.1 `append()` — Add one item to the end

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)   # ['apple', 'banana', 'cherry']
```

6.5.2 `insert()` — Add one item at a specific position

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "blueberry")   # insert at index 1
print(fruits)   # ['apple', 'blueberry', 'banana', 'cherry']
```

Everything from index 1 onward shifts one position to the right.

6.5.3 `extend()` — Add all items from another list

```
fruits = ["apple", "banana"]
more_fruits = ["cherry", "date"]
fruits.extend(more_fruits)
print(fruits)   # ['apple', 'banana', 'cherry', 'date']
```

Or use `+` to combine two lists into a new one:

```
combined = ["apple", "banana"] + ["cherry", "date"]
print(combined)   # ['apple', 'banana', 'cherry', 'date']
```

Your turn: Start with an empty list. Append five numbers to it one at a time using a loop and `range()`. Print the final list.

6.6 Removing Items

6.6.1 `remove()` — Remove by value

Removes the *first* occurrence of the item.

```
fruits = ["apple", "banana", "cherry", "banana"]
fruits.remove("banana")
print(fruits)   # ['apple', 'cherry', 'banana']
```

If the item doesn't exist, Python raises a `ValueError`.

6.6.2 `pop()` — Remove by index (and return the value)

```
fruits = ["apple", "banana", "cherry"]
removed = fruits.pop(1)    # removes index 1
print(removed)    # banana
print(fruits)     # ['apple', 'cherry']
```

Without an index, `pop()` removes and returns the last item — useful for treating a list like a stack.

```
fruits = ["apple", "banana", "cherry"]
last = fruits.pop()
print(last)     # cherry
print(fruits)  # ['apple', 'banana']
```

6.6.3 `del` — Remove by index or slice

```
fruits = ["apple", "banana", "cherry", "date"]
del fruits[1]
print(fruits)    # ['apple', 'cherry', 'date']

del fruits[0:2]
print(fruits)    # ['date']
```

6.6.4 `clear()` — Remove everything

```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits)    # []
```

Your turn: Create a list of 5 items. Remove the first item using `pop(0)`. Remove a specific item by value using `remove()`. Print the list after each operation.

6.7 Useful Functions for Lists

6.7.1 `len()` — How many items?

```
fruits = ["apple", "banana", "cherry"]
print(len(fruits))    # 3
```

6.7.2 `sum()`, `min()`, `max()` — Math on a list of numbers

```
scores = [88, 72, 95, 61, 84]
```

```
print(sum(scores))    # 400
print(min(scores))    # 61
print(max(scores))    # 95
print(sum(scores) / len(scores))    # 80.0  (average)
```

6.7.3 `sorted()` — Sort without changing the original

```
scores = [88, 72, 95, 61, 84]
sorted_scores = sorted(scores)
print(sorted_scores)    # [61, 72, 84, 88, 95]
print(scores)           # [88, 72, 95, 61, 84]  (unchanged)
```

To sort in descending order:

```
print(sorted(scores, reverse=True))    # [95, 88, 84, 72, 61]
```

6.7.4 `.sort()` — Sort in place (changes the original)

```
scores = [88, 72, 95, 61, 84]
scores.sort()
print(scores)    # [61, 72, 84, 88, 95]
```

6.7.5 `reversed()` — Iterate in reverse order

```
fruits = ["apple", "banana", "cherry"]
for fruit in reversed(fruits):
    print(fruit)
```

Output:

```
cherry
banana
apple
```

6.8 More List Methods

6.8.1 `count()` — How many times does a value appear?

```
numbers = [1, 2, 3, 2, 4, 2, 5]
print(numbers.count(2))    # 3
```

6.8.2 `index()` — What position is this value at?

```
fruits = ["apple", "banana", "cherry"]
print(fruits.index("banana"))    # 1
```

Raises `ValueError` if the item isn't in the list.

6.8.3 `copy()` — Make a shallow copy

```
original = [1, 2, 3]
copy = original.copy()
copy.append(4)

print(original)    # [1, 2, 3]  (unchanged)
print(copy)        # [1, 2, 3, 4]
```

Why does this matter? Because if you do `copy = original`, you don't get a copy — both variables point to the *same* list. Changing one changes the other. `copy()` gives you an independent duplicate.

```
original = [1, 2, 3]
alias = original      # NOT a copy --- same list
alias.append(99)

print(original)    # [1, 2, 3, 99]  (original was changed!)
```

This surprises almost every beginner at least once. Now you know.

6.9 Checking if an Item is in a List

Use the `in` keyword.

```
fruits = ["apple", "banana", "cherry"]

print("banana" in fruits)     # True
print("grape" in fruits)      # False
print("grape" not in fruits)  # True
```

Combine with `if` for practical use:

```
shopping_cart = ["bread", "milk", "eggs"]

item = input("Enter an item to check: ")

if item in shopping_cart:
    print(f"{item} is in your cart.")
else:
    print(f"{item} is not in your cart.")
```

6.10 Looping Over a List

6.10.1 Loop over values directly

```
names = ["Alice", "Bob", "Carlos"]

for name in names:
    print(f"Hello, {name}!")
```

Output:

```
Hello, Alice!
Hello, Bob!
Hello, Carlos!
```

6.10.2 Loop with index using `enumerate()`

When you need both the index and the value, use `enumerate()`. It gives you both on each iteration.

```
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
```

Output:

```
0: apple
1: banana
2: cherry
```

Start the count from 1 instead of 0:

```
for index, fruit in enumerate(fruits, start=1):
    print(f"{index}. {fruit}")
```

Output:

```
1. apple
2. banana
3. cherry
```

This is cleaner than using `range(len(fruits))` and indexing manually.

6.11 Nested Lists — Lists Inside Lists

A list can contain other lists. This is how you represent a grid, a table, or a matrix.

```
grid = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(grid[0])      # [1, 2, 3]
print(grid[1][2])   # 6  (row 1, column 2)
```

Loop over a nested list with nested loops:

```
for row in grid:
    for value in row:
        print(value, end="  ")
    print()
```

Output:

```
1  2  3
4  5  6
7  8  9
```

Real use case: a tic-tac-toe board, a seating chart, a spreadsheet.

6.12 List Comprehensions — A Powerful Shortcut

A list comprehension creates a new list by applying an expression to every item in a sequence. It's one of Python's most elegant features.

The pattern is: `[expression for item in sequence]`

```
# The long way
squares = []
for n in range(1, 6):
    squares.append(n ** 2)
print(squares)    # [1, 4, 9, 16, 25]

# The list comprehension way
squares = [n ** 2 for n in range(1, 6)]
print(squares)    # [1, 4, 9, 16, 25]
```

Same result. Half the lines.

You can add a condition to filter items:

```
# Only even squares
even_squares = [n ** 2 for n in range(1, 11) if n % 2 == 0]
print(even_squares)    # [4, 16, 36, 64, 100]

# Uppercase only the long names
names = ["Alice", "Bob", "Christopher", "Di", "Alejandro"]
long_names = [name.upper() for name in names if len(name) > 4]
print(long_names)    # ['ALICE', 'CHRISTOPHER', 'ALEJANDRO']
```

List comprehensions are covered in full depth in Chapter 14. For now, just know they exist and start recognizing the pattern.

Your turn: Use a list comprehension to create a list of all numbers from 1 to 50 that are divisible by 3. Print the result.

6.13 Putting It All Together: Student Grade Tracker

Here's a program that uses almost everything from this chapter.

```
# Student grade tracker
students = []
grades = []

print("Enter 5 student names and their grades.")

for i in range(5):
    name = input(f"Student {i + 1} name: ")
    grade = float(input(f"Grade for {name}: "))
    students.append(name)
    grades.append(grade)

print("\n--- Results ---")

for i, name in enumerate(students):
    status = "Pass" if grades[i] >= 50 else "Fail"
    print(f"{name}: {grades[i]} --- {status}")

print(f"\nHighest grade: {max(grades)}")
print(f"Lowest grade:  {min(grades)}")
print(f"Average grade: {round(sum(grades) / len(grades), 1)}")
print(f"Top student:   {students[grades.index(max(grades))]}")
```

Sample output:

```
--- Results ---
Alice: 88.0 --- Pass
Bob: 45.0 --- Fail
Carlos: 92.0 --- Pass
Diana: 71.0 --- Pass
Eve: 55.0 --- Pass

Highest grade: 92.0
Lowest grade:  45.0
Average grade: 70.2
Top student:   Carlos
```

`grades.index(max(grades))` finds the position of the highest grade, then uses that same position to find the student name. That's two list operations working together.

6.14 What You Learned in This Chapter

- A list stores many values in one variable, in order, using `[]`.
- Indexing: `list[0]` is the first item. `list[-1]` is the last. Counting starts at 0.
- Slicing: `list[start:stop]` extracts a portion. `list[::-1]` reverses.
- Lists are mutable — you can change items by assigning to an index.
- Adding items: `append()` (end), `insert()` (position), `extend()` (add list), `+` (new list).
- Removing items: `remove()` (by value), `pop()` (by index), `del`, `clear()`.
- Useful functions: `len()`, `sum()`, `min()`, `max()`, `sorted()`, `reversed()`.
- Useful methods: `sort()`, `count()`, `index()`, `copy()`.
- Check membership with `in` and `not in`.
- Loop with `for item in list` or `for index, item in enumerate(list)`.
- Nested lists represent grids and tables.
- List comprehensions create lists in one readable line.
- `copy = original` is not a copy — use `.copy()` to get a real independent duplicate.

6.15 What's Next?

You can now store and manipulate collections of data. The next step is organizing your *code*.

In Chapter 7 you'll learn about **functions** — a way to give a name to a block

of code and reuse it wherever you need it. Functions turn a 200-line program into something clean and readable. They also make it possible to break a big problem into small, solvable pieces — which is exactly how professional programmers think.

Your turn: Write a program that asks the user to enter numbers until they type `-1` (use a `while` loop). Store each number (except `-1`) in a list. After they stop, print: the list, the total, the average, the highest, and the lowest. If they entered no numbers, print "No numbers entered."

Chapter 7

Chapter 7: Functions — Reusable Blocks of Code

You've been writing programs that run top to bottom — one long sequence of instructions. That works for small programs. But as programs grow, that approach falls apart fast.

Imagine you need to calculate a discount in five different places in your program. Without functions, you copy the same four lines five times. Then the discount formula changes — and you have to find and fix all five copies. Miss one, and your program gives wrong answers.

Functions solve this. You write the logic once, give it a name, and call that name wherever you need it. Change the function once, and every place that calls it gets the update automatically.

That's the idea. But functions do much more than just avoid repetition — they let you break a complex problem into small, named, solvable pieces. Every professional programmer thinks this way.

7.1 What is a Function?

A function is a named block of code that you define once and can run (call) as many times as you want.

You've already used functions — `print()`, `input()`, `len()`, `range()`, `int()`, `round()`. Those are built-in functions that Python provides. Now you'll write your own.

In plain English: A function is like a recipe. You write it once. Whenever you want that dish, you follow the recipe — you don't rewrite it from scratch.

7.2 Defining Your First Function

Use the `def` keyword, then the function name, then parentheses, then a colon. The code inside the function is indented.

```
def say_hello():
    print("Hello!")
    print("Nice to meet you.")

say_hello()
say_hello()
say_hello()
```

Output:

```
Hello!
Nice to meet you.
Hello!
Nice to meet you.
Hello!
Nice to meet you.
```

The function runs every time you call it. You wrote the two `print` lines once, but ran them three times.

Important: You must *define* a function before you *call* it. If you call it before the `def`, Python raises a `NameError`.

```
say_hello()   # NameError --- Python doesn't know this yet

def say_hello():
    print("Hello!")
```

Always define first, call second.

Your turn: Write a function called `print_divider` that prints a line of 30 dashes (- x 30). Call it three times. You'll use this kind of function to format output in real programs.

7.3 Parameters — Passing Information In

A function that always does the exact same thing has limited use. Parameters let you pass information in so the function can work with different data each time.

```
def greet(name):
    print(f"Hello, {name}! Welcome.")

greet("Alice")
greet("Bob")
greet("Carlos")
```

Output:

```
Hello, Alice! Welcome.
Hello, Bob! Welcome.
Hello, Carlos! Welcome.
```

`name` is the **parameter** — a placeholder in the function definition. `"Alice"`, `"Bob"`, `"Carlos"` are the **arguments** — the actual values you pass in when you call the function.

You can have multiple parameters:

```
def describe_person(name, age, city):
    print(f"{name} is {age} years old and lives in {city}.")

describe_person("Diana", 29, "Lagos")
describe_person("Kenji", 41, "Tokyo")
```

Output:

```
Diana is 29 years old and lives in Lagos.
Kenji is 41 years old and lives in Tokyo.
```

By default, you must pass arguments in the same order as the parameters. `describe_person("Diana", 29, "Lagos")` assigns `"Diana"` to `name`, `29` to `age`, `"Lagos"` to `city`.

Your turn: Write a function called `calculate_area` that takes `length` and `width` as parameters and prints the area (`length x width`). Call it with three different pairs of values.

7.4 Return Values — Getting Information Back

So far our functions print results. But printing and *returning* are two very different things.

- **Printing** shows a value on screen. The rest of your program can't use it.
- **Returning** sends a value back to wherever the function was called. Your program can store it, calculate with it, pass it to another function.

```
def add(a, b):
    return a + b

result = add(10, 5)
print(result)          # 15
print(add(3, 7) * 2)   # 20
```

The `return` statement does two things: it sends a value back, and it immediately exits the function. Any code after `return` in the same block never runs.

```
def check_positive(number):
    if number > 0:
        return True
    return False

print(check_positive(5))    # True
print(check_positive(-3))   # False
```

A function that returns something is far more useful than one that just prints, because you can use the result in calculations, conditions, lists, or anywhere else.

```
def celsius_to_fahrenheit(c):
    return (c * 9/5) + 32

temps_celsius = [0, 20, 37, 100]

for temp in temps_celsius:
    f = celsius_to_fahrenheit(temp)
    print(f"{temp}°C = {f}°F")
```

Output:

```
0°C = 32.0°F
20°C = 68.0°F
37°C = 98.6°F
100°C = 212.0°F
```

What if a function has no `return` statement? It implicitly returns `None` — Python's way of saying "nothing."

```
def say_hi():
    print("Hi!")

result = say_hi()
print(result)    # None
```

Your turn: Write a function called `calculate_discount` that takes a `price` and a `discount_percent`, and *returns* the final price after the discount. Test it with a few prices and discounts, storing and printing the results.

7.5 Default Parameter Values

Sometimes you want a parameter to have a sensible default — a value it uses when the caller doesn't provide one.

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")              # uses default greeting
greet("Bob", "Good morning")  # overrides the default
greet("Carlos", "¡Hola")
```

Output:

```
Hello, Alice!
Good morning, Bob!
¡Hola, Carlos!
```

Parameters with defaults must come *after* parameters without defaults.

```
def make_coffee(size, strength="medium", milk=False):
    print(f"{size} coffee, strength: {strength}, milk: {milk}")

make_coffee("large")
make_coffee("small", "strong")
make_coffee("medium", "light", True)
```

Output:

```
large coffee, strength: medium, milk: False
small coffee, strength: strong, milk: False
medium coffee, strength: light, milk: True
```

Your turn: Write a function called `power` that takes a `base` and an `exponent` (default `2`). It should return `base ** exponent`. Call it with just a base (it should square it), and then with both a base and an exponent.

7.6 Keyword Arguments

When calling a function, you can name the arguments explicitly. This makes the call more readable and lets you pass them in any order.

```
def describe_book(title, author, year):
    print(f'"{title}" by {author} ({year})')

# Positional --- order matters
describe_book("Dune", "Frank Herbert", 1965)

# Keyword --- order doesn't matter
describe_book(author="George Orwell", year=1949, title="1984")
```

Output:

```
"Dune" by Frank Herbert (1965)
"1984" by George Orwell (1949)
```

You can mix positional and keyword arguments, but positional arguments must come first.

```
describe_book("Brave New World", year=1932, author="Aldous Huxley")
```

7.7 `*args` — Any Number of Arguments

What if you don't know how many arguments the caller will pass? Use `*args`. It collects all extra positional arguments into a tuple.

```
def total(*args):
    print(f"Arguments received: {args}")
    return sum(args)

print(total(1, 2, 3))          # 6
print(total(10, 20, 30, 40))   # 100
print(total(5))                # 5
```

Output:

```
Arguments received: (1, 2, 3)
```

```
Arguments received: (10, 20, 30, 40)
Arguments received: (5,)
6
100
5
```

Inside the function, `args` is a tuple containing all the values passed in. You can loop over it, pass it to `sum()`, or do anything you'd do with a regular tuple.

```
def greet_all(*names):
    for name in names:
        print(f"Hello, {name}!")

greet_all("Alice", "Bob", "Carlos", "Diana")
```

Output:

```
Hello, Alice!
Hello, Bob!
Hello, Carlos!
Hello, Diana!
```

7.8 `**kwargs` — Any Number of Keyword Arguments

`**kwargs` collects extra keyword arguments into a dictionary.

```
def show_profile(**kwargs):
    for key, value in kwargs.items():
        print(f"  {key}: {value}")

show_profile(name="Alice", age=30, city="Paris", job="Engineer")
```

Output:

```
  name: Alice
  age: 30
  city: Paris
  job: Engineer
```

Inside the function, `kwargs` is a regular dictionary. You can use any keyword arguments the caller wants to pass — you don't have to define them in advance.

You can combine all four: regular parameters, defaults, `*args`, and `**kwargs`.

```
def log_event(level, *messages, **metadata):
    print(f"[{level}]", " ".join(messages))
```

```
    for key, value in metadata.items():
        print(f"  {key}={value}")

log_event("INFO", "User logged in", user="alice", ip="192.168.1.1")
```

Output:

```
[INFO] User logged in
  user=alice
  ip=192.168.1.1
```

7.9 Scope — Where Variables Live

Every variable in Python has a **scope** — the part of the program where it exists and can be accessed.

7.9.1 Local scope

Variables created inside a function exist only inside that function. They're destroyed when the function returns.

```
def my_function():
    x = 10   # local variable
    print(x)

my_function()   # 10
print(x)        # NameError --- x doesn't exist here
```

7.9.2 Global scope

Variables created outside any function are in the global scope. They exist for the entire program.

```
name = "Alice"   # global variable

def greet():
    print(f"Hello, {name}!")   # can READ a global variable

greet()   # Hello, Alice!
```

A function can *read* a global variable. But if you try to *assign* to it inside the function, Python creates a new local variable with that name instead — the global is untouched.

```
count = 0

def increment():
    count = count + 1    # UnboundLocalError! Python thinks count is local
```

7.9.3 The `global` keyword

To modify a global variable from inside a function, declare it with `global`.

```
count = 0

def increment():
    global count
    count = count + 1

increment()
increment()
increment()
print(count)    # 3
```

Use `global` sparingly. In most cases, a better design is to pass the value in as a parameter and return the updated value — that keeps functions self-contained and predictable.

```
# Better approach --- no global needed
def increment(count):
    return count + 1

count = 0
count = increment(count)
count = increment(count)
count = increment(count)
print(count)    # 3
```

7.10 Docstrings — Documenting Your Functions

A **docstring** is a string at the very start of a function that explains what it does. Write it with triple quotes.

```
def celsius_to_fahrenheit(celsius):
    """
    Convert a temperature from Celsius to Fahrenheit.
```

```
    Parameters:
        celsius (float): Temperature in Celsius.

    Returns:
        float: Temperature in Fahrenheit.
    """
    return (celsius * 9/5) + 32
```

Access a function's docstring with `help()` or `.__doc__`:

```
help(celsius_to_fahrenheit)
print(celsius_to_fahrenheit.__doc__)
```

Good docstrings answer three questions: what does this function do, what does it take as input, and what does it return? Write them for any function someone else (or future you) will use.

7.11 Calling Functions from Functions

Functions can call other functions. This is how you build complex behavior from simple pieces.

```
def square(n):
    return n ** 2

def sum_of_squares(a, b):
    return square(a) + square(b)

def hypotenuse(a, b):
    """Return the hypotenuse of a right triangle with sides a and b."""
    return sum_of_squares(a, b) ** 0.5

print(hypotenuse(3, 4))     # 5.0
print(hypotenuse(5, 12))    # 13.0
```

Each function does one small thing. Combined, they calculate the hypotenuse. This is called **decomposition** — breaking a problem into small, focused functions. It's one of the most important skills in programming.

7.12 Returning Multiple Values

A Python function can return more than one value. Separate them with commas and Python returns them as a tuple.

```python
def min_max(numbers):
    """Return both the minimum and maximum of a list."""
    return min(numbers), max(numbers)

scores = [88, 72, 95, 61, 84]
lowest, highest = min_max(scores)

print(f"Lowest: {lowest}")     # 61
print(f"Highest: {highest}")   # 95
```

The `lowest, highest = min_max(scores)` line **unpacks** the tuple into two variables in one step. You'll see this pattern everywhere in real Python code.

7.13 Putting It All Together: A BMI Calculator

```python
def calculate_bmi(weight_kg, height_m):
    """
    Calculate Body Mass Index (BMI).

    Parameters:
        weight_kg (float): Weight in kilograms.
        height_m (float): Height in metres.

    Returns:
        float: BMI rounded to 1 decimal place.
    """
    return round(weight_kg / height_m ** 2, 1)


def bmi_category(bmi):
    """Return the BMI category as a string."""
    if bmi < 18.5:
        return "Underweight"
    elif bmi < 25:
        return "Normal weight"
    elif bmi < 30:
        return "Overweight"
    else:
        return "Obese"


def print_bmi_report(name, weight_kg, height_m):
```

```
    """Print a complete BMI report for one person."""
    bmi = calculate_bmi(weight_kg, height_m)
    category = bmi_category(bmi)
    print(f"Name:     {name}")
    print(f"BMI:      {bmi}")
    print(f"Category: {category}")
    print()


print_bmi_report("Alice", 65, 1.70)
print_bmi_report("Bob", 90, 1.75)
print_bmi_report("Carlos", 55, 1.80)
```

Output:

```
Name:     Alice
BMI:      22.5
Category: Normal weight

Name:     Bob
BMI:      29.4
Category: Overweight

Name:     Carlos
BMI:      17.0
Category: Underweight
```

Three small functions, each doing one job. `print_bmi_report` calls both of the others. The result is clean, readable, and easy to test or change.

7.14 What You Learned in This Chapter

- `def` defines a function. Call it by writing its name with parentheses.
- **Parameters** are placeholders in the definition. **Arguments** are the values you pass when calling.
- `return` sends a value back. A function with no `return` returns `None`.
- **Default parameters** give a fallback value when the caller doesn't provide one.
- **Keyword arguments** let you pass values by name, in any order.
- `*args` collects any number of positional arguments into a tuple.
- `**kwargs` collects any number of keyword arguments into a dictionary.
- **Scope:** local variables exist only inside their function. Global variables exist everywhere but should be modified with `global` only when necessary.

- **Docstrings** document what a function does, what it takes, and what it returns.
- Functions can call other functions — break big problems into small pieces.
- A function can return multiple values, unpacked with `a, b = function()`.

7.15 What's Next?

Your programs can now store collections of data (lists) and organize logic into reusable functions. But everything disappears when the program stops. If you run your grade tracker, enter 30 students, and close the terminal — all that data is gone.

In Chapter 8, you'll learn how to **read and write files**, so your data can survive after the program ends. You'll save results to a file, read them back later, and start building programs that remember things between runs.

Your turn: Write a function called `summarize` that takes a list of numbers and returns a dictionary with four keys: `"count"`, `"total"`, `"average"`, and `"max"`. Then write a second function called `print_summary` that takes that dictionary and prints each value neatly. Call both functions with a list of your choice.

Chapter 8

Chapter 8: Working with Files

Every program you've written so far has the same problem: the moment it stops running, everything is gone. Variables, lists, results — all erased. Run the grade tracker from Chapter 6 and enter thirty students — close the terminal, and they've vanished.

Files fix this. A file stores data on your hard drive permanently. Your program can write data to a file when it finishes, and read it back the next time it runs. That's how every real application works — notes apps, spreadsheets, web servers, games. They all read and write files.

By the end of this chapter you'll be able to read any text file, write data to a file, append to existing content, work with file paths, handle CSV data, and deal gracefully with files that don't exist.

8.1 How Files Work in Python

Working with a file follows three steps every time:

1. **Open** the file — tell Python which file and what you want to do (read, write, or append).
2. **Do the work** — read from it or write to it.
3. **Close** the file — release it so other programs can use it and your data is saved.

Python gives you a shortcut for steps 1 and 3 together — the `with` statement. You'll use it for almost everything.

8.2 File Modes

When you open a file, you tell Python *how* you want to use it. This is called the **mode**.

Mode	Meaning	File must exist?	Creates file?	Overwrites?
`"r"`	Read	Yes	No	No
`"w"`	Write	No	Yes	Yes
`"a"`	Append	No	Yes	No
`"r+"`	Read and write	Yes	No	No
`"x"`	Create (fails if exists)	No	Yes	No

The two you'll use 95% of the time are `"r"` (read) and `"w"` (write). `"a"` is essential when you want to add to a file without erasing what's already there.

8.3 Reading a File

8.3.1 `read()` — Read the whole file at once

First, create a file to read. Open any text editor, type this, and save it as `notes.txt` in the same folder as your Python file:

```
Python is fun.
Files make data last.
I am learning a lot.
```

Now read it:

```
with open("notes.txt", "r") as file:
    content = file.read()

print(content)
```

Output:

```
Python is fun.
Files make data last.
I am learning a lot.
```

`with open("notes.txt", "r") as file:` opens the file for reading and assigns it to the variable `file`. Everything indented under `with` can use `file`.

When the indented block ends, Python automatically closes the file — even if an error occurs.

`file.read()` returns the entire contents of the file as one big string, including all newline characters (`\n`).

8.3.2 `readlines()` — Read into a list of lines

```
with open("notes.txt", "r") as file:
    lines = file.readlines()

print(lines)
print(len(lines))
```

Output:

```
['Python is fun.\n', 'Files make data last.\n', 'I am learning a lot.\n']
3
```

`readlines()` gives you a list where each item is one line of the file. Notice the `\n` at the end of each line — that's the newline character. Strip it off with `.strip()` when you don't need it.

```
with open("notes.txt", "r") as file:
    lines = file.readlines()

for line in lines:
    print(line.strip())
```

Output:

```
Python is fun.
Files make data last.
I am learning a lot.
```

8.3.3 Looping line by line — The memory-efficient way

For large files, reading everything at once wastes memory. Loop over the file directly instead:

```
with open("notes.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Output:

```
Python is fun.
Files make data last.
I am learning a lot.
```

Python reads one line at a time. This works for files of any size — even files too large to fit in memory. Get into this habit.

Your turn: Create a file called `numbers.txt` with five numbers, one per line (just type them in a text editor). Write a program that reads the file line by line, converts each line to a float, and prints the sum of all the numbers.

8.4 Writing to a File

Open with `"w"` and use `file.write()`.

```
with open("output.txt", "w") as file:
    file.write("Hello from Python!\n")
    file.write("This is the second line.\n")
    file.write("And the third.\n")
```

After this runs, `output.txt` exists on your drive with those three lines. Open it in a text editor to check.

Critical warning: Opening a file with `"w"` **destroys all existing content**. If `output.txt` already had data, it's gone the moment you open it with `"w"`. Always be sure that's what you want.

You need to add `\n` yourself to start new lines — `write()` does not add them automatically.

```
# Write a list of items to a file
shopping = ["bread", "milk", "eggs", "butter", "coffee"]

with open("shopping.txt", "w") as file:
    for item in shopping:
        file.write(item + "\n")
```

`shopping.txt` will contain:

```
bread
milk
eggs
butter
coffee
```

Your turn: Write a program that asks the user to enter five items (use a loop and `input()`). Write each item to a file called `my_list.txt`, one per line. Open the file in a text editor to confirm it worked.

8.5 Appending to a File

`"w"` overwrites. `"a"` (append) adds to the end without touching what's already there.

```
with open("log.txt", "a") as file:
    file.write("Program started.\n")

# Run it again --- this line gets added, not replaced
with open("log.txt", "a") as file:
    file.write("Program started again.\n")
```

After two runs, `log.txt` contains:

```
Program started.
Program started again.
```

Append mode is perfect for log files, diaries, and any record you want to grow over time.

```
from datetime import datetime

def log(message):
    """Append a timestamped message to the log file."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    with open("app.log", "a") as file:
        file.write(f"[{timestamp}] {message}\n")

log("User logged in")
log("User viewed dashboard")
log("User logged out")
```

`app.log` will contain something like:

```
[2026-03-09 14:23:01] User logged in
[2026-03-09 14:23:45] User viewed dashboard
[2026-03-09 14:24:12] User logged out
```

Every call adds a line. The file grows. That's a real application log.

8.6 File Paths: Absolute vs Relative

When you write `open("notes.txt")`, Python looks for `notes.txt` in the **current directory** — the folder your program is running from. This is a **relative path**.

A **relative path** is relative to where your program is:

```
open("notes.txt")              # same folder as the script
open("data/notes.txt")         # in a subfolder called data
open("../notes.txt")           # one folder up
```

An **absolute path** gives the full location from the root of your drive:

```
open("C:/Users/Alice/Documents/notes.txt")    # Windows
open("/home/alice/documents/notes.txt")        # Mac/Linux
```

The modern way: `pathlib`

The `pathlib` module (covered fully in Chapter 16) gives you a clean, cross-platform way to work with file paths. Here's a taste:

```
from pathlib import Path

data_folder = Path("data")
file_path = data_folder / "notes.txt"    # works on Windows AND Mac/Linux

with open(file_path, "r") as file:
    print(file.read())
```

The `/` operator on `Path` objects joins path components — and it uses the right separator for whatever operating system you're on. Get into this habit over using raw strings for paths.

8.7 Handling Files That Don't Exist

If you try to open a file that doesn't exist in `"r"` mode, Python raises a `FileNotFoundError` and your program crashes.

```
with open("missing.txt", "r") as file:    # FileNotFoundError!
    content = file.read()
```

You'll learn full error handling in Chapter 13, but here's the essential pattern for files:

```
try:
    with open("notes.txt", "r") as file:
        content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: The file 'notes.txt' was not found.")
    print("Please create the file and try again.")
```

Now instead of crashing, you get a friendly message. Always wrap file reads in a `try/except` in real programs.

You can also check if a file exists before opening it:

```
from pathlib import Path

path = Path("notes.txt")

if path.exists():
    with open(path, "r") as file:
        print(file.read())
else:
    print("File not found.")
```

Your turn: Write a program that tries to open a file called `config.txt`. If it exists, read and print its contents. If it doesn't exist, create it and write three lines of default settings to it, then print "Config file created."

8.8 Working with CSV Files

CSV stands for "Comma-Separated Values." It's the most common format for storing tabular data — rows and columns, like a spreadsheet. Almost every data source (Excel, Google Sheets, databases) can export to CSV.

A CSV file looks like this (`students.csv`):

```
name,grade,city
Alice,88,London
Bob,72,Paris
Carlos,95,Madrid
Diana,61,Tokyo
```

8.8.1 Reading a CSV file

Python's `csv` module makes this clean:

```
import csv

with open("students.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output:

```
['name', 'grade', 'city']
['Alice', '88', 'London']
['Bob', '72', 'Paris']
```

```
['Carlos', '95', 'Madrid']
['Diana', '61', 'Tokyo']
```

Each row becomes a list of strings. Skip the header row by calling `next(reader)` before the loop:

```
import csv

with open("students.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)    # skip the header row

    for row in reader:
        name, grade, city = row
        print(f"{name} scored {grade} and is from {city}.")
```

Output:

```
Alice scored 88 and is from London.
Bob scored 72 and is from Paris.
Carlos scored 95 and is from Madrid.
Diana scored 61 and is from Tokyo.
```

8.8.2 `DictReader` — Rows as dictionaries

Even better: `csv.DictReader` turns each row into a dictionary, using the header row as keys.

```
import csv

with open("students.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(f"{row['name']}: {row['grade']} ({row['city']})")
```

Output:

```
Alice: 88 (London)
Bob: 72 (Paris)
Carlos: 95 (Madrid)
Diana: 61 (Tokyo)
```

No need to remember column positions — you access by name. Much less error-prone.

8.8.3 Writing a CSV file

```
import csv
```

```
students = [
    ["name", "grade", "city"],
    ["Alice", 88, "London"],
    ["Bob", 72, "Paris"],
    ["Carlos", 95, "Madrid"],
]

with open("output_students.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(students)
```

The `newline=""` argument prevents extra blank lines on Windows. Always include it when writing CSV on Windows.

Your turn: Create a file called `products.csv` with columns `name`, `price`, `quantity`. Add five products. Write a Python program that reads the file and prints the total value of inventory (price x quantity for each product, summed).

8.9 Putting It All Together: A Persistent To-Do List

Here's a program that saves and loads a to-do list from a file — so your tasks survive between runs.

```
from pathlib import Path

TODO_FILE = Path("todos.txt")


def load_todos():
    """Load todos from file. Return empty list if file doesn't exist."""
    if not TODO_FILE.exists():
        return []
    with open(TODO_FILE, "r") as file:
        return [line.strip() for line in file if line.strip()]


def save_todos(todos):
    """Save the list of todos to file."""
    with open(TODO_FILE, "w") as file:
        for todo in todos:
            file.write(todo + "\n")
```

```
def show_todos(todos):
    """Print all todos with numbers."""
    if not todos:
        print("No todos yet.")
        return
    for i, todo in enumerate(todos, start=1):
        print(f"  {i}. {todo}")


todos = load_todos()

while True:
    print("\n--- To-Do List ---")
    show_todos(todos)
    print("\n1. Add  2. Remove  3. Quit")
    choice = input("Choice: ").strip()

    if choice == "1":
        task = input("New task: ").strip()
        if task:
            todos.append(task)
            save_todos(todos)
            print("Task added.")

    elif choice == "2":
        show_todos(todos)
        try:
            n = int(input("Remove number: "))
            removed = todos.pop(n - 1)
            save_todos(todos)
            print(f"Removed: {removed}")
        except (ValueError, IndexError):
            print("Invalid number.")

    elif choice == "3":
        print("Goodbye!")
        break
    else:
        print("Invalid choice.")
```

Run this program, add some tasks, quit, and run it again. Your tasks are still there. That's the difference files make.

8.10 What You Learned in This Chapter

- Files store data permanently — they survive after your program ends.
- File modes: `"r"` (read), `"w"` (write — overwrites!), `"a"` (append — adds to end).
- Always use `with open(...) as file:` — it closes the file automatically.
- Reading: `file.read()` (whole file), `file.readlines()` (list of lines), `for line in file` (one line at a time — best for large files).
- Writing: `file.write(text)` — you must add `\n` yourself for new lines.
- Relative paths are relative to where the script runs. Absolute paths are the full location.
- Use `pathlib.Path` for cross-platform path handling.
- Wrap file opens in `try/except FileNotFoundError` to handle missing files gracefully.
- `csv.reader` reads CSV rows as lists. `csv.DictReader` reads them as dictionaries.
- `csv.writer` writes CSV files.

8.11 What's Next?

You can now read, write, and persist data. The next chapter brings everything together — variables, lists, loops, functions, and files — in a complete beginner project: a to-do list app built from scratch, step by step, with every decision explained.

Your turn: Write a program that reads a CSV file of products (name, price), applies a 10% discount to every price, and writes the updated data to a new CSV file called `discounted.csv`. Print a summary showing the original total value and the discounted total value.

Chapter 9

Chapter 9: A Beginner Project — Score Keeper

You've covered eight chapters. You know variables, operators, conditionals, loops, lists, functions, and files. That's a real toolkit. Now it's time to use it.

This chapter builds a complete project from scratch — a **Score Keeper** for a two-team game. You'll plan it, build it piece by piece, test it at each step, and end up with a fully working program that saves results to a file.

The goal isn't just to get the program working. It's to show you *how a programmer thinks*: start small, add one feature at a time, test constantly, and build up to something complete.

9.1 What We're Building

A Score Keeper that:

- Asks for two team names at the start
- Keeps a running score for both teams
- Lets the user add a point for either team after each round
- Shows the current score after every update
- Announces the winner when the game ends
- Saves the final result to a file so there's a record

Simple enough to finish in one sitting. Complex enough to use everything you've learned.

9.2 Step 1: Plan Before You Code

Professional programmers don't just start typing. They spend a minute thinking about what the program needs to do and what it needs to *know*.

What does the program need to store? - Two team names (strings) - Two scores (integers)

What does the program need to do? - Get the team names from the user - Repeatedly: ask who scored, update the right score, show the scores - Detect when the user wants to end the game - Show the winner - Save the result to a file

What's the flow?

1. Get team names
2. Loop:
    - Show current score
    - Ask: team 1, team 2, or done?
    - If team 1 or 2: add a point and continue
    - If done: break
3. Show the winner
4. Save results

That's the whole program, before a single line of code. Writing it out like this — even just as bullet points — saves you from getting lost once the code gets longer.

9.3 Step 2: Get the Team Names

Start with just this part. Run it. Make sure it works before moving on.

```
# score_keeper.py

team1_name = input("Enter name for Team 1: ").strip()
team2_name = input("Enter name for Team 2: ").strip()

print(f"\nGame: {team1_name} vs {team2_name}")
print("Let's play!")
```

Sample run:

```
Enter name for Team 1: Eagles
Enter name for Team 2: Tigers

Game: Eagles vs Tigers
```

```
Let's play!
```

Test it. Does it show both names? Good. Move on.

The `.strip()` removes any accidental spaces the user might type before or after the name. Small detail — big difference when comparing strings later.

9.4 Step 3: Set Up the Scores and the Main Loop

Add the score variables and the loop structure.

```
# score_keeper.py

team1_name = input("Enter name for Team 1: ").strip()
team2_name = input("Enter name for Team 2: ").strip()

team1_score = 0
team2_score = 0

print(f"\nGame: {team1_name} vs {team2_name}")
print("Commands: '1' to score for Team 1, '2' for Team 2, 'q' to quit.\n")

while True:
    print(f"  {team1_name}: {team1_score}  |  {team2_name}: {team2_score}")
    command = input("Who scored? ").strip().lower()

    if command == "1":
        team1_score += 1
        print(f"  Point for {team1_name}!")
    elif command == "2":
        team2_score += 1
        print(f"  Point for {team2_name}!")
    elif command == "q":
        print("\nGame over!")
        break
    else:
        print("  Invalid input. Type '1', '2', or 'q'.")

print(f"\nFinal score --- {team1_name}: {team1_score}  |  {team2_name}: {team2_
```

Sample run:

```
Enter name for Team 1: Eagles
```

```
Enter name for Team 2: Tigers

Game: Eagles vs Tigers
Commands: '1' to score for Team 1, '2' for Team 2, 'q' to quit.

  Eagles: 0  |  Tigers: 0
Who scored? 1
  Point for Eagles!
  Eagles: 1  |  Tigers: 0
Who scored? 2
  Point for Tigers!
  Eagles: 1  |  Tigers: 1
Who scored? 1
  Point for Eagles!
  Eagles: 2  |  Tigers: 1
Who scored? q

Game over!

Final score --- Eagles: 2  |  Tigers: 1
```

Run this. Score a few points for each team. Quit. Does it work exactly as expected? Good. Now we add the winner announcement.

9.5 Step 4: Announce the Winner

Add this right after the `break` but before the final score print — or better, after the final score print. Think about where it belongs logically: *after* the game ends and *after* showing the final score.

```python
# After the final score line, add:

if team1_score > team2_score:
    print(f"\nWinner: {team1_name}! Congratulations!")
elif team2_score > team1_score:
    print(f"\nWinner: {team2_name}! Congratulations!")
else:
    print("\nIt's a tie!")
```

Sample output for a tied game:

```
Final score --- Eagles: 3  |  Tigers: 3

It's a tie!
```

Three conditions: team1 wins, team2 wins, or a tie. All three handled. Test all three by playing short games with known outcomes.

9.6 Step 5: Save the Result to a File

After every game, write the result to `results.txt` using append mode so each game's result is added to the history, not overwritten.

```
from datetime import datetime

def save_result(team1_name, team1_score, team2_name, team2_score):
    """Append the game result to results.txt with a timestamp."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M")

    if team1_score > team2_score:
        winner = team1_name
    elif team2_score > team1_score:
        winner = team2_name
    else:
        winner = "Tie"

    result_line = (
        f"{timestamp} | "
        f"{team1_name} {team1_score} - {team2_score} {team2_name} | "
        f"Winner: {winner}\n"
    )

    with open("results.txt", "a") as file:
        file.write(result_line)

    print(f"Result saved to results.txt.")
```

Call this function after announcing the winner:

```
save_result(team1_name, team1_score, team2_name, team2_score)
```

Play two or three games. Then open `results.txt` in a text editor. You'll see something like:

```
2026-03-09 14:30 | Eagles 3 - 2 Tigers | Winner: Eagles
2026-03-09 14:35 | Lions 1 - 1 Bears | Winner: Tie
2026-03-09 14:40 | Eagles 5 - 3 Lions | Winner: Eagles
```

A permanent record of every game played.

9.7 Step 6: The Complete Program

Here's the full program, clean and organized:

```
from datetime import datetime


def get_team_names():
    """Ask for two team names and return them."""
    name1 = input("Enter name for Team 1: ").strip()
    name2 = input("Enter name for Team 2: ").strip()
    return name1, name2


def show_score(team1_name, team1_score, team2_name, team2_score):
    """Print the current score."""
    print(f"  {team1_name}: {team1_score}  |  {team2_name}: {team2_score}")


def announce_winner(team1_name, team1_score, team2_name, team2_score):
    """Print the winner or declare a tie."""
    print(f"\nFinal: {team1_name} {team1_score} --- {team2_name} {team2_score}")
    if team1_score > team2_score:
        print(f"Winner: {team1_name}! Congratulations!")
    elif team2_score > team1_score:
        print(f"Winner: {team2_name}! Congratulations!")
    else:
        print("It's a tie!")


def save_result(team1_name, team1_score, team2_name, team2_score):
    """Append the game result to results.txt with a timestamp."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M")

    if team1_score > team2_score:
        winner = team1_name
    elif team2_score > team1_score:
        winner = team2_name
    else:
        winner = "Tie"

    line = (
        f"{timestamp} | "
        f"{team1_name} {team1_score} - {team2_score} {team2_name} | "
        f"Winner: {winner}\n"
```

```
    )
    with open("results.txt", "a") as file:
        file.write(line)

    print("Result saved to results.txt.")


def play_game():
    """Run one full game of score keeper."""
    team1_name, team2_name = get_team_names()

    team1_score = 0
    team2_score = 0

    print(f"\nGame: {team1_name} vs {team2_name}")
    print("Type '1' to score for Team 1, '2' for Team 2, 'q' to end the game.\n

    while True:
        show_score(team1_name, team1_score, team2_name, team2_score)
        command = input("Who scored? ").strip().lower()

        if command == "1":
            team1_score += 1
            print(f"  Point for {team1_name}!\n")
        elif command == "2":
            team2_score += 1
            print(f"  Point for {team2_name}!\n")
        elif command == "q":
            break
        else:
            print("  Invalid. Type '1', '2', or 'q'.\n")

    announce_winner(team1_name, team1_score, team2_name, team2_score)
    save_result(team1_name, team1_score, team2_name, team2_score)


def main():
    """Main entry point --- play games until the user quits."""
    print("=== Score Keeper ===\n")

    while True:
        play_game()
        again = input("\nPlay again? (y/n): ").strip().lower()
        if again != "y":
```

```
            print("\nThanks for playing. Goodbye!")
            break


main()
```

A complete, working program. Functions make it readable — each one does exactly one job. `main()` ties them together.

9.8 What the Code Teaches

Look at this program and notice the patterns you've seen in every chapter:

Pattern	Where it appears
Variables	`team1_name`, `team1_score`, `team2_name`, `team2_score`
f-strings	Every `print` with team names and scores
`if/elif/else`	Checking who scored, announcing the winner
`while True` + `break`	The main game loop
Functions with parameters and return	`get_team_names()`, `save_result()`, etc.
File append mode	Saving the result history
`datetime` module	Timestamping each result
`.strip().lower()`	Cleaning user input

There's nothing in this program you haven't seen before. It's just the same building blocks, combined with intention.

9.9 How to Extend This Project

Once a program works, the real learning comes from extending it. Here are ideas in order of difficulty:

Beginner extensions:

- Add a "show history" option that reads and prints `results.txt` before asking to play again.
- Add input validation so team names can't be empty.

- Add a maximum score — the game ends automatically when someone reaches 10 points.

Intermediate extensions:

- Support more than two teams (use a dictionary mapping name to score).
- Track how many games each team has won across all sessions (requires more advanced file parsing).
- Add a rematch feature that swaps which team goes first.

Try one now. Pick the simplest — adding a maximum score — and implement it. You already know everything you need:

```
MAX_SCORE = 10

# Inside the while loop, after updating the score:
if team1_score >= MAX_SCORE or team2_score >= MAX_SCORE:
    print(f"\nSomeone reached {MAX_SCORE} points!")
    break
```

Make the change, test it, and see it work. That satisfaction — writing new code that does exactly what you intended — is what programming feels like when things click.

9.10 Ideas for Your Own Project

The best way to learn from here is to build something *you* care about. Here are three project ideas using only what you've learned so far:

Quiz Game

Store questions and answers in two lists. Loop through them. Ask each question with `input()`. Compare the answer to the correct one. Keep a score. Print results at the end. Bonus: save the high score to a file.

Simple Diary

Ask the user for today's entry using `input()`. Append it to `diary.txt` with a date and time stamp. Add an option to show all previous entries by reading and printing the file. This teaches append mode and date formatting.

Unit Converter

Ask what to convert (km to miles, kg to pounds, Celsius to Fahrenheit). Ask for the value. Calculate and print the result. Use a dictionary to store conversion factors. Loop so the user can convert multiple values in one session.

All three use variables, lists, loops, functions, and optionally files. Start with one, get it working, then extend it.

9.11 What You Learned in This Chapter

- Real programs are built in steps: plan, build a little, test, build more.
- Functions make programs readable — one function, one job.
- You can use everything from Chapters 1–8 together in one program.
- `while True` + `break` is the standard pattern for interactive menus.
- File append mode (`"a"`) builds a growing history without overwriting.
- Extending a working program is the fastest way to deepen your understanding.

9.12 What's Next?

Part 1 of this book is complete. You've gone from zero to writing a real, working, file-saving Python program. That's not nothing — that's a foundation most people never build properly.

Part 2 starts now, and it goes deeper. In Chapter 10 you'll learn about **strings in depth** — all the methods Python gives you to search, slice, transform, and format text. You've been using strings since Chapter 1; it's time to master them.

Your turn: Choose one of the three project ideas above (quiz, diary, or unit converter) and build it from scratch. Don't copy from the book — plan it yourself, write it step by step, test at each step. When it works, add one extension. That's what being a programmer looks like.

Chapter 10

Chapter 10: Strings in Depth

You've been using strings since Chapter 1. Every name, message, and label in your programs has been a string. But so far you've only scratched the surface.

Strings in Python are extraordinarily powerful. Python gives you dozens of built-in methods to search, split, transform, format, and validate text — and you'll use them constantly. Almost every real program deals with text: reading user input, processing files, building messages, parsing data. Master strings and a huge part of Python becomes easy.

This chapter covers everything.

10.1 Strings Are Sequences

A string is not just a blob of text — it's a sequence of characters. That means everything you learned about sequences in Chapter 6 (lists) applies here too: indexing, slicing, and looping.

```
name = "Python"

print(name[0])     # P
print(name[1])     # y
print(name[-1])    # n  (last character)
print(name[-2])    # o  (second to last)
print(len(name))   # 6
```

10.1.1 Slicing strings

```
language = "Python 3.12"

print(language[0:6])     # Python
print(language[7:])      # 3.12
print(language[:6])      # Python
print(language[::2])     # Pto .2  (every other character)
print(language[::-1])    # 21.3 nohtyP  (reversed)
```

10.1.2 Looping over a string

```
for char in "Hello":
    print(char)
```

Output:

```
H
e
l
l
o
```

10.1.3 Strings are immutable

Unlike lists, you cannot change a character in a string by assigning to an index. Strings are **immutable** — once created, they cannot be modified.

```
name = "Python"
name[0] = "J"   # TypeError: 'str' object does not support item assignment
```

To "change" a string, you create a new one:

```
name = "Python"
name = "J" + name[1:]
print(name)    # Jython
```

Your turn: Take the string `"Hello, World!"`. Using slicing, extract and print just `"World"`.

10.2 String Methods

A method is a function that belongs to an object. You call it with a dot: `string.method()`. String methods never modify the original string — they always return a new one.

10.2.1 Case methods

```
message = "Hello, World!"

print(message.upper())       # HELLO, WORLD!
print(message.lower())       # hello, world!
print(message.title())       # Hello, World!
print(message.capitalize()) # Hello, world!
print(message.swapcase())    # hELLO, wORLD!
```

Real use: normalizing user input before comparison.

```
answer = input("Continue? (yes/no): ").strip().lower()

if answer == "yes":
    print("Continuing...")
elif answer == "no":
    print("Stopping.")
else:
    print("Please type yes or no.")
```

`.lower()` means `"Yes"`, `"YES"`, `"yes"` all match `"yes"`. Without it, the user has to type the exact case you expect.

10.2.2 Stripping whitespace

```
text = "   hello world   "

print(text.strip())     # "hello world"   (both ends)
print(text.lstrip())    # "hello world   " (left only)
print(text.rstrip())    # "   hello world"  (right only)
```

Always `.strip()` user input. People accidentally add spaces all the time.

You can also strip specific characters:

```
url = "https://example.com/"
print(url.strip("/"))    # https://example.com
```

10.2.3 `split()` — Break a string into a list

```
sentence = "Python is easy to learn"
words = sentence.split()
print(words)    # ['Python', 'is', 'easy', 'to', 'learn']
```

Without arguments, `split()` splits on any whitespace. Pass a delimiter to split on something specific:

```
csv_line = "Alice,88,London"
parts = csv_line.split(",")
print(parts)    # ['Alice', '88', 'London']

date = "2026-03-09"
year, month, day = date.split("-")
print(year, month, day)    # 2026 03 09
```

`split()` with unpacking is one of the most useful patterns in Python for parsing text data.

10.2.4 `join()` — Combine a list into a string

`join()` is the opposite of `split()`. It takes a list of strings and joins them with a separator.

```
words = ["Python", "is", "fun"]
sentence = " ".join(words)
print(sentence)    # Python is fun

items = ["bread", "milk", "eggs"]
print(", ".join(items))    # bread, milk, eggs
print(" | ".join(items))   # bread | milk | eggs
print("\n".join(items))    # each item on its own line
```

The pattern to remember: separator goes *before* `.join()`, list goes *inside* the parentheses.

10.2.5 `replace()` — Swap text

```
text = "I love Java. Java is great."
new_text = text.replace("Java", "Python")
print(new_text)    # I love Python. Python is great.
```

Replace only the first occurrence:

```
print(text.replace("Java", "Python", 1))    # I love Python. Java is great.
```

10.2.6 `find()` and `index()` — Search for a substring

`find()` returns the index of the first occurrence of a substring. Returns `-1` if not found.

```
email = "alice@example.com"
at_position = email.find("@")
print(at_position)    # 5
```

`index()` does the same but raises a `ValueError` if not found (instead of returning `-1`). Use `find()` when "not found" is a normal situation; use `index()` when not finding it would be a bug.

```
text = "Hello, World!"
print(text.find("World"))     # 7
print(text.find("Python"))    # -1  (not found)
```

10.2.7 `startswith()` and `endswith()`

```
filename = "report_2026.csv"

print(filename.startswith("report"))    # True
print(filename.endswith(".csv"))        # True
print(filename.endswith(".pdf"))        # False
```

Pass a tuple to check multiple options at once:

```
print(filename.endswith((".csv", ".xlsx", ".txt")))    # True
```

Real use: filtering files by extension.

```
files = ["data.csv", "notes.txt", "image.png", "sales.csv", "readme.md"]
csv_files = [f for f in files if f.endswith(".csv")]
print(csv_files)    # ['data.csv', 'sales.csv']
```

10.2.8 `count()` — How many times does a substring appear?

```
text = "banana"
print(text.count("a"))     # 3
print(text.count("an"))    # 2
```

10.2.9 `in` — Check if a substring exists

```
sentence = "The quick brown fox"

print("fox" in sentence)       # True
print("cat" in sentence)       # False
print("quick" in sentence)     # True
```

This is often cleaner than `find()` when you just need a yes/no answer:

```
email = input("Enter your email: ")

if "@" not in email or "." not in email:
    print("That doesn't look like a valid email.")
```

Your turn: Take this string: `"  the Quick Brown FOX jumps OVER the lazy DOG  "`. In one chain of method calls, strip the whitespace, convert to title case, and print the result.

10.3 String Formatting

Python gives you three ways to format strings. You met f-strings in Chapter 2 — now let's see all three and understand why f-strings are the right choice for almost everything.

10.3.1 1. `%` formatting (old style — avoid in new code)

```
name = "Alice"
age = 30
print("My name is %s and I am %d years old." % (name, age))
```

`%s` for strings, `%d` for integers, `%f` for floats. You'll see this in older Python code. Don't write it in new code.

10.3.2 2. `.format()` (better — still common)

```
name = "Alice"
age = 30
print("My name is {} and I am {} years old.".format(name, age))

# Named placeholders
print("My name is {name} and I am {age} years old.".format(name=name, age=age))
```

More readable than `%`, but more verbose than f-strings.

10.3.3 3. F-strings (best — use these)

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

F-strings are: - **Faster** — evaluated at runtime, not parsed - **Cleaner** — expression goes right in the string - **More powerful** — any Python expression works inside `{}`

```
price = 49.99
quantity = 3

print(f"Total: ${price * quantity:.2f}")    # Total: $149.97
```

The `:.2f` inside the `{}` formats the number to 2 decimal places.

10.3.4 F-string formatting cheat sheet

```
pi = 3.14159265

print(f"{pi:.2f}")        # 3.14      (2 decimal places)
print(f"{pi:.4f}")        # 3.1416    (4 decimal places)
print(f"{pi:10.2f}")      # '      3.14'  (width 10, right-aligned)
print(f"{pi:<10.2f}")     # '3.14      '  (left-aligned)
print(f"{pi:^10.2f}")     # '   3.14   '  (centered)

n = 1000000
print(f"{n:,}")           # 1,000,000  (thousands separator)
print(f"{n:_}")           # 1_000_000  (underscore separator)

pct = 0.87654
print(f"{pct:.1%}")       # 87.7%   (percentage)

# Padding and alignment with strings
name = "Alice"
print(f"|{name:10}|")     # |Alice     |  (left-aligned, width 10)
print(f"|{name:>10}|")    # |     Alice|  (right-aligned)
print(f"|{name:^10}|")    # |  Alice   |  (centered)
print(f"|{name:*^10}|")   # |**Alice** |  (centered, fill with *)
```

Your turn: Create a simple receipt printer. You have three items with prices. Print a formatted receipt where item names are left-aligned in a 20-character column and prices are right-aligned in an 8-character column, showing 2 decimal places. Add a total at the bottom.

10.4 Checking String Content

Python has a set of methods that check *what kind of characters* a string contains. They all return `True` or `False`.

```
print("12345".isdigit())      # True  --- all digits
print("12.5".isdigit())       # False --- period is not a digit
print("hello".isalpha())      # True  --- all letters
print("hello1".isalpha())     # False --- has a digit
print("hello1".isalnum())     # True  --- letters and digits only
print("   ".isspace())        # True  --- all whitespace
print("Hello World".istitle()) # True --- title case
```

```
print("HELLO".isupper())      # True  --- all uppercase
print("hello".islower())      # True  --- all lowercase
```

Real use: validating user input.

```
def get_positive_integer(prompt):
    """Keep asking until the user enters a positive integer."""
    while True:
        value = input(prompt).strip()
        if value.isdigit():
            return int(value)
        print("Please enter a whole number greater than zero.")

age = get_positive_integer("Enter your age: ")
print(f"You are {age} years old.")
```

`isdigit()` is a clean way to validate numeric input without using `try/except` (though `try/except` handles negatives and decimals better — you'll pick the right tool for each situation).

10.5 Multiline Strings and Raw Strings

You saw these in Chapter 2. Here's the full picture.

10.5.1 Triple-quoted strings

```
poem = """
Roses are red,
Violets are blue,
Python is great,
And so are you.
"""
print(poem)
```

Triple quotes preserve all whitespace and newlines exactly as written. Useful for long messages, email templates, SQL queries, and HTML snippets embedded in Python.

10.5.2 Raw strings

A raw string treats backslashes as literal characters — no escape processing.

```
# Normal string --- \n is a newline, \t is a tab
print("First line\nSecond line")
```

```
# Raw string --- \n is literally backslash-n
print(r"First line\nSecond line")
```

Output:

```
First line
Second line
First line\nSecond line
```

Essential for Windows file paths and regular expressions:

```
# Windows path --- without raw string you'd need "C:\\Users\\Alice\\file.txt"
path = r"C:\Users\Alice\Documents\file.txt"

# Regular expression pattern --- backslashes are common
import re
pattern = r"\d{3}-\d{4}"   # matches phone number format like 555-1234
```

10.6 Unicode and Encoding

Every character in Python is a Unicode character. Unicode is a standard that assigns a unique number to every character in every writing system on earth — Latin letters, Arabic, Chinese, emoji, everything.

```
print("Hello")          # English
print("□□□□□")      # Japanese
print("Привет")         # Russian
print("مرحبا")          # Arabic
print("")             # Snake emoji
```

All of these work in Python 3 strings.

10.6.1 `ord()` and `chr()`

`ord()` gives you the Unicode code point (number) of a character. `chr()` gives you the character for a code point.

```
print(ord("A"))     # 65
print(ord("a"))     # 97
print(ord(""))    # 128013

print(chr(65))      # A
print(chr(9829))    # ♥
```

10.6.2 Encoding

When Python writes strings to a file or sends them over a network, it must convert them to bytes. The most common encoding is **UTF-8**, and it's what Python uses by default.

```
text = "Hello, □□"
encoded = text.encode("utf-8")
print(encoded)    # b'Hello, \xe4\xb8\x96\xe7\x95\x8c'

decoded = encoded.decode("utf-8")
print(decoded)    # Hello, □□
```

In practice, you almost never need to think about this. Python handles it for you. The one time it matters: if you're opening a file and see strange characters, specify the encoding explicitly:

```
with open("data.txt", "r", encoding="utf-8") as file:
    content = file.read()
```

If you're working with files from Windows, you may sometimes encounter `encoding="latin-1"` or `encoding="cp1252"`. UTF-8 is always the right choice for new files.

10.7 String Methods Reference

Here's a quick reference of every method covered in this chapter:

Method	What it does	Example
`.upper()`	All uppercase	`"hi".upper()` -> `"HI"`
`.lower()`	All lowercase	`"HI".lower()` -> `"hi"`
`.title()`	Title Case	`"hello world".title()` -> `"Hello World"`
`.capitalize()`	First letter uppercase	`"hello".capitalize()` -> `"Hello"`
`.strip()`	Remove leading/trailing whitespace	`"  hi  ".strip()` -> `"hi"`
`.lstrip()`	Remove leading whitespace	
`.rstrip()`	Remove trailing whitespace	
`.split(sep)`	Split into list	`"a,b,c".split(",")` -> `["a","b","c"]`

Method	What it does	Example
`.join(list)`	Join list into string	`",".join(["a","b"])` -> `"a,b"`
`.replace(old, new)`	Replace substring	`"hi".replace("h","H")` -> `"Hi"`
`.find(sub)`	First index of substring, -1 if not found	`"hello".find("l")` -> `2`
`.index(sub)`	First index, raises ValueError if not found	
`.startswith(prefix)`	Starts with?	`"hello".startswith("he")` -> `True`
`.endswith(suffix)`	Ends with?	`"hello".endswith("lo")` -> `True`
`.count(sub)`	Count occurrences	`"banana".count("a")` -> `3`
`.isdigit()`	All digits?	`"123".isdigit()` -> `True`
`.isalpha()`	All letters?	`"abc".isalpha()` -> `True`
`.isalnum()`	Letters and digits only?	`"abc1".isalnum()` -> `True`
`.isspace()`	All whitespace?	`"  ".isspace()` -> `True`
`.isupper()`	All uppercase?	`"HI".isupper()` -> `True`
`.islower()`	All lowercase?	`"hi".islower()` -> `True`

10.8 Putting It All Together: A Text Analyzer

```
def analyze_text(text):
    """Print a detailed analysis of a text string."""
    words = text.split()
    sentences = text.replace("!", ".").replace("?", ".").split(".")
    sentences = [s.strip() for s in sentences if s.strip()]

    print(f"--- Text Analysis ---")
    print(f"Characters (with spaces):    {len(text)}")
    print(f"Characters (no spaces):      {len(text.replace(' ', ''))}")
    print(f"Words:                       {len(words)}")
    print(f"Sentences:                   {len(sentences)}")
```

```
    print(f"Unique words:                {len(set(w.lower().strip('.,!?') for w in
    print(f"Average word length:         {sum(len(w) for w in words) / len(words):

    most_common = max(set(words), key=words.count)
    print(f"Most common word:            '{most_common}' ({words.count(most_common

    print(f"\nFirst 50 chars:  '{text[:50]}'")
    print(f"Uppercase:       '{text.upper()[:50]}'")

sample = (
    "Python is a great language. Python is easy to learn. "
    "Many people use Python for data science and web development. "
    "Python is growing fast!"
)

analyze_text(sample)
```

Output:

```
--- Text Analysis ---
Characters (with spaces):    122
Characters (no spaces):      103
Words:                       21
Sentences:                   4
Unique words:                15
Average word length:         5.3
Most common word:            'Python' (4x)

First 50 chars:  'Python is a great language. Python is easy to lea'
Uppercase:       'PYTHON IS A GREAT LANGUAGE. PYTHON IS EASY TO LEA'
```

Every technique from this chapter, working together on a real task.

10.9 What You Learned in This Chapter

- Strings are immutable sequences — you can index, slice, and loop over them, but not change them in place.
- Key methods: `upper()`, `lower()`, `title()`, `strip()`, `split()`, `join()`, `replace()`, `find()`, `startswith()`, `endswith()`, `count()`
- Content checks: `isdigit()`, `isalpha()`, `isalnum()`, `isspace()`, `isupper()`, `islower()`
- Three formatting styles: `%` (old), `.format()` (fine), f-strings (best — use these)

- F-string format spec: `:.2f` (floats), `:,` (thousands), `:.1%` (percent), `:<`, `:>`, `:^` (alignment)
- Triple-quoted strings preserve newlines and whitespace.
- Raw strings (`r""`) treat backslashes literally — essential for paths and regex.
- Python strings are Unicode. `ord()` and `chr()` convert between characters and code points.
- Specify `encoding="utf-8"` when opening files to avoid encoding issues.

10.10 What's Next?

Strings are sequences of characters. But Python has two more sequence types you haven't met yet — and they behave quite differently.

In Chapter 11 you'll learn about **tuples** (like lists, but unchangeable) and **sets** (unordered collections with no duplicates). They sound simple, but knowing when to use each one — and why — is what separates programmers who just write code from programmers who write *good* code.

Your turn: Write a function called `clean_name(name)` that takes a raw name string and returns a cleaned version: stripped of whitespace, converted to title case, and with any double spaces replaced with single spaces. Test it with inputs like `"  alice  smith  "`, `"BOB  JONES"`, and `"  carlos  "`.

Chapter 11

Chapter 11: Tuples and Sets

You know lists. Lists are ordered, changeable, and incredibly useful. But Python has two more collection types that solve different problems — and knowing which one to reach for is a mark of a thoughtful programmer.

Tuples are like lists that you can never change. Once you create a tuple, it's locked. That sounds like a limitation, but it's actually a feature — you use it to signal "this data is fixed and shouldn't change."

Sets are unordered collections with no duplicates. They're designed for membership testing and set math — finding what's common, what's different, and what's unique.

Both are lightweight, fast, and used constantly in real Python code. By the end of this chapter you'll know what they are, when to use them, and how to choose between a tuple, a set, and a list.

11.1 Tuples — Immutable Sequences

11.1.1 Creating a tuple

Use parentheses with comma-separated values:

```
point = (3, 7)
colors = ("red", "green", "blue")
person = ("Alice", 30, "London")
empty = ()
```

You can also create a tuple without parentheses — the comma is what makes a tuple, not the parentheses:

```
point = 3, 7
print(point)         # (3, 7)
print(type(point))   # <class 'tuple'>
```

One-item tuple: You need a trailing comma, otherwise Python just sees parentheses around a value:

```
not_a_tuple = (42)
is_a_tuple  = (42,)

print(type(not_a_tuple))   # <class 'int'>
print(type(is_a_tuple))    # <class 'tuple'>
```

11.1.2 Indexing and slicing — same as lists

```
colors = ("red", "green", "blue")

print(colors[0])     # red
print(colors[-1])    # blue
print(colors[1:])    # ('green', 'blue')
print(len(colors))   # 3
```

11.1.3 Tuples are immutable

```
colors = ("red", "green", "blue")
colors[0] = "yellow"    # TypeError: 'tuple' object does not support item assign
```

You can't change a tuple after creation. Period. This is intentional — it's the whole point.

11.1.4 Looping over a tuple

```
rgb = (255, 128, 0)

for value in rgb:
    print(value)
```

Output:

```
255
128
0
```

Everything you can do with a list in a `for` loop works with a tuple.

11.1.5 Tuple methods

Tuples have only two methods (because you can't change them, most list methods don't apply):

```
numbers = (1, 2, 3, 2, 4, 2, 5)

print(numbers.count(2))    # 3  --- how many times 2 appears
print(numbers.index(4))    # 4  --- position of first 4
```

11.1.6 Tuple packing and unpacking

Packing is putting multiple values into a tuple:

```
coordinates = 40.7128, -74.0060    # New York City lat/long
print(coordinates)    # (40.7128, -74.006)
```

Unpacking is the reverse — splitting a tuple into individual variables:

```
x, y = (3, 7)
print(x)    # 3
print(y)    # 7
```

```
name, age, city = ("Alice", 30, "London")
print(f"{name} is {age} and lives in {city}.")
# Alice is 30 and lives in London.
```

Unpacking is one of Python's most elegant features. You've already seen it when a function returns multiple values:

```
def get_dimensions():
    return 1920, 1080    # returns a tuple

width, height = get_dimensions()
print(f"{width} x {height}")    # 1920 x 1080
```

Swap two variables using tuple unpacking — no temporary variable needed:

```
a = 10
b = 20

a, b = b, a    # swap!

print(a)    # 20
print(b)    # 10
```

In most languages, swapping requires a temporary variable. In Python, tuple unpacking makes it a one-liner. You'll use this more than you'd expect.

Extended unpacking with `*`:

```
first, *middle, last = (1, 2, 3, 4, 5)

print(first)     # 1
print(middle)    # [2, 3, 4]
print(last)      # 5
```

The `*` soaks up everything that doesn't fit into the named variables. The `middle` variable becomes a list.

```
head, *tail = ("apple", "banana", "cherry", "date")

print(head)    # apple
print(tail)    # ['banana', 'cherry', 'date']
```

Your turn: Create a tuple with five numbers. Unpack the first and last into named variables using extended unpacking. Print all three parts.

11.2 When to Use a Tuple Instead of a List

This is the question beginners always ask. Here's the honest answer:

Use a tuple when the data is fixed and shouldn't change. Use a list when the data might grow, shrink, or be updated.

Situation	Use
RGB color value: `(255, 128, 0)`	Tuple — it's a fixed set of three values
GPS coordinates: `(40.7128, -74.0060)`	Tuple — lat/long don't change
Days of the week	Tuple — there are always seven
A shopping cart	List — items get added and removed
Student grades	List — new grades get added
Returning multiple values from a function	Tuple
Dictionary keys	Tuple (lists can't be dict keys; tuples can)

Tuples as dictionary keys is a real practical reason to use them:

```
# Map grid coordinates to a value
grid = {
    (0, 0): "start",
    (1, 0): "wall",
    (1, 1): "open",
```

```
    (2, 2): "goal",
}

print(grid[(0, 0)])    # start
print(grid[(2, 2)])    # goal
```

A list cannot be used as a dictionary key — only immutable types can. Tuples are immutable, so they can.

Tuples are slightly faster and use less memory than lists. For large amounts of fixed data, this matters.

Your turn: A record store has albums stored as `(title, artist, year)` tuples in a list. Create five album tuples, put them in a list, and loop through the list printing each album's details in a formatted string.

11.3 Sets — Unique, Unordered Collections

A **set** stores unique values with no defined order. If you add a duplicate, it's silently ignored. There's no first item or last item — sets don't track position.

11.3.1 Creating a set

Use curly braces `{}`:

```
fruits = {"apple", "banana", "cherry"}
print(fruits)    # {'apple', 'cherry', 'banana'}  (order not guaranteed)
```

Or use `set()` to create a set from another iterable:

```
numbers = set([1, 2, 3, 2, 1, 4])
print(numbers)    # {1, 2, 3, 4}  (duplicates removed)
```

Important: An empty `{}` creates a dict, not a set. Use `set()` for an empty set:

```
empty_dict = {}
empty_set  = set()
```

11.3.2 Duplicates are automatically removed

This is the most common reason to use a set — you have a list with duplicates and you want only the unique values.

```
votes = ["Alice", "Bob", "Alice", "Carlos", "Bob", "Alice", "Diana"]
unique_voters = set(votes)
print(unique_voters)      # {'Alice', 'Bob', 'Carlos', 'Diana'}
print(len(unique_voters))  # 4
```

11.3.3 Adding and removing

```
fruits = {"apple", "banana", "cherry"}

fruits.add("date")
print(fruits)   # {'apple', 'banana', 'cherry', 'date'}

fruits.add("apple")   # duplicate --- silently ignored
print(fruits)   # still 4 items

fruits.remove("banana")   # raises KeyError if not found
print(fruits)

fruits.discard("grape")   # safe remove --- no error if not found
print(fruits)
```

Use `discard()` when you're not sure if the item is there. Use `remove()` when it must be there (and you want an error if it's not).

11.3.4 Membership testing — this is where sets shine

```
allowed_extensions = {".jpg", ".png", ".gif", ".webp"}

filename = "photo.jpg"
ext = filename[filename.rfind("."):]

if ext in allowed_extensions:
    print("File type accepted.")
else:
    print("File type not allowed.")
```

Checking `in` for a **set** is dramatically faster than checking `in` for a **list** — especially for large collections. A list scans every element one by one. A set uses a hash table and finds the answer almost instantly regardless of size.

For small collections the difference doesn't matter. For thousands or millions of items, it's the difference between fast and unusably slow.

```
# Use a set for fast lookups
blocked_users = {"spammer42", "troll99", "badactor"}

username = input("Enter username: ")

if username in blocked_users:
    print("Access denied.")
else:
    print(f"Welcome, {username}!")
```

Your turn: You have a list of 10 words, some repeated. Using a set, find how many unique words there are. Then check if the word `"python"` is in the set.

11.4 Set Operations — The Real Power

Sets support mathematical set operations: union, intersection, difference, and symmetric difference. These are incredibly useful for comparing collections.

11.4.1 Union — everything in either set

```
python_devs = {"Alice", "Bob", "Carlos", "Diana"}
java_devs   = {"Carlos", "Diana", "Eve", "Frank"}

all_devs = python_devs | java_devs
# or: python_devs.union(java_devs)

print(all_devs)
# {'Alice', 'Bob', 'Carlos', 'Diana', 'Eve', 'Frank'}
```

11.4.2 Intersection — only what's in both

```
both = python_devs & java_devs
# or: python_devs.intersection(java_devs)

print(both)   # {'Carlos', 'Diana'}
```

11.4.3 Difference — in the first but not the second

```
only_python = python_devs - java_devs
# or: python_devs.difference(java_devs)

print(only_python)   # {'Alice', 'Bob'}

only_java = java_devs - python_devs
print(only_java)   # {'Eve', 'Frank'}
```

11.4.4 Symmetric difference — in one or the other, but not both

```
exclusive = python_devs ^ java_devs
# or: python_devs.symmetric_difference(java_devs)
```

```
print(exclusive)    # {'Alice', 'Bob', 'Eve', 'Frank'}
```

11.4.5 Checking subset and superset

```
admins = {"Alice", "Bob"}
all_users = {"Alice", "Bob", "Carlos", "Diana"}

print(admins.issubset(all_users))      # True  --- all admins are users
print(all_users.issuperset(admins))    # True  --- all_users contains all admins
print(admins.isdisjoint({"Carlos"}))   # True  --- no overlap
```

11.4.6 A real example: comparing survey responses

```
survey_week1 = {"Alice", "Bob", "Carlos", "Diana", "Eve"}
survey_week2 = {"Bob", "Diana", "Frank", "Grace", "Alice"}

returned     = survey_week1 & survey_week2
new_this_week = survey_week2 - survey_week1
dropped_off  = survey_week1 - survey_week2
total_reached = survey_week1 | survey_week2

print(f"Responded both weeks: {returned}")
print(f"New this week:        {new_this_week}")
print(f"Didn't return:        {dropped_off}")
print(f"Total unique people:  {len(total_reached)}")
```

Output (order may vary):

```
Responded both weeks: {'Alice', 'Bob', 'Diana'}
New this week:        {'Frank', 'Grace'}
Didn't return:        {'Carlos', 'Eve'}
Total unique people:  7
```

Four lines of set operations replace what would otherwise take dozens of lines of loops and conditionals.

Your turn: You have two lists of student names — one for Math class and one for Science class. Using sets, find: students in both classes, students only in Math, students only in Science, and total unique students across both classes.

11.5 Updating Sets In Place

The set operators (`|`, `&`, `-`, `^`) return new sets. The update methods modify the set in place:

```
tags = {"python", "programming"}

tags.update({"web", "api"})          # add multiple items (union in place)
tags.intersection_update({"python", "web", "data"})  # keep only common items
tags.difference_update({"web"})      # remove items in another set

print(tags)    # {'python'}
```

11.6 `frozenset` — An Immutable Set

Just as tuples are immutable lists, `frozenset` is an immutable set. You can't add or remove items after creation. Why would you want this?

1. A `frozenset` can be used as a dictionary key (a regular set cannot).
2. A `frozenset` can be an element inside another set.

```
vowels = frozenset("aeiou")
print(vowels)            # frozenset({'a', 'e', 'i', 'o', 'u'})
print("a" in vowels)    # True

# Can be used as a dict key
valid_types = {
    frozenset({"jpg", "png"}): "image",
    frozenset({"mp3", "wav"}): "audio",
}
```

You'll use `frozenset` rarely, but it's good to know it exists when you need a hashable, immutable set.

11.7 Choosing Between List, Tuple, and Set

Need	Use
Ordered, changeable collection	`list`
Ordered, fixed collection	`tuple`
Unique values, no order needed	`set`
Fast membership testing	`set`

Need	Use
Dictionary key	`tuple` or `frozenset`
Multiple return values from a function	`tuple`
Remove duplicates from a list	`set(list)`
Set math (union, intersection, etc.)	`set`

The most common pattern you'll use from this chapter:

```
# Remove duplicates from a list while preserving order
items = ["banana", "apple", "banana", "cherry", "apple"]
seen = set()
unique = []
for item in items:
    if item not in seen:
        unique.append(item)
        seen.add(item)

print(unique)   # ['banana', 'apple', 'cherry']
```

Or the quick version when order doesn't matter:

```
items = ["banana", "apple", "banana", "cherry", "apple"]
unique = list(set(items))
print(unique)   # ['apple', 'banana', 'cherry']  (order not guaranteed)
```

11.8 Putting It All Together: Analyzing Word Frequency

```
def analyze_words(text):
    """Analyze word statistics in a block of text."""
    # Clean and split into words
    words = text.lower().split()
    cleaned = [w.strip(".,!?;:\"'") for w in words]

    total_words    = len(cleaned)
    unique_words   = set(cleaned)
    unique_count   = len(unique_words)
    most_common    = max(unique_words, key=cleaned.count)

    # Common words to ignore
    stopwords = frozenset({"the", "a", "an", "is", "in", "it",
                           "to", "and", "of", "for", "on", "are"})
```

```
    content_words  = unique_words - stopwords

    print(f"Total words:          {total_words}")
    print(f"Unique words:         {unique_count}")
    print(f"Content words:        {len(content_words)}")
    print(f"Most common word:     '{most_common}' ({cleaned.count(most_common)}x)")
    print(f"Vocabulary richness:  {unique_count / total_words:.1%}")


text = """
Python is a programming language. Python is easy to learn and Python is
powerful. Many programmers use Python for data science, web development,
and automation. Python is growing in popularity every year.
"""

analyze_words(text)
```

Output:

```
Total words:          31
Unique words:         19
Content words:        14
Most common word:     'python' (5x)
Vocabulary richness:  61.3%
```

Strings, lists, sets, and a frozenset — all working together in thirty lines.

11.9 What You Learned in This Chapter

- A **tuple** is an immutable, ordered sequence. Created with `()` or just commas.
- Tuples support indexing, slicing, `len()`, `count()`, and `index()`.
- Tuple **unpacking** assigns multiple variables at once: `a, b = (1, 2)`.
- Extended unpacking: `first, *rest = tuple` collects the remainder into a list.
- Swapping variables: `a, b = b, a` — no temporary variable needed.
- Use tuples for fixed data, multiple return values, and dictionary keys.
- A **set** is an unordered collection of unique values. Created with `{}` or `set()`.
- Sets automatically remove duplicates.
- `add()` adds one item. `discard()` removes without error. `remove()` raises `KeyError`.
- Membership testing with `in` is much faster on sets than lists.

- Set operations: `|` union, `&` intersection, `-` difference, `^` symmetric difference.
- `frozenset` is an immutable set — usable as a dict key or set element.
- Remove list duplicates: `list(set(my_list))` (order not preserved).

11.10 What's Next?

Tuples and sets handle ordered-but-fixed and unordered-unique data. But the most versatile collection in Python — the one you'll use in almost every real program — is the dictionary.

In Chapter 12 you'll learn **dictionaries**: key-value pairs that let you look up any piece of data by a meaningful name rather than a numeric index. Think of a dictionary as a real dictionary — you look up a word (the key) and get its definition (the value). Incredibly powerful, incredibly common.

Your turn: Write a program that reads a sentence from the user, finds the unique characters (excluding spaces), and prints them in sorted order. Then print how many unique characters there are. Bonus: also print which letters are vowels using set intersection.

Chapter 12

Chapter 12: Dictionaries — Key-Value Data

Lists store items by position. You get item number 0, item number 1, item number 2. That works when position is meaningful. But often you want to look something up by *name*, not by index.

Imagine a contact list. With a list, you'd need to remember that position 0 is the name, position 1 is the phone number, position 2 is the email. Fragile and unclear. With a dictionary, you look up `contact["phone"]` and get the phone number directly. No guessing, no counting.

Dictionaries are Python's most versatile data structure. You'll use them constantly — for configuration, for counting, for grouping data, for caching, for representing real-world objects. Master this chapter and a huge portion of real Python code becomes easy to read and write.

12.1 What is a Dictionary?

A dictionary stores **key-value pairs**. Every value has a unique key that you use to look it up. Think of a real dictionary: you look up a word (the key) and find its definition (the value).

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "London"
}
```

```
print(person)
```

Output:

```
{'name': 'Alice', 'age': 30, 'city': 'London'}
```

The syntax: curly braces `{}`, with `key: value` pairs separated by commas. Keys are usually strings (though any immutable type works). Values can be anything — strings, numbers, lists, other dictionaries, functions.

In plain English: A dictionary is like a labeled filing cabinet. Each drawer has a label (the key). You open the drawer to find what's inside (the value). You don't need to count to drawer 3 — you just ask for the drawer labeled "phone".

12.2 Creating a Dictionary

```
# Empty dictionary
empty = {}

# String keys
config = {
    "host": "localhost",
    "port": 5432,
    "database": "myapp",
    "debug": True
}

# Integer keys
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Mixed value types
student = {
    "name": "Carlos",
    "grade": 88,
    "subjects": ["Math", "Physics", "Chemistry"],
    "active": True
}
```

You can also create a dictionary from a list of key-value pairs using `dict()`:

```
pairs = [("name", "Alice"), ("age", 30), ("city", "London")]
person = dict(pairs)
print(person)   # {'name': 'Alice', 'age': 30, 'city': 'London'}
```

Or with keyword arguments:

```
person = dict(name="Alice", age=30, city="London")
print(person)    # {'name': 'Alice', 'age': 30, 'city': 'London'}
```

12.3 Accessing Values

Use the key in square brackets:

```
person = {"name": "Alice", "age": 30, "city": "London"}

print(person["name"])    # Alice
print(person["age"])     # 30
```

What if the key doesn't exist?

```
print(person["email"])    # KeyError: 'email'
```

A `KeyError` crashes your program. Use `.get()` to avoid this.

12.3.1 `.get()` — Safe access with a default

```
print(person.get("email"))              # None   (no crash)
print(person.get("email", "N/A"))       # N/A    (your default)
print(person.get("name", "Unknown"))    # Alice  (key exists, returns it)
```

`.get()` is one of the most used dictionary methods. Use it whenever the key might not be there.

Your turn: Create a dictionary for a movie with keys `title`, `director`, `year`, and `rating`. Print each value using both square bracket access and `.get()`. Try `.get()` with a key that doesn't exist and a default value.

12.4 Adding and Updating Values

Assign to a key to add or update:

```
person = {"name": "Alice", "age": 30}

# Update existing key
person["age"] = 31
print(person)    # {'name': 'Alice', 'age': 31}

# Add new key
person["email"] = "alice@example.com"
print(person)    # {'name': 'Alice', 'age': 31, 'email': 'alice@example.com'}
```

If the key exists, the value is overwritten. If it doesn't, a new key-value pair is added.

12.4.1 `update()` — Add or overwrite multiple pairs at once

```
person = {"name": "Alice", "age": 30}

person.update({"age": 31, "city": "Paris", "job": "Engineer"})
print(person)
# {'name': 'Alice', 'age': 31, 'city': 'Paris', 'job': 'Engineer'}
```

`update()` is useful when you have a batch of changes.

12.5 Deleting Keys

```
person = {"name": "Alice", "age": 30, "city": "London", "temp": "delete me"}

# del --- raises KeyError if key doesn't exist
del person["temp"]

# pop() --- removes and returns the value; optional default avoids KeyError
city = person.pop("city")
print(city)      # London
print(person)    # {'name': 'Alice', 'age': 30}

# pop with default --- no error if key missing
result = person.pop("missing_key", "not found")
print(result)    # not found

# clear() --- remove everything
person.clear()
print(person)    # {}
```

12.6 Checking if a Key Exists

```
person = {"name": "Alice", "age": 30}

print("name" in person)     # True
```

```
print("email" in person)    # False
print("email" not in person)  # True
```

Always check before accessing if you're not sure the key is there — or use `.get()`.

```
# Safe pattern
if "email" in person:
    print(person["email"])
else:
    print("No email on file.")
```

12.7 Looping Over a Dictionary

Python gives you three ways to loop over a dictionary.

12.7.1 `.keys()` — Loop over keys

```
scores = {"Alice": 88, "Bob": 72, "Carlos": 95}

for name in scores.keys():
    print(name)
```

Output:

```
Alice
Bob
Carlos
```

In fact, looping over a dictionary directly also gives you the keys:

```
for name in scores:    # same as scores.keys()
    print(name)
```

12.7.2 `.values()` — Loop over values

```
for score in scores.values():
    print(score)
```

Output:

```
88
72
95
```

12.7.3 `.items()` — Loop over key-value pairs (most common)

```
for name, score in scores.items():
    print(f"{name}: {score}")
```

Output:

```
Alice: 88
Bob: 72
Carlos: 95
```

`.items()` returns each pair as a tuple, which you unpack directly in the `for` loop. This is the pattern you'll use most often — it gives you both the key and the value on every iteration.

Your turn: Create a dictionary of five countries and their populations. Loop over `.items()` and print each country and population in a formatted string. Also print the total world population of those five countries.

12.8 Dictionary Methods Reference

```
d = {"a": 1, "b": 2, "c": 3}

print(d.keys())     # dict_keys(['a', 'b', 'c'])
print(d.values())   # dict_values([1, 2, 3])
print(d.items())    # dict_items([('a', 1), ('b', 2), ('c', 3)])

print(len(d))       # 3

d2 = d.copy()       # shallow copy --- independent from d
```

12.9 Nested Dictionaries

Dictionary values can be dictionaries themselves. This is how you represent structured, hierarchical data.

```
team = {
    "Alice": {"age": 30, "role": "Engineer", "salary": 95000},
    "Bob":   {"age": 25, "role": "Designer",  "salary": 75000},
    "Carlos":{"age": 35, "role": "Manager",   "salary": 110000},
}
```

```
# Access nested value
print(team["Alice"]["role"])      # Engineer
print(team["Carlos"]["salary"])   # 110000

# Loop over the nested structure
for name, info in team.items():
    print(f"{name} --- {info['role']} (${info['salary']:,})")
```

Output:

```
Alice --- Engineer ($95,000)
Bob --- Designer ($75,000)
Carlos --- Manager ($110,000)
```

Nested dictionaries are everywhere in real data: API responses, configuration files, JSON data. Get comfortable with them.

Adding to a nested dict:

```
team["Diana"] = {"age": 28, "role": "QA Engineer", "salary": 80000}
team["Alice"]["salary"] = 100000   # raise Alice's salary
```

Your turn: Build a nested dictionary representing a small school. Top-level keys are class names (`"Math"`, `"Science"`). Each value is a dictionary with `"teacher"` and `"students"` (a list of names). Print the teacher and number of students for each class.

12.10 Dictionary Comprehensions

Just like list comprehensions, you can create dictionaries in a single readable line.

The pattern: `{key_expr: value_expr for item in iterable}`

```
# Squares dictionary
squares = {n: n**2 for n in range(1, 6)}
print(squares)   # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Invert a dictionary (swap keys and values)
original = {"a": 1, "b": 2, "c": 3}
inverted = {value: key for key, value in original.items()}
print(inverted)   # {1: 'a', 2: 'b', 3: 'c'}

# Filter: keep only passing scores
scores = {"Alice": 88, "Bob": 45, "Carlos": 92, "Diana": 38, "Eve": 76}
passing = {name: score for name, score in scores.items() if score >= 50}
print(passing)   # {'Alice': 88, 'Carlos': 92, 'Eve': 76}
```

```
# Convert a list of words to a dict of word -> length
words = ["apple", "banana", "cherry", "date"]
lengths = {word: len(word) for word in words}
print(lengths)   # {'apple': 5, 'banana': 6, 'cherry': 6, 'date': 4}
```

Your turn: Given a list of temperatures in Celsius `[0, 10, 20, 30, 37, 100]`, create a dictionary mapping each Celsius value to its Fahrenheit equivalent using a dictionary comprehension.

12.11 The Counting Pattern

One of the most common uses of a dictionary is counting occurrences of items.

```
words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
counts = {}

for word in words:
    if word in counts:
        counts[word] += 1
    else:
        counts[word] = 1

print(counts)   # {'apple': 3, 'banana': 2, 'cherry': 1}
```

A cleaner version using `.get()`:

```
counts = {}
for word in words:
    counts[word] = counts.get(word, 0) + 1

print(counts)   # {'apple': 3, 'banana': 2, 'cherry': 1}
```

`counts.get(word, 0)` returns the current count (or 0 if not yet seen), then adds 1. This is the idiomatic Python counting pattern — you'll use it constantly.

12.12 `defaultdict` and `Counter` from `collections`

Python's standard library has two tools that make dictionary-based counting and grouping even easier.

12.12.1 `Counter` — Count things instantly

```
from collections import Counter

words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
counts = Counter(words)

print(counts)
# Counter({'apple': 3, 'banana': 2, 'cherry': 1})

print(counts["apple"])    # 3
print(counts["grape"])    # 0  (not KeyError --- returns 0 for missing keys)

# Most common
print(counts.most_common(2))
# [('apple', 3), ('banana', 2)]

# Count characters in a string
letter_counts = Counter("mississippi")
print(letter_counts)
# Counter({'s': 4, 'i': 4, 'p': 2, 'm': 1})

print(letter_counts.most_common(3))
# [('s', 4), ('i', 4), ('p', 2)]
```

`Counter` is a subclass of `dict`. It works anywhere a dict does, and it adds the `.most_common()` method.

12.12.2 `defaultdict` — Never get a KeyError again

`defaultdict` works like a regular dict, but if you access a missing key it automatically creates it with a default value.

```
from collections import defaultdict

# defaultdict with list --- groups items
groups = defaultdict(list)

students = [("Math", "Alice"), ("Science", "Bob"),
            ("Math", "Carlos"), ("Science", "Diana"), ("Math", "Eve")]

for subject, name in students:
    groups[subject].append(name)

print(dict(groups))
# {'Math': ['Alice', 'Carlos', 'Eve'], 'Science': ['Bob', 'Diana']}
```

Without `defaultdict`, you'd need to check `if subject in groups` before appending. With it, the missing key is created with an empty list automatically.

```
# defaultdict with int --- counting
from collections import defaultdict

word_count = defaultdict(int)

for word in ["apple", "banana", "apple", "cherry"]:
    word_count[word] += 1

print(dict(word_count))
# {'apple': 2, 'banana': 1, 'cherry': 1}
```

Your turn: Use `Counter` to count the frequency of each character in the string `"programming"`. Print the three most common characters.

12.13 Merging Dictionaries

12.13.1 Python 3.9+: the `|` operator

```
defaults = {"theme": "light", "font_size": 12, "language": "en"}
user_prefs = {"theme": "dark", "font_size": 14}

settings = defaults | user_prefs
print(settings)
# {'theme': 'dark', 'font_size': 14, 'language': 'en'}
```

The right-hand dict's values win on conflicts. This is perfect for layering defaults with user overrides.

12.13.2 `{**dict1, **dict2}` — Works in all Python 3 versions

```
settings = {**defaults, **user_prefs}
print(settings)
# {'theme': 'dark', 'font_size': 14, 'language': 'en'}
```

`**` unpacks a dictionary into key-value pairs. Spreading two dicts inside `{}` merges them.

12.14 Putting It All Together: A Word Frequency Analyzer

```
from collections import Counter


def word_frequency(text, top_n=10):
    """
    Analyze word frequency in a text.
    Returns the top_n most common words and their counts.
    """
    # Clean the text and split into words
    stop_words = {"the", "a", "an", "is", "in", "it", "to", "and",
                  "of", "for", "on", "are", "was", "that", "this",
                  "with", "as", "at", "be", "by", "from", "or"}

    words = text.lower().split()
    cleaned = [
        word.strip(".,!?;:\"'()-")
        for word in words
        if word.strip(".,!?;:\"'()-") not in stop_words
        and len(word.strip(".,!?;:\"'()-")) > 1
    ]

    counts = Counter(cleaned)
    total  = sum(counts.values())

    print(f"Total meaningful words: {total}")
    print(f"Unique words:           {len(counts)}")
    print(f"\nTop {top_n} words:")
    print(f"{'Word':<15} {'Count':>5} {'Frequency':>10}")
    print("-" * 33)

    for word, count in counts.most_common(top_n):
        freq = count / total
        print(f"{word:<15} {count:>5} {freq:>10.1%}")

    return counts


sample = """
Python is an interpreted high-level programming language. Python's design
philosophy emphasizes code readability. Python is dynamically typed and
garbage-collected. It supports multiple programming paradigms, including
```

```
structured, object-oriented and functional programming. Python is often
described as a batteries-included language due to its comprehensive
standard library. Python is consistently ranked as one of the most popular
programming languages.
"""

frequency = word_frequency(sample, top_n=8)
```

Output:

```
Total meaningful words: 42
Unique words:           36

Top 8 words:
Word            Count  Frequency
---------------------------------
python              5      11.9%
programming         3       7.1%
language            2       4.8%
design              1       2.4%
philosophy          1       2.4%
emphasizes          1       2.4%
code                1       2.4%
readability         1       2.4%
```

`Counter`, dictionary comprehension, f-string formatting, and loop unpacking — all working together.

12.15 What You Learned in This Chapter

- A dictionary stores **key-value pairs** using `{key: value}` syntax.
- Access values with `dict[key]` (raises `KeyError` if missing) or `dict.get(key, default)` (safe).
- Add or update: `dict[key] = value`. Delete: `del dict[key]` or `dict.pop(key)`.
- Check membership: `key in dict`.
- Loop with `.keys()`, `.values()`, or `.items()` (most common — gives both).
- Nested dictionaries represent hierarchical data.
- Dictionary comprehensions: `{k: v for k, v in iterable if condition}`.
- The counting pattern: `counts[word] = counts.get(word, 0) + 1`.
- `Counter` from `collections` — instant counting with `.most_common()`.
- `defaultdict` — auto-creates missing keys with a factory function.

- Merge dicts with `|` (Python 3.9+) or `{**d1, **d2}`.

12.16 What's Next?

You now have all five core Python data structures: strings, lists, tuples, sets, and dictionaries. Together they can represent almost any data you'll encounter.

But programs fail. Files don't exist. Users type the wrong thing. Network connections drop. In Chapter 13 you'll learn **error handling** — how to use `try`, `except`, `finally`, and `raise` to write programs that deal gracefully with problems instead of crashing. It's the difference between code that works in a demo and code that works in production.

Your turn: Write a program that reads a text file (you create it with a few paragraphs of any text), counts the frequency of every word using a `Counter`, and prints: the 10 most common words, the 10 least common words, the total word count, and the unique word count. Handle the case where the file doesn't exist with a friendly error message.

Chapter 13

Chapter 13: Error Handling — `try`, `except`, `finally`

Every program you've written so far assumes the best. The file exists. The user types a number when you ask for one. The network is up. The data is clean.

Reality is messier. Files get deleted. Users type letters when you expect numbers. Servers go down. Data arrives corrupted. In the real world, things go wrong constantly — and the programs that survive are the ones written to handle it.

Python gives you a clean, readable way to deal with errors: the `try` / `except` block. Instead of letting your program crash, you *catch* the error, deal with it, and keep going — or fail gracefully with a useful message.

This is one of the most important chapters in the book. Error handling is what separates code that works in a demo from code that works in production.

13.1 Why Programs Crash

When Python encounters a problem it can't resolve, it raises an **exception** — an error object that describes what went wrong. If nothing catches it, the program stops and prints a traceback.

You've already seen several:

```
print(10 / 0)
```

```
# ZeroDivisionError: division by zero

int("hello")
# ValueError: invalid literal for int() with base 10: 'hello'

name = "Alice"
print(name[100])
# IndexError: string index out of range

d = {"a": 1}
print(d["b"])
# KeyError: 'b'

open("missing_file.txt")
# FileNotFoundError: [Errno 2] No such file or directory: 'missing_file.txt'
```

Each error has a **type** (the class name before the colon) and a **message** (what comes after). Both give you information about what went wrong and where.

The key insight: exceptions are not disasters. They're Python's way of saying "I hit a situation I can't handle on my own. Can you?" With `try` / `except`, you can.

13.2 `try` and `except` — The Basic Pattern

Wrap the risky code in a `try` block. If an exception occurs, execution jumps immediately to the matching `except` block. If no exception occurs, the `except` block is skipped.

```
try:
    number = int(input("Enter a number: "))
    print(f"You entered: {number}")
except ValueError:
    print("That's not a valid number. Please try again.")

print("Program continues.")
```

Sample run 1 — user types `42`:

```
Enter a number: 42
You entered: 42
Program continues.
```

Sample run 2 — user types `hello`:

```
Enter a number: hello
```

```
That's not a valid number. Please try again.
Program continues.
```

Without the `try` / `except`, the second run would crash with a traceback and never reach the last line. With it, the program handles the problem and keeps going.

The flow: 1. Python tries the `try` block line by line. 2. If an exception occurs, it stops there and jumps to the matching `except`. 3. If no exception occurs, it skips all `except` blocks. 4. Either way, code after the `try` / `except` continues running.

13.3 Common Exception Types

Python has dozens of built-in exception types. Here are the ones you'll encounter most:

Exception	When it occurs
`ValueError`	Wrong type of value (e.g. `int("hello")`)
`TypeError`	Wrong type entirely (e.g. `"hi" + 5`)
`ZeroDivisionError`	Division or modulo by zero
`IndexError`	List/string index out of range
`KeyError`	Dictionary key doesn't exist
`FileNotFoundError`	File or directory doesn't exist
`PermissionError`	No permission to read/write a file
`AttributeError`	Object doesn't have that attribute/method
`NameError`	Variable or function name not defined
`RecursionError`	Too many nested function calls
`StopIteration`	Iterator is exhausted
`OverflowError`	Number too large for a float
`MemoryError`	Not enough memory
`OSError`	General operating system error (parent of `FileNotFoundError`, `PermissionError`, etc.)

13.4 Catching Specific Exceptions

Catch specific exception types so you can respond appropriately to each situation.

```
def safe_divide(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        return None
    except TypeError:
        print(f"Error: Cannot divide {type(a).__name__} by {type(b).__name__}.")
        return None

print(safe_divide(10, 2))     # 5.0
print(safe_divide(10, 0))     # Error: Cannot divide by zero. -> None
print(safe_divide(10, "x"))   # Error: Cannot divide int by str. -> None
```

13.4.1 Catching multiple exceptions in one block

```
try:
    value = int(input("Enter a number: "))
    result = 100 / value
    print(f"100 / {value} = {result}")
except (ValueError, ZeroDivisionError):
    print("Please enter a non-zero integer.")
```

Use a tuple of exception types when the same response is appropriate for multiple errors.

13.4.2 Catching all exceptions with `Exception`

```
try:
    risky_operation()
except Exception as e:
    print(f"Something went wrong: {e}")
```

`Exception` is the base class for almost all built-in exceptions. Catching it handles anything unexpected. The `as e` gives you access to the exception object so you can print its message.

Important: Be specific when you can. Catching `Exception` too broadly can hide bugs — you catch an error you didn't expect and your program silently does the wrong thing.

```python
# Too broad --- catches everything, even bugs you didn't anticipate
try:
    process_data(data)
except Exception:
    pass   # <- never do this silently

# Better --- catch what you expect, let unexpected errors propagate
try:
    process_data(data)
except ValueError as e:
    print(f"Bad data: {e}")
except FileNotFoundError:
    print("Data file missing.")
```

13.5 The `as` Keyword — Accessing the Exception Object

Add `as e` (or any name) after the exception type to capture the exception object itself.

```python
try:
    with open("data.txt", "r") as file:
        content = file.read()
except FileNotFoundError as e:
    print(f"File error: {e}")
    print(f"Error type: {type(e).__name__}")
```

Output:

```
File error: [Errno 2] No such file or directory: 'data.txt'
Error type: FileNotFoundError
```

The exception object's string representation is its message. `type(e).__name__` gives you the class name as a string.

13.6 `else` — Runs Only When No Exception Occurred

The `else` block runs if the `try` block completed without raising any exception.

```python
try:
    number = int(input("Enter a number: "))
```

```
except ValueError:
    print("That's not a number.")
else:
    # Only runs if no ValueError was raised
    print(f"Success! You entered {number}.")
    squared = number ** 2
    print(f"Its square is {squared}.")
```

Sample run — user types `7`:

```
Enter a number: 7
Success! You entered 7.
Its square is 49.
```

Sample run — user types `abc`:

```
Enter a number: abc
That's not a number.
```

Why use `else` instead of just putting the code at the end of `try`? Because any exception inside `else` won't be caught by the `except` blocks above it — which is usually what you want. The `try` block should only contain the risky code, not the code that depends on success.

13.7 `finally` — Always Runs

The `finally` block runs no matter what — whether an exception occurred or not, whether it was caught or not. It's for cleanup code that must always execute.

```
def read_file(filename):
    file = None
    try:
        file = open(filename, "r")
        content = file.read()
        return content
    except FileNotFoundError:
        print(f"File '{filename}' not found.")
        return None
    finally:
        if file:
            file.close()
            print("File closed.")    # always runs
```

With `with open(...)`, you usually don't need `finally` for files — the context manager handles it. But `finally` is essential for:

- Closing database connections
- Releasing locks
- Cleaning up temporary files
- Logging that an operation ended (regardless of success)

```
import time

def timed_operation():
    start = time.time()
    try:
        print("Starting operation...")
        result = some_risky_function()
        return result
    except Exception as e:
        print(f"Operation failed: {e}")
        return None
    finally:
        elapsed = time.time() - start
        print(f"Operation took {elapsed:.3f} seconds.")    # always prints
```

The full structure:

```
try:
    # risky code
except SomeError:
    # handle specific error
except AnotherError as e:
    # handle another error
else:
    # runs only if no exception
finally:
    # always runs
```

You don't need all four every time. Use what the situation calls for.

13.8 `raise` — Raising Your Own Exceptions

You can raise exceptions yourself using the `raise` keyword. This is how you signal that something has gone wrong in your own code.

```
def set_age(age):
    if not isinstance(age, int):
        raise TypeError(f"Age must be an integer, got {type(age).__name__}.")
    if age < 0 or age > 150:
        raise ValueError(f"Age {age} is out of the valid range (0--150).")
```

```
    return age

try:
    set_age(25)     # fine
    set_age(-5)     # raises ValueError
except ValueError as e:
    print(f"Invalid age: {e}")
```

Output:

```
Invalid age: Age -5 is out of the valid range (0--150).
```

Raising exceptions from your own functions is how you enforce rules. Instead of returning `None` and hoping the caller checks, you raise an exception that *forces* the caller to deal with the problem.

13.8.1 Re-raising an exception

Sometimes you want to catch an exception, do something (like log it), and then let it propagate as if you never caught it.

```
import logging

def process_payment(amount):
    try:
        result = charge_card(amount)
        return result
    except Exception as e:
        logging.error(f"Payment failed for amount {amount}: {e}")
        raise   # re-raise the same exception
```

`raise` with no arguments inside an `except` block re-raises the current exception. The caller will still see the exception — you just got a chance to log it first.

13.9 Creating Custom Exceptions

For larger programs, you'll want your own exception classes. This lets callers catch your specific errors separately from generic Python errors.

```
class InsufficientFundsError(Exception):
    """Raised when a withdrawal exceeds the account balance."""

    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
```

```
        super().__init__(
            f"Cannot withdraw ${amount:.2f}. Balance is only ${balance:.2f}."
        )


class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive.")
        self.balance += amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if amount > self.balance:
            raise InsufficientFundsError(self.balance, amount)
        self.balance -= amount
        return amount


account = BankAccount("Alice", 100.0)

try:
    account.withdraw(150.0)
except InsufficientFundsError as e:
    print(f"Transaction declined: {e}")
    print(f"  Balance: ${e.balance:.2f}")
    print(f"  Requested: ${e.amount:.2f}")
except ValueError as e:
    print(f"Invalid amount: {e}")
```

Output:

```
Transaction declined: Cannot withdraw $150.00. Balance is only $100.00.
  Balance: $100.00
  Requested: $150.00
```

Custom exceptions carry extra information (here, `balance` and `amount`) that callers can use to respond intelligently. You'll see this pattern in every serious Python library.

13.10 Exception Hierarchy

Python exceptions form a hierarchy. Every exception ultimately inherits from `BaseException`. Most inherit from `Exception`.

```
BaseException
├── SystemExit            <- sys.exit() raises this
├── KeyboardInterrupt     <- Ctrl+C raises this
└── Exception             <- catch this for normal errors
    ├── ValueError
    ├── TypeError
    ├── OSError
    │   ├── FileNotFoundError
    │   └── PermissionError
    ├── IndexError
    ├── KeyError
    ├── ZeroDivisionError
    ├── AttributeError
    ├── NameError
    └── ... (many more)
```

When you catch a parent class, you catch all its children too:

```
try:
    open("missing.txt")
except OSError:
    print("An OS error occurred.")    # catches FileNotFoundError too
```

This matters when you want to handle a family of errors together without listing every possible subtype.

13.11 Practical Patterns

13.11.1 Pattern 1: Retry loop

Keep trying until the user gives valid input:

```
def get_integer(prompt, min_val=None, max_val=None):
    """Keep asking until the user enters a valid integer in the given range."""
    while True:
        try:
            value = int(input(prompt))
        except ValueError:
            print("Please enter a whole number.")
            continue
```

```
        if min_val is not None and value < min_val:
            print(f"Please enter a number >= {min_val}.")
        elif max_val is not None and value > max_val:
            print(f"Please enter a number <= {max_val}.")
        else:
            return value

age = get_integer("Enter your age: ", min_val=0, max_val=120)
print(f"Your age is {age}.")
```

Sample run:

```
Enter your age: abc
Please enter a whole number.
Enter your age: -5
Please enter a number >= 0.
Enter your age: 200
Please enter a number <= 120.
Enter your age: 28
Your age is 28.
```

13.11.2 Pattern 2: Try multiple approaches

```
def load_config(primary="config.json", fallback="config_default.json"):
    """Try the primary config, fall back to the default if missing."""
    for path in (primary, fallback):
        try:
            with open(path) as f:
                print(f"Loaded config from: {path}")
                return f.read()
        except FileNotFoundError:
            print(f"Not found: {path}")

    raise FileNotFoundError("No config file found.")
```

13.11.3 Pattern 3: Context manager for cleanup

```
from contextlib import suppress

# suppress is a context manager that silently ignores specific exceptions
with suppress(FileNotFoundError):
    import os
    os.remove("temp_file.txt")    # no crash if file doesn't exist
```

`contextlib.suppress` is a clean alternative to a full `try` / `except` / `pass` when you genuinely want to ignore a specific error.

13.12 Best Practices

1. Be specific — catch only what you expect.

```
# Bad
try:
    result = process(data)
except Exception:
    pass

# Good
try:
    result = process(data)
except ValueError as e:
    log_error(e)
    result = default_value
```

2. Never silently swallow exceptions.

```
# Bad --- hides bugs
except Exception:
    pass

# Good --- at minimum, log it
except Exception as e:
    logger.error(f"Unexpected error: {e}")
    raise
```

3. Keep `try` blocks short. Only wrap the line(s) that might fail, not the entire function.

```
# Bad --- too broad
try:
    data = load_file(path)
    processed = transform(data)
    save_result(processed)
except Exception:
    print("Something failed.")

# Good --- specific
try:
    data = load_file(path)
```

```
except FileNotFoundError:
    print(f"File not found: {path}")
    return

processed = transform(data)
save_result(processed)
```

4. Use `finally` for cleanup, not just error handling.

5. Create custom exceptions for your domain logic. Generic `ValueError` is fine for basic validation. For complex business rules, a named exception makes your code self-documenting.

13.13 Putting It All Together: A Robust CSV Loader

```
import csv
from pathlib import Path


class CSVError(Exception):
    """Raised when a CSV file cannot be loaded or parsed."""


def load_students(filepath):
    """
    Load student data from a CSV file.

    Returns a list of dicts with 'name', 'grade', 'city' keys.
    Raises CSVError if the file can't be read or has bad data.
    """
    path = Path(filepath)

    if not path.exists():
        raise CSVError(f"File not found: {filepath}")

    if path.suffix.lower() != ".csv":
        raise CSVError(f"Expected a .csv file, got: {path.suffix}")

    students = []
    try:
        with open(path, "r", encoding="utf-8") as file:
            reader = csv.DictReader(file)
```

```
            required_fields = {"name", "grade", "city"}
            if not required_fields.issubset(reader.fieldnames or set()):
                raise CSVError(
                    f"CSV missing required columns. "
                    f"Expected: {required_fields}, "
                    f"Got: {set(reader.fieldnames or [])}"
                )

            for line_num, row in enumerate(reader, start=2):
                try:
                    grade = float(row["grade"])
                except ValueError:
                    raise CSVError(
                        f"Line {line_num}: invalid grade '{row['grade']}' "
                        f"for student '{row['name']}'."
                    )
                students.append({
                    "name": row["name"].strip(),
                    "grade": grade,
                    "city": row["city"].strip(),
                })

    except PermissionError:
        raise CSVError(f"Permission denied: cannot read {filepath}")

    return students


def print_report(students):
    """Print a formatted student report."""
    print(f"{'Name':<15} {'Grade':>6} {'City':<12}")
    print("-" * 35)
    for s in students:
        print(f"{s['name']:<15} {s['grade']:>6.1f} {s['city']:<12}")
    grades = [s["grade"] for s in students]
    print("-" * 35)
    print(f"{'Average':<15} {sum(grades)/len(grades):>6.1f}")
    print(f"{'Highest':<15} {max(grades):>6.1f}")
    print(f"{'Lowest':<15} {min(grades):>6.1f}")


# Main program
try:
```

```
    students = load_students("students.csv")
    print_report(students)
except CSVError as e:
    print(f"Could not load student data:\n  {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
    raise
```

A real-world pattern: a custom exception, specific error messages at each failure point, short `try` blocks, and a clean separation between error handling and program logic.

13.14 What You Learned in This Chapter

- Exceptions are Python's way of signaling problems — they're not failures, they're information.
- `try` / `except` catches exceptions and lets your program respond instead of crash.
- Catch specific exception types (`ValueError`, `FileNotFoundError`, etc.) rather than bare `Exception`.
- `as e` captures the exception object — use `str(e)` or `type(e).__name__` for details.
- `else` runs only when no exception occurred — for code that depends on success.
- `finally` always runs — for cleanup that must happen regardless of outcome.
- `raise ExceptionType("message")` signals a problem from your own code.
- Bare `raise` inside an `except` block re-raises the current exception.
- Create custom exceptions by subclassing `Exception` — carry domain-specific information.
- Never silently swallow exceptions with `pass` — at minimum, log them.
- Keep `try` blocks short — only wrap the risky line(s).

13.15 What's Next?

You can now write programs that handle failure gracefully. The next chapter takes a different direction — not error handling, but **elegance**.

In Chapter 14 you'll master **comprehensions**: list comprehensions, dictionary comprehensions, set comprehensions, and generator expressions. You've

seen list comprehensions in Chapter 6 and dictionary comprehensions in Chapter 12. Now you'll learn all the forms, understand when to use them vs. loops, and see how they can make your code dramatically shorter and clearer.

Your turn: Write a function called `safe_open_json(filepath)` that tries to open and parse a JSON file (`import json`, then `json.load(file)`). Handle three specific errors: `FileNotFoundError` (file doesn't exist), `json.JSONDecodeError` (file isn't valid JSON), and `PermissionError` (no read access). Return the parsed data on success or `None` on failure, printing a specific message for each error type.

Chapter 14

Chapter 14: Comprehensions — Elegant, Fast Code

You've been writing loops to build lists and dictionaries. They work — but Python has a more elegant way that experienced programmers reach for constantly: **comprehensions**.

A comprehension lets you create a collection in a single, readable line. What takes four lines of loop code becomes one line that reads almost like English. They're faster, shorter, and when written well, clearer than the equivalent loop.

By the end of this chapter you'll use list comprehensions, dictionary comprehensions, set comprehensions, and generator expressions fluently — and you'll know when to use each one.

14.1 List Comprehensions — Deep Dive

You've seen the basic form. Let's go deep.

14.1.1 The pattern

```
[expression for item in iterable]
```

Read it as: "Give me `expression` for each `item` in `iterable`."

```
# The loop way
squares = []
```

```
for n in range(1, 6):
    squares.append(n ** 2)
print(squares)    # [1, 4, 9, 16, 25]

# The comprehension way
squares = [n ** 2 for n in range(1, 6)]
print(squares)    # [1, 4, 9, 16, 25]
```

Same result. Half the lines. And once you're used to reading comprehensions, the single line is actually *clearer* — because the intent is stated in one place.

14.1.2 The expression can be anything

```
names = ["alice", "bob", "carlos", "diana"]

# Capitalize each name
capitalized = [name.title() for name in names]
print(capitalized)    # ['Alice', 'Bob', 'Carlos', 'Diana']

# Length of each name
lengths = [len(name) for name in names]
print(lengths)    # [5, 3, 6, 5]

# Tuple of (name, length) for each
pairs = [(name.title(), len(name)) for name in names]
print(pairs)    # [('Alice', 5), ('Bob', 3), ('Carlos', 6), ('Diana', 5)]
```

14.1.3 Filtering with a condition

Add an `if` clause at the end to filter which items get included:

```
[expression for item in iterable if condition]
```

```
numbers = [1, -3, 7, -2, 4, -8, 5, 0]

# Only positive numbers
positives = [n for n in numbers if n > 0]
print(positives)    # [1, 7, 4, 5]

# Squares of even numbers
even_squares = [n ** 2 for n in range(1, 11) if n % 2 == 0]
print(even_squares)    # [4, 16, 36, 64, 100]

# Words longer than 4 characters
words = ["cat", "elephant", "ant", "hippopotamus", "bee", "giraffe"]
```

```
long_words = [w for w in words if len(w) > 4]
print(long_words)   # ['elephant', 'hippopotamus', 'giraffe']
```

14.1.4 `if/else` inside the expression (not the filter)

You can also use `if/else` *inside* the expression to transform values conditionally. Notice the position — this goes *before* `for`, not after.

```
[value_if_true if condition else value_if_false for item in iterable]
```

```
numbers = [1, -3, 7, -2, 4, -8, 5, 0]

# Replace negatives with 0
clamped = [n if n > 0 else 0 for n in numbers]
print(clamped)   # [1, 0, 7, 0, 4, 0, 5, 0]

# Label each number
labels = ["positive" if n > 0 else "negative" if n < 0 else "zero"
          for n in numbers]
print(labels)
# ['positive', 'negative', 'positive', 'negative', 'positive', 'negative', 'pos
```

This is the inline conditional (also called a ternary expression). The rule: - `if condition` at the **end** = filter (exclude items) - `if/else` before `for` = transform (include all items, change some)

14.1.5 Nested comprehensions

You can nest loops inside a comprehension — useful for flattening or combining sequences.

```
# Flatten a nested list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [n for row in matrix for n in row]
print(flat)   # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Read the nested comprehension left to right, same as nested loops: “for each row in matrix, for each n in row, give me n.”

```
# All pairs (i, j) where i != j, from 1 to 4
pairs = [(i, j) for i in range(1, 5) for j in range(1, 5) if i != j]
print(pairs)
# [(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), ...]

# Transpose a matrix (rows become columns)
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
transposed = [[row[i] for row in matrix] for i in range(3)]
```

```
print(transposed)
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

A word of caution: nested comprehensions get hard to read quickly. If it takes more than a moment to understand, use nested loops instead. Readability beats cleverness.

Your turn: Given a list of sentences (strings), write a comprehension that produces a list of all words across all sentences that are longer than 3 characters. Use nested comprehension: one loop over sentences, one over words in each sentence.

14.2 Dictionary Comprehensions

The pattern:

```
{key_expr: value_expr for item in iterable if condition}
```

```
# Word -> length mapping
words = ["apple", "banana", "cherry", "date"]
word_lengths = {word: len(word) for word in words}
print(word_lengths)
# {'apple': 5, 'banana': 6, 'cherry': 6, 'date': 4}

# Squares dictionary
squares = {n: n**2 for n in range(1, 11)}
print(squares)
# {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}

# Invert a dictionary
original = {"a": 1, "b": 2, "c": 3}
inverted = {v: k for k, v in original.items()}
print(inverted)    # {1: 'a', 2: 'b', 3: 'c'}

# Filter: only passing grades
scores = {"Alice": 88, "Bob": 45, "Carlos": 92, "Diana": 38, "Eve": 76}
passing = {name: score for name, score in scores.items() if score >= 50}
print(passing)    # {'Alice': 88, 'Carlos': 92, 'Eve': 76}

# Transform values
prices_usd = {"apple": 1.20, "banana": 0.50, "cherry": 3.00}
exchange_rate = 0.92
prices_eur = {item: round(price * exchange_rate, 2)
              for item, price in prices_usd.items()}
print(prices_eur)    # {'apple': 1.1, 'banana': 0.46, 'cherry': 2.76}
```

```python
# From two parallel lists
names  = ["Alice", "Bob", "Carlos"]
grades = [88, 72, 95]
grade_book = {name: grade for name, grade in zip(names, grades)}
print(grade_book)   # {'Alice': 88, 'Bob': 72, 'Carlos': 95}
```

`zip()` pairs up two iterables element by element — here it pairs each name with its grade. It's one of the most useful built-in functions for comprehensions.

Your turn: You have a list of file names: `["report.pdf", "data.csv", "image.jpg", "notes.txt", "backup.csv"]`. Create a dictionary where each filename maps to its extension (without the dot). Use a comprehension.

14.3 Set Comprehensions

The pattern:

```
{expression for item in iterable if condition}
```

It looks exactly like a dict comprehension but has no `:` — just a single expression.

```python
# Unique word lengths
words = ["cat", "elephant", "ant", "hippopotamus", "bee", "giraffe", "dog"]
unique_lengths = {len(w) for w in words}
print(unique_lengths)   # {3, 7, 4, 12}  (order not guaranteed)

# Unique first letters
names = ["Alice", "Bob", "Anna", "Carlos", "Beth", "Alex"]
first_letters = {name[0] for name in names}
print(first_letters)   # {'A', 'B', 'C'}

# Squares of odd numbers as a set
odd_squares = {n**2 for n in range(1, 11) if n % 2 != 0}
print(odd_squares)   # {1, 9, 25, 49, 81}
```

Use a set comprehension when you want unique values and don't care about order. It's like `set(list_comprehension)` but cleaner.

14.4 Generator Expressions — Lazy Evaluation

A **generator expression** looks like a list comprehension but with parentheses instead of brackets. The crucial difference: it doesn't create the entire list

in memory. Instead it produces values one at a time, on demand.

```
# List comprehension --- creates the entire list in memory NOW
squares_list = [n**2 for n in range(1, 1_000_001)]   # 1 million items in RAM

# Generator expression --- creates nothing yet; produces values as needed
squares_gen = (n**2 for n in range(1, 1_000_001))    # barely any memory used
```

You iterate over a generator with `for`, `next()`, or by passing it to a function:

```
gen = (n**2 for n in range(1, 6))

print(next(gen))    # 1
print(next(gen))    # 4
print(next(gen))    # 9
# generator remembers where it left off

# Sum without building the list
total = sum(n**2 for n in range(1, 1_000_001))
print(total)    # 333333833333500000
```

Notice: `sum(n**2 for n in range(...))` — no double brackets needed when a generator expression is the only argument to a function. Python accepts the generator directly.

```
# Find the first even square over 50
gen = (n**2 for n in range(1, 100))
result = next(x for x in gen if x > 50 and x % 2 == 0)
print(result)    # 64
```

14.4.1 When to use generators vs list comprehensions

Situation	Use
You need to iterate once and move on	Generator expression
You'll iterate multiple times	List comprehension
You'll index into the result (`result[3]`)	List comprehension
You need `len()`	List comprehension
Processing millions of items	Generator expression
Passing to `sum()`, `max()`, `min()`, `any()`, `all()`	Generator expression

The golden rule: if you only need to go through the values once, use a generator. If you need to use them multiple times or by index, use a list.

```
# any() and all() with generators --- short-circuit evaluation
numbers = [2, 4, 6, 7, 8, 10]

# any() stops as soon as it finds one True
has_odd = any(n % 2 != 0 for n in numbers)
print(has_odd)    # True  (stops at 7)

# all() stops as soon as it finds one False
all_even = all(n % 2 == 0 for n in numbers)
print(all_even)  # False  (stops at 7)
```

`any()` and `all()` with generator expressions are incredibly efficient — they stop the moment the answer is clear, without processing the rest.

14.5 Comprehensions vs Loops — When to Use Which

Comprehensions are not always the right tool. Here's how to decide.

Use a comprehension when: - You're creating a new list/dict/set by transforming or filtering an existing iterable. - The logic fits comfortably on one or two lines. - The expression is clear without needing a comment.

Use a loop when: - You have complex multi-step logic per item. - You need `break` or `continue`. - The result involves more than one collection being updated. - You're doing something for its *side effects* (printing, writing to a file) — not building a value.

```
# Good use of comprehension --- simple transformation
emails = [user["email"].lower() for user in users if user["active"]]

# Bad use --- too complex, hard to read
results = [
    (item["id"], item["price"] * (1 - item.get("discount", 0)) * tax_rate)
    for item in inventory
    if item["in_stock"] and item["price"] * (1 - item.get("discount", 0)) > 10
]

# Better as a loop --- readable, debuggable
results = []
for item in inventory:
    if not item["in_stock"]:
        continue
    discount = item.get("discount", 0)
```

```
    price = item["price"] * (1 - discount)
    if price > 10:
        results.append((item["id"], price * tax_rate))
```

The second comprehension is technically valid but takes real effort to parse. The loop is longer but instantly understandable. Choose readability.

14.6 `zip()`, `enumerate()`, and `map()` in Comprehensions

These built-ins pair perfectly with comprehensions.

14.6.1 `zip()` — Pair up two or more iterables

```
names  = ["Alice", "Bob", "Carlos"]
scores = [88, 72, 95]
cities = ["London", "Paris", "Madrid"]

# Three parallel lists -> list of tuples
combined = [(name, score, city) for name, score, city in zip(names, scores, cities
print(combined)
# [('Alice', 88, 'London'), ('Bob', 72, 'Paris'), ('Carlos', 95, 'Madrid')]

# Directly into a dict
grade_book = {name: score for name, score in zip(names, scores)}
print(grade_book)    # {'Alice': 88, 'Bob': 72, 'Carlos': 95}
```

14.6.2 `enumerate()` — Index + value

```
fruits = ["apple", "banana", "cherry"]

indexed = {i: fruit for i, fruit in enumerate(fruits, start=1)}
print(indexed)    # {1: 'apple', 2: 'banana', 3: 'cherry'}
```

14.6.3 `map()` — Apply a function to every item

```
words = ["hello", "world", "python"]
upper = list(map(str.upper, words))
print(upper)    # ['HELLO', 'WORLD', 'PYTHON']

# Equivalent comprehension (usually preferred for readability)
upper = [w.upper() for w in words]
```

`map()` returns an iterator (like a generator). Wrap it in `list()` to get a list. List comprehensions are usually clearer — but `map()` is worth knowing because you'll see it in real code.

14.7 Putting It All Together: A Data Pipeline

Here's a realistic data processing pipeline using comprehensions throughout:

```
# Raw data --- imagine this came from a CSV or API
raw_students = [
    {"name": "  Alice  ", "grade": "88", "city": "London", "active": True},
    {"name": "Bob",       "grade": "45", "city": "Paris",  "active": True},
    {"name": "Carlos",    "grade": "92", "city": "Madrid", "active": False},
    {"name": "  Diana ",  "grade": "N/A","city": "Tokyo",  "active": True},
    {"name": "Eve",       "grade": "76", "city": "Sydney", "active": True},
]


def parse_grade(value):
    """Parse a grade string to float, return None if invalid."""
    try:
        return float(value)
    except ValueError:
        return None


# Step 1: Clean and parse --- only active students with valid grades
students = [
    {
        "name":  s["name"].strip().title(),
        "grade": parse_grade(s["grade"]),
        "city":  s["city"],
    }
    for s in raw_students
    if s["active"] and parse_grade(s["grade"]) is not None
]

print("Active students with valid grades:")
for s in students:
    print(f"  {s['name']}: {s['grade']}")

# Step 2: Build a grade lookup dictionary
```

```python
grade_lookup = {s["name"]: s["grade"] for s in students}

# Step 3: Classify grades
classifications = {
    s["name"]: "Pass" if s["grade"] >= 50 else "Fail"
    for s in students
}

# Step 4: Get unique cities
cities = {s["city"] for s in students}

# Step 5: Calculate stats using generator expressions
grades = [s["grade"] for s in students]
average = sum(grades) / len(grades)
passing_count = sum(1 for g in grades if g >= 50)

print(f"\nGrade lookup: {grade_lookup}")
print(f"Classifications: {classifications}")
print(f"Cities represented: {cities}")
print(f"Average grade: {average:.1f}")
print(f"Passing: {passing_count}/{len(students)}")
```

Output:

```
Active students with valid grades:
  Alice: 88.0
  Bob: 45.0
  Eve: 76.0

Grade lookup: {'Alice': 88.0, 'Bob': 45.0, 'Eve': 76.0}
Classifications: {'Alice': 'Pass', 'Bob': 'Fail', 'Eve': 'Pass'}
Cities represented: {'London', 'Paris', 'Sydney'}
Average grade: 69.7
Passing: 2/3
```

List comprehension, dict comprehension, set comprehension, and generator expression — all in a twenty-five line pipeline that cleans, filters, transforms, and summarizes real messy data.

14.8 Quick Reference

```python
# List comprehension
result = [expr for item in iterable]
result = [expr for item in iterable if condition]
```

```
result = [a if cond else b for item in iterable]

# Dict comprehension
result = {key: value for item in iterable}
result = {key: value for item in iterable if condition}
result = {k: v for k, v in some_dict.items()}

# Set comprehension
result = {expr for item in iterable}
result = {expr for item in iterable if condition}

# Generator expression
result = (expr for item in iterable)
result = (expr for item in iterable if condition)

# Useful combinations
sum(expr for item in iterable)
max(expr for item in iterable)
any(condition for item in iterable)
all(condition for item in iterable)
list(map(func, iterable))
{k: v for k, v in zip(keys, values)}
```

14.9 What You Learned in This Chapter

- A **list comprehension** creates a list: `[expr for item in iterable if condition]`.
- Filter with `if condition` at the end; transform conditionally with `if/else` before `for`.
- Nested comprehensions handle nested loops — but readability has limits.
- A **dictionary comprehension** creates a dict: `{key: value for item in iterable}`.
- A **set comprehension** creates a set: `{expr for item in iterable}`.
- A **generator expression** uses `()` and produces values lazily — no memory overhead.
- Use generators with `sum()`, `any()`, `all()`, `max()`, `min()` — they stop early when possible.
- `zip()` pairs up iterables; perfect for building dicts from parallel lists.
- Comprehensions are for building values. Loops are for side effects and complex logic.
- If the comprehension takes a moment to parse, use a loop.

14.10 What's Next?

Your code is becoming more expressive and efficient. Now it's time to learn how to *organize* it.

In Chapter 15 you'll learn about **modules and packages** — how Python's import system works, how to use the standard library, how to install third-party packages with `pip`, and how to organize your own code into modules. This is the foundation for writing programs that scale beyond a single file.

Your turn: You have a list of dictionaries, each representing a product: `{"name": "...", "price": ..., "category": "..."}`. Using only comprehensions: 1. Build a list of product names where price > 20. 2. Build a dict mapping name -> price for all products. 3. Build a set of all unique categories. 4. Calculate the average price using a generator expression. 5. Find the most expensive product name using `max()` with a generator.

Chapter 15

Chapter 15: Modules and Packages

Every program you've written so far lives in one file. That works for small programs. But as programs grow — hundreds of lines, then thousands — one file becomes impossible to navigate, understand, or maintain.

Python's answer is **modules** and **packages**: a way to split code across multiple files, import what you need, and build on the work of millions of other programmers through the standard library and third-party packages.

This chapter teaches you how Python's import system works, how to write your own modules, how to structure a package, how to install third-party libraries with `pip`, and how to manage environments so different projects don't interfere with each other.

15.1 What is a Module?

A module is simply a Python file. Any `.py` file you write is a module. Its name is the filename without the `.py` extension.

When you write:

```
import math
```

Python finds a file called `math.py` somewhere on your system, runs it, and makes everything defined in it available to you as `math.something`.

You've been using modules since Chapter 3 (`import math`) and Chapter 9 (`from`

`datetime import datetime`). Now you'll understand exactly what's happening.

15.2 Three Ways to Import

15.2.1 `import module` — Import the whole module

```
import math

print(math.pi)          # 3.141592653589793
print(math.sqrt(16))    # 4.0
print(math.ceil(4.2))   # 5
```

You access everything through the module name. This keeps namespaces clean — `math.sqrt` is unambiguous.

15.2.2 `from module import name` — Import specific names

```
from math import pi, sqrt, ceil

print(pi)       # 3.141592653589793
print(sqrt(16)) # 4.0
print(ceil(4.2))# 5
```

The names are imported directly into your namespace — no `math.` prefix needed. Useful when you use the same function constantly and the shorter name is clear.

Be careful: if you import something that clashes with a name you already defined, the import wins (or your definition wins, depending on order). Keep imports at the top of the file where they're visible.

15.2.3 `import module as alias` — Import with a shorter name

```
import numpy as np          # convention in data science
import pandas as pd         # convention
import matplotlib.pyplot as plt

# Now use the alias
arr = np.array([1, 2, 3])
```

Aliases are used throughout the Python ecosystem for popular libraries. `np`, `pd`, `plt` are so universal that code using them is immediately recognizable to any data scientist.

You can alias specific imports too:

```
from collections import defaultdict as dd
from datetime import datetime as dt

now = dt.now()
```

15.2.4 `from module import *` — Import everything (avoid this)

```
from math import *    # imports everything math defines

print(sqrt(16))    # works, but...
```

This pollutes your namespace with dozens of names you may not expect. If two modules both define `sqrt`, one silently overwrites the other. Use it only in interactive sessions for quick exploration, never in production code.

15.3 How Python Finds Modules

When you write `import something`, Python searches these locations in order:

1. **The current directory** — your script's folder
2. **`PYTHONPATH`** — an environment variable listing extra directories
3. **The standard library** — Python's built-in modules
4. **Site-packages** — where `pip` installs third-party packages

You can see the full search path:

```
import sys
print(sys.path)
```

This list of directories is where Python looks. If your module isn't in any of them, you get a `ModuleNotFoundError`.

15.4 Writing Your Own Module

Any `.py` file is a module. Create a file called `greetings.py`:

```
# greetings.py

DEFAULT_GREETING = "Hello"
```

```
def greet(name, greeting=DEFAULT_GREETING):
    """Return a greeting string."""
    return f"{greeting}, {name}!"


def farewell(name):
    """Return a farewell string."""
    return f"Goodbye, {name}. See you soon!"


def greet_all(names, greeting=DEFAULT_GREETING):
    """Return a list of greeting strings."""
    return [greet(name, greeting) for name in names]
```

Now in another file in the same folder, import it:

```
# main.py

import greetings

print(greetings.greet("Alice"))
print(greetings.farewell("Bob"))
print(greetings.DEFAULT_GREETING)

messages = greetings.greet_all(["Carlos", "Diana", "Eve"])
for msg in messages:
    print(msg)
```

Output:

```
Hello, Alice!
Goodbye, Bob. See you soon!
Hello
Hello, Carlos!
Hello, Diana!
Hello, Eve!
```

That's it. A module is just a file with functions, variables, and classes. Import it by name and use what's inside.

15.5 `if __name__ == "__main__":`

This is one of the most important Python idioms. Every module has a special variable called `__name__`. When a file is *run directly*, `__name__` equals `"__main__"`. When a file is *imported*, `__name__` equals the module's name.

```python
# greetings.py

DEFAULT_GREETING = "Hello"


def greet(name, greeting=DEFAULT_GREETING):
    return f"{greeting}, {name}!"


def farewell(name):
    return f"Goodbye, {name}. See you soon!"


# This block only runs when the file is executed directly
# It does NOT run when the module is imported
if __name__ == "__main__":
    print("Running greetings module directly.")
    print(greet("World"))
    print(farewell("World"))
```

Run `python greetings.py` directly -> the `if __name__` block runs. Import it from `main.py` -> it doesn't run.

This pattern lets you write modules that are also runnable scripts. Put your tests, demos, or CLI entry point inside `if __name__ == "__main__":` and the module stays importable without side effects.

Your turn: Create a module called `calculator.py` with four functions: `add`, `subtract`, `multiply`, `divide`. Add a `if __name__ == "__main__":` block that tests each function and prints the results. Import the module from a second file and use all four functions.

15.6 What is a Package?

A **package** is a folder containing multiple modules, plus a special file called `__init__.py`.

```
myproject/
├── main.py
└── utils/
    ├── __init__.py
    ├── math_tools.py
    ├── string_tools.py
    └── file_tools.py
```

The `__init__.py` file tells Python "this folder is a package." It can be empty, or it can define what gets exported when someone does `import utils`.

```
# utils/math_tools.py
def square(n):
    return n ** 2

def cube(n):
    return n ** 3

# utils/string_tools.py
def titlecase(text):
    return text.title()

def word_count(text):
    return len(text.split())

# main.py
from utils import math_tools
from utils.string_tools import titlecase, word_count

print(math_tools.square(5))        # 25
print(math_tools.cube(3))          # 27
print(titlecase("hello world"))    # Hello World
print(word_count("one two three")) # 3
```

15.6.1 Controlling what `import utils` exports

In `utils/__init__.py`, you can import the things you want to expose at the package level:

```
# utils/__init__.py
from .math_tools import square, cube
from .string_tools import titlecase, word_count
```

Now users can do:

```
from utils import square, titlecase
```

Without needing to know which submodule each function lives in. This is how libraries like `requests` and `pandas` work — you import from the top-level package, not the internal submodules.

15.7 The Standard Library

Python ships with a massive standard library — hundreds of modules covering almost everything you'd need. You don't need to install anything; they're included with Python.

Here's a taste of what's available:

Module	What it does
`math`	Mathematical functions and constants
`os`	Operating system interface — files, paths, environment
`sys`	Python interpreter — version, path, argv
`datetime`	Dates, times, timedeltas
`random`	Random numbers and choices
`json`	Read and write JSON
`csv`	Read and write CSV
`re`	Regular expressions
`pathlib`	Modern file path handling
`collections`	Counter, defaultdict, deque, OrderedDict, namedtuple
`itertools`	Powerful iteration tools
`functools`	Higher-order functions — lru_cache, reduce, partial
`string`	String constants and Template
`time`	Time access and conversions
`hashlib`	Secure hash functions
`urllib`	URL handling and HTTP requests (basic)
`sqlite3`	SQLite database
`logging`	Logging framework
`unittest`	Testing framework
`argparse`	Command-line argument parsing
`threading`	Thread-based parallelism
`multiprocessing`	Process-based parallelism
`asyncio`	Async I/O
`abc`	Abstract base classes
`dataclasses`	Dataclass decorator
`typing`	Type hints
`contextlib`	Context manager utilities
`copy`	Shallow and deep copy
`io`	In-memory file-like objects
`struct`	Binary data packing/unpacking
`socket`	Low-level networking
`http`	HTTP server and client
`email`	Email composition and parsing
`zipfile`	ZIP archive reading/writing
`tarfile`	TAR archive reading/writing

Module	What it does
`shutil`	High-level file operations
`tempfile`	Temporary files and directories
`subprocess`	Run shell commands
`signal`	Signal handling
`traceback`	Print and inspect tracebacks

Chapter 16 covers the most useful ones in detail.

15.8 Installing Third-Party Packages with `pip`

The standard library is huge, but the real Python ecosystem is the **Python Package Index (PyPI)** — over 500,000 packages built by the community. `pip` is the tool that installs them.

15.8.1 Installing a package

Open your terminal and run:

```
pip install requests
```

After installation, import it normally:

```
import requests

response = requests.get("https://api.github.com")
print(response.status_code)    # 200
print(response.json()["current_user_url"])
```

15.8.2 Installing a specific version

```
pip install requests==2.31.0
```

15.8.3 Upgrading a package

```
pip install --upgrade requests
```

15.8.4 Uninstalling a package

```
pip uninstall requests
```

15.8.5 Listing installed packages

```
pip list
```

15.8.6 Seeing where a package is installed

```
pip show requests
```

15.9 `requirements.txt` — Sharing Your Environment

When you share your project with someone else (or deploy it to a server), they need to know which packages to install and which versions you used. The standard way is a `requirements.txt` file.

15.9.1 Creating it

```
pip freeze > requirements.txt
```

This writes all installed packages and their exact versions to `requirements.txt`:

```
certifi==2024.2.2
charset-normalizer==3.3.2
idna==3.6
requests==2.31.0
urllib3==2.2.0
```

15.9.2 Installing from it

```
pip install -r requirements.txt
```

This installs every package listed. One command restores your entire environment.

15.10 Virtual Environments — Isolating Projects

Here's a problem: Project A needs `requests==2.25.0`. Project B needs `requests==2.31.0`. If you install both, one overwrites the other. Your system Python can only hold one version of any package at a time.

A **virtual environment** solves this. It's an isolated Python installation for one project — its own copy of Python, its own packages, completely separate from your system and every other project.

15.10.1 Creating a virtual environment

```
python -m venv venv
```

This creates a folder called `venv` in your current directory containing a full isolated Python installation.

15.10.2 Activating it

Windows:

```
venv\Scripts\activate
```

Mac/Linux:

```
source venv/bin/activate
```

After activation, your terminal prompt changes to show `(venv)`. Every `pip install` now installs into this environment only.

15.10.3 Using it

```
(venv) pip install requests flask pandas
(venv) pip freeze > requirements.txt
(venv) python main.py
```

15.10.4 Deactivating it

```
deactivate
```

15.10.5 The standard project layout

```
my_project/
├── venv/              <- virtual environment (NEVER commit to git)
├── .gitignore           <- include venv/ here
├── requirements.txt     <- commit this
├── README.md
├── main.py
└── utils/
    ├── __init__.py
    └── helpers.py
```

Always add `venv/` to `.gitignore` — it's hundreds of megabytes and specific to your machine. Commit `requirements.txt` instead so anyone who clones

your project can recreate the environment with `pip install -r requirements.txt`.

15.11 Relative Imports

Inside a package, you can use **relative imports** to import from sibling modules without spelling out the full package path.

```
# utils/math_tools.py

# Absolute import --- works from anywhere
from utils.string_tools import titlecase

# Relative import --- works only inside the package
from .string_tools import titlecase   # same package
from ..other_package import something  # parent package
```

The `.` means "current package." `..` means "parent package." Relative imports make refactoring easier — if you rename the package, relative imports inside it don't need to change.

15.12 `__init__.py` in Depth

In Python 3.3+, packages don't actually *require* `__init__.py` — Python will find the folder as a "namespace package" without it. But it's still best practice to include one because:

1. It signals clearly that this folder is a Python package.
2. It lets you control what the package exports.
3. It lets you run initialization code when the package is imported.

```
# utils/__init__.py

# Package version
__version__ = "1.0.0"
__author__ = "Alice Smith"

# Expose the most important things at the top level
from .math_tools import square, cube
from .string_tools import titlecase, word_count
```

```
# Run once when the package is imported
print(f"utils package v{__version__} loaded.")
```

15.13 Putting It All Together: A Utility Package

Here's a complete mini-package structure:

```
weather_app/
├── main.py
├── requirements.txt
└── weather/
    ├── __init__.py
    ├── fetcher.py
    ├── parser.py
    └── formatter.py
```

```
# weather/fetcher.py
import requests

def fetch_weather(city, api_key):
    """Fetch current weather data for a city."""
    url = f"https://api.openweathermap.org/data/2.5/weather"
    params = {"q": city, "appid": api_key, "units": "metric"}
    response = requests.get(url, params=params, timeout=10)
    response.raise_for_status()
    return response.json()
```

```
# weather/parser.py
def parse_weather(data):
    """Extract key fields from raw API response."""
    return {
        "city":        data["name"],
        "country":     data["sys"]["country"],
        "temp":        data["main"]["temp"],
        "feels_like":  data["main"]["feels_like"],
        "humidity":    data["main"]["humidity"],
        "description": data["weather"][0]["description"].title(),
        "wind_speed":  data["wind"]["speed"],
    }
```

```
# weather/formatter.py
def format_report(weather):
    """Format parsed weather data into a readable string."""
```

```
    return (
        f"Weather in {weather['city']}, {weather['country']}\n"
        f"  {weather['description']}\n"
        f"  Temperature: {weather['temp']}°C "
        f"(feels like {weather['feels_like']}°C)\n"
        f"  Humidity:    {weather['humidity']}%\n"
        f"  Wind speed:  {weather['wind_speed']} m/s"
    )
```

```
# weather/__init__.py
from .fetcher import fetch_weather
from .parser import parse_weather
from .formatter import format_report

__version__ = "1.0.0"
```

```
# main.py
from weather import fetch_weather, parse_weather, format_report

API_KEY = "your_api_key_here"

city = input("Enter city name: ")
try:
    raw = fetch_weather(city, API_KEY)
    parsed = parse_weather(raw)
    print(format_report(parsed))
except Exception as e:
    print(f"Could not get weather: {e}")
```

Each file does one job. `main.py` orchestrates. The package is importable by anyone. This is how real Python applications are structured.

15.14 What You Learned in This Chapter

- A **module** is any `.py` file. Import it with `import name`.
- Three import styles: `import module`, `from module import name`, `import module as alias`.
- `from module import *` pollutes your namespace — avoid it in production code.
- Python searches for modules in: current directory -> `PYTHONPATH` -> standard library -> site-packages.
- `if __name__ == "__main__":` runs only when a file is executed directly, not when imported.
- A **package** is a folder with `__init__.py` containing multiple modules.

- `__init__.py` controls what the package exposes and can run initialization code.
- Relative imports use `.` (current package) and `..` (parent package).
- The **standard library** has hundreds of modules — check it before installing anything.
- `pip install package` installs from PyPI. `pip freeze > requirements.txt` saves your dependencies.
- **Virtual environments** (`python -m venv venv`) isolate each project's packages.
- Always add `venv/` to `.gitignore`. Commit `requirements.txt` instead.

15.15 What's Next?

Now that you know how to import modules, Chapter 16 gives you a guided tour of the most useful ones in Python's standard library: `os`, `sys`, `datetime`, `random`, `json`, `re`, `itertools`, `functools`, and `pathlib`. You'll use these in almost every real project you write.

Your turn: Create a package called `converters` with three modules: - `temperature.py` — functions for Celsius ☐ Fahrenheit ☐ Kelvin - `distance.py` — functions for km ☐ miles, m ☐ feet - `currency.py` — a simple function that multiplies by an exchange rate

Create an `__init__.py` that exposes the most important functions at the package level. Write a `main.py` that imports from the package and lets the user pick a conversion type, enter a value, and see the result.

Chapter 16

Chapter 16: Useful Standard Library Modules

Python ships with a massive standard library — "batteries included" is the official motto. You don't need to install anything for these; they're part of every Python installation.

This chapter is a practical tour of the modules you'll reach for most often. By the end you'll know what each one does, when to use it, and the key functions to remember. Think of this chapter as a toolbox reference — you don't need to memorize everything, but knowing what tools exist means you'll never reinvent something Python already built.

16.1 `os` and `os.path` — The File System

The `os` module lets you interact with the operating system: create and delete files and directories, read environment variables, navigate paths, and run system commands.

```
import os

# Current working directory
print(os.getcwd())          # C:\Users\Alice\projects

# List files in a directory
print(os.listdir("."))       # ['main.py', 'data', 'README.md']
print(os.listdir("data"))    # ['sales.csv', 'users.csv']
```

```python
# Create a directory (raises FileExistsError if it exists)
os.mkdir("output")

# Create nested directories
os.makedirs("output/reports/2026", exist_ok=True)   # no error if exists

# Delete a file
os.remove("temp.txt")

# Delete an empty directory
os.rmdir("old_folder")

# Delete a directory and all its contents (use carefully!)
import shutil
shutil.rmtree("old_folder")

# Rename or move
os.rename("old_name.txt", "new_name.txt")

# Check if a path exists
print(os.path.exists("data.csv"))    # True or False

# Is it a file? A directory?
print(os.path.isfile("data.csv"))    # True
print(os.path.isdir("output"))       # True

# Join paths safely (handles / vs \ automatically)
path = os.path.join("data", "2026", "sales.csv")
print(path)   # data/2026/sales.csv  (or data\2026\sales.csv on Windows)

# Split a path into directory and filename
folder, filename = os.path.split("/home/alice/data/sales.csv")
print(folder)    # /home/alice/data
print(filename)  # sales.csv

# Get just the extension
name, ext = os.path.splitext("report.pdf")
print(name)   # report
print(ext)    # .pdf

# Absolute path
print(os.path.abspath("data.csv"))   # full path from root
```

16.1.1 Environment variables

```
import os

# Read an environment variable
api_key = os.environ.get("API_KEY", "not_set")
print(api_key)

# Set one for the current process
os.environ["MY_VAR"] = "hello"

# All environment variables
for key, value in os.environ.items():
    print(f"{key}={value}")
```

Environment variables are the standard way to keep secrets (API keys, passwords) out of your code. Never hardcode credentials — read them from the environment.

16.2 `pathlib` — Modern File Paths

`pathlib` is the modern, object-oriented replacement for `os.path`. Use it for all new code.

```
from pathlib import Path

# Create a path object
p = Path("data/sales.csv")

print(p.name)        # sales.csv
print(p.stem)        # sales
print(p.suffix)      # .csv
print(p.parent)      # data
print(p.is_file())   # True/False
print(p.exists())    # True/False

# Join paths with /
base = Path("data")
full = base / "2026" / "sales.csv"
print(full)    # data/2026/sales.csv

# Read and write files
text = full.read_text(encoding="utf-8")
```

```
full.write_text("new content", encoding="utf-8")

# Create directories
Path("output/reports").mkdir(parents=True, exist_ok=True)

# List files matching a pattern
for csv_file in Path("data").glob("*.csv"):
    print(csv_file)

# Recursive glob
for py_file in Path(".").rglob("*.py"):
    print(py_file)

# Absolute path
print(p.resolve())    # full absolute path

# Home directory
print(Path.home())    # /home/alice or C:\Users\Alice
```

Why `pathlib` over `os.path`? The `/` operator for joining, readable attributes (`.name`, `.stem`, `.suffix`), built-in read/write, and `glob()` built in. It's cleaner in every way.

16.3 `sys` — The Python Interpreter

```
import sys

# Python version
print(sys.version)        # 3.12.4 (main, Jun 6 2024, ...)
print(sys.version_info)   # sys.version_info(major=3, minor=12, ...)

# Minimum version check
if sys.version_info < (3, 10):
    print("This script requires Python 3.10 or higher.")
    sys.exit(1)

# Command-line arguments
# If you run: python script.py hello world
print(sys.argv)         # ['script.py', 'hello', 'world']
print(sys.argv[0])      # script.py  (the script name)
print(sys.argv[1:])     # ['hello', 'world']  (the arguments)
```

```
# The module search path
print(sys.path)

# Exit the program with a status code
# 0 = success, anything else = error
sys.exit(0)

# Standard streams
sys.stdout.write("Hello\n")    # same as print()
sys.stderr.write("Error!\n")   # writes to stderr

# Size of an object in bytes
print(sys.getsizeof([]))        # 56
print(sys.getsizeof([1,2,3]))   # 88

# Platform
print(sys.platform)    # 'win32', 'darwin', 'linux'
```

A simple command-line argument example:

```
import sys

if len(sys.argv) != 3:
    print(f"Usage: python {sys.argv[0]} <name> <age>")
    sys.exit(1)

name = sys.argv[1]
age = int(sys.argv[2])
print(f"Hello, {name}! You are {age} years old.")
```

Run it: `python greet.py Alice 30` -> `Hello, Alice! You are 30 years old.`

16.4 `datetime` — Dates and Times

```
from datetime import datetime, date, time, timedelta

# Current date and time
now = datetime.now()
print(now)                          # 2026-03-09 14:30:45.123456
print(now.year)                     # 2026
print(now.month)                    # 3
print(now.day)                      # 9
print(now.hour)                     # 14
```

```
print(now.minute)                    # 30
print(now.second)                    # 45

# Create a specific datetime
birthday = datetime(1990, 7, 15, 8, 30)
print(birthday)    # 1990-07-15 08:30:00

# Just a date
today = date.today()
print(today)          # 2026-03-09
print(today.weekday())  # 0=Monday ... 6=Sunday

# Formatting --- strftime (string from time)
print(now.strftime("%Y-%m-%d"))              # 2026-03-09
print(now.strftime("%d/%m/%Y %H:%M"))        # 09/03/2026 14:30
print(now.strftime("%B %d, %Y"))             # March 09, 2026
print(now.strftime("%A, %B %d"))             # Monday, March 09

# Parsing --- strptime (string parse time)
date_str = "2026-03-09"
parsed = datetime.strptime(date_str, "%Y-%m-%d")
print(parsed)    # 2026-03-09 00:00:00

# Arithmetic with timedelta
tomorrow  = today + timedelta(days=1)
last_week = today - timedelta(weeks=1)
in_90_days = today + timedelta(days=90)

print(f"Tomorrow:    {tomorrow}")
print(f"Last week:   {last_week}")
print(f"In 90 days:  {in_90_days}")

# Difference between two dates
start = date(2026, 1, 1)
end   = date(2026, 12, 31)
diff  = end - start
print(f"Days in 2026: {diff.days}")    # 364

# Age calculator
def calculate_age(birthdate):
    today = date.today()
    years = today.year - birthdate.year
    if (today.month, today.day) < (birthdate.month, birthdate.day):
        years -= 1
```

```
    return years

print(calculate_age(date(1990, 7, 15)))    # e.g. 35
```

16.4.1 Common `strftime` format codes

Code	Meaning	Example
`%Y`	4-digit year	2026
`%m`	Zero-padded month	03
`%d`	Zero-padded day	09
`%H`	Hour (24h)	14
`%M`	Minute	30
`%S`	Second	45
`%A`	Full weekday name	Monday
`%B`	Full month name	March
`%I`	Hour (12h)	02
`%p`	AM/PM	PM

16.5 `random` — Randomness

```
import random

# Random float between 0.0 and 1.0
print(random.random())            # 0.7342...

# Random float in a range
print(random.uniform(1.0, 10.0)) # 7.234...

# Random integer (inclusive on both ends)
print(random.randint(1, 6))       # die roll: 1, 2, 3, 4, 5, or 6

# Random integer from range (exclusive end, like range())
print(random.randrange(0, 100, 5))  # 0, 5, 10, ..., 95

# Choose one random item from a sequence
fruits = ["apple", "banana", "cherry", "date"]
print(random.choice(fruits))      # one random fruit

# Choose multiple items (without replacement)
print(random.sample(fruits, 2))  # 2 unique random fruits
```

```
# Choose multiple items (with replacement)
print(random.choices(fruits, k=3))  # 3 items, may repeat

# Shuffle a list in place
deck = list(range(1, 53))
random.shuffle(deck)
print(deck[:5])   # first 5 cards after shuffle

# Reproducible randomness --- set a seed
random.seed(42)
print(random.randint(1, 100))   # always the same number when seed=42
```

Seeding is essential for testing — same seed, same sequence, reproducible results every time.

16.6 `json` — Reading and Writing JSON

JSON (JavaScript Object Notation) is the most common data format for APIs and configuration files.

```
import json

# Python dict -> JSON string
data = {
    "name": "Alice",
    "age": 30,
    "scores": [88, 92, 76],
    "active": True,
    "address": None
}

json_string = json.dumps(data)
print(json_string)
# {"name": "Alice", "age": 30, "scores": [88, 92, 76], "active": true, "address":

# Pretty-printed
pretty = json.dumps(data, indent=2)
print(pretty)

# JSON string -> Python dict
back = json.loads(json_string)
print(back["name"])   # Alice
```

```
print(type(back))      # <class 'dict'>

# Write JSON to a file
with open("data.json", "w") as f:
    json.dump(data, f, indent=2)

# Read JSON from a file
with open("data.json", "r") as f:
    loaded = json.load(f)

print(loaded["scores"])    # [88, 92, 76]
```

16.6.1 JSON ☐ Python type mapping

JSON	Python
object `{}`	`dict`
array `[]`	`list`
string `""`	`str`
number	`int` or `float`
`true` / `false`	`True` / `False`
`null`	`None`

16.7 `re` — Regular Expressions

Regular expressions let you search, match, and transform text using patterns. They're concise but powerful — one pattern can match thousands of variations.

```
import re

text = "Contact us at support@example.com or sales@company.org"

# Search for one match
match = re.search(r"\b\w+@\w+\.\w+\b", text)
if match:
    print(match.group())    # support@example.com

# Find all matches
emails = re.findall(r"\b\w+@\w+\.\w+\b", text)
print(emails)    # ['support@example.com', 'sales@company.org']

# Replace matches
```

```
cleaned = re.sub(r"\b\w+@\w+\.\w+\b", "[EMAIL]", text)
print(cleaned)   # Contact us at [EMAIL] or [EMAIL]

# Match from the start of a string
phone = "555-1234"
if re.match(r"\d{3}-\d{4}", phone):
    print("Valid phone format")

# Split on a pattern
parts = re.split(r"[,;\s]+", "one, two;  three    four")
print(parts)   # ['one', 'two', 'three', 'four']

# Groups --- capture parts of a match
date_text = "Today is 2026-03-09."
match = re.search(r"(\d{4})-(\d{2})-(\d{2})", date_text)
if match:
    year, month, day = match.groups()
    print(f"Year: {year}, Month: {month}, Day: {day}")
```

16.7.1 Compile for repeated use

```
# Compile a pattern once, use it many times (faster in loops)
email_pattern = re.compile(r"\b[\w.+-]+@[\w-]+\.[a-z]{2,}\b", re.IGNORECASE)

addresses = [
    "alice@example.com",
    "not-an-email",
    "Bob@Company.ORG",
    "bad@",
]

for addr in addresses:
    if email_pattern.match(addr):
        print(f"  Valid: {addr}")
    else:
        print(f"Invalid: {addr}")
```

Regular expressions are covered in full depth in Chapter 28. This section gives you enough to be productive.

16.8 `itertools` — Powerful Iteration Tools

`itertools` provides fast, memory-efficient tools for working with iterators. All of them return iterators (lazy) — wrap in `list()` to see the values.

```
import itertools

# chain --- combine multiple iterables into one
combined = list(itertools.chain([1, 2, 3], [4, 5], [6]))
print(combined)    # [1, 2, 3, 4, 5, 6]

# chain.from_iterable --- flatten one level
nested = [[1, 2], [3, 4], [5, 6]]
flat = list(itertools.chain.from_iterable(nested))
print(flat)    # [1, 2, 3, 4, 5, 6]

# product --- Cartesian product (all combinations)
sizes = ["S", "M", "L"]
colors = ["red", "blue"]
variants = list(itertools.product(sizes, colors))
print(variants)
# [('S','red'), ('S','blue'), ('M','red'), ('M','blue'), ('L','red'), ('L','blu

# permutations --- all orderings
print(list(itertools.permutations("ABC", 2)))
# [('A','B'), ('A','C'), ('B','A'), ('B','C'), ('C','A'), ('C','B')]

# combinations --- all subsets (order doesn't matter)
print(list(itertools.combinations("ABCD", 2)))
# [('A','B'), ('A','C'), ('A','D'), ('B','C'), ('B','D'), ('C','D')]

# combinations_with_replacement --- same but allows repeats
print(list(itertools.combinations_with_replacement("ABC", 2)))
# [('A','A'), ('A','B'), ('A','C'), ('B','B'), ('B','C'), ('C','C')]

# groupby --- group consecutive items by a key
data = [
    {"name": "Alice",  "dept": "Engineering"},
    {"name": "Bob",    "dept": "Engineering"},
    {"name": "Carlos", "dept": "Marketing"},
    {"name": "Diana",  "dept": "Marketing"},
    {"name": "Eve",    "dept": "Engineering"},
]
# Note: groupby only groups CONSECUTIVE equal keys --- sort first
data.sort(key=lambda x: x["dept"])
```

```python
for dept, members in itertools.groupby(data, key=lambda x: x["dept"]):
    names = [m["name"] for m in members]
    print(f"{dept}: {names}")
# Engineering: ['Alice', 'Bob', 'Eve']
# Marketing: ['Carlos', 'Diana']

# islice --- slice an iterator (no index needed)
gen = (n**2 for n in range(1, 1000))
first_five = list(itertools.islice(gen, 5))
print(first_five)   # [1, 4, 9, 16, 25]

# cycle --- repeat a sequence forever (always use with islice or break)
colors_cycle = itertools.cycle(["red", "green", "blue"])
first_seven = list(itertools.islice(colors_cycle, 7))
print(first_seven)   # ['red', 'green', 'blue', 'red', 'green', 'blue', 'red']

# count --- infinite counter
counter = itertools.count(start=10, step=5)
print(list(itertools.islice(counter, 5)))   # [10, 15, 20, 25, 30]
```

16.9 `functools` — Higher-Order Functions

```python
from functools import reduce, lru_cache, partial

# reduce --- apply a function cumulatively
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda a, b: a * b, numbers)
print(product)   # 120  (1x2x3x4x5)

total = reduce(lambda a, b: a + b, numbers)
print(total)     # 15
```

16.9.1 `lru_cache` — Memoization (cache function results)

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    """Calculate the nth Fibonacci number with caching."""
    if n < 2:
        return n
```

```
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))    # 55
print(fibonacci(50))    # 12586269025  (instant --- cached)

# See cache statistics
print(fibonacci.cache_info())
# CacheInfo(hits=48, misses=51, maxsize=None, currsize=51)
```

Without `lru_cache`, `fibonacci(50)` would make billions of recursive calls. With it, each value is computed once and reused. This is called **memoization** and it's one of the most powerful optimizations in programming.

16.9.2 `partial` — Pre-fill function arguments

```
from functools import partial

def power(base, exponent):
    return base ** exponent

# Create a new function with exponent pre-filled
square = partial(power, exponent=2)
cube   = partial(power, exponent=3)

print(square(5))    # 25
print(cube(3))      # 27

# Useful for passing to map/filter
numbers = [1, 2, 3, 4, 5]
squares = list(map(square, numbers))
print(squares)    # [1, 4, 9, 16, 25]
```

16.9.3 `wraps` — Preserve function metadata in decorators

```
from functools import wraps

def my_decorator(func):
    @wraps(func)    # preserves func's name, docstring, etc.
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
```

You'll use this in Chapter 22 (Decorators).

16.10 Putting It All Together: A File Report Tool

```
import os
import sys
import json
from pathlib import Path
from datetime import datetime
from collections import defaultdict, Counter


def analyze_directory(directory):
    """
    Analyze all files in a directory and return a report dict.
    """
    path = Path(directory)

    if not path.exists():
        print(f"Error: '{directory}' does not exist.")
        sys.exit(1)

    if not path.is_dir():
        print(f"Error: '{directory}' is not a directory.")
        sys.exit(1)

    files = list(path.rglob("*"))
    files = [f for f in files if f.is_file()]

    total_size = sum(f.stat().st_size for f in files)
    extensions = Counter(f.suffix.lower() or "(no ext)" for f in files)
    by_extension = defaultdict(list)

    for f in files:
        by_extension[f.suffix.lower() or "(no ext)"].append(f.name)

    # Find newest and oldest files
    by_mtime = sorted(files, key=lambda f: f.stat().st_mtime)
    oldest = by_mtime[0]  if files else None
    newest = by_mtime[-1] if files else None

    return {
```

```
        "directory":   str(path.resolve()),
        "generated":   datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "total_files": len(files),
        "total_size_kb": round(total_size / 1024, 1),
        "extensions":  dict(extensions.most_common()),
        "oldest_file": oldest.name if oldest else None,
        "newest_file": newest.name if newest else None,
    }


def print_report(report):
    print(f"\n{'='*50}")
    print(f"Directory Report: {report['directory']}")
    print(f"Generated:        {report['generated']}")
    print(f"{'='*50}")
    print(f"Total files:    {report['total_files']}")
    print(f"Total size:     {report['total_size_kb']} KB")
    print(f"\nFile types:")
    for ext, count in report["extensions"].items():
        print(f"  {ext:<15} {count:>4} file(s)")
    print(f"\nOldest file:  {report['oldest_file']}")
    print(f"Newest file:  {report['newest_file']}")


# Entry point
if len(sys.argv) > 1:
    target = sys.argv[1]
else:
    target = "."

report = analyze_directory(target)
print_report(report)

# Save report to JSON
output_path = Path("directory_report.json")
with open(output_path, "w") as f:
    json.dump(report, f, indent=2)
print(f"\nReport saved to: {output_path.resolve()}")
```

Run it with: `python report.py C:\Users\Alice\Documents`

Uses: `pathlib`, `sys`, `os`, `json`, `datetime`, `collections.Counter`, `collections.defaultdict` — seven modules from this chapter in one useful tool.

16.11 What You Learned in This Chapter

- **`os`** — file system operations, environment variables, path manipulation.
- **`pathlib`** — modern path objects: `Path / "subdir"`, `.name`, `.stem`, `.suffix`, `.glob()`, `.read_text()`.
- **`sys`** — Python version, command-line args (`sys.argv`), exit codes (`sys.exit()`), platform.
- **`datetime`** — create, format (`strftime`), parse (`strptime`), and do math with dates/times using `timedelta`.
- **`random`** — `randint`, `choice`, `sample`, `shuffle`, `seed` for reproducibility.
- **`json`** — `dumps`/`loads` for strings, `dump`/`load` for files, Python↔JSON type mapping.
- **`re`** — `search`, `findall`, `sub`, `match`, `split`, groups, compiled patterns.
- **`itertools`** — `chain`, `product`, `permutations`, `combinations`, `groupby`, `islice`, `cycle`, `count`.
- **`functools`** — `reduce`, `lru_cache` (memoization), `partial` (pre-filled functions), `wraps`.

16.12 What's Next?

Part 2 is complete. You've gone deep into Python's essential tools — strings, tuples, sets, dictionaries, error handling, comprehensions, modules, and the standard library.

Part 3 is where programming gets really interesting. In Chapter 17 you'll learn **object-oriented programming** — how to design your own types, bundle data and behavior together into classes, and think about code the way professional software engineers do. It's one of the most powerful ideas in programming, and Python makes it approachable.

Your turn: Build a command-line tool called `rename_files.py` that takes a directory path as `sys.argv[1]` and a pattern as `sys.argv[2]` (e.g. `"*.txt"`). List all matching files using `pathlib.glob()`, show the user how many were found, and ask for a prefix to add to each filename. Rename them all using `os.rename()` and save a log of all renames (old name -> new name, with timestamp) to a JSON file.

Chapter 17

Chapter 17: Classes and Objects — Introduction to OOP

Everything you've built so far uses Python's built-in types — strings, lists, dictionaries, numbers. They're powerful, but they're generic. A dictionary can represent a bank account, but it can't *enforce* that the balance never goes negative. A list can hold students, but it can't know how to calculate their average grade.

Object-oriented programming (OOP) lets you create your own types. You define what data a type holds and what it can do — bundled together into one unit called a **class**. Once you have a class, you create **objects** (also called instances) from it, each with its own data but sharing the same behavior.

This is how almost all serious Python programs are structured. By the end of this chapter you'll understand what OOP is, why it matters, and how to write your own classes with data, methods, and useful string representations.

17.1 What is Object-Oriented Programming?

OOP is a way of organizing code around *things* rather than *steps*.

Instead of thinking "first do this, then do that," you think "what are the things in this program, and what can they do?"

A bank account *is a thing*. It has data (a balance, an owner name) and it can

do things (deposit, withdraw, show its balance). In OOP you design a `BankAccount` class that captures both.

The three core ideas:

1. **Encapsulation** — bundle data and the code that works on it together. The account knows its own balance and its own rules.
2. **Inheritance** — build new types from existing ones. A `SavingsAccount` is a `BankAccount` with extra behavior.
3. **Polymorphism** — different types respond to the same operation in their own way.

This chapter covers encapsulation. Chapters 18 and 19 cover inheritance and polymorphism.

In plain English: A class is a blueprint. An object is something built from that blueprint. The blueprint for a house describes how many rooms it has and what you can do in them. The actual house you live in is an object built from that blueprint.

17.2 Your First Class

Use the `class` keyword, then the class name (by convention: `CapitalizedWords`, also called PascalCase), then a colon.

```
class Dog:
    pass    # empty class for now
```

That's a valid class. Create an object from it:

```
my_dog = Dog()
print(type(my_dog))    # <class '__main__.Dog'>
```

`Dog()` calls the class like a function to create an instance. `my_dog` is now a `Dog` object.

17.3 `__init__` — The Constructor

The `__init__` method runs automatically every time you create a new object. It's where you set up the object's initial data.

```
class Dog:
    def __init__(self, name, breed, age):
        self.name  = name
```

```
        self.breed = breed
        self.age   = age
```

Now create some dogs:

```
dog1 = Dog("Rex",    "German Shepherd", 3)
dog2 = Dog("Bella", "Labrador",          5)
dog3 = Dog("Max",    "Poodle",           1)
```

Each object has its own data:

```
print(dog1.name)     # Rex
print(dog2.breed)    # Labrador
print(dog3.age)      # 1
```

`dog1`, `dog2`, and `dog3` are three separate objects. They were all built from the same `Dog` blueprint, but each has its own `name`, `breed`, and `age`.

17.4 `self` — What It Means and Why It's There

Every method in a class receives the object itself as the first argument. By convention this is named `self`. Python passes it automatically — you never provide it when calling a method.

```
class Dog:
    def __init__(self, name, breed, age):
        self.name  = name    # self.name is an instance attribute
        self.breed = breed   # stored on THIS specific object
        self.age   = age
```

When you write `self.name = name`, you're saying: "Store the value of `name` on this specific instance of Dog." Every `Dog` object has its own `self.name`, its own `self.breed`, its own `self.age`.

Think of `self` as "me" — the object that the method belongs to.

In plain English: `self` is how a method says "I'm talking about this specific object, not some other one." When `dog1.bark()` runs, `self` is `dog1`. When `dog2.bark()` runs, `self` is `dog2`.

17.5 Instance Attributes vs Class Attributes

Instance attributes are set on `self` — each object has its own copy.

Class attributes are set directly on the class — shared by all instances.

```
class Dog:
    species = "Canis lupus familiaris"   # class attribute --- shared by all dogs

    def __init__(self, name, breed, age):
        self.name  = name    # instance attribute --- unique to each dog
        self.breed = breed
        self.age   = age

dog1 = Dog("Rex", "German Shepherd", 3)
dog2 = Dog("Bella", "Labrador", 5)

print(dog1.species)    # Canis lupus familiaris
print(dog2.species)    # Canis lupus familiaris  (same --- shared)
print(Dog.species)     # Canis lupus familiaris  (access via class directly)

print(dog1.name)       # Rex
print(dog2.name)       # Bella  (different --- each has its own)
```

If you set a class attribute via an instance (`dog1.species = "something"`), Python creates a new instance attribute that shadows the class attribute — the class attribute itself is unchanged.

Use class attributes for data that's the same for every instance. Use instance attributes for data that varies per instance.

17.6 Methods — What Objects Can Do

A method is a function defined inside a class. It always receives `self` as its first parameter.

```
class Dog:
    def __init__(self, name, breed, age):
        self.name  = name
        self.breed = breed
        self.age   = age

    def bark(self):
        print(f"{self.name} says: Woof!")

    def describe(self):
        print(f"{self.name} is a {self.age}-year-old {self.breed}.")

    def birthday(self):
        self.age += 1
```

```
        print(f"Happy birthday {self.name}! Now {self.age} years old.")

    def is_puppy(self):
        return self.age < 2
```

Call methods on objects:

```
dog1 = Dog("Rex", "German Shepherd", 3)
dog2 = Dog("Max", "Poodle", 1)

dog1.bark()          # Rex says: Woof!
dog1.describe()      # Rex is a 3-year-old German Shepherd.
dog1.birthday()      # Happy birthday Rex! Now 4 years old.

print(dog2.is_puppy())    # True  (age is 1, under 2)
print(dog1.is_puppy())    # False (age is now 4)
```

Methods can read instance attributes (`self.name`), modify them (`self.age += 1`), return values (`return self.age < 2`), and call other methods on `self`.

Your turn: Add a method `human_years()` to the `Dog` class that returns the dog's age multiplied by 7. Call it on two different dogs and print the result.

17.7 `__str__` and `__repr__` — Representing Objects as Text

When you print an object without defining `__str__`, Python shows something ugly:

```
dog = Dog("Rex", "German Shepherd", 3)
print(dog)    # <__main__.Dog object at 0x000001A2B3C4D5E6>
```

Define `__str__` to control what `print()` and f-strings show:

```
class Dog:
    def __init__(self, name, breed, age):
        self.name  = name
        self.breed = breed
        self.age   = age

    def __str__(self):
        return f"Dog(name={self.name!r}, breed={self.breed!r}, age={self.age})"
```

Now:

```
dog = Dog("Rex", "German Shepherd", 3)
print(dog)          # Dog(name='Rex', breed='German Shepherd', age=3)
print(f"{dog}")     # Dog(name='Rex', breed='German Shepherd', age=3)
```

`__str__` vs `__repr__`:

- `__str__` is for humans — the friendly, readable version. Used by `print()` and f-strings.
- `__repr__` is for developers — the technical version, ideally showing how to recreate the object. Used in the REPL and by `repr()`.

```
def __repr__(self):
    return f"Dog({self.name!r}, {self.breed!r}, {self.age})"
```

The `!r` format spec inside f-strings calls `repr()` on the value — it adds quotes around strings, so the output is unambiguous.

If you only define one, define `__repr__` — Python uses it as a fallback for `__str__` too.

17.8 A Complete, Realistic Class

Let's build a `BankAccount` class that enforces real rules:

```
class BankAccount:
    """A simple bank account with deposit, withdrawal, and history."""

    interest_rate = 0.02   # class attribute --- 2% for all accounts

    def __init__(self, owner, initial_balance=0.0):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative.")
        self.owner    = owner
        self._balance = float(initial_balance)   # _balance: "private by conventio
        self._history = []
        self._log(f"Account opened with ${initial_balance:.2f}")

    # ── Properties ──────────────────────────────────
    @property
    def balance(self):
        """Read-only access to the balance."""
        return self._balance

    # ── Public methods ──────────────────────────────
    def deposit(self, amount):
```

```
        """Deposit a positive amount."""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive.")
        self._balance += amount
        self._log(f"Deposited  ${amount:.2f} -> balance ${self._balance:.2f}")

    def withdraw(self, amount):
        """Withdraw amount if sufficient funds exist."""
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if amount > self._balance:
            raise ValueError(
                f"Insufficient funds. Balance: ${self._balance:.2f}, "
                f"Requested: ${amount:.2f}"
            )
        self._balance -= amount
        self._log(f"Withdrew   ${amount:.2f} -> balance ${self._balance:.2f}")

    def apply_interest(self):
        """Add interest to the account."""
        interest = self._balance * self.interest_rate
        self._balance += interest
        self._log(f"Interest   ${interest:.2f} -> balance ${self._balance:.2f}")

    def print_history(self):
        """Print full transaction history."""
        print(f"\nAccount history for {self.owner}:")
        for entry in self._history:
            print(f"  {entry}")

    # ── Dunder methods ──────────────────────────────
    def __str__(self):
        return f"BankAccount({self.owner!r}, balance=${self._balance:.2f})"

    def __repr__(self):
        return f"BankAccount({self.owner!r}, {self._balance:.2f})"

    # ── Private helper ──────────────────────────────
    def _log(self, message):
        """Record a transaction (private --- not for external use)."""
        self._history.append(message)
```

Using it:

```
account = BankAccount("Alice", 500.00)
```

```
account.deposit(200.00)
account.withdraw(75.50)
account.apply_interest()
account.print_history()
print(account)
print(f"Balance: ${account.balance:.2f}")
```

Output:

```
Account history for Alice:
  Account opened with $500.00
  Deposited  $200.00 -> balance $700.00
  Withdrew   $75.50 -> balance $624.50
  Interest   $12.49 -> balance $636.99

BankAccount('Alice', balance=$636.99)
Balance: $636.99
```

Notice several things: - `_balance` and `_history` start with `_` — the Python convention for "don't touch this from outside the class." - `@property` makes `balance` readable as `account.balance` without parentheses, but prevents direct assignment. - The class enforces its own rules — you can't deposit a negative amount or overdraw. - `_log` is a private helper — it's an implementation detail, not part of the public interface.

Your turn: Add a `transfer(self, other_account, amount)` method to `BankAccount` that withdraws from `self` and deposits into `other_account`. Handle the case where there are insufficient funds. Test it with two accounts.

17.9 Multiple Objects, Same Class

Every object created from a class is independent. Changes to one don't affect the others.

```
alice_account = BankAccount("Alice", 1000.00)
bob_account   = BankAccount("Bob",    500.00)

alice_account.deposit(250.00)
bob_account.withdraw(100.00)

print(alice_account)    # BankAccount('Alice', balance=$1250.00)
print(bob_account)      # BankAccount('Bob', balance=$400.00)
```

Alice's deposit didn't touch Bob's balance. They're completely separate objects sharing only the class definition (the blueprint) — not the data.

17.10 Objects in Collections

Objects work perfectly inside lists, dictionaries, and anywhere else you'd use a value:

```
accounts = [
    BankAccount("Alice",  1000.00),
    BankAccount("Bob",     500.00),
    BankAccount("Carlos", 2500.00),
    BankAccount("Diana",   750.00),
]

# Apply interest to all accounts
for acc in accounts:
    acc.apply_interest()

# Find the richest account
richest = max(accounts, key=lambda acc: acc.balance)
print(f"Richest: {richest.owner} with ${richest.balance:.2f}")

# Total deposits across all accounts
total = sum(acc.balance for acc in accounts)
print(f"Total:   ${total:.2f}")
```

17.11 Comparing Objects

By default, two objects are only equal if they're the *same object* in memory:

```
a = Dog("Rex", "German Shepherd", 3)
b = Dog("Rex", "German Shepherd", 3)

print(a == b)    # False --- different objects, even with same data
print(a is b)    # False
print(a == a)    # True --- same object
```

To make == compare by value, define `__eq__`:

```
class Dog:
    def __init__(self, name, breed, age):
        self.name  = name
        self.breed = breed
```

```
        self.age   = age

    def __eq__(self, other):
        if not isinstance(other, Dog):
            return NotImplemented
        return (self.name, self.breed, self.age) == (other.name, other.breed, othe

    def __repr__(self):
        return f"Dog({self.name!r}, {self.breed!r}, {self.age})"
```

Now:

```
a = Dog("Rex", "German Shepherd", 3)
b = Dog("Rex", "German Shepherd", 3)
c = Dog("Bella", "Labrador", 5)

print(a == b)   # True  --- same data
print(a == c)   # False --- different data
```

You'll learn all the comparison dunder methods (`__lt__`, `__le__`, `__gt__`, etc.) in Chapter 19.

17.12 Putting It All Together: A Student Grade Tracker

```
class Student:
    """Represents a student with a name and a list of grades."""

    def __init__(self, name, student_id):
        self.name       = name
        self.student_id = student_id
        self._grades    = []

    def add_grade(self, subject, score):
        if not 0 <= score <= 100:
            raise ValueError(f"Score must be between 0 and 100, got {score}.")
        self._grades.append({"subject": subject, "score": score})

    @property
    def average(self):
        if not self._grades:
            return 0.0
        return sum(g["score"] for g in self._grades) / len(self._grades)
```

```
    @property
    def highest(self):
        if not self._grades:
            return None
        return max(self._grades, key=lambda g: g["score"])

    @property
    def lowest(self):
        if not self._grades:
            return None
        return min(self._grades, key=lambda g: g["score"])

    @property
    def letter_grade(self):
        avg = self.average
        if avg >= 90: return "A"
        if avg >= 80: return "B"
        if avg >= 70: return "C"
        if avg >= 60: return "D"
        return "F"

    def report(self):
        print(f"\nStudent: {self.name} (ID: {self.student_id})")
        print(f"{'Subject':<20} {'Score':>5}")
        print("-" * 27)
        for g in self._grades:
            print(f"{g['subject']:<20} {g['score']:>5}")
        print("-" * 27)
        print(f"{'Average':<20} {self.average:>5.1f}  ({self.letter_grade})")
        if self.highest:
            print(f"Best:  {self.highest['subject']} ({self.highest['score']})")
            print(f"Worst: {self.lowest['subject']} ({self.lowest['score']})")

    def __str__(self):
        return f"{self.name} (avg: {self.average:.1f}, grade: {self.letter_grad

    def __repr__(self):
        return f"Student({self.name!r}, {self.student_id!r})"


# Create students
alice = Student("Alice Smith", "S001")
alice.add_grade("Math", 92)
```

```
alice.add_grade("Physics", 88)
alice.add_grade("Chemistry", 76)
alice.add_grade("English", 95)
alice.add_grade("History", 84)

bob = Student("Bob Jones", "S002")
bob.add_grade("Math", 65)
bob.add_grade("Physics", 72)
bob.add_grade("Chemistry", 58)
bob.add_grade("English", 80)
bob.add_grade("History", 61)

# Print reports
alice.report()
bob.report()

# Compare and sort
students = [alice, bob]
students.sort(key=lambda s: s.average, reverse=True)

print("\nRankings:")
for rank, student in enumerate(students, start=1):
    print(f"  {rank}. {student}")
```

Output:

```
Student: Alice Smith (ID: S001)
Subject              Score
--------------------------
Math                    92
Physics                 88
Chemistry               76
English                 95
History                 84
--------------------------
Average               87.0  (B)
Best:  English (95)
Worst: Chemistry (76)

Student: Bob Jones (ID: S002)
Subject              Score
--------------------------
Math                    65
Physics                 72
Chemistry               58
```

```
English                  80
History                  61
--------------------------
Average               67.2  (D)
Best:  English (80)
Worst: Chemistry (58)

Rankings:
  1. Alice Smith (avg: 87.0, grade: B)
  2. Bob Jones (avg: 67.2, grade: D)
```

17.13 What You Learned in This Chapter

- A **class** is a blueprint. An **object** (instance) is built from that blueprint.
- Define classes with `class ClassName:` — use PascalCase for names.
- `__init__` is the constructor — it runs automatically when an object is created.
- `self` refers to the current instance — Python passes it automatically.
- **Instance attributes** (`self.x`) are unique to each object. **Class attributes** are shared by all.
- **Methods** are functions inside a class — always receive `self` as the first parameter.
- `__str__` controls `print()` output. `__repr__` controls the developer representation.
- `_name` (single underscore prefix) signals "private by convention" — don't use from outside.
- `@property` makes a method behave like an attribute — readable without `()`.
- `__eq__` lets you define what `==` means for your objects.
- Objects work inside lists, dicts, and any other collection.
- Classes enforce their own rules through validation in methods.

17.14 What's Next?

You've built your first classes. The real power of OOP comes when you build classes *from* other classes.

In Chapter 18 you'll learn **inheritance** — how a `SavingsAccount` can be a `BankAccount` with extra behavior, how a `GuideDog` can be a `Dog` with extra capabilities, and how `super()` lets child classes build on their parent's work without

repeating it. You'll also learn **polymorphism** — how different classes can respond to the same method call in their own way.

Your turn: Design a `Rectangle` class with `width` and `height` attributes. Add methods: `area()`, `perimeter()`, `is_square()`, and `scale(factor)` (which multiplies both dimensions by a factor). Add `__str__` and `__repr__`. Then create a `Square` class (not using inheritance yet — just a class) that only takes one `side` argument and sets `width = height = side`. Compare a `Rectangle(4, 4)` and a `Square(4)` — do they have the same area and perimeter?

Chapter 18

Chapter 18: Inheritance and Polymorphism

In Chapter 17 you built classes from scratch. Each class was independent — a `Dog`, a `BankAccount`, a `Student`. That works, but real programs have *families* of related things.

A `SavingsAccount` is a `BankAccount` with an interest rate. A `GuideDog` is a `Dog` with extra training. A `Manager` is an `Employee` with extra responsibilities. They share most of their behavior, but differ in specific ways.

Inheritance lets you build a new class on top of an existing one. The new class gets everything the parent has, and you add or change only what's different. This is one of the most powerful ideas in OOP — it eliminates duplication and lets you build complex systems from simple, layered pieces.

Polymorphism lets different classes respond to the same method call in their own way. A `Dog` and a `Cat` both have a `speak()` method, but one barks and one meows. Your code can call `speak()` without caring which animal it has.

18.1 What is Inheritance?

Inheritance creates an **is-a** relationship. A `SavingsAccount` *is a* `BankAccount`. A `GuideDog` *is a* `Dog`. The new class (child, subclass) inherits all the attributes and methods of the existing class (parent, superclass).

Without inheritance — duplicating code:

```
class Animal:
    def __init__(self, name, age):
```

```
        self.name = name
        self.age  = age

    def breathe(self):
        print(f"{self.name} breathes.")

    def sleep(self):
        print(f"{self.name} sleeps.")


# Without inheritance, you copy these methods into Dog and Cat
class Dog:
    def __init__(self, name, age):
        self.name = name    # duplicated
        self.age  = age     # duplicated

    def breathe(self):      # duplicated
        print(f"{self.name} breathes.")

    def sleep(self):        # duplicated
        print(f"{self.name} sleeps.")

    def bark(self):
        print(f"{self.name} says: Woof!")
```

With inheritance — define once, reuse everywhere:

```
class Dog(Animal):    # Dog inherits from Animal
    def bark(self):
        print(f"{self.name} says: Woof!")
```

`Dog` inherits `__init__`, `breathe`, and `sleep` from `Animal`. You only write what's new — `bark`. If `Animal.sleep()` changes, `Dog` gets the update automatically.

18.2 Creating a Child Class

Put the parent class name in parentheses:

```
class Animal:
    def __init__(self, name, species, age):
        self.name    = name
        self.species = species
        self.age     = age
```

```
    def breathe(self):
        print(f"{self.name} breathes.")

    def sleep(self):
        print(f"{self.name} is sleeping.")

    def describe(self):
        print(f"{self.name} is a {self.age}-year-old {self.species}.")

    def __str__(self):
        return f"{self.species}({self.name!r}, age={self.age})"


class Dog(Animal):
    def bark(self):
        print(f"{self.name} says: Woof!")

    def fetch(self, item):
        print(f"{self.name} fetches the {item}!")


class Cat(Animal):
    def meow(self):
        print(f"{self.name} says: Meow!")

    def purr(self):
        print(f"{self.name} purrs contentedly.")
```

`Dog` and `Cat` inherit everything from `Animal` and add their own methods:

```
rex   = Dog("Rex",   "German Shepherd", 3)
bella = Cat("Bella", "Persian Cat",     5)

rex.describe()     # Rex is a 3-year-old German Shepherd.
rex.breathe()      # Rex breathes.
rex.bark()         # Rex says: Woof!
rex.fetch("ball") # Rex fetches the ball!

bella.describe()  # Bella is a 5-year-old Persian Cat.
bella.meow()       # Bella says: Meow!
bella.purr()       # Bella purrs contentedly.

print(rex)         # German Shepherd('Rex', age=3)
print(bella)       # Persian Cat('Bella', age=5)
```

`rex` and `bella` can use both their own methods and everything from `Animal`.

18.3 `super()` — Calling the Parent's Method

When you define `__init__` in a child class, you need to call the parent's `__init__` too — otherwise the parent's setup code never runs and the object won't have the parent's attributes.

`super()` gives you access to the parent class:

```
class Animal:
    def __init__(self, name, species, age):
        self.name    = name
        self.species = species
        self.age     = age


class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, "Dog", age)    # call Animal.__init__
        self.breed = breed                    # add Dog-specific attribute

    def describe(self):
        print(f"{self.name} is a {self.age}-year-old {self.breed}.")

    def __str__(self):
        return f"Dog({self.name!r}, {self.breed!r}, age={self.age})"

rex = Dog("Rex", 3, "German Shepherd")
print(rex.name)     # Rex      (set by Animal.__init__ via super())
print(rex.age)      # 3        (set by Animal.__init__ via super())
print(rex.breed)    # German Shepherd  (set by Dog.__init__)
rex.describe()      # Rex is a 3-year-old German Shepherd.
```

The rule: When a child class has its own `__init__`, call `super().__init__(...)` first to initialize the parent's part of the object. Then add the child's own attributes.

`super()` also works for other methods — call `super().method()` to run the parent version and then extend it:

```
class Animal:
    def describe(self):
        print(f"Name: {self.name}, Age: {self.age}")


class Dog(Animal):
```

```
    def describe(self):
        super().describe()             # run Animal's version first
        print(f"Breed: {self.breed}") # then add Dog's extra info
```

18.4 Overriding Methods

A child class can **override** any parent method — define a method with the same name. The child's version replaces the parent's for that class.

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age  = age

    def speak(self):
        print(f"{self.name} makes a sound.")

    def __str__(self):
        return f"{type(self).__name__}({self.name!r})"


class Dog(Animal):
    def speak(self):                           # overrides Animal.speak
        print(f"{self.name} says: Woof!")


class Cat(Animal):
    def speak(self):                           # overrides Animal.speak
        print(f"{self.name} says: Meow!")


class Duck(Animal):
    def speak(self):                           # overrides Animal.speak
        print(f"{self.name} says: Quack!")


class Fish(Animal):
    pass                                       # no override --- uses Animal.spea

animals = [
    Dog("Rex",    3),
    Cat("Bella", 5),
    Duck("Donald", 7),
```

```
    Fish("Nemo", 1),
]

for animal in animals:
    animal.speak()
```

Output:

```
Rex says: Woof!
Bella says: Meow!
Donald says: Quack!
Nemo makes a sound.
```

This is **polymorphism** — the same `speak()` call on the same loop produces different behavior depending on which class the object belongs to. The loop doesn't know or care whether it has a dog, cat, or duck. It just calls `speak()` and each object knows what to do.

`type(self).__name__` in `__str__` above automatically uses the actual class name — `"Dog"`, `"Cat"`, etc. — without hardcoding it. This works correctly in child classes without overriding `__str__`.

18.5 `isinstance()` and `issubclass()`

`isinstance(obj, Class)` — is `obj` an instance of `Class` (or a subclass of it)?

```
rex = Dog("Rex", 3)

print(isinstance(rex, Dog))      # True  --- rex IS a Dog
print(isinstance(rex, Animal))   # True  --- Dog is a subclass of Animal
print(isinstance(rex, Cat))      # False --- rex is not a Cat
```

`issubclass(Child, Parent)` — is `Child` a subclass of `Parent`?

```
print(issubclass(Dog, Animal))     # True
print(issubclass(Cat, Animal))     # True
print(issubclass(Dog, Cat))        # False
print(issubclass(Animal, Animal)) # True --- a class is a subclass of itself
```

These are useful for writing code that handles multiple types gracefully:

```
def process_animal(animal):
    if isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Cat):
        animal.meow()
```

```
    else:
        animal.speak()
```

But here's the thing: if you're checking `isinstance` in a lot of places, that's often a sign you should use polymorphism instead — let each class define its own behavior so you don't have to check.

18.6 Inheritance Chains — Multiple Levels

Inheritance can go multiple levels deep. A `GuideDog` is a `Dog` which is an `Animal`.

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age  = age

    def breathe(self):
        print(f"{self.name} breathes.")


class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def bark(self):
        print(f"{self.name} says: Woof!")


class GuideDog(Dog):
    def __init__(self, name, age, breed, handler):
        super().__init__(name, age, breed)
        self.handler = handler

    def guide(self):
        print(f"{self.name} guides {self.handler} safely.")

    def __str__(self):
        return (f"GuideDog({self.name!r}, {self.breed!r}, "
                f"handler={self.handler!r})")

buddy = GuideDog("Buddy", 4, "Labrador", "John")

buddy.breathe()     # inherited from Animal
```

```
buddy.bark()        # inherited from Dog
buddy.guide()       # defined in GuideDog
print(buddy)        # GuideDog('Buddy', 'Labrador', handler='John')

print(isinstance(buddy, GuideDog))   # True
print(isinstance(buddy, Dog))        # True
print(isinstance(buddy, Animal))     # True
```

`GuideDog` inherits from `Dog`, which inherits from `Animal`. `buddy` has access to everything from all three levels.

18.7 Multiple Inheritance

Python allows a class to inherit from more than one parent class.

```
class Flyable:
    def fly(self):
        print(f"{self.name} is flying!")

    def land(self):
        print(f"{self.name} has landed.")


class Swimmable:
    def swim(self):
        print(f"{self.name} is swimming!")


class Duck(Animal, Flyable, Swimmable):
    def __init__(self, name, age):
        super().__init__(name, age)

    def speak(self):
        print(f"{self.name} says: Quack!")

donald = Duck("Donald", 3)
donald.speak()     # Donald says: Quack!
donald.fly()       # Donald is flying!
donald.swim()      # Donald is swimming!
donald.breathe()   # Donald breathes.  (from Animal)
```

`Duck` inherits from `Animal`, `Flyable`, and `Swimmable` simultaneously.

Be careful with multiple inheritance. When two parent classes define

the same method name, Python's **Method Resolution Order (MRO)** determines which one runs. Check it with:

```
print(Duck.__mro__)
# (<class 'Duck'>, <class 'Animal'>, <class 'Flyable'>, <class 'Swimmable'>, <c
```

Python resolves methods left to right through the MRO chain. Keep multiple inheritance simple — preferably use it only for **mixins** (small classes that add one focused capability, like `Flyable` and `Swimmable` above). When inheritance gets complex, prefer composition (holding another object as an attribute) over deep inheritance trees.

18.8 Abstract Base Classes — Enforcing a Contract

Sometimes you want to define a class that *requires* child classes to implement certain methods. You can't use the parent class directly — it's just a template.

Python's `abc` module provides this:

```
from abc import ABC, abstractmethod


class Shape(ABC):
    """Abstract base class for all shapes."""

    @abstractmethod
    def area(self):
        """Return the area of the shape."""

    @abstractmethod
    def perimeter(self):
        """Return the perimeter of the shape."""

    def describe(self):
        """Non-abstract --- available to all shapes."""
        print(f"{type(self).__name__}: area={self.area():.2f}, "
              f"perimeter={self.perimeter():.2f}")
```

Any class that inherits from `Shape` **must** implement `area()` and `perimeter()`. If it doesn't, Python raises a `TypeError` when you try to create an instance.

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

```
    def area(self):
        import math
        return math.pi * self.radius ** 2

    def perimeter(self):
        import math
        return 2 * math.pi * self.radius

    def __str__(self):
        return f"Circle(radius={self.radius})"


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width  = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def __str__(self):
        return f"Rectangle({self.width}x{self.height})"


class Triangle(Shape):
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c

    def area(self):
        # Heron's formula
        s = (self.a + self.b + self.c) / 2
        return (s * (s-self.a) * (s-self.b) * (s-self.c)) ** 0.5

    def perimeter(self):
        return self.a + self.b + self.c

    def __str__(self):
        return f"Triangle({self.a}, {self.b}, {self.c})"

# This would raise TypeError --- Shape is abstract
```

```
# s = Shape()

shapes = [
    Circle(5),
    Rectangle(4, 6),
    Triangle(3, 4, 5),
    Circle(2.5),
    Rectangle(10, 3),
]

for shape in shapes:
    shape.describe()

print(f"\nLargest area: {max(shapes, key=lambda s: s.area())}")
```

Output:

```
Circle: area=78.54, perimeter=31.42
Rectangle: area=24.00, perimeter=20.00
Triangle: area=6.00, perimeter=12.00
Circle: area=19.63, perimeter=15.71
Rectangle: area=30.00, perimeter=26.00

Largest area: Circle(radius=5)
```

The `for` loop calls `.describe()` on every shape. Each shape calculates its own `area()` and `perimeter()` — polymorphism again. The loop doesn't know or care about the specific type.

Abstract base classes enforce a **contract**: "any class that inherits from me must provide these methods." This is how libraries and frameworks define interfaces that your code must implement.

Your turn: Add a `Square` class that inherits from `Rectangle`. Its `__init__` takes only a `side` argument and sets `width = height = side`. Override `__str__` to return `"Square(side=4)"`. Add it to the shapes list and verify it works with `describe()`.

18.9 A Real-World Inheritance Example: Payment System

```
from abc import ABC, abstractmethod
from datetime import datetime
```

```
class PaymentMethod(ABC):
    """Abstract base class for all payment methods."""

    def __init__(self, owner):
        self.owner = owner
        self._transactions = []

    @abstractmethod
    def process(self, amount, description):
        """Process a payment. Must be implemented by subclasses."""

    def _record(self, amount, description, status):
        self._transactions.append({
            "timestamp":   datetime.now().strftime("%Y-%m-%d %H:%M"),
            "amount":      amount,
            "description": description,
            "status":      status,
        })

    def history(self):
        print(f"\nTransaction history for {self.owner}:")
        for t in self._transactions:
            print(f"  [{t['timestamp']}] {t['description']}: "
                  f"${t['amount']:.2f} --- {t['status']}")

    def __str__(self):
        return f"{type(self).__name__}(owner={self.owner!r})"


class CreditCard(PaymentMethod):
    def __init__(self, owner, card_number, credit_limit):
        super().__init__(owner)
        self._card_number  = card_number
        self._credit_limit = credit_limit
        self._balance      = 0.0

    def process(self, amount, description):
        if self._balance + amount > self._credit_limit:
            self._record(amount, description, "DECLINED --- over limit")
            print(f"DECLINED: {description} ${amount:.2f} --- credit limit reached
            return False
        self._balance += amount
        self._record(amount, description, "APPROVED")
```

```
            print(f"APPROVED: {description} ${amount:.2f} via credit card.")
            return True

        def pay_bill(self, amount):
            self._balance = max(0, self._balance - amount)
            print(f"Bill payment: ${amount:.2f}. Remaining balance: ${self._balance

        def __str__(self):
            masked = "*" * 12 + self._card_number[-4:]
            return f"CreditCard({self.owner!r}, {masked}, limit=${self._credit_limi


    class BankTransfer(PaymentMethod):
        def __init__(self, owner, account_number, balance):
            super().__init__(owner)
            self._account = account_number
            self._balance = float(balance)

        def process(self, amount, description):
            if amount > self._balance:
                self._record(amount, description, "DECLINED --- insufficient funds"
                print(f"DECLINED: {description} ${amount:.2f} --- insufficient fund
                return False
            self._balance -= amount
            self._record(amount, description, "APPROVED")
            print(f"APPROVED: {description} ${amount:.2f} via bank transfer.")
            return True

        def __str__(self):
            return f"BankTransfer({self.owner!r}, balance=${self._balance:.2f})"


    class DigitalWallet(PaymentMethod):
        def __init__(self, owner, initial_balance=0.0):
            super().__init__(owner)
            self._balance = float(initial_balance)

        def top_up(self, amount):
            self._balance += amount
            print(f"Wallet topped up: ${amount:.2f}. New balance: ${self._balance:.

        def process(self, amount, description):
            if amount > self._balance:
                self._record(amount, description, "DECLINED --- low balance")
```

```
            print(f"DECLINED: {description} ${amount:.2f} --- wallet balance too l
            return False
        self._balance -= amount
        self._record(amount, description, "APPROVED")
        print(f"APPROVED: {description} ${amount:.2f} via digital wallet.")
        return True

    def __str__(self):
        return f"DigitalWallet({self.owner!r}, balance=${self._balance:.2f})"


def checkout(payment_method, cart):
    """Process a list of (description, amount) purchases."""
    print(f"\nChecking out with: {payment_method}")
    total = 0
    for description, amount in cart:
        if payment_method.process(amount, description):
            total += amount
    print(f"Total charged: ${total:.2f}")
    payment_method.history()


# --- Demonstrate polymorphism ---
card   = CreditCard("Alice",   "1234567890123456", credit_limit=500.0)
bank   = BankTransfer("Bob",   "ACC-9876",         balance=300.0)
wallet = DigitalWallet("Carlos", initial_balance=150.0)

cart = [
    ("Coffee",       4.50),
    ("Lunch",       12.75),
    ("Books",       45.00),
    ("Electronics", 600.00),   # this will be declined for credit card
]

# Same checkout() function --- different payment method --- different behavior
checkout(card,   cart)
checkout(wallet, cart)
```

Three completely different payment methods, one `checkout()` function. Polymorphism in action.

18.10 What You Learned in This Chapter

- **Inheritance** creates an is-a relationship: child class gets all parent attributes and methods.
- Define a child class: `class Child(Parent):`.
- `super().__init__(...)` calls the parent's constructor — always do this in a child's `__init__`.
- `super().method()` calls the parent's version of any method.
- **Overriding** replaces a parent method with a new version in the child class.
- **Polymorphism** — different classes respond to the same method call differently.
- `isinstance(obj, Class)` checks if an object is an instance of a class or its subclasses.
- `issubclass(Child, Parent)` checks inheritance relationships.
- Inheritance chains can go multiple levels deep — each `super()` call walks up the chain.
- **Multiple inheritance** lets a class inherit from several parents. Use mixins for focused capabilities.
- **Method Resolution Order (MRO)** — Python's left-to-right rule for resolving method names in multiple inheritance.
- **Abstract base classes** (from `abc`) enforce contracts — any subclass must implement `@abstractmethod` methods.
- If you can't instantiate an abstract class directly, that's intentional — it's just a blueprint.

18.11 What's Next?

You know how to build class hierarchies. Chapter 19 goes deeper into OOP — **encapsulation and special methods**. You'll learn about properties, private attributes, and the full range of dunder methods that let your objects work with Python's built-in operators: `+`, `[]`, `len()`, `in`, comparisons, iteration, and more. This is where your classes start feeling like native Python types.

Your turn: Build a `Vehicle` base class with `make`, `model`, `year`, and `fuel_level` attributes. Add `fuel()` (add fuel) and `drive(distance)` (consumes fuel) methods. Create three subclasses: `Car` (regular fuel consumption), `Truck` (double fuel consumption), and `ElectricCar` (uses charge instead of fuel, with a different display). Override `drive()` in each. Make them all work with a `road_trip(vehicle, stops)` function that drives between each stop and prints a status update. Use `isinstance()` to give a special message for `ElectricCar` about charging.

Chapter 19

Chapter 19: OOP in Depth — Encapsulation and Special Methods

You know how to create classes, add methods, and use inheritance. That's the foundation. This chapter goes deeper — into the features that make Python OOP genuinely powerful and elegant.

You'll learn how to protect data inside objects, how to use properties for controlled access, how to make your objects work with Python's built-in operators (`+`, `[]`, `len()`, `in`, comparisons), how to make objects iterable, how to write class and static methods, and how `dataclasses` can save you from writing boilerplate.

By the end of this chapter, your classes will feel like native Python types — not just containers for functions.

19.1 Encapsulation — Protecting Data

Encapsulation means bundling data and the code that works on it together, and controlling access from outside.

In Python this is done by convention, not enforcement. There are two levels:

19.1.1 Single underscore `_name` — "private by convention"

A single underscore signals: "this is an implementation detail — don't use it from outside the class." Python won't stop you, but it's a clear message.

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius    # _celsius: treat as private


temp = Temperature(25)
print(temp._celsius)    # works, but you're breaking the contract
```

19.1.2 Double underscore `__name` — name mangling

A double underscore triggers **name mangling** — Python renames the attribute to `_ClassName__name`. This makes accidental access harder (but not impossible).

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner    = owner
        self.__balance = balance    # name-mangled to _BankAccount__balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount


account = BankAccount("Alice", 1000)
print(account.get_balance())       # 1000  [x]

# Direct access fails
# print(account.__balance)         # AttributeError
print(account._BankAccount__balance)  # 1000 --- still accessible if you know t
```

Use __ sparingly — mainly when you're writing a library and want to prevent accidental name clashes in subclasses. For most code, `_name` convention is sufficient.

19.2 Properties — Controlled Attribute Access

A **property** lets you expose an attribute-like interface while running code behind the scenes. Callers write `obj.balance` (no parentheses), but you control what happens when they read, write, or delete it.

19.2.1 `@property` — A readable "attribute"

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """The radius of the circle."""
        return self._radius

    @property
    def diameter(self):
        return self._radius * 2

    @property
    def area(self):
        import math
        return math.pi * self._radius ** 2

    @property
    def circumference(self):
        import math
        return 2 * math.pi * self._radius

c = Circle(5)
print(c.radius)          # 5
print(c.diameter)        # 10
print(f"{c.area:.2f}")   # 78.54

c.radius = 10            # AttributeError --- no setter defined yet
```

19.2.2 `@property.setter` — Allow controlled writing

```
class Circle:
    def __init__(self, radius):
        self.radius = radius    # calls the setter immediately

    @property
```

```
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError(f"Radius must be positive, got {value}.")
        self._radius = value

    @property
    def area(self):
        import math
        return math.pi * self._radius ** 2

    def __str__(self):
        return f"Circle(radius={self._radius})"

c = Circle(5)
print(c.radius)     # 5

c.radius = 10
print(c.radius)     # 10

c.radius = -3       # ValueError: Radius must be positive, got -3.
```

The setter validates before storing. The caller just writes `c.radius = 10` — clean and natural. The validation is invisible.

19.2.3 `@property.deleter` — Handle deletion

```
class Connection:
    def __init__(self, host):
        self._host    = host
        self._socket  = None

    @property
    def host(self):
        return self._host

    @host.deleter
    def host(self):
        print(f"Disconnecting from {self._host}...")
        self._host   = None
        self._socket = None
```

```
conn = Connection("example.com")
del conn.host    # Disconnecting from example.com...
```

19.2.4 Computed properties

Properties are perfect for values that should be calculated rather than stored:

```
class Rectangle:
    def __init__(self, width, height):
        self.width  = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

    @property
    def perimeter(self):
        return 2 * (self.width + self.height)

    @property
    def is_square(self):
        return self.width == self.height

    def __repr__(self):
        return f"Rectangle({self.width}, {self.height})"

r = Rectangle(4, 6)
print(r.area)       # 24
print(r.perimeter)  # 20
print(r.is_square)  # False

r.width = 6
print(r.is_square)  # True  --- recalculated from current values
```

`area`, `perimeter`, and `is_square` are always fresh — they're computed from the current `width` and `height` each time. No risk of stale data.

Your turn: Build a `Temperature` class that stores the value in Celsius internally. Add `celsius`, `fahrenheit`, and `kelvin` as properties. Add setters for `celsius` and `fahrenheit` (both should update `_celsius`). Make it impossible to set a temperature below absolute zero (-273.15°C).

19.3 Dunder (Magic) Methods — Making Objects Feel Native

Dunder methods (double underscore on both sides — `__method__`) let your objects work with Python's built-in operators and functions. When Python sees `a + b`, it calls `a.__add__(b)`. When it sees `len(x)`, it calls `x.__len__()`. You define these methods to make your objects behave naturally.

19.3.1 `__len__` — `len()` support

```
class Playlist:
    def __init__(self, name):
        self.name   = name
        self._songs = []

    def add(self, song):
        self._songs.append(song)

    def __len__(self):
        return len(self._songs)

    def __str__(self):
        return f"Playlist({self.name!r}, {len(self)} songs)"


p = Playlist("Favourites")
p.add("Bohemian Rhapsody")
p.add("Hotel California")
p.add("Stairway to Heaven")

print(len(p))     # 3
print(p)          # Playlist('Favourites', 3 songs)
```

19.3.2 `__getitem__` and `__setitem__` — Index access `[]`

```
class Playlist:
    def __init__(self, name):
        self.name   = name
        self._songs = []

    def add(self, song):
        self._songs.append(song)

    def __len__(self):
```

```
        return len(self._songs)

    def __getitem__(self, index):
        return self._songs[index]

    def __setitem__(self, index, value):
        self._songs[index] = value

    def __delitem__(self, index):
        del self._songs[index]


p = Playlist("Favourites")
p.add("Song A")
p.add("Song B")
p.add("Song C")

print(p[0])      # Song A
print(p[-1])     # Song C
print(p[1:3])    # ['Song B', 'Song C']  --- slicing works too!

p[0] = "Song D"
print(p[0])      # Song D

del p[1]
print(len(p))    # 2
```

Once you define `__getitem__`, Python also gives you slicing, reverse iteration, and `for item in obj` automatically.

19.3.3 `__contains__` — `in` operator

```
def __contains__(self, song):
    return song in self._songs


p = Playlist("Favourites")
p.add("Bohemian Rhapsody")

print("Bohemian Rhapsody" in p)    # True
print("Thriller" in p)             # False
```

19.3.4 `__iter__` and `__next__` — Making objects iterable

Define `__iter__` to make your object work in `for` loops:

```
class Playlist:
    def __init__(self, name):
        self.name   = name
        self._songs = []

    def add(self, song):
        self._songs.append(song)

    def __iter__(self):
        return iter(self._songs)    # delegate to the list's iterator

    def __len__(self):
        return len(self._songs)

p = Playlist("Favourites")
p.add("Song A")
p.add("Song B")
p.add("Song C")

for song in p:
    print(song)

# Also works with list comprehensions, zip, enumerate, etc.
titles = [song.upper() for song in p]
print(titles)   # ['SONG A', 'SONG B', 'SONG C']
```

For a custom iterator that tracks state, implement both `__iter__` (returns `self`) and `__next__` (returns the next value, raises `StopIteration` when done):

```
class Countdown:
    def __init__(self, start):
        self.start   = start
        self.current = start

    def __iter__(self):
        self.current = self.start   # reset on each iteration
        return self

    def __next__(self):
        if self.current < 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value
```

```
for n in Countdown(5):
    print(n, end=" ")    # 5 4 3 2 1 0
```

19.3.5 Arithmetic operators

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __rmul__(self, scalar):
        return self.__mul__(scalar)    # handles scalar * vector

    def __neg__(self):
        return Vector(-self.x, -self.y)

    def __abs__(self):
        return (self.x**2 + self.y**2) ** 0.5

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(1, 4)

print(v1 + v2)      # Vector(3, 7)
print(v1 - v2)      # Vector(1, -1)
print(v1 * 3)       # Vector(6, 9)
print(3 * v1)       # Vector(6, 9)  --- via __rmul__
print(-v1)          # Vector(-2, -3)
print(abs(v1))      # 3.605551275...
print(v1 == v2)     # False
```

19.3.6 Comparison operators

```
class Product:
    def __init__(self, name, price):
        self.name  = name
        self.price = price

    def __eq__(self, other):
        return self.price == other.price

    def __lt__(self, other):
        return self.price < other.price

    def __le__(self, other):
        return self.price <= other.price

    def __gt__(self, other):
        return self.price > other.price

    def __ge__(self, other):
        return self.price >= other.price

    def __repr__(self):
        return f"Product({self.name!r}, ${self.price:.2f})"

apple  = Product("Apple",  1.20)
banana = Product("Banana", 0.50)
cherry = Product("Cherry", 3.00)

print(apple > banana)     # True
print(cherry < apple)     # False
print(apple == apple)     # True

products = [cherry, apple, banana]
products.sort()   # sort() uses __lt__
print(products)   # [Product('Banana', $0.50), Product('Apple', $1.20), Product

print(min(products))    # Product('Banana', $0.50)
print(max(products))    # Product('Cherry', $3.00)
```

Once you define comparison dunders, `sort()`, `min()`, `max()`, and `sorted()` all work on your objects automatically.

Tip: Instead of implementing all six comparison methods manually, use `@functools.total_ordering`. Define `__eq__` and one of `__lt__`, `__le__`, `__gt__`, `__ge__`, and Python fills in the rest:

```
from functools import total_ordering

@total_ordering
class Product:
    def __init__(self, name, price):
        self.name  = name
        self.price = price

    def __eq__(self, other):
        return self.price == other.price

    def __lt__(self, other):
        return self.price < other.price
```

19.3.7 `__call__` — Making objects callable

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, value):
        return value * self.factor


double = Multiplier(2)
triple = Multiplier(3)

print(double(5))     # 10
print(triple(7))     # 21

numbers = [1, 2, 3, 4, 5]
doubled = list(map(double, numbers))
print(doubled)    # [2, 4, 6, 8, 10]
```

`double` is an object, but it acts like a function. You can use it anywhere a function is expected.

19.3.8 `__enter__` and `__exit__` — Context managers

Define these to make your object work with `with`:

```
class Timer:
    """Context manager that times a block of code."""
    import time as _time
```

```
    def __enter__(self):
        self._start = self._time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        elapsed = self._time.time() - self._start
        print(f"Elapsed: {elapsed:.4f} seconds")
        return False    # don't suppress exceptions


with Timer():
    total = sum(n**2 for n in range(1_000_000))
    print(f"Sum: {total}")
```

Output:

```
Sum: 333332833333500000
Elapsed: 0.0842 seconds
```

The `__exit__` method receives exception info if one occurred. Return `True` to suppress the exception, `False` (or `None`) to let it propagate.

19.4 Class Methods and Static Methods

19.4.1 `@classmethod` — Works on the class, not an instance

A class method receives the *class itself* as the first argument (conventionally named `cls`), not an instance. Use it for alternative constructors — different ways to create an object.

```
class Date:
    def __init__(self, year, month, day):
        self.year  = year
        self.month = month
        self.day   = day

    @classmethod
    def from_string(cls, date_string):
        """Create a Date from a string like '2026-03-09'."""
        year, month, day = map(int, date_string.split("-"))
        return cls(year, month, day)

    @classmethod
```

```
    def today(cls):
        """Create a Date for today."""
        from datetime import date
        d = date.today()
        return cls(d.year, d.month, d.day)

    def __repr__(self):
        return f"Date({self.year}, {self.month:02d}, {self.day:02d})"

    def __str__(self):
        return f"{self.year}-{self.month:02d}-{self.day:02d}"

d1 = Date(2026, 3, 9)
d2 = Date.from_string("2026-07-04")
d3 = Date.today()

print(d1)    # 2026-03-09
print(d2)    # 2026-07-04
print(d3)    # today's date
```

`from_string` and `today` are alternative ways to create a `Date`. They're on the class because they don't need an existing instance — they *create* an instance.

19.4.2 `@staticmethod` — Belongs to the class, needs neither `self` nor `cls`

A static method is a plain function that lives in the class's namespace for organizational reasons. It receives neither `self` nor `cls`.

```
class MathUtils:
    @staticmethod
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True

    @staticmethod
    def clamp(value, minimum, maximum):
        return max(minimum, min(maximum, value))

    @staticmethod
    def lerp(a, b, t):
        """Linear interpolation between a and b at position t (0.0--1.0)."""
```

```
        return a + (b - a) * t


print(MathUtils.is_prime(17))       # True
print(MathUtils.clamp(150, 0, 100))  # 100
print(MathUtils.lerp(0, 100, 0.25))  # 25.0
```

Use `@staticmethod` for utility functions that are logically related to the class but don't need access to the instance or class.

19.5 Dataclasses — The Modern Shortcut

Writing `__init__`, `__repr__`, and `__eq__` for every class gets repetitive. The `@dataclass` decorator generates them automatically from annotated class variables.

```
from dataclasses import dataclass, field


@dataclass
class Point:
    x: float
    y: float

    def distance_to(self, other):
        return ((self.x - other.x)**2 + (self.y - other.y)**2) ** 0.5


p1 = Point(3.0, 4.0)
p2 = Point(0.0, 0.0)

print(p1)                       # Point(x=3.0, y=4.0)
print(p1 == Point(3.0, 4.0))    # True  --- __eq__ is generated
print(p1.distance_to(p2))       # 5.0
```

19.5.1 Default values and field options

```
from dataclasses import dataclass, field
from datetime import datetime


@dataclass
class Employee:
```

```
    name:        str
    department:  str
    salary:      float = 50_000.0
    start_date:  str   = field(default_factory=lambda: datetime.now().strftime("%Y
    skills:      list  = field(default_factory=list)   # mutable default --- must

    def add_skill(self, skill):
        self.skills.append(skill)

    def annual_bonus(self, percent=10):
        return self.salary * percent / 100


emp = Employee("Alice", "Engineering", 95_000.0)
emp.add_skill("Python")
emp.add_skill("FastAPI")

print(emp)
# Employee(name='Alice', department='Engineering', salary=95000.0, start_date='202

print(emp.annual_bonus())   # 9500.0
```

19.5.2 Frozen dataclasses — immutable objects

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Color:
    r: int
    g: int
    b: int

    def to_hex(self):
        return f"#{self.r:02X}{self.g:02X}{self.b:02X}"


red   = Color(255, 0, 0)
green = Color(0, 255, 0)
blue  = Color(0, 0, 255)

print(red.to_hex())    # #FF0000
print(green.to_hex())  # #00FF00
```

```
# Frozen --- can't change
red.r = 100   # FrozenInstanceError
```

Frozen dataclasses are immutable and hashable — they can be used as dictionary keys or in sets.

19.5.3 Ordering with dataclasses

```
@dataclass(order=True)
class Student:
    gpa:  float
    name: str


students = [
    Student(3.8, "Alice"),
    Student(3.5, "Bob"),
    Student(3.9, "Carlos"),
]

students.sort()
for s in students:
    print(s)
# Student(gpa=3.5, name='Bob')
# Student(gpa=3.8, name='Alice')
# Student(gpa=3.9, name='Carlos')
```

`order=True` generates all comparison methods based on the fields in declaration order.

19.6 Putting It All Together: A Full Playlist Class

```
from dataclasses import dataclass, field
from functools import total_ordering


@dataclass(frozen=True)
@total_ordering
class Song:
    title:  str
    artist: str
    duration: float   # seconds
```

```
    @property
    def duration_str(self):
        m, s = divmod(int(self.duration), 60)
        return f"{m}:{s:02d}"

    def __lt__(self, other):
        return self.duration < other.duration

    def __str__(self):
        return f"{self.title} --- {self.artist} ({self.duration_str})"


class Playlist:
    def __init__(self, name):
        self.name   = name
        self._songs = []

    def add(self, song):
        if not isinstance(song, Song):
            raise TypeError(f"Expected Song, got {type(song).__name__}.")
        self._songs.append(song)

    def remove(self, title):
        before = len(self._songs)
        self._songs = [s for s in self._songs if s.title != title]
        if len(self._songs) == before:
            raise ValueError(f"No song titled {title!r} found.")

    @property
    def total_duration(self):
        return sum(s.duration for s in self._songs)

    @property
    def total_duration_str(self):
        total = int(self.total_duration)
        h, rem = divmod(total, 3600)
        m, s   = divmod(rem, 60)
        return f"{h}h {m}m {s}s" if h else f"{m}m {s}s"

    def shuffle(self):
        import random
        shuffled = self._songs.copy()
        random.shuffle(shuffled)
```

```
        return Playlist._from_songs(self.name + " (shuffled)", shuffled)

    @classmethod
    def _from_songs(cls, name, songs):
        p = cls(name)
        p._songs = songs
        return p

    def __len__(self):
        return len(self._songs)

    def __iter__(self):
        return iter(self._songs)

    def __getitem__(self, index):
        return self._songs[index]

    def __contains__(self, song):
        return song in self._songs

    def __str__(self):
        return f"Playlist({self.name!r}, {len(self)} songs, {self.total_duratio

    def __repr__(self):
        return f"Playlist({self.name!r})"


# Build a playlist
p = Playlist("Evening Chill")

p.add(Song("Moonlight Sonata",  "Beethoven",    900.0))
p.add(Song("Clair de Lune",     "Debussy",      300.0))
p.add(Song("Gymnopédie No. 1",  "Satie",        195.0))
p.add(Song("The Four Seasons",  "Vivaldi",     2400.0))
p.add(Song("Für Elise",         "Beethoven",    175.0))

print(p)
print(f"Longest: {max(p)}")
print(f"Shortest: {min(p)}")

print(f"\nAll songs:")
for i, song in enumerate(p, 1):
    print(f"  {i}. {song}")
```

```
beethoven_songs = [s for s in p if s.artist == "Beethoven"]
print(f"\nBeethoven: {len(beethoven_songs)} song(s)")

moon = Song("Moonlight Sonata", "Beethoven", 900.0)
print(f"\n'{moon.title}' in playlist: {moon in p}")
```

Output:

```
Playlist('Evening Chill', 5 songs, 65m 50s)
Longest: The Four Seasons --- Vivaldi (40:00)
Shortest: Für Elise --- Beethoven (2:55)

All songs:
  1. Moonlight Sonata --- Beethoven (15:00)
  2. Clair de Lune --- Debussy (5:00)
  3. Gymnopédie No. 1 --- Satie (3:15)
  4. The Four Seasons --- Vivaldi (40:00)
  5. Für Elise --- Beethoven (2:55)

Beethoven: 2 song(s)
'Moonlight Sonata' in playlist: True
```

`Song` is a frozen dataclass with `@total_ordering`. `Playlist` implements `__len__`, `__iter__`, `__getitem__`, `__contains__`, `__str__`, `__repr__`, a classmethod, and a computed property. The result feels completely native.

19.7 Dunder Methods Quick Reference

Dunder	Triggered by
`__init__`	`ClassName()`
`__str__`	`str(obj)`, `print(obj)`, f-strings
`__repr__`	`repr(obj)`, REPL display
`__len__`	`len(obj)`
`__getitem__`	`obj[key]`
`__setitem__`	`obj[key] = value`
`__delitem__`	`del obj[key]`
`__contains__`	`item in obj`
`__iter__`	`for x in obj`
`__next__`	`next(obj)`
`__call__`	`obj(args)`
`__add__`	`obj + other`
`__sub__`	`obj - other`

Dunder	Triggered by
`__mul__`	`obj * other`
`__rmul__`	`other * obj`
`__neg__`	`-obj`
`__abs__`	`abs(obj)`
`__eq__`	`obj == other`
`__lt__`	`obj < other`
`__le__`	`obj <= other`
`__gt__`	`obj > other`
`__ge__`	`obj >= other`
`__enter__`	`with obj:`
`__exit__`	end of `with` block
`__bool__`	`bool(obj)`, `if obj:`
`__hash__`	`hash(obj)`, dict key, set member

19.8 What You Learned in This Chapter

- `_name` is private by convention. `__name` triggers name mangling — use sparingly.
- `@property` exposes a method as an attribute. Add `.setter` and `.deleter` for full control.
- Computed properties are always fresh — ideal for derived values.
- Dunder methods make objects feel native: `__len__`, `__getitem__`, `__iter__`, `__contains__`.
- Arithmetic dunders: `__add__`, `__sub__`, `__mul__`, `__rmul__`, `__neg__`, `__abs__`.
- Comparison dunders: `__eq__`, `__lt__`, etc. `@total_ordering` fills in the rest from two methods.
- `__call__` makes objects callable like functions.
- `__enter__` / `__exit__` make objects work with `with` — context managers.
- `@classmethod` — receives `cls`, used for alternative constructors.
- `@staticmethod` — no `self` or `cls`, utility functions in the class namespace.
- `@dataclass` generates `__init__`, `__repr__`, `__eq__` automatically. `frozen=True` makes objects immutable and hashable. `order=True` generates comparison methods.

19.9 What's Next?

Part 3's final chapter is Chapter 20 — an intermediate OOP project that puts everything together: a mini library system with multiple classes, inheritance, encapsulation, properties, and file persistence. It's the capstone for everything in Chapters 17, 18, and 19.

Your turn: Build a `Matrix` class that wraps a 2D list of numbers. Implement: `__init__(rows, cols, fill=0)`, `__getitem__` and `__setitem__` using `(row, col)` tuple keys, `__add__` for matrix addition, `__mul__` for scalar multiplication, `__str__` for a formatted grid display, `__len__` returning the number of rows, and a `transpose()` method. Test it by creating two 3x3 matrices, adding them, multiplying by a scalar, and printing the transpose.

Chapter 20

Chapter 20: Intermediate OOP Project — Build a Mini Library System

The last three chapters gave you the full OOP toolkit: classes, inheritance, polymorphism, encapsulation, properties, and dunder methods. Now it's time to use all of it together.

This chapter builds a complete mini library system from scratch — the kind of system a small library or bookstore might use. You'll design the classes, implement them step by step, connect them, add file persistence, and end up with a real working program.

Unlike the quick examples in previous chapters, this one is a full project. Take your time with it. Read each step, understand the design decisions, then type it out yourself.

20.1 What We're Building

A library management system with:

- **Books** — with title, author, ISBN, and availability status
- **Digital books** — extend Book with file format and download size
- **Members** — registered library members who borrow and return books
- **Library** — the main system that manages everything

Features: - Check books in and out - Search by title, author, or ISBN - Show a member's borrowing history - Save and load the entire state to/from a JSON

file - A simple menu-driven interface

20.2 Step 1: Design the Classes First

Before writing code, sketch what each class needs:

```
Book
├── isbn (str)          --- unique identifier
├── title (str)
├── author (str)
├── year (int)
├── available (bool)    --- is it on the shelf?
├── borrower (str|None) --- member_id if borrowed
├── checkout(member_id) -> None
├── return_book()       -> None
└── to_dict() / from_dict() --- for JSON persistence

DigitalBook(Book)
├── file_format (str)   --- "EPUB", "PDF", etc.
├── size_mb (float)
└── download()          -> str

Member
├── member_id (str)
├── name (str)
├── email (str)
├── borrowed (list[str]) --- list of ISBNs currently borrowed
├── history (list[dict]) --- full borrowing history
├── borrow(isbn)         -> None
├── return_book(isbn)    -> None
└── to_dict() / from_dict()

Library
├── name (str)
├── books (dict[isbn -> Book])
├── members (dict[member_id -> Member])
├── add_book(book)
├── remove_book(isbn)
├── register_member(member)
├── checkout(member_id, isbn)
├── return_book(member_id, isbn)
├── search(query)        -> list[Book]
```

```
├── save(filepath)
└── load(filepath)
```

Good design starts on paper (or a text file), not in code. Now let's build it.

20.3 Step 2: The `Book` Class

```python
# library_system.py

from datetime import datetime


class Book:
    """Represents a physical book in the library."""

    def __init__(self, isbn, title, author, year):
        self.isbn      = isbn
        self.title     = title
        self.author    = author
        self.year      = year
        self.available = True
        self.borrower  = None   # member_id of current borrower

    def checkout(self, member_id):
        """Mark the book as checked out by member_id."""
        if not self.available:
            raise ValueError(
                f"'{self.title}' is already checked out by member {self.borrowe
            )
        self.available = False
        self.borrower  = member_id

    def return_book(self):
        """Mark the book as returned."""
        if self.available:
            raise ValueError(f"'{self.title}' is not currently checked out.")
        self.available = True
        self.borrower  = None

    def to_dict(self):
        """Serialize to a dictionary for JSON storage."""
        return {
```

```
            "type":      "Book",
            "isbn":      self.isbn,
            "title":     self.title,
            "author":    self.author,
            "year":      self.year,
            "available": self.available,
            "borrower":  self.borrower,
        }

    @classmethod
    def from_dict(cls, data):
        """Reconstruct a Book from a dictionary."""
        book           = cls(data["isbn"], data["title"], data["author"], data["ye
        book.available = data["available"]
        book.borrower  = data["borrower"]
        return book

    def __str__(self):
        status = "Available" if self.available else f"Checked out (member: {self.b
        return f"[{self.isbn}] {self.title} by {self.author} ({self.year}) --- {st

    def __repr__(self):
        return f"Book({self.isbn!r}, {self.title!r})"

    def __eq__(self, other):
        if not isinstance(other, Book):
            return NotImplemented
        return self.isbn == other.isbn
```

Test it:

```
b1 = Book("978-0-06-112008-4", "To Kill a Mockingbird", "Harper Lee", 1960)
b2 = Book("978-0-7432-7356-5", "1984", "George Orwell", 1949)

print(b1)
b1.checkout("M001")
print(b1)
b1.return_book()
print(b1)
```

Output:

```
[978-0-06-112008-4] To Kill a Mockingbird by Harper Lee (1960) --
- Available
[978-0-06-112008-4] To Kill a Mockingbird by Harper Lee (1960) --
- Checked out (member: M001)
```

```
[978-0-06-112008-4] To Kill a Mockingbird by Harper Lee (1960) --
- Available
```

20.4 Step 3: The `DigitalBook` Subclass

```
class DigitalBook(Book):
    """A book available in digital format --- always available (no checkout lim

    VALID_FORMATS = {"EPUB", "PDF", "MOBI", "AZW3"}

    def __init__(self, isbn, title, author, year, file_format, size_mb):
        super().__init__(isbn, title, author, year)
        if file_format.upper() not in self.VALID_FORMATS:
            raise ValueError(
                f"Invalid format {file_format!r}. Must be one of {self.VALID_FO
            )
        self.file_format = file_format.upper()
        self.size_mb     = float(size_mb)

    def checkout(self, member_id):
        """Digital books can be borrowed by multiple people simultaneously."""
        self.borrower = member_id   # track last downloader
        # available stays True --- digital copies are unlimited

    def return_book(self):
        """Returning a digital book --- nothing to do physically."""
        pass   # nothing changes for digital

    def download(self):
        """Simulate downloading the book."""
        return (
            f"Downloading '{self.title}' ({self.file_format}, "
            f"{self.size_mb:.1f} MB)... Done."
        )

    def to_dict(self):
        data = super().to_dict()
        data["type"]        = "DigitalBook"
        data["file_format"] = self.file_format
        data["size_mb"]     = self.size_mb
        return data
```

```
    @classmethod
    def from_dict(cls, data):
        book = cls(
            data["isbn"], data["title"], data["author"], data["year"],
            data["file_format"], data["size_mb"]
        )
        book.available = data["available"]
        book.borrower  = data["borrower"]
        return book

    def __str__(self):
        return (
            f"[{self.isbn}] {self.title} by {self.author} ({self.year}) "
            f"[{self.file_format}, {self.size_mb:.1f} MB]"
        )

    def __repr__(self):
        return f"DigitalBook({self.isbn!r}, {self.title!r}, {self.file_format!r})"
```

Test it:

```
db = DigitalBook("978-0-14-028329-7", "The Great Gatsby", "F. Scott Fitzgerald",
                 1925, "EPUB", 0.8)
print(db)
print(db.download())
```

Output:

```
[978-0-14-028329-7] The Great Gatsby by F. Scott Fitzgerald (1925) [EPUB, 0.8 MB]
Downloading 'The Great Gatsby' (EPUB, 0.8 MB)... Done.
```

20.5 Step 4: The `Member` Class

```
class Member:
    """A registered library member."""

    MAX_BOOKS = 5   # class attribute --- borrowing limit

    def __init__(self, member_id, name, email):
        self.member_id = member_id
        self.name      = name
        self.email     = email
        self._borrowed = []    # list of ISBNs currently borrowed
        self._history  = []    # full history of all transactions
```

```
    @property
    def borrowed(self):
        """Read-only view of currently borrowed ISBNs."""
        return list(self._borrowed)

    @property
    def borrowed_count(self):
        return len(self._borrowed)

    def borrow(self, isbn):
        if self.borrowed_count >= self.MAX_BOOKS:
            raise ValueError(
                f"{self.name} has reached the borrowing limit of {self.MAX_BOOK
            )
        if isbn in self._borrowed:
            raise ValueError(f"{self.name} already has book {isbn!r}.")
        self._borrowed.append(isbn)
        self._history.append({
            "action":    "borrowed",
            "isbn":      isbn,
            "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M"),
        })

    def return_book(self, isbn):
        if isbn not in self._borrowed:
            raise ValueError(f"{self.name} does not have book {isbn!r} checked
        self._borrowed.remove(isbn)
        self._history.append({
            "action":    "returned",
            "isbn":      isbn,
            "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M"),
        })

    def print_history(self):
        print(f"\nBorrowing history for {self.name} ({self.member_id}):")
        if not self._history:
            print("  No transactions yet.")
            return
        for entry in self._history:
            print(f"  [{entry['timestamp']}] {entry['action'].capitalize()}: {e

    def to_dict(self):
        return {
```

```
            "member_id": self.member_id,
            "name":      self.name,
            "email":     self.email,
            "borrowed":  self._borrowed,
            "history":   self._history,
        }

    @classmethod
    def from_dict(cls, data):
        m            = cls(data["member_id"], data["name"], data["email"])
        m._borrowed  = data["borrowed"]
        m._history   = data["history"]
        return m

    def __str__(self):
        return (f"Member({self.member_id}) {self.name} <{self.email}> "
                f"--- {self.borrowed_count}/{self.MAX_BOOKS} books")

    def __repr__(self):
        return f"Member({self.member_id!r}, {self.name!r})"
```

20.6 Step 5: The `Library` Class

```
import json
from pathlib import Path


class Library:
    """The main library system."""

    def __init__(self, name):
        self.name    = name
        self._books  = {}     # isbn -> Book
        self._members = {}    # member_id -> Member

    # ── Book management ──────────────────────────────────────────

    def add_book(self, book):
        if book.isbn in self._books:
            raise ValueError(f"Book with ISBN {book.isbn!r} already exists.")
        self._books[book.isbn] = book
        print(f"Added: {book.title!r}")
```

```
    def remove_book(self, isbn):
        book = self._get_book(isbn)
        if not book.available:
            raise ValueError(f"Cannot remove '{book.title}' --- it is currently
        del self._books[isbn]
        print(f"Removed: {book.title!r}")

    def _get_book(self, isbn):
        if isbn not in self._books:
            raise KeyError(f"No book with ISBN {isbn!r} found.")
        return self._books[isbn]

    # ── Member management ──────────────────────────────────────────────

    def register_member(self, member):
        if member.member_id in self._members:
            raise ValueError(f"Member {member.member_id!r} already registered."
        self._members[member.member_id] = member
        print(f"Registered member: {member.name}")

    def _get_member(self, member_id):
        if member_id not in self._members:
            raise KeyError(f"No member with ID {member_id!r} found.")
        return self._members[member_id]

    # ── Transactions ──────────────────────────────────────────────────

    def checkout(self, member_id, isbn):
        member = self._get_member(member_id)
        book   = self._get_book(isbn)
        book.checkout(member_id)
        member.borrow(isbn)

        if isinstance(book, DigitalBook):
            print(book.download())
        else:
            print(f"[x] '{book.title}' checked out to {member.name}.")

    def return_book(self, member_id, isbn):
        member = self._get_member(member_id)
        book   = self._get_book(isbn)
        book.return_book()
        member.return_book(isbn)
```

```
        print(f"[x] '{book.title}' returned by {member.name}.")

    # ── Search ──────────────────────────────────────────────────────

    def search(self, query):
        """Search books by title, author, or ISBN (case-insensitive)."""
        q = query.lower()
        return [
            book for book in self._books.values()
            if q in book.title.lower()
            or q in book.author.lower()
            or q in book.isbn
        ]

    def available_books(self):
        """Return all available (not checked-out) physical books."""
        return [b for b in self._books.values()
                if b.available and not isinstance(b, DigitalBook)]

    # ── Display ─────────────────────────────────────────────────────

    def print_catalogue(self):
        print(f"\n{'='*60}")
        print(f"  {self.name} --- Catalogue ({len(self._books)} books)")
        print(f"{'='*60}")
        for book in sorted(self._books.values(), key=lambda b: b.title):
            marker = "" if isinstance(book, DigitalBook) else ""
            print(f"  {marker} {book}")
        print()

    def print_members(self):
        print(f"\n{'='*60}")
        print(f"  Registered Members ({len(self._members)})")
        print(f"{'='*60}")
        for member in self._members.values():
            print(f"  {member}")
        print()

    def print_summary(self):
        total      = len(self._books)
        digital    = sum(1 for b in self._books.values() if isinstance(b, DigitalB
        physical   = total - digital
        available  = sum(1 for b in self._books.values()
                         if b.available and not isinstance(b, DigitalBook))
```

```
        checked_out = physical - available

        print(f"\n{self.name} Summary:")
        print(f"  Total books:     {total}  ({physical} physical, {digital} dig
        print(f"  Available:       {available}")
        print(f"  Checked out:     {checked_out}")
        print(f"  Members:         {len(self._members)}")

    # ── Persistence ──────────────────────────────────────────────────────

    def save(self, filepath="library.json"):
        data = {
            "name":    self.name,
            "books":   [book.to_dict() for book in self._books.values()],
            "members": [m.to_dict() for m in self._members.values()],
        }
        with open(filepath, "w", encoding="utf-8") as f:
            json.dump(data, f, indent=2)
        print(f"Library saved to {filepath}.")

    @classmethod
    def load(cls, filepath="library.json"):
        path = Path(filepath)
        if not path.exists():
            raise FileNotFoundError(f"No save file found at {filepath!r}.")

        with open(filepath, "r", encoding="utf-8") as f:
            data = json.load(f)

        library = cls(data["name"])

        for book_data in data["books"]:
            if book_data["type"] == "DigitalBook":
                book = DigitalBook.from_dict(book_data)
            else:
                book = Book.from_dict(book_data)
            library._books[book.isbn] = book

        for member_data in data["members"]:
            member = Member.from_dict(member_data)
            library._members[member.member_id] = member

        print(f"Library loaded from {filepath}.")
        return library
```

20.7 Step 6: The Menu Interface

```
def run_library():
    """Interactive menu-driven library interface."""

    SAVE_FILE = "library.json"

    # Load existing data or start fresh
    try:
        lib = Library.load(SAVE_FILE)
    except FileNotFoundError:
        lib = Library("City Library")
        _seed_data(lib)

    while True:
        print(f"\n{'—'*40}")
        print(f"  {lib.name}")
        print(f"{'—'*40}")
        print("  1. View catalogue")
        print("  2. Search books")
        print("  3. Checkout a book")
        print("  4. Return a book")
        print("  5. View member history")
        print("  6. View summary")
        print("  7. Save and quit")
        print(f"{'—'*40}")

        choice = input("Choice: ").strip()

        try:
            if choice == "1":
                lib.print_catalogue()

            elif choice == "2":
                query = input("Search (title / author / ISBN): ").strip()
                results = lib.search(query)
                if results:
                    print(f"\n{len(results)} result(s):")
                    for book in results:
                        print(f"  {book}")
                else:
```

```
                    print("No books found.")

            elif choice == "3":
                member_id = input("Member ID: ").strip()
                isbn      = input("Book ISBN: ").strip()
                lib.checkout(member_id, isbn)

            elif choice == "4":
                member_id = input("Member ID: ").strip()
                isbn      = input("Book ISBN: ").strip()
                lib.return_book(member_id, isbn)

            elif choice == "5":
                member_id = input("Member ID: ").strip()
                member = lib._get_member(member_id)
                member.print_history()

            elif choice == "6":
                lib.print_summary()

            elif choice == "7":
                lib.save(SAVE_FILE)
                print("Goodbye!")
                break

            else:
                print("Invalid choice. Enter 1--7.")

        except (KeyError, ValueError) as e:
            print(f"Error: {e}")


def _seed_data(lib):
    """Populate a fresh library with sample data."""
    books = [
        Book("978-0-06-112008-4", "To Kill a Mockingbird", "Harper Lee",
        Book("978-0-7432-7356-5", "1984",                  "George Orwell",
        Book("978-0-14-028329-7", "The Great Gatsby",      "F. Scott Fitzgerald
        Book("978-0-06-093546-9", "Brave New World",       "Aldous Huxley",
        Book("978-0-7432-7357-2", "Animal Farm",           "George Orwell",
        DigitalBook("978-0-14-303943-3", "Crime and Punishment",
                    "Fyodor Dostoevsky", 1866, "EPUB", 1.2),
        DigitalBook("978-0-14-044913-6", "The Odyssey",
                    "Homer", -800, "PDF", 2.1),
```

```
    ]

    members = [
        Member("M001", "Alice Smith",   "alice@example.com"),
        Member("M002", "Bob Jones",     "bob@example.com"),
        Member("M003", "Carlos Rivera", "carlos@example.com"),
    ]

    for book in books:
        lib.add_book(book)

    for member in members:
        lib.register_member(member)

    print("\nLibrary initialized with sample data.")
```

20.8 Step 7: Run It

```
if __name__ == "__main__":
    run_library()
```

Save everything as `library_system.py`. Run it:

```
python library_system.py
```

Walk through these steps to test every feature:

1. View catalogue — see all books
2. Search "Orwell" — find both his books
3. Checkout ISBN `978-0-7432-7356-5` for member `M001` — 1984 checked out
4. View catalogue again — see "Checked out" status
5. Checkout `978-0-14-303943-3` for member `M001` — digital book downloads
6. View member history for `M001` — see both transactions
7. Return `978-0-7432-7356-5` for `M001` — 1984 back on shelf
8. Save and quit — `library.json` written
9. Run again — data loads from file, state preserved

20.9 What This Project Demonstrates

Look at every OOP concept from the last three chapters in this one program:

Concept	Where
Classes with `__init__`	`Book`, `Member`, `Library`
Inheritance	`DigitalBook(Book)`
Method overriding	`DigitalBook.checkout()`, `DigitalBook.return_book()`
`super()`	`DigitalBook.__init__`, `DigitalBook.to_dict()`
`isinstance()`	Library sorts digital vs physical, triggers download
Encapsulation	`_borrowed`, `_history`, `_books`, `_members`
`@property`	`Member.borrowed`, `Member.borrowed_count`
Class attributes	`Member.MAX_BOOKS`
`@classmethod`	`Book.from_dict()`, `Library.load()`
`__str__` / `__repr__`	All four classes
`__eq__`	`Book.__eq__` compares by ISBN
Error handling	`try/except` in every transaction
File persistence	`json.dump` / `json.load` via `to_dict`/`from_dict`
Polymorphism	`Library.checkout()` calls `book.checkout()` regardless of type

A real system. Real design decisions. Real OOP.

20.10 Extensions to Try

Once the base system works, push further:

Medium extensions: - Add a due date when a book is checked out (use `datetime` + `timedelta`). Flag overdue books. - Add a `Librarian` subclass of `Member` with permission to add and remove books. - Add a `search_by_availability()` method that returns only currently available books.

Harder extensions: - Replace the JSON file with an SQLite database using `sqlite3` (covered in Chapter 27). - Add a `Reservation` system — members can reserve a book that's currently checked out. - Build a web API for the library using FastAPI (covered in Chapter 41).

20.11 What You Learned in This Chapter

- Real OOP projects start with design — sketch your classes and their relationships before writing code.
- Each class has a single responsibility: `Book` manages book state, `Member` manages member state, `Library` coordinates them.
- Inheritance (`DigitalBook`) adds or overrides behavior without duplicating code.
- Serialization (`to_dict` / `from_dict`) is a clean pattern for JSON persistence.
- `@classmethod` with `from_dict` creates alternative constructors — common in real libraries.
- `isinstance()` in `Library.checkout()` enables different behavior for physical vs digital books without `if` chains everywhere.
- Error handling in every transaction keeps the program alive when something goes wrong.
- A menu loop is the simplest way to build an interactive CLI program.

20.12 What's Next?

Part 3 is complete. You've gone from writing your first class to building a multi-class system with inheritance, persistence, and a working interface.

Part 4 starts now — **Functional Programming and Advanced Functions**. In Chapter 21 you'll learn about first-class functions and closures: what it means that functions are objects in Python, how to pass them around, and how closures let functions remember their surrounding environment. It's a completely different way of thinking about code — and it leads directly to decorators, generators, and some of Python's most elegant patterns.

Your turn: Add a due-date system. When a book is checked out, record the date. Add a `due_date` property to `Book` that returns 14 days after checkout. Add an `overdue_books()` method to `Library` that returns all books past their due date. Test it by manipulating the checkout date directly to simulate overdue books.

Chapter 21

Chapter 21: First-Class Functions and Closures

You know functions well — you've been writing them since Chapter 7. But there's something about Python functions that most beginners don't fully grasp until much later: **functions are objects**.

Not metaphorically. Literally. In Python, a function is a value — the same kind of value as an integer or a string. You can store a function in a variable, put it in a list, pass it to another function, or return it from a function. This is called **first-class functions**, and it unlocks a whole new way of writing code.

This chapter also covers **closures** — functions that remember the environment they were created in, even after that environment no longer exists. Closures are the foundation of decorators (Chapter 22), which are used everywhere in real Python code.

21.1 Functions Are Objects

Let's start with a simple demonstration:

```
def greet(name):
    return f"Hello, {name}!"

# Functions are objects --- you can assign them to variables
say_hello = greet

print(say_hello("Alice"))    # Hello, Alice!
print(greet is say_hello)    # True --- same object, two names
```

`greet` and `say_hello` both point to the same function object. Writing `say_hello = greet` doesn't call the function — it just creates another name for it.

A function has a type, attributes, and an identity, just like any other object:

```
def add(a, b):
    return a + b

print(type(add))          # <class 'function'>
print(add.__name__)       # add
print(add.__doc__)        # None (no docstring)
print(id(add))            # some memory address
```

21.2 Passing Functions as Arguments

Because functions are objects, you can pass them to other functions as arguments. A function that accepts or returns another function is called a **higher-order function**.

```
def apply(func, value):
    """Apply func to value and return the result."""
    return func(value)

def double(n):
    return n * 2

def square(n):
    return n ** 2

def negate(n):
    return -n

print(apply(double, 5))    # 10
print(apply(square, 5))    # 25
print(apply(negate, 5))    # -5
```

The same `apply` function does completely different things depending on which function you pass. The behavior is data — you supply it at call time.

21.2.1 A practical example: applying transformations to a list

```
def transform(items, func):
    """Return a new list with func applied to every item."""
    return [func(item) for item in items]

numbers = [1, 2, 3, 4, 5]

print(transform(numbers, double))    # [2, 4, 6, 8, 10]
print(transform(numbers, square))    # [1, 4, 9, 16, 25]
print(transform(numbers, negate))    # [-1, -2, -3, -4, -5]
```

This is exactly what Python's built-in `map()` does. You just built your own version.

21.2.2 Sorting with a key function

You've already used this pattern — `sorted()` and `.sort()` both accept a `key` function:

```
students = [
    {"name": "Alice",  "gpa": 3.8},
    {"name": "Bob",    "gpa": 3.5},
    {"name": "Carlos", "gpa": 3.9},
    {"name": "Diana",  "gpa": 3.7},
]

# Sort by GPA (ascending)
by_gpa = sorted(students, key=lambda s: s["gpa"])

# Sort by name
by_name = sorted(students, key=lambda s: s["name"])

# Sort by GPA descending
top_students = sorted(students, key=lambda s: s["gpa"], reverse=True)

for s in top_students:
    print(f"{s['name']}: {s['gpa']}")
```

Output:

```
Carlos: 3.9
Alice: 3.8
Diana: 3.7
Bob: 3.5
```

The `key` parameter *is* a function. Python calls it on each item to get the sort value. Any callable works here.

Your turn: Write a function `apply_all(value, *funcs)` that applies a list of functions to a value in sequence. For example, `apply_all(5, double, square, negate)` should return `-(5*2)^2 = -100`. Test it with several combinations.

21.3 Storing Functions in Data Structures

Functions can live in lists, dictionaries, and any other collection.

```
def add(a, b):      return a + b
def subtract(a, b): return a - b
def multiply(a, b): return a * b
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero.")
    return a / b

# Dictionary of operations
operations = {
    "+": add,
    "-": subtract,
    "*": multiply,
    "/": divide,
}

# A simple calculator using the dispatch table
def calculate(a, op, b):
    if op not in operations:
        raise ValueError(f"Unknown operator: {op!r}")
    return operations[op](a, b)

print(calculate(10, "+", 5))   # 15
print(calculate(10, "*", 3))   # 30
print(calculate(10, "/", 4))   # 2.5
```

This pattern — a dictionary that maps strings to functions — is called a **dispatch table**. It replaces a long `if/elif` chain with a clean lookup. You'll see it constantly in real code.

```
# List of validators --- run each one and collect errors
def is_not_empty(value):
```

```
    return len(value.strip()) > 0, "Field cannot be empty."

def is_long_enough(value):
    return len(value) >= 8, "Must be at least 8 characters."

def has_uppercase(value):
    return any(c.isupper() for c in value), "Must contain an uppercase letter."

def has_digit(value):
    return any(c.isdigit() for c in value), "Must contain a digit."


validators = [is_not_empty, is_long_enough, has_uppercase, has_digit]

def validate_password(password):
    errors = []
    for validator in validators:
        passed, message = validator(password)
        if not passed:
            errors.append(message)
    return errors


errors = validate_password("hello")
for e in errors:
    print(f"  [-] {e}")

print()
errors = validate_password("Hello123")
if not errors:
    print("  [x] Password is valid.")
```

Output:

```
  [-] Must be at least 8 characters.
  [-] Must contain an uppercase letter.
  [-] Must contain a digit.

  [x] Password is valid.
```

A list of validation functions, looped over. Adding a new rule means adding one function to the list — no `if` chain to modify.

21.4 Returning Functions from Functions

A function can return another function. This is the foundation of closures and decorators.

```
def make_multiplier(factor):
    """Return a function that multiplies its argument by factor."""
    def multiplier(n):
        return n * factor
    return multiplier    # return the function, don't call it


double = make_multiplier(2)
triple = make_multiplier(3)
times_ten = make_multiplier(10)

print(double(5))      # 10
print(triple(5))      # 15
print(times_ten(5))   # 50
```

`make_multiplier(2)` doesn't return `10` — it returns a *function* that multiplies by 2. That function is stored in `double` and called later.

Notice that `multiplier` uses `factor` from the outer function `make_multiplier`. Even after `make_multiplier` finishes running, the inner function *remembers* the value of `factor`. That's a **closure**.

21.5 Closures — Functions That Remember

A **closure** is a function that captures variables from its enclosing scope and remembers them even after that scope has ended.

```
def make_counter(start=0):
    count = start

    def counter():
        nonlocal count
        count += 1
        return count

    return counter


count_a = make_counter()
```

```
count_b = make_counter(10)

print(count_a())    # 1
print(count_a())    # 2
print(count_a())    # 3
print(count_b())    # 11
print(count_b())    # 12
print(count_a())    # 4  --- count_a has its own independent count
```

`count_a` and `count_b` are two separate closure instances. Each one has its own `count` variable, captured from its own call to `make_counter`. They don't interfere with each other.

The `nonlocal` keyword is required here — it tells Python that `count` refers to the variable from the enclosing function, not a new local variable.

What makes a closure?

A closure exists when: 1. There is a nested function (a function inside a function). 2. The inner function references a variable from the outer function. 3. The outer function returns the inner function.

The inner function *closes over* the outer variable — that's where the name comes from.

21.5.1 Inspecting a closure

```
def make_adder(n):
    def adder(x):
        return x + n
    return adder

add5  = make_adder(5)
add10 = make_adder(10)

print(add5(3))     # 8
print(add10(3))    # 13

# Inspect the closed-over variables
print(add5.__closure__)                       # (<cell at 0x...>,)
print(add5.__closure__[0].cell_contents)      # 5
print(add10.__closure__[0].cell_contents)     # 10
```

The `__closure__` attribute holds the captured variables. Each one is a "cell" — a box that holds the value.

21.6 Factory Functions

A **factory function** is a function that creates and returns other functions, customized by the arguments you pass. It's one of the most useful closure patterns.

```
def make_validator(min_val, max_val, name="value"):
    """Return a validator function for a specific range."""
    def validate(value):
        if not isinstance(value, (int, float)):
            raise TypeError(f"{name} must be a number.")
        if value < min_val or value > max_val:
            raise ValueError(
                f"{name} must be between {min_val} and {max_val}, got {value}."
            )
        return True
    return validate


validate_age      = make_validator(0,   120, "Age")
validate_score    = make_validator(0,   100, "Score")
validate_rating   = make_validator(1,     5, "Rating")
validate_temp_c   = make_validator(-273.15, 1e6, "Temperature")

try:
    validate_age(25)        # passes
    validate_score(105)     # raises ValueError
except ValueError as e:
    print(e)    # Score must be between 0 and 100, got 105.

try:
    validate_rating("five")  # raises TypeError
except TypeError as e:
    print(e)    # Rating must be a number.
```

Each validator is a specialized function created by the factory. They all share the same validation logic but have different ranges and names.

```
def make_formatter(prefix="", suffix="", width=0, fill=" "):
    """Return a string formatter function."""
    def format_value(value):
        text = f"{prefix}{value}{suffix}"
        return text.center(width, fill) if width else text
    return format_value

format_currency = make_formatter(prefix="$", suffix="", width=0)
format_percent  = make_formatter(prefix="", suffix="%")
```

```
format_centered = make_formatter(prefix="[ ", suffix=" ]", width=30, fill="—")

print(format_currency(1234.56))     # $1234.56
print(format_percent(87.5))         # 87.5%
print(format_centered("Python"))    # ————————[ Python ]————————
```

21.7 The `nonlocal` Keyword

`nonlocal` lets an inner function modify a variable from the enclosing (but not global) scope. Without it, assignment creates a new local variable.

```
def make_accumulator():
    total = 0

    def add(value):
        nonlocal total        # refer to the outer total
        total += value
        return total

    return add


acc = make_accumulator()
print(acc(10))    # 10
print(acc(5))     # 15
print(acc(20))    # 35
```

Without `nonlocal total`, `total += value` would raise `UnboundLocalError` — Python would treat `total` as a local variable that doesn't exist yet.

Compare to the `global` keyword — `nonlocal` goes one level up (to the enclosing function), while `global` goes all the way to the module level. Always prefer `nonlocal` for closures; use `global` only when truly necessary.

21.8 Closures vs Classes

Closures and classes often solve the same problem — maintaining state across multiple calls. Here's the same counter written both ways:

```
# Closure version
def make_counter(start=0, step=1):
    count = start
```

```
    def counter():
        nonlocal count
        count += step
        return count
    return counter

c = make_counter(0, 2)
print(c(), c(), c())    # 2 4 6


# Class version
class Counter:
    def __init__(self, start=0, step=1):
        self.count = start
        self.step  = step

    def __call__(self):
        self.count += self.step
        return self.count

c = Counter(0, 2)
print(c(), c(), c())    # 2 4 6
```

Both work. Choose based on complexity: - **Closure:** Simple, one function, minimal state. Less code, harder to inspect or extend. - **Class:** Multiple methods, more state, easier to test, inherit, or add features to.

21.9 Partial Application with Closures

Closures are a natural way to do **partial application** — fixing some arguments of a function to create a simpler, more specific version.

```
def power(base, exponent):
    return base ** exponent

def make_power(exponent):
    def powered(base):
        return power(base, exponent)
    return powered

square  = make_power(2)
cube    = make_power(3)
to_tenth = make_power(10)
```

```
print(square(5))      # 25
print(cube(3))        # 27
print(to_tenth(2))    # 1024
```

Python's `functools.partial` does this more cleanly (you saw it in Chapter 16). But understanding the closure version helps you understand *why* `partial` works the way it does.

21.10 Memoization with Closures

A **memoization** cache remembers the results of expensive function calls. If you call the function with the same arguments again, it returns the cached result instead of recalculating.

```
def memoize(func):
    """Wrap func with a memoization cache."""
    cache = {}

    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    wrapper.__name__ = func.__name__
    wrapper.cache    = cache   # expose the cache for inspection
    return wrapper


def slow_fibonacci(n):
    if n < 2:
        return n
    return slow_fibonacci(n - 1) + slow_fibonacci(n - 2)


fast_fibonacci = memoize(slow_fibonacci)

import time

start = time.time()
print(fast_fibonacci(35))    # 9227465
print(f"Time: {time.time() - start:.4f}s")
```

```
print(f"Cache size: {len(fast_fibonacci.cache)} entries")
```

`memoize` is a closure. The `cache` dictionary is created in `memoize`, captured by `wrapper`, and persists across all calls to `wrapper`. This is almost exactly what `@functools.lru_cache` does internally.

21.11 Putting It All Together: A Pipeline Builder

```
def pipeline(*funcs):
    """
    Return a function that passes its input through each function in sequence.
    The output of each function becomes the input of the next.
    """
    def run(value):
        for func in funcs:
            value = func(value)
        return value
    return run


# String processing pipeline
clean      = str.strip
lowercase  = str.lower
capitalize = str.capitalize

def remove_punctuation(text):
    return "".join(c for c in text if c.isalpha() or c.isspace())

def normalize_spaces(text):
    return " ".join(text.split())


clean_text = pipeline(clean, lowercase, remove_punctuation, normalize_spaces, capi

raw_inputs = [
    "  HELLO,  WORLD!!!  ",
    "   Python    is   AWESOME.  ",
    "  too many    spaces   and CAPS  ",
]

for raw in raw_inputs:
```

```
    print(f"Before: {raw!r}")
    print(f"After:  {clean_text(raw)!r}")
    print()
```

Output:

```
Before: '  HELLO,  WORLD!!!  '
After:  'Hello world'

Before: '   Python    is   AWESOME.  '
After:  'Python is awesome'

Before: '  too many    spaces   and CAPS  '
After:  'Too many spaces and caps'
```

`pipeline` is a factory function that returns a closure. Each pipeline captures a different set of functions in its `funcs` tuple. You can build as many pipelines as you need, each doing something different.

```
# Number processing pipeline
import math

normalize = lambda x: (x - min_val) / (max_val - min_val)

numbers = [10, 25, 5, 40, 15, 30]
min_val = min(numbers)
max_val = max(numbers)

process_numbers = pipeline(
    lambda x: x ** 2,
    lambda x: x - 100,
    abs,
    math.sqrt,
    lambda x: round(x, 2),
)

results = [process_numbers(n) for n in numbers]
print(results)    # [9.49, 5.0, 9.95, 17.32, 6.24, 13.23]
```

21.12 What You Learned in This Chapter

- Functions are first-class objects — they have type, identity, and can be stored in variables.

- **Higher-order functions** accept functions as arguments or return functions.
- A **dispatch table** (dict of functions) replaces long `if/elif` chains.
- Storing functions in a list lets you apply a pipeline of operations with a simple loop.
- A function can **return another function** — this is the foundation of closures and decorators.
- A **closure** is an inner function that captures variables from its enclosing scope, remembering them after the outer function has returned.
- `nonlocal` lets an inner function modify an enclosing variable. `global` reaches the module level.
- **Factory functions** create specialized functions from a common template.
- Closures are an alternative to classes for lightweight stateful functions.
- **Partial application** — fixing some arguments to create a more specific function — is naturally done with closures (or `functools.partial`).
- **Memoization** — caching function results — is a classic closure pattern, and the basis of `functools.lru_cache`.

21.13 What's Next?

Closures lead directly to one of Python's most powerful and widely-used features: **decorators**.

In Chapter 22 you'll learn what decorators are, how to write them, how to stack them, how to make them accept arguments, and how they're used everywhere in real Python code — for logging, timing, authentication, caching, retry logic, and more. If you understood closures, decorators will click immediately.

Your turn: Write a `retry(times)` factory function. It takes a number `times` and returns a decorator that wraps a function so it retries up to `times` times if it raises an exception. After all retries are exhausted, re-raise the last exception. Test it with a function that randomly fails (use `random.random() < 0.7` to simulate a 70% failure rate). Print a message each time it retries.

Chapter 22

Chapter 22: Decorators

In Chapter 21 you built a `memoize` function that wrapped another function with a cache. You wrote it like this:

```
fast_fibonacci = memoize(slow_fibonacci)
```

That's a decorator — you just didn't use the special syntax yet. A **decorator** is a function that takes another function, wraps it with extra behavior, and returns the wrapped version. Python gives you the `@` symbol to apply decorators cleanly.

This chapter shows you exactly how decorators work, how to write your own, how to stack them, how to pass arguments to them, and where they show up constantly in real Python code.

22.1 The Long Way First

Before the `@` syntax, let's see the mechanics explicitly:

```
def shout(func):
    """Wrap func so it prints 'START' before and 'DONE' after."""
    def wrapper(*args, **kwargs):
        print("START")
        result = func(*args, **kwargs)
        print("DONE")
        return result
    return wrapper


def greet(name):
    print(f"Hello, {name}!")
```

```
# Manually wrap
greet = shout(greet)

greet("Alice")
```

Output:

```
START
Hello, Alice!
DONE
```

`shout` takes a function, defines an inner `wrapper` that adds behavior before and after the original call, and returns `wrapper`. After `greet = shout(greet)`, `greet` *is* the wrapper. The original function is captured inside the closure.

22.2 The `@` Syntax

The `@decorator` line is exactly equivalent to `func = decorator(func)`. It's just cleaner:

```
@shout
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:

```
START
Hello, Alice!
DONE
```

Same result. Python processes `@shout` before the function is bound to any name. When Python sees:

```
@shout
def greet(name):
    ...
```

it runs this under the hood:

```
def greet(name):
    ...
greet = shout(greet)
```

That's the complete rule. No magic beyond that.

22.3 Preserving the Wrapped Function's Identity

There's one problem with naive decorators. After wrapping, the function loses its name and docstring:

```
def shout(func):
    def wrapper(*args, **kwargs):
        print("START")
        result = func(*args, **kwargs)
        print("DONE")
        return result
    return wrapper

@shout
def greet(name):
    """Greet someone warmly."""
    print(f"Hello, {name}!")

print(greet.__name__)    # wrapper  <- wrong!
print(greet.__doc__)     # None     <- wrong!
```

Fix this with `functools.wraps`:

```
import functools

def shout(func):
    @functools.wraps(func)    # copies __name__, __doc__, __module__, etc.
    def wrapper(*args, **kwargs):
        print("START")
        result = func(*args, **kwargs)
        print("DONE")
        return result
    return wrapper

@shout
def greet(name):
    """Greet someone warmly."""
    print(f"Hello, {name}!")

print(greet.__name__)    # greet  [x]
print(greet.__doc__)     # Greet someone warmly.  [x]
```

Always use `@functools.wraps(func)` inside your wrapper. It's one line, and it makes debugging, logging, and introspection work correctly. Consider it mandatory.

22.4 Useful Decorator Patterns

22.4.1 Timing

```
import functools
import time

def timer(func):
    """Print how long the function took to run."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start  = time.perf_counter()
        result = func(*args, **kwargs)
        end    = time.perf_counter()
        print(f"{func.__name__!r} took {end - start:.6f}s")
        return result
    return wrapper


@timer
def slow_sum(n):
    """Sum integers from 0 to n."""
    return sum(range(n))


print(slow_sum(10_000_000))
```

Output:

```
'slow_sum' took 0.312471s
3333333350000000
```

22.4.2 Logging

```
import functools

def log_calls(func):
    """Log every call: arguments in, result out."""
    @functools.wraps(func)
```

```
    def wrapper(*args, **kwargs):
        args_repr   = [repr(a) for a in args]
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]
        signature   = ", ".join(args_repr + kwargs_repr)
        print(f"CALL  {func.__name__}({signature})")
        result = func(*args, **kwargs)
        print(f"RETURN {func.__name__} -> {result!r}")
        return result
    return wrapper


@log_calls
def add(a, b):
    return a + b

@log_calls
def greet(name, loud=False):
    message = f"Hello, {name}!"
    return message.upper() if loud else message


add(3, 4)
greet("Alice")
greet("Bob", loud=True)
```

Output:

```
CALL  add(3, 4)
RETURN add -> 7
CALL  greet('Alice')
RETURN greet -> 'Hello, Alice!'
CALL  greet('Bob', loud=True)
RETURN greet -> 'HELLO, BOB!'
```

22.4.3 Retry on failure

```
import functools
import time

def retry(times=3, delay=0, exceptions=(Exception,)):
    """
    Retry the function up to `times` times if it raises one of `exceptions`.
    Wait `delay` seconds between retries.
    """
    def decorator(func):
```

```
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            last_error = None
            for attempt in range(1, times + 1):
                try:
                    return func(*args, **kwargs)
                except exceptions as e:
                    last_error = e
                    print(f"Attempt {attempt}/{times} failed: {e}")
                    if attempt < times and delay:
                        time.sleep(delay)
            raise last_error
        return wrapper
    return decorator


import random

@retry(times=5, exceptions=(ValueError,))
def unstable_parse(text):
    """Simulate a parser that sometimes fails."""
    if random.random() < 0.7:
        raise ValueError("Parse error (simulated)")
    return int(text)


try:
    result = unstable_parse("42")
    print(f"Result: {result}")
except ValueError:
    print("All retries exhausted.")
```

Notice that `retry` is a **decorator factory** — a function that *returns* a decorator. The three levels are:

1. `retry(times=5)` — outer function, accepts configuration, returns `decorator`
2. `decorator(func)` — middle function, accepts the function, returns `wrapper`
3. `wrapper(*args, **kwargs)` — inner function, runs when the decorated function is called

22.4.4 Input validation

```
import functools

def validate_positive(*param_names):
    """Raise ValueError if any named parameter is not positive."""
    def decorator(func):
        import inspect
        sig = inspect.signature(func)

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            bound = sig.bind(*args, **kwargs)
            bound.apply_defaults()
            for name in param_names:
                if name in bound.arguments:
                    value = bound.arguments[name]
                    if value <= 0:
                        raise ValueError(
                            f"Parameter {name!r} must be positive, got {value}.
                        )
            return func(*args, **kwargs)
        return wrapper
    return decorator


@validate_positive("width", "height")
def make_rectangle(width, height):
    return width * height

print(make_rectangle(4, 5))    # 20

try:
    make_rectangle(-1, 5)
except ValueError as e:
    print(e)    # Parameter 'width' must be positive, got -1.
```

22.4.5 Caching (your own `lru_cache`)

```
import functools

def simple_cache(func):
    """Cache results in a plain dictionary (no size limit)."""
    cache = {}
```

```
    @functools.wraps(func)
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    wrapper.cache_info  = lambda: f"{len(cache)} items cached"
    wrapper.cache_clear = lambda: cache.clear()
    return wrapper


@simple_cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


print(fibonacci(50))            # 12586269025
print(wrapper.cache_info())     # <- AttributeError (wrapper is not in scope)
print(fibonacci.cache_info())   # 51 items cached
```

For production use, prefer `@functools.lru_cache(maxsize=128)` — it's faster, thread-safe, and has a size limit to prevent unbounded memory growth.

22.5 Decorator Factories (Decorators with Arguments)

A plain decorator takes one argument — the function. When you need to pass configuration to the decorator, you need an extra level of nesting:

```
# Pattern:
def decorator_factory(config):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # use config here
            return func(*args, **kwargs)
        return wrapper
    return decorator

@decorator_factory(config_value)
```

```
def my_function():
    ...
```

The `@decorator_factory(config_value)` line calls `decorator_factory(config_value)` first, gets back `decorator`, and then applies `decorator` to `my_function`.

22.5.1 Rate limiter

```
import functools
import time
from collections import deque

def rate_limit(calls_per_second):
    """Allow at most `calls_per_second` calls per second."""
    min_interval = 1.0 / calls_per_second
    last_called   = [0.0]   # list so closure can mutate it

    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            elapsed = time.time() - last_called[0]
            to_wait = min_interval - elapsed
            if to_wait > 0:
                time.sleep(to_wait)
            last_called[0] = time.time()
            return func(*args, **kwargs)
        return wrapper
    return decorator


@rate_limit(calls_per_second=2)
def fetch_data(url):
    print(f"Fetching {url} at {time.time():.3f}")


for url in ["http://a.com", "http://b.com", "http://c.com"]:
    fetch_data(url)
```

Output (approximately):

```
Fetching http://a.com at 1741500001.234
Fetching http://b.com at 1741500001.734
Fetching http://c.com at 1741500002.234
```

Each call waits at least 0.5s — two calls per second maximum.

22.6 Stacking Decorators

You can apply multiple decorators to one function. They apply **bottom-up** (innermost first):

```
import functools

def bold(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return f"<b>{func(*args, **kwargs)}</b>"
    return wrapper

def italic(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return f"<i>{func(*args, **kwargs)}</i>"
    return wrapper

def underline(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return f"<u>{func(*args, **kwargs)}</u>"
    return wrapper


@bold
@italic
@underline
def greet(name):
    return f"Hello, {name}!"


print(greet("Alice"))
```

Output:

```
<b><i><u>Hello, Alice!</u></i></b>
```

The order is: `greet` -> `underline(greet)` -> `italic(underline(greet))` -> `bold(italic(underline(greet)))`.

When called, the outermost decorator runs first: `bold` wraps `italic` wraps `underline` wraps `greet`.

A more practical stacking example:

```
@timer
```

```
@log_calls
@retry(times=3)
def call_api(endpoint):
    # would normally make an HTTP request
    import random
    if random.random() < 0.5:
        raise ConnectionError("Network timeout")
    return {"status": "ok", "endpoint": endpoint}
```

Execution order: `timer` starts the clock -> `log_calls` logs the call -> `retry` handles failures -> `call_api` runs.

22.7 Class-Based Decorators

You can also implement a decorator as a class that implements `__call__`:

```
import functools

class CountCalls:
    """Decorator that counts how many times a function is called."""
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func  = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"{self.func.__name__!r} called {self.count} time(s)")
        return self.func(*args, **kwargs)


@CountCalls
def say_hello():
    print("Hello!")


say_hello()
say_hello()
say_hello()
print(say_hello.count)    # 3
```

Output:

```
'say_hello' called 1 time(s)
```

```
Hello!
'say_hello' called 2 time(s)
Hello!
'say_hello' called 3 time(s)
Hello!
3
```

Class-based decorators are useful when the decorator needs to maintain significant state or have methods of its own. They're less common than function-based decorators but good to know.

22.8 Decorators in the Real World

You've already used decorators — you just may not have realized it. Every time you've written `@property`, `@staticmethod`, `@classmethod`, or `@functools.lru_cache`, you've been using a decorator.

Here's a quick map of decorators you'll encounter constantly:

Decorator	Where	Purpose
`@property`	Classes	Turn a method into a read-only attribute
`@staticmethod`	Classes	Method that doesn't receive `self` or `cls`
`@classmethod`	Classes	Method that receives `cls` instead of `self`
`@functools.lru_cache`	Anywhere	Memoize with a size limit
`@functools.wraps`	Inside decorators	Copy metadata from wrapped function
`@dataclasses.dataclass`	Classes	Auto-generate `__init__`, `__repr__`, etc.
`@abstractmethod`	ABCs	Mark method as required in subclasses
`@app.route("/")`	Flask/FastAPI	Register a URL route
`@pytest.mark.parametrize`	Tests	Run test with multiple input sets
`@login_required`	Django/Flask	Restrict view to logged-in users
`@app.on_event("startup")`	FastAPI	Run code on server startup

Once you understand how decorators work, all of these become obvious.

22.9 Project: A Decorator Toolkit

Let's build a small, reusable collection of decorators that you'd actually use in a real project.

```
"""
toolkit.py --- A collection of practical decorators.
"""
import functools
import time
import logging

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")
logger = logging.getLogger(__name__)


# ── 1. Timer ──────────────────────────────────────────────────────────

def timer(func):
    """Log execution time."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start  = time.perf_counter()
        result = func(*args, **kwargs)
        ms     = (time.perf_counter() - start) * 1000
        logger.info(f"{func.__name__} completed in {ms:.2f}ms")
        return result
    return wrapper


# ── 2. Retry ──────────────────────────────────────────────────────────

def retry(times=3, delay=0.5, exceptions=(Exception,), on_retry=None):
    """Retry function up to `times` times on `exceptions`."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            last_exc = None
            for attempt in range(1, times + 1):
                try:
```

```
                    return func(*args, **kwargs)
                except exceptions as e:
                    last_exc = e
                    if on_retry:
                        on_retry(attempt, e)
                    if attempt < times:
                        time.sleep(delay * attempt)    # exponential back-off
            raise last_exc
        return wrapper
    return decorator


# ── 3. Cache (with TTL) ──────────────────────────────────────────

def cache(ttl_seconds=None):
    """Cache results. Expire entries after ttl_seconds if given."""
    def decorator(func):
        store = {}    # {args: (result, timestamp)}

        @functools.wraps(func)
        def wrapper(*args):
            now = time.monotonic()
            if args in store:
                result, ts = store[args]
                if ttl_seconds is None or (now - ts) < ttl_seconds:
                    return result
            result       = func(*args)
            store[args]  = (result, now)
            return result

        wrapper.cache_clear = store.clear
        return wrapper
    return decorator


# ── 4. Deprecated ────────────────────────────────────────────────

def deprecated(message=""):
    """Emit a warning when the decorated function is called."""
    import warnings
    def decorator(func):
        msg = f"{func.__name__} is deprecated. {message}".strip()
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
```

```
            warnings.warn(msg, DeprecationWarning, stacklevel=2)
            return func(*args, **kwargs)
        return wrapper
    return decorator


# ── 5. Require types ──────────────────────────────────────────────

def require_types(**type_map):
    """
    Validate parameter types at call time.
    Usage: @require_types(name=str, age=int)
    """
    def decorator(func):
        import inspect
        sig = inspect.signature(func)

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            bound = sig.bind(*args, **kwargs)
            bound.apply_defaults()
            for param, expected in type_map.items():
                if param in bound.arguments:
                    value = bound.arguments[param]
                    if not isinstance(value, expected):
                        raise TypeError(
                            f"{func.__name__}(): parameter {param!r} expected "
                            f"{expected.__name__}, got {type(value).__name__}."
                        )
            return func(*args, **kwargs)
        return wrapper
    return decorator


# ── Demo ──────────────────────────────────────────────────────────

@timer
@retry(times=3, delay=0.1)
def fetch_user(user_id):
    import random
    if random.random() < 0.5:
        raise ConnectionError("Simulated network error")
    return {"id": user_id, "name": "Alice"}
```

```
@cache(ttl_seconds=5)
def get_config(key):
    print(f"  (computing config for {key!r})")
    return f"value_for_{key}"


@require_types(name=str, age=int)
def register(name, age):
    print(f"Registered: {name}, age {age}")


@deprecated("Use register() instead.")
def add_user(name):
    print(f"Adding user: {name}")


if __name__ == "__main__":
    # Timer + retry
    try:
        user = fetch_user(42)
        print(f"Got user: {user}")
    except ConnectionError as e:
        print(f"Failed after 3 retries: {e}")

    print()

    # Cache with TTL
    print(get_config("theme"))      # computed
    print(get_config("theme"))      # cached
    print(get_config("language"))   # computed

    print()

    # Type checking
    register("Alice", 30)
    try:
        register("Bob", "thirty")
    except TypeError as e:
        print(e)

    print()

    # Deprecated
```

```
import warnings
with warnings.catch_warnings(record=True) as w:
    warnings.simplefilter("always")
    add_user("Carol")
    if w:
        print(f"Warning: {w[0].message}")
```

22.10 What You Learned in This Chapter

- A **decorator** is a function that wraps another function with extra behavior.
- `@decorator` is shorthand for `func = decorator(func)`.
- Always use `@functools.wraps(func)` inside your wrapper to preserve `__name__` and `__doc__`.
- **Common patterns:** timing, logging, retry, caching, input validation.
- A **decorator factory** adds an extra nesting level so you can pass configuration to the decorator.
- **Stacking decorators** works bottom-up: the bottom decorator applies first, the top runs outermost.
- **Class-based decorators** implement `__call__` and are useful for decorators with significant internal state.
- Decorators are everywhere: `@property`, `@classmethod`, `@lru_cache`, `@app.route`, `@login_required`.

22.11 What's Next?

Chapter 23 covers **Generators and Iterators** — the machinery that powers `for` loops, `zip()`, `enumerate()`, comprehensions, and lazy pipelines. You'll write your own `range`, your own `zip`, and your own infinite sequences. If you've ever wondered how Python loops work at the lowest level, this is where you find out.

Chapter 23

Chapter 23: Generators and Iterators

Every time you write a `for` loop in Python, something called an **iterator** is doing the work. When you call `range(1_000_000)`, Python doesn't create a list of a million numbers — it creates an object that produces one number at a time, on demand. That's a **generator**.

Understanding generators and iterators explains how Python loops actually work, lets you process huge datasets without running out of memory, and gives you a tool for writing elegant, lazy pipelines.

23.1 How `for` Loops Actually Work

You've written hundreds of `for` loops. Here's what Python does behind the scenes:

```
for item in [1, 2, 3]:
    print(item)
```

Python silently translates this into something like:

```
_iter = iter([1, 2, 3])    # get an iterator from the list
while True:
    try:
        item = next(_iter)    # get the next value
        print(item)
    except StopIteration:    # no more values -> stop the loop
        break
```

Two built-in functions do all the work: - `iter(obj)` — calls `obj.__iter__()` and returns an iterator - `next(iterator)` — calls `iterator.__next__()` and returns the next value, or raises `StopIteration` when exhausted

```
numbers = [10, 20, 30]
it = iter(numbers)

print(next(it))    # 10
print(next(it))    # 20
print(next(it))    # 30

try:
    next(it)
except StopIteration:
    print("No more values.")
```

Any object that implements `__iter__` and `__next__` is an iterator. Lists, tuples, strings, dicts, files — all iterable. But they are not iterators themselves; you call `iter()` on them to get an iterator.

```
nums = [1, 2, 3]
print(hasattr(nums, "__iter__"))     # True  --- it's iterable
print(hasattr(nums, "__next__"))     # False --- it's NOT an iterator

it = iter(nums)
print(hasattr(it, "__iter__"))     # True
print(hasattr(it, "__next__"))     # True  --- now it IS an iterator
```

23.2 Writing an Iterator Class

You can make any class iterable by implementing `__iter__` and `__next__`:

```
class Countdown:
    """Counts down from start to 0."""
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self    # the object is its own iterator

    def __next__(self):
        if self.current < 0:
            raise StopIteration
        value = self.current
```

```
        self.current -= 1
        return value


for n in Countdown(5):
    print(n, end=" ")
print()    # 5 4 3 2 1 0
```

This works — but it's verbose. Generators let you write the same thing in a fraction of the code.

23.3 Generator Functions: `yield`

A **generator function** looks like a normal function but uses `yield` instead of `return`. When called, it returns a **generator object** — an iterator that produces values one at a time.

```
def countdown(start):
    while start >= 0:
        yield start
        start -= 1


gen = countdown(5)
print(type(gen))    # <class 'generator'>

print(next(gen))    # 5
print(next(gen))    # 4
print(next(gen))    # 3

# Or just loop over it
for n in countdown(3):
    print(n, end=" ")    # 3 2 1 0
print()
```

What `yield` does: 1. Produces a value (like `return`) 2. **Pauses** the function, saving its entire state (local variables, which line it's on) 3. On the next `next()` call, **resumes** from exactly where it paused

This is the key insight: the function's state is frozen between yields. Local variables keep their values. The instruction pointer doesn't move. Nothing runs until you ask for the next value.

```
def trace_generator():
    print("Start")
```

```
    yield 1
    print("After first yield")
    yield 2
    print("After second yield")
    yield 3
    print("After third yield --- generator done")


gen = trace_generator()
print(f"Got: {next(gen)}")    # prints "Start", returns 1
print(f"Got: {next(gen)}")    # prints "After first yield", returns 2
print(f"Got: {next(gen)}")    # prints "After second yield", returns 3

try:
    next(gen)                 # prints "After third yield", then raises StopIter
except StopIteration:
    print("Generator exhausted.")
```

Output:

```
Start
Got: 1
After first yield
Got: 2
After second yield
Got: 3
After third yield --- generator done
Generator exhausted.
```

23.4 Memory Efficiency: The Big Win

A list stores everything in memory at once. A generator produces one item at a time and immediately discards it.

```
import sys

# List: all million numbers in memory
big_list = [x ** 2 for x in range(1_000_000)]
print(f"List size: {sys.getsizeof(big_list):,} bytes")    # ~8,000,056 bytes

# Generator: produces one number at a time
def squares(n):
    for x in range(n):
        yield x ** 2
```

```
big_gen = squares(1_000_000)
print(f"Generator size: {sys.getsizeof(big_gen):,} bytes")    # 112 bytes
```

The generator is 112 bytes regardless of how many items it will eventually produce. The list grows with `n`. For large data, this is the difference between a program that works and one that crashes.

```
# Sum a billion numbers --- impossible with a list on most machines
total = sum(x ** 2 for x in range(1_000_000_000))
print(total)    # works fine --- never more than a few integers in memory at once
```

23.5 Common Generator Patterns

23.5.1 Infinite sequences

```
def naturals(start=1):
    """Generate 1, 2, 3, 4, ... forever."""
    n = start
    while True:
        yield n
        n += 1


def take(n, iterable):
    """Return the first n items from an iterable."""
    for _, item in zip(range(n), iterable):
        yield item


print(list(take(10, naturals())))    # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

An infinite generator is safe because values are produced only when requested. As long as you `take` a finite number, it terminates.

```
def fibonacci():
    """Generate the Fibonacci sequence forever."""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b


fibs = fibonacci()
```

```
print([next(fibs) for _ in range(12)])
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

23.5.2 Reading large files line by line

```
def read_large_file(path):
    """Yield one line at a time --- never loads the whole file."""
    with open(path) as f:
        for line in f:
            yield line.rstrip("\n")


# Only one line in memory at any time, even for a 10GB file
for line in read_large_file("huge_log.txt"):
    if "ERROR" in line:
        print(line)
```

This is how you process log files that are larger than your RAM.

23.5.3 Filtering and transforming

```
def where(iterable, predicate):
    """Yield only items where predicate(item) is True."""
    for item in iterable:
        if predicate(item):
            yield item


def select(iterable, transform):
    """Yield transform(item) for every item."""
    for item in iterable:
        yield transform(item)


numbers = range(1, 21)
evens   = where(numbers, lambda n: n % 2 == 0)
doubled = select(evens, lambda n: n * 2)

print(list(doubled))    # [4, 8, 12, 16, 20, 24, 28, 32, 36, 40]
```

These are your own `filter()` and `map()` — written as generators.

23.5.4 Batching

```
def batches(iterable, size):
    """Yield successive non-overlapping chunks of `size`."""
    batch = []
    for item in iterable:
        batch.append(item)
        if len(batch) == size:
            yield batch
            batch = []
    if batch:
        yield batch


numbers = range(1, 22)
for batch in batches(numbers, 5):
    print(batch)
```

Output:

```
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[21]
```

Useful for sending data to an API in batches, writing CSV in chunks, etc.

23.6 Generator Expressions

You've seen list comprehensions. A **generator expression** is the same syntax with parentheses instead of brackets:

```
# List comprehension --- all values computed now, stored in memory
squares_list = [x ** 2 for x in range(1_000_000)]

# Generator expression --- values produced on demand
squares_gen  = (x ** 2 for x in range(1_000_000))

# Both support iteration
print(next(squares_gen))    # 0
print(next(squares_gen))    # 1
print(next(squares_gen))    # 4
```

When you pass a generator expression as the sole argument to a function, you

can drop the extra parentheses:

```
# Both are equivalent
total = sum((x ** 2 for x in range(100)))
total = sum(x ** 2 for x in range(100))    # cleaner

# max/min/any/all all work this way
largest_even = max(x for x in range(100) if x % 2 == 0)    # 98
has_negative = any(x < 0 for x in [1, 2, -3, 4])           # True
all_positive = all(x > 0 for x in [1, 2,  3, 4])           # True
```

23.7 `yield from` — Delegating to Sub-generators

`yield from` lets one generator delegate to another:

```
def first_five():
    yield from range(1, 6)

def second_five():
    yield from range(6, 11)

def one_to_ten():
    yield from first_five()
    yield from second_five()


print(list(one_to_ten()))    # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This is cleaner than looping and re-yielding:

```
# Instead of this:
def flatten_messy(nested):
    for sublist in nested:
        for item in sublist:
            yield item

# Write this:
def flatten(nested):
    for sublist in nested:
        yield from sublist


data = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
print(list(flatten(data)))    # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

23.7.1 Recursive flattening with `yield from`

```
def deep_flatten(obj):
    """Flatten arbitrarily nested lists."""
    if isinstance(obj, list):
        for item in obj:
            yield from deep_flatten(item)
    else:
        yield obj


deep = [1, [2, [3, [4, [5]]]]]
print(list(deep_flatten(deep)))    # [1, 2, 3, 4, 5]
```

Recursive generators with `yield from` are one of the most elegant patterns in Python.

23.8 Sending Values into a Generator

Generators can receive values too, using the `.send()` method. This turns them into **coroutines** — functions that both produce and consume values.

```
def accumulator():
    """Running total --- receives numbers and yields the running sum."""
    total = 0
    while True:
        value = yield total    # yield current total, receive next value
        if value is None:
            break
        total += value


acc = accumulator()
next(acc)          # prime the generator (advance to first yield)

print(acc.send(10))    # 10
print(acc.send(5))     # 15
print(acc.send(20))    # 35
print(acc.send(3))     # 38
acc.close()            # clean shutdown
```

You must call `next(gen)` once first to advance the generator to its first `yield`. After that, `.send(value)` resumes the generator and passes `value` back as the result of the `yield` expression.

This is the foundation of Python's `async/await` system — but that's a topic for a later chapter.

23.9 `itertools` — Your Generator Toolkit

Python's `itertools` module is a collection of fast, memory-efficient generator-based tools. You met a few in Chapter 16; here's a deeper look at the ones you'll use most.

```
import itertools

# ── Infinite iterators ──────────────────────────────────

# count(start, step) --- like range but infinite
for n in itertools.islice(itertools.count(0, 5), 5):
    print(n, end=" ")    # 0 5 10 15 20
print()

# cycle(iterable) --- repeat forever
colors = itertools.cycle(["red", "green", "blue"])
for _ in range(7):
    print(next(colors), end=" ")    # red green blue red green blue red
print()

# repeat(value, times) --- repeat a value
print(list(itertools.repeat("Python", 3)))    # ['Python', 'Python', 'Python']


# ── Slicing and selecting ──────────────────────────────

# islice --- slice any iterator (like list slicing, but lazy)
gen = (x ** 2 for x in naturals())
print(list(itertools.islice(gen, 5, 15)))    # squares from index 5 to 14


# ── Combining iterators ────────────────────────────────

# chain --- concatenate multiple iterables
combined = itertools.chain([1, 2], [3, 4], [5, 6])
print(list(combined))    # [1, 2, 3, 4, 5, 6]

# zip_longest --- zip with a fill value when lengths differ
```

```
a = [1, 2, 3]
b = ["a", "b"]
print(list(itertools.zip_longest(a, b, fillvalue="?")))
# [(1, 'a'), (2, 'b'), (3, '?')]

# product --- Cartesian product (nested loops replaced)
suits  = ["♠", "♥", "♦", "♣"]
values = ["A", "2", "K"]
cards  = list(itertools.product(values, suits))
print(cards[:4])   # [('A', '♠'), ('A', '♥'), ('A', '♦'), ('A', '♣')]

# combinations and permutations
items = ["a", "b", "c", "d"]
print(list(itertools.combinations(items, 2)))
# [('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]

print(list(itertools.permutations(["x", "y", "z"], 2)))
# [('x', 'y'), ('x', 'z'), ('y', 'x'), ('y', 'z'), ('z', 'x'), ('z', 'y')]


# ── Grouping ──────────────────────────────────────────────────────────

# groupby --- group consecutive items by a key (data must be sorted first)
data = [
    {"name": "Alice",  "dept": "Eng"},
    {"name": "Bob",    "dept": "Eng"},
    {"name": "Carlos", "dept": "HR"},
    {"name": "Diana",  "dept": "HR"},
    {"name": "Eve",    "dept": "Sales"},
]
data.sort(key=lambda x: x["dept"])

for dept, members in itertools.groupby(data, key=lambda x: x["dept"]):
    names = [m["name"] for m in members]
    print(f"{dept}: {names}")
```

Output:

```
Eng: ['Alice', 'Bob']
HR: ['Carlos', 'Diana']
Sales: ['Eve']
```

23.10 Building a Lazy Data Pipeline

Generators compose beautifully. Each stage produces values one at a time; the next stage consumes them. Nothing runs until you pull from the end of the pipeline.

```
import csv
import itertools

def read_csv(path):
    """Yield one dict per row."""
    with open(path, newline="") as f:
        yield from csv.DictReader(f)


def where(iterable, predicate):
    return (item for item in iterable if predicate(item))


def select(iterable, *fields):
    return ({f: item[f] for f in fields} for item in iterable)


def convert(iterable, **converters):
    for item in iterable:
        row = dict(item)
        for field, func in converters.items():
            if field in row:
                row[field] = func(row[field])
        yield row


def take(n, iterable):
    yield from itertools.islice(iterable, n)


# Pipeline --- nothing runs until list() at the end
pipeline = (
    read_csv("sales.csv")
    |> where(lambda r: r["region"] == "EMEA")        # not real Python syntax..
)
```

Python doesn't have a `|>` pipe operator, but we chain calls:

```
def run_pipeline(path):
    rows     = read_csv(path)
```

```
    emea      = where(rows, lambda r: r["region"] == "EMEA")
    trimmed   = select(emea, "date", "product", "amount", "region")
    typed     = convert(trimmed, amount=float)
    top10     = take(10, typed)

    for row in top10:
        print(row)

# Each generator is lazy --- the file is read one line at a time,
# filtered, projected, converted, and printed, all without ever
# holding more than one row in memory (beyond what the generators need).
```

This is how production ETL pipelines work — reading gigabytes of CSV without materializing any of it in RAM.

23.11 Project: Build Your Own `range`

Let's build a complete iterator class that matches the behavior of Python's built-in `range`:

```
class MyRange:
    """
    A lazy range iterator matching Python's built-in range.
    Supports: iteration, len(), in operator, negative step, slicing preview.
    """
    def __init__(self, start, stop=None, step=1):
        if stop is None:
            start, stop = 0, start
        if step == 0:
            raise ValueError("MyRange() step argument must not be zero.")
        self.start = start
        self.stop  = stop
        self.step  = step

    def __iter__(self):
        current = self.start
        if self.step > 0:
            while current < self.stop:
                yield current
                current += self.step
        else:
            while current > self.stop:
                yield current
```

```
                current += self.step

    def __len__(self):
        if self.step > 0:
            return max(0, (self.stop - self.start + self.step - 1) // self.step
        else:
            return max(0, (self.start - self.stop - self.step - 1) // (-self.st

    def __contains__(self, value):
        if self.step > 0:
            return self.start <= value < self.stop and (value - self.start) % s
        else:
            return self.stop < value <= self.start and (self.start - value) % (

    def __repr__(self):
        if self.step == 1:
            return f"MyRange({self.start}, {self.stop})"
        return f"MyRange({self.start}, {self.stop}, {self.step})"


# Basic usage
print(list(MyRange(5)))           # [0, 1, 2, 3, 4]
print(list(MyRange(2, 10, 2)))    # [2, 4, 6, 8]
print(list(MyRange(10, 0, -2)))   # [10, 8, 6, 4, 2]

print(len(MyRange(0, 100, 3)))    # 34
print(6 in MyRange(0, 10, 2))     # True
print(7 in MyRange(0, 10, 2))     # False

# Lazy --- no list ever created
total = sum(MyRange(1_000_000))
print(total)    # 499999500000
```

`__iter__` is a generator function — it uses `yield`. The class doesn't store any list. Every item is computed on demand.

23.12 What You Learned in This Chapter

- A `for` loop calls `iter()` to get an iterator, then `next()` repeatedly until `StopIteration`.
- Any object with `__iter__` and `__next__` is an iterator.
- A **generator function** uses `yield` to pause and resume, producing val-

ues on demand.
- Generators use constant memory regardless of how many values they produce.
- Generator expressions `(x for x in ...)` are lazy — no parentheses needed when passed alone to a function.
- `yield from` delegates to a sub-generator — cleaner than a loop.
- `.send(value)` passes a value into a paused generator.
- `itertools` provides fast, composable generator tools: `count`, `cycle`, `chain`, `islice`, `groupby`, `product`, `combinations`, `permutations`.
- Generators compose into lazy data pipelines that can process arbitrarily large data with constant memory.

23.13 What's Next?

Chapter 24 covers **Context Managers** — the `with` statement and how it works. You've been using `with open(...)` since Chapter 8. Now you'll learn to write your own context managers both with `__enter__`/`__exit__` and with `@contextlib.contextmanager`, a generator-based shortcut that makes them trivial to write.

Chapter 24

Chapter 24: Context Managers

You've been writing `with open("file.txt") as f:` since Chapter 8. It closes the file automatically — even if your code crashes. That's a **context manager** at work.

Context managers handle setup and teardown automatically. They guarantee that cleanup always happens, no matter what. Files get closed. Database connections get released. Locks get unlocked. Timers get stopped. You never forget, because you can't.

This chapter shows you exactly how they work, how to write your own using both classes and generators, and the patterns that make them so useful in real code.

24.1 The Problem They Solve

Without a context manager, you have to remember to clean up:

```
f = open("data.txt")
data = f.read()
# ... do something with data ...
f.close()   # easy to forget, or skip if an exception is raised
```

If an exception happens between `open()` and `close()`, the file never closes. Your program leaks a file handle. Do this enough times and the OS refuses to open any more files.

The `try/finally` pattern fixes it:

```
f = open("data.txt")
try:
    data = f.read()
    # ... work with data ...
finally:
    f.close()    # always runs, even if an exception occurs
```

This is correct — but verbose and easy to get wrong. The `with` statement is cleaner:

```
with open("data.txt") as f:
    data = f.read()
    # ... work with data ...
# file is always closed here, guaranteed
```

Same behavior. Zero extra code. The context manager handles the `try/finally` for you.

24.2 How `with` Works

The `with` statement calls two methods on the context manager object:

1. `__enter__()` — runs at the start of the `with` block. The return value is bound to the `as` variable.
2. `__exit__(exc_type, exc_val, exc_tb)` — runs when the block ends, whether normally or due to an exception.

```
with expression as variable:
    body
```

Python translates this into roughly:

```
_cm = expression
variable = _cm.__enter__()
try:
    body
except:
    if not _cm.__exit__(*sys.exc_info()):
        raise
else:
    _cm.__exit__(None, None, None)
```

If `__exit__` returns `True`, the exception is suppressed. If it returns `False` (or `None`), the exception propagates.

24.3 Writing a Context Manager Class

Implement `__enter__` and `__exit__`:

```
class ManagedFile:
    """Context manager for opening files safely."""
    def __init__(self, path, mode="r"):
        self.path = path
        self.mode = mode
        self.file = None

    def __enter__(self):
        print(f"Opening {self.path!r}")
        self.file = open(self.path, self.mode)
        return self.file   # bound to the 'as' variable

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Closing {self.path!r}")
        if self.file:
            self.file.close()
        if exc_type:
            print(f"  Exception suppressed: {exc_val}")
        return False   # don't suppress exceptions


with ManagedFile("data.txt", "w") as f:
    f.write("Hello from a context manager!\n")

# Prints:
# Opening 'data.txt'
# Closing 'data.txt'
```

24.3.1 Handling exceptions in `__exit__`

`__exit__` receives three arguments: the exception type, value, and traceback. If no exception occurred, all three are `None`.

```
class SuppressErrors:
    """Silently ignore specific exception types."""
    def __init__(self, *exceptions):
        self.exceptions = exceptions

    def __enter__(self):
        return self
```

```
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type and issubclass(exc_type, self.exceptions):
            print(f"Suppressed: {exc_val}")
            return True   # swallow the exception
        return False      # let other exceptions propagate


with SuppressErrors(FileNotFoundError, PermissionError):
    open("does_not_exist.txt")   # FileNotFoundError is suppressed

print("Still running after the error.")

# Output:
# Suppressed: [Errno 2] No such file or directory: 'does_not_exist.txt'
# Still running after the error.
```

This is exactly what `contextlib.suppress` does internally.

24.4 Writing a Context Manager with `@contextlib.contextmanager`

Writing a class every time is overkill. The `contextlib.contextmanager` decorator lets you write a context manager as a simple generator function:

- Everything **before** `yield` is the setup (`__enter__`).
- The `yield` value is what gets bound to the `as` variable.
- Everything **after** `yield` is the teardown (`__exit__`).

```
from contextlib import contextmanager

@contextmanager
def managed_file(path, mode="r"):
    print(f"Opening {path!r}")
    f = open(path, mode)
    try:
        yield f           # give the caller the file object
    finally:
        print(f"Closing {path!r}")
        f.close()         # always runs


with managed_file("data.txt", "w") as f:
    f.write("Written via contextmanager!\n")
```

```
# Opening 'data.txt'
# Closing 'data.txt'
```

The `try/finally` inside the generator ensures cleanup happens even if the body of the `with` block raises an exception. You must always wrap the `yield` in `try/finally` when there's something to clean up.

24.5 Practical Context Manager Patterns

24.5.1 Timer

```
from contextlib import contextmanager
import time

@contextmanager
def timer(label=""):
    start = time.perf_counter()
    try:
        yield
    finally:
        elapsed = time.perf_counter() - start
        tag = f"[{label}] " if label else ""
        print(f"{tag}Elapsed: {elapsed:.4f}s")


with timer("Sorting"):
    data = sorted(range(1_000_000), reverse=True)

# [Sorting] Elapsed: 0.0821s
```

24.5.2 Temporary directory

```
from contextlib import contextmanager
import tempfile
import shutil
import os

@contextmanager
def temp_directory():
    """Create a temp directory, yield its path, then delete it."""
    path = tempfile.mkdtemp()
    try:
```

```
        yield path
    finally:
        shutil.rmtree(path, ignore_errors=True)


with temp_directory() as tmpdir:
    test_file = os.path.join(tmpdir, "test.txt")
    with open(test_file, "w") as f:
        f.write("Temporary content")
    print(f"Temp dir: {tmpdir}")
    print(f"File exists: {os.path.exists(test_file)}")

print(f"Dir still exists after: {os.path.exists(tmpdir)}")
# False --- deleted automatically
```

24.5.3 Changing directory temporarily

```
from contextlib import contextmanager
import os

@contextmanager
def working_directory(path):
    """Temporarily change the current working directory."""
    original = os.getcwd()
    try:
        os.chdir(path)
        yield
    finally:
        os.chdir(original)


print(os.getcwd())                      # C:\python-book
with working_directory("C:\\Users"):
    print(os.getcwd())                  # C:\Users
print(os.getcwd())                      # C:\python-book (restored)
```

24.5.4 Redirecting stdout

```
from contextlib import contextmanager, redirect_stdout
import io

@contextmanager
def capture_output():
    """Capture everything printed inside the block."""
```

```
    buffer = io.StringIO()
    with redirect_stdout(buffer):
        yield buffer
    # buffer.getvalue() has everything that was printed


with capture_output() as output:
    print("Hello")
    print("World")
    for i in range(3):
        print(f"Line {i}")

captured = output.getvalue()
print(f"Captured {len(captured.splitlines())} lines:")
print(repr(captured))
```

Output:

```
Captured 5 lines:
'Hello\nWorld\nLine 0\nLine 1\nLine 2\n'
```

24.5.5 Database transaction

```
from contextlib import contextmanager
import sqlite3

@contextmanager
def transaction(connection):
    """Commit on success, rollback on failure."""
    try:
        yield connection
        connection.commit()
        print("Transaction committed.")
    except Exception as e:
        connection.rollback()
        print(f"Transaction rolled back: {e}")
        raise


conn = sqlite3.connect(":memory:")
conn.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)")

# Successful transaction
with transaction(conn):
    conn.execute("INSERT INTO users (name) VALUES (?)", ("Alice",))
```

```
    conn.execute("INSERT INTO users (name) VALUES (?)", ("Bob",))

# Failed transaction --- both inserts rolled back
try:
    with transaction(conn):
        conn.execute("INSERT INTO users (name) VALUES (?)", ("Carlos",))
        raise ValueError("Simulated error mid-transaction")
except ValueError:
    pass

rows = conn.execute("SELECT * FROM users").fetchall()
print(rows)    # [(1, 'Alice'), (2, 'Bob')] --- Carlos never committed
conn.close()
```

24.5.6 Lock management

```
from contextlib import contextmanager
import threading

@contextmanager
def locked(lock):
    """Acquire a lock, yield, then release it."""
    lock.acquire()
    try:
        yield
    finally:
        lock.release()


# (Same as using: with lock:)
my_lock = threading.Lock()
shared_data = []

def append_safely(item):
    with locked(my_lock):
        shared_data.append(item)
```

24.6 `contextlib` Utilities You Should Know

Python's `contextlib` module has several ready-made tools:

```
from contextlib import (
    suppress,
```

```
    nullcontext,
    ExitStack,
    asynccontextmanager,
)

# suppress --- ignore specific exceptions
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove("does_not_exist.txt")    # silently ignored
# equivalent to our SuppressErrors class from earlier


# nullcontext --- a no-op context manager (useful for optional context managers
def process(data, lock=None):
    ctx = lock if lock is not None else nullcontext()
    with ctx:
        # do something with data
        return data * 2

process(5)            # no lock
process(5, my_lock)   # with lock


# ExitStack --- dynamically manage multiple context managers
files = ["a.txt", "b.txt", "c.txt"]

with ExitStack() as stack:
    handles = [stack.enter_context(open(f, "w")) for f in files]
    for i, handle in enumerate(handles):
        handle.write(f"Content of file {i}\n")
# All three files closed automatically --- even if one write fails
```

`ExitStack` is particularly useful when you don't know at write time how many context managers you need.

24.7 Nested and Multiple Context Managers

You can use multiple context managers in one `with` statement:

```
# Old way --- nested
with open("input.txt") as infile:
    with open("output.txt", "w") as outfile:
```

```
        outfile.write(infile.read())

# New way --- comma-separated (Python 3.1+)
with open("input.txt") as infile, open("output.txt", "w") as outfile:
    outfile.write(infile.read())
```

Both are equivalent. The comma form is preferred — it's flatter and just as clear.

24.8 Project: A Robust Database Connection Manager

Let's build a complete context manager that handles SQLite connections with automatic transaction management, connection pooling awareness, and detailed logging.

```
"""
db.py --- A context manager for robust SQLite access.
"""
from contextlib import contextmanager
import sqlite3
import logging
import time

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")
log = logging.getLogger("db")


class Database:
    """
    Context manager wrapping a SQLite connection.

    Usage:
        with Database("myapp.db") as db:
            db.execute("INSERT INTO ...")
            rows = db.query("SELECT ...")
    """

    def __init__(self, path, timeout=30, isolation_level="DEFERRED"):
        self.path            = path
        self.timeout         = timeout
        self.isolation_level = isolation_level
        self._conn           = None
```

```
    # ── Context manager protocol ──────────────────────────────────────

    def __enter__(self):
        log.info(f"Connecting to {self.path!r}")
        self._conn = sqlite3.connect(
            self.path,
            timeout=self.timeout,
            isolation_level=self.isolation_level,
            check_same_thread=False,
        )
        self._conn.row_factory = sqlite3.Row   # rows behave like dicts
        self._conn.execute("PRAGMA journal_mode=WAL")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self._conn:
            if exc_type is None:
                log.info("Committing transaction")
                self._conn.commit()
            else:
                log.warning(f"Rolling back due to: {exc_val}")
                self._conn.rollback()
            self._conn.close()
            self._conn = None
            log.info(f"Connection to {self.path!r} closed")
        return False   # don't suppress exceptions

    # ── Query helpers ─────────────────────────────────────────────────

    def execute(self, sql, params=()):
        """Run a write statement. Returns the cursor."""
        if self._conn is None:
            raise RuntimeError("Not inside a 'with' block.")
        return self._conn.execute(sql, params)

    def executemany(self, sql, seq_of_params):
        if self._conn is None:
            raise RuntimeError("Not inside a 'with' block.")
        return self._conn.executemany(sql, seq_of_params)

    def query(self, sql, params=()):
        """Run a SELECT and return a list of Row objects."""
        return self.execute(sql, params).fetchall()
```

```
    def query_one(self, sql, params=()):
        """Return the first matching row, or None."""
        return self.execute(sql, params).fetchone()

    def table_exists(self, name):
        row = self.query_one(
            "SELECT 1 FROM sqlite_master WHERE type='table' AND name=?",
            (name,),
        )
        return row is not None

    # ── Schema helpers ──────────────────────────────────────────

    @contextmanager
    def savepoint(self, name="sp"):
        """Nested transaction using SQLite savepoints."""
        self.execute(f"SAVEPOINT {name}")
        try:
            yield self
            self.execute(f"RELEASE SAVEPOINT {name}")
        except Exception:
            self.execute(f"ROLLBACK TO SAVEPOINT {name}")
            raise


# ── Demo ────────────────────────────────────────────────────────

def setup_schema(db):
    if not db.table_exists("users"):
        db.execute("""
            CREATE TABLE users (
                id    INTEGER PRIMARY KEY AUTOINCREMENT,
                name  TEXT    NOT NULL,
                email TEXT    UNIQUE NOT NULL,
                score REAL    DEFAULT 0
            )
        """)
        log.info("Created table 'users'")


def seed_data(db):
    users = [
        ("Alice",   "alice@example.com",   95.5),
```

```
        ("Bob",     "bob@example.com",     82.0),
        ("Carlos",  "carlos@example.com",  91.3),
        ("Diana",   "diana@example.com",   88.7),
    ]
    db.executemany(
        "INSERT OR IGNORE INTO users (name, email, score) VALUES (?, ?, ?)",
        users,
    )
    log.info(f"Seeded {len(users)} users")


if __name__ == "__main__":
    DB_PATH = ":memory:"

    # Normal operation --- commits automatically
    with Database(DB_PATH) as db:
        setup_schema(db)
        seed_data(db)

        rows = db.query("SELECT name, score FROM users ORDER BY score DESC")
        print("\nTop scorers:")
        for row in rows:
            print(f"  {row['name']:<10} {row['score']}")

    # Savepoint --- nested transaction
    with Database(DB_PATH) as db:
        setup_schema(db)    # table already exists, no-op

        try:
            with db.savepoint("add_user"):
                db.execute(
                    "INSERT INTO users (name, email, score) VALUES (?, ?, ?)",
                    ("Eve", "eve@example.com", 99.9),
                )
                # Simulate a conflict
                db.execute(
                    "INSERT INTO users (name, email, score) VALUES (?, ?, ?)",
                    ("Eve", "eve@example.com", 50.0),    # duplicate email
                )
        except sqlite3.IntegrityError as e:
            log.warning(f"Savepoint rolled back: {e}")

        # Eve was not added (savepoint rolled back)
        count = db.query_one("SELECT COUNT(*) AS n FROM users")
```

```
print(f"\nUser count after conflict: {count['n']}")    # 4
```

24.9 What You Learned in This Chapter

- Context managers guarantee setup and teardown code runs, even when exceptions occur.
- The `with` statement calls `__enter__()` at the start and `__exit__()` at the end.
- If `__exit__` returns `True`, exceptions are suppressed; `False` lets them propagate.
- Write context managers as **classes** (implement `__enter__` and `__exit__`) or as **generator functions** with `@contextmanager` (wrap `yield` in `try/finally`).
- Common patterns: timers, temp directories, working directory changes, stdout capture, database transactions, lock management.
- `contextlib.suppress` — ignore specific exception types cleanly.
- `contextlib.nullcontext` — a no-op for optional context managers.
- `contextlib.ExitStack` — manage a dynamic number of context managers.
- Multiple context managers can share one `with` line using commas.

24.10 What's Next?

Chapter 25 covers **Functional Programming** — `map`, `filter`, `reduce`, pure functions, immutability, and using `functools` to write code that is easier to reason about, test, and compose. You already know the building blocks from earlier chapters; this chapter ties them together into a coherent style.

Chapter 25

Chapter 25: Functional Programming

You've been writing Python in an **imperative** style — telling the computer *how* to do things step by step. Functional programming is a different approach: you describe *what* you want using functions that take inputs and return outputs, with no side effects and no shared state.

Python is not a purely functional language. But it borrows the best ideas from functional programming, and knowing them makes your code shorter, easier to test, and easier to reason about.

This chapter covers pure functions, immutability, `map`, `filter`, `reduce`, `functools`, and how to think functionally in a Python context.

25.1 Pure Functions

A **pure function** always returns the same output for the same input, and causes no side effects. No printing, no modifying globals, no writing to files — just input in, output out.

```
# Pure --- same input always gives same output
def add(a, b):
    return a + b

def square(n):
    return n ** 2
```

```
def to_celsius(fahrenheit):
    return (fahrenheit - 32) * 5 / 9

# Impure --- depends on or modifies external state
total = 0

def add_to_total(n):    # modifies global state
    global total
    total += n

def get_timestamp():    # result depends on the clock, not just inputs
    import time
    return time.time()

def save_user(user):    # side effect: writes to disk
    with open("users.txt", "a") as f:
        f.write(str(user) + "\n")
```

Pure functions are easy to test (just call them with inputs, check outputs), easy to reason about (they don't depend on anything outside themselves), and safe to run in parallel (no shared state to corrupt).

You can't write a whole program with only pure functions — at some point you need to read input, write output, interact with the world. The functional style says: *push the impure parts to the edges, keep the core pure.*

```
# Functional core, imperative shell
# ──────────────────────────────────

# Pure core --- no side effects
def parse_line(line):
    parts = line.strip().split(",")
    return {"name": parts[0], "score": float(parts[1])}

def is_passing(record):
    return record["score"] >= 60

def format_record(record):
    status = "PASS" if is_passing(record) else "FAIL"
    return f"{record['name']:<15} {record['score']:>6.1f}  {status}"

def compute_average(records):
    if not records:
        return 0.0
    return sum(r["score"] for r in records) / len(records)
```

```
# Impure shell --- reads/writes files, talks to the world
def process_scores(input_path, output_path):
    with open(input_path) as f:
        lines = f.readlines()

    records = [parse_line(line) for line in lines if line.strip()]
    passing = [r for r in records if is_passing(r)]
    avg     = compute_average(records)

    with open(output_path, "w") as f:
        for r in records:
            f.write(format_record(r) + "\n")
        f.write(f"\nAverage: {avg:.1f}\n")
        f.write(f"Passing: {len(passing)}/{len(records)}\n")

    return avg, len(passing)
```

The pure functions (`parse_line`, `is_passing`, `format_record`, `compute_average`) are trivial to unit-test. The impure shell (`process_scores`) is easy to read because it just calls the pure functions in order.

25.2 Immutability

Functional programming favors **immutable data** — data that doesn't change after creation. Instead of modifying existing data, you create new data.

```
# Mutable style --- modify in place
def add_tax_mutable(items, rate):
    for item in items:
        item["price"] *= (1 + rate)    # modifies the original dicts
    return items

# Immutable style --- create new data, leave original unchanged
def add_tax(items, rate):
    return [
        {**item, "price": item["price"] * (1 + rate)}
        for item in items
    ]


cart = [
    {"name": "Book",   "price": 20.00},
```

```
    {"name": "Pen",    "price":  2.50},
    {"name": "Laptop", "price": 999.99},
]

cart_with_tax = add_tax(cart, 0.08)

print(cart[0]["price"])              # 20.00  --- original unchanged
print(cart_with_tax[0]["price"])     # 21.60  --- new data
```

Immutability prevents a whole class of bugs where one part of your code modifies data that another part expects to stay the same. It makes functions predictable and programs easier to debug.

Python's built-in immutable types — `tuple`, `frozenset`, `str` — enforce this at the language level. For custom classes, `@dataclass(frozen=True)` (from Chapter 19) gives you immutable objects.

25.3 `map` — Transform Every Element

`map(func, iterable)` applies a function to every item in an iterable and returns a lazy iterator of results.

```
numbers = [1, 2, 3, 4, 5]

# Imperative
squared = []
for n in numbers:
    squared.append(n ** 2)

# map
squared = list(map(lambda n: n ** 2, numbers))

# List comprehension (often preferred in Python)
squared = [n ** 2 for n in numbers]
```

All three do the same thing. `map` is lazy — it doesn't compute anything until you iterate:

```
result = map(str, range(5))    # nothing computed yet
print(result)                   # <map object at 0x...>
print(list(result))             # ['0', '1', '2', '3', '4']
```

`map` with named functions reads very naturally:

```
def celsius_to_fahrenheit(c):
    return c * 9 / 5 + 32
```

```
temps_c = [0, 20, 37, 100]
temps_f = list(map(celsius_to_fahrenheit, temps_c))
print(temps_f)    # [32.0, 68.0, 98.6, 212.0]
```

`map` also accepts multiple iterables — the function must accept that many arguments:

```
a = [1, 2, 3]
b = [10, 20, 30]

sums = list(map(lambda x, y: x + y, a, b))
print(sums)    # [11, 22, 33]

# Equivalent with zip
sums = [x + y for x, y in zip(a, b)]
```

25.4 `filter` — Keep Only Matching Elements

`filter(func, iterable)` returns a lazy iterator of items for which `func(item)` is truthy.

```
numbers = range(-5, 6)

# Keep only positives
positives = list(filter(lambda n: n > 0, numbers))
print(positives)    # [1, 2, 3, 4, 5]

# Equivalent comprehension
positives = [n for n in numbers if n > 0]

words = ["apple", "banana", "cherry", "date", "elderberry", "fig"]

long_words = list(filter(lambda w: len(w) > 5, words))
print(long_words)    # ['banana', 'cherry', 'elderberry']
```

Pass `None` as the function to filter out falsy values:

```
data = [0, 1, "", "hello", None, [], [1, 2], False, True]
truthy = list(filter(None, data))
print(truthy)    # [1, 'hello', [1, 2], True]
```

25.5 `reduce` — Fold a Sequence into One Value

`reduce(func, iterable)` repeatedly applies `func` to pairs of elements until one value remains. It's in `functools`:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

total   = reduce(lambda acc, x: acc + x, numbers)    # ((((1+2)+3)+4)+5) = 15
product = reduce(lambda acc, x: acc * x, numbers)    # ((((1*2)*3)*4)*5) = 120
maximum = reduce(lambda acc, x: acc if acc > x else x, numbers)    # 5

print(total, product, maximum)    # 15  120  5
```

You can provide an initial value as the third argument:

```
# Flatten a list of lists
nested = [[1, 2], [3, 4], [5, 6]]
flat   = reduce(lambda acc, lst: acc + lst, nested, [])
print(flat)    # [1, 2, 3, 4, 5, 6]

# Count words in a list of sentences
sentences = ["hello world", "foo bar baz", "python"]
word_count = reduce(lambda acc, s: acc + len(s.split()), sentences, 0)
print(word_count)    # 6
```

In Python, `reduce` is less common than explicit loops or comprehensions because readability matters more here. Use it when the fold pattern is genuinely the clearest expression of the idea.

25.6 `functools` — Your Functional Toolkit

`functools` has several powerful tools for functional-style programming.

25.6.1 `partial` — Fix some arguments

`functools.partial` creates a new function with some arguments pre-filled:

```
from functools import partial

def power(base, exponent):
    return base ** exponent

square  = partial(power, exponent=2)
```

```
cube    = partial(power, exponent=3)
two_to  = partial(power, base=2)

print(square(5))     # 25
print(cube(3))       # 27
print(two_to(10))    # 1024
```

```
# Useful with map --- apply a function with a fixed extra argument
from functools import partial

def clamp(value, low, high):
    return max(low, min(high, value))

clamp_score = partial(clamp, low=0, high=100)

raw_scores = [85, -5, 110, 72, 0, 100, 150]
clamped    = list(map(clamp_score, raw_scores))
print(clamped)    # [85, 0, 100, 72, 0, 100, 100]
```

25.6.2 `lru_cache` — Memoize with a size limit

```
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(100))    # 354224848179261915075
print(fibonacci.cache_info())
# CacheInfo(hits=98, misses=101, maxsize=128, currsize=101)
```

`@lru_cache` is thread-safe, respects `maxsize`, and exposes `cache_info()` and `cache_clear()`. Use it whenever a pure function is called repeatedly with the same arguments.

25.6.3 `cache` — Unbounded memoization (Python 3.9+)

```
from functools import cache

@cache
def expensive(n):
    return sum(range(n))
```

```
# Same as @lru_cache(maxsize=None) but slightly faster
```

25.6.4 `total_ordering` — Fill in comparison methods

```
from functools import total_ordering

@total_ordering
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    def __eq__(self, other):
        return self.celsius == other.celsius

    def __lt__(self, other):
        return self.celsius < other.celsius

    # total_ordering fills in: __le__, __gt__, __ge__


temps = [Temperature(100), Temperature(0), Temperature(37)]
print(sorted(temps)[0].celsius)      # 0
print(max(temps).celsius)            # 100
print(Temperature(20) <= Temperature(37))    # True
```

25.6.5 `reduce` — Already covered above

25.6.6 `singledispatch` — Function overloading by type

```
from functools import singledispatch

@singledispatch
def process(value):
    raise TypeError(f"Cannot process type {type(value).__name__}")

@process.register(int)
def _(value):
    return f"Integer: {value * 2}"

@process.register(str)
def _(value):
    return f"String: {value.upper()}"

@process.register(list)
def _(value):
```

```
    return f"List of {len(value)} items"

@process.register(float)
def _(value):
    return f"Float: {value:.2f}"


print(process(42))           # Integer: 84
print(process("hello"))      # String: HELLO
print(process([1, 2, 3]))    # List of 3 items
print(process(3.14))         # Float: 3.14
```

25.7 Lambda Functions

A **lambda** is an anonymous function defined in a single expression. It's useful for short, one-off functions passed as arguments.

```
# Named function
def double(n):
    return n * 2

# Lambda equivalent
double = lambda n: n * 2

# In practice, lambdas shine as inline arguments
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

print(sorted(numbers, key=lambda n: -n))        # [9, 6, 5, 4, 3, 2, 1, 1]

people = [("Alice", 30), ("Bob", 25), ("Carlos", 35)]
print(sorted(people, key=lambda p: p[1]))       # sorted by age
```

When to use lambdas vs named functions: - Use a lambda when the function is very short, used once, and its meaning is obvious from context. - Use a named function when the function is more than one expression, used multiple times, or needs a docstring.

```
# Good lambda use --- short, obvious
evens = list(filter(lambda n: n % 2 == 0, range(10)))

# Bad lambda use --- hard to read, should be a named function
result = sorted(data, key=lambda x: (x["dept"].lower(), -x["salary"], x["name"]

# Better
```

```
def sort_key(employee):
    return (employee["dept"].lower(), -employee["salary"], employee["name"])

result = sorted(data, key=sort_key)
```

25.8 Function Composition

Composition means combining functions so the output of one feeds into the input of the next.

```
def compose(*funcs):
    """Return a function that applies funcs right-to-left."""
    from functools import reduce
    def composed(value):
        return reduce(lambda v, f: f(v), reversed(funcs), value)
    return composed

def pipe(*funcs):
    """Return a function that applies funcs left-to-right."""
    from functools import reduce
    def piped(value):
        return reduce(lambda v, f: f(v), funcs, value)
    return piped


import math

normalize   = lambda x: x / 100
to_radians  = math.radians
sine        = math.sin
to_percent  = lambda x: x * 100

# pipe: normalize -> radians -> sin -> percent
process = pipe(normalize, to_radians, sine, to_percent)
print(f"{process(9000):.2f}%")    # 0.00%  (sin of 90 degrees = 1.0 -> 100%)

process2 = pipe(normalize, to_radians, sine, to_percent)
print(f"{process2(4500):.2f}%")   # 70.71%  (sin 45° ≈ 0.707)

# Text processing pipeline
strip      = str.strip
lower      = str.lower
capitalize = str.capitalize
```

```
def remove_extra_spaces(text):
    return " ".join(text.split())

clean_title = pipe(strip, lower, remove_extra_spaces, capitalize)

titles = [
    "  PYTHON  ZERO  TO  HERO  ",
    "  the  QUICK  brown  FOX  ",
    "  FUNCTIONAL  programming  IN  python  ",
]

for title in titles:
    print(clean_title(title))
```

Output:

```
Python zero to hero
The quick brown fox
Functional programming in python
```

25.9 Putting It All Together: A Data Processing Library

```
"""
dataflow.py --- A small functional data processing library.
"""
from functools import reduce, partial
from typing import Callable, Iterable, Any


def pipeline(*funcs: Callable) -> Callable:
    """Compose functions left-to-right into a single callable."""
    def run(value):
        return reduce(lambda v, f: f(v), funcs, value)
    return run


def pluck(field: str) -> Callable:
    """Return a function that extracts field from a dict."""
    return lambda record: record[field]
```

```
def where(predicate: Callable) -> Callable:
    """Return a function that filters an iterable."""
    return lambda items: (item for item in items if predicate(item))


def transform(func: Callable) -> Callable:
    """Return a function that maps func over an iterable."""
    return lambda items: (func(item) for item in items)


def sort_by(key: Callable, reverse: bool = False) -> Callable:
    """Return a function that sorts an iterable."""
    return lambda items: sorted(items, key=key, reverse=reverse)


def group_by(key: Callable) -> Callable:
    """Return a function that groups items into a dict by key(item)."""
    def grouper(items):
        groups = {}
        for item in items:
            k = key(item)
            groups.setdefault(k, []).append(item)
        return groups
    return grouper


def aggregate(func: Callable) -> Callable:
    """Return a function that reduces an iterable to one value."""
    return lambda items: func(list(items))


def to_list(items: Iterable) -> list:
    return list(items)


# ── Demo ──────────────────────────────────────────────────────────────

employees = [
    {"name": "Alice",   "dept": "Eng",   "salary": 95_000},
    {"name": "Bob",     "dept": "Eng",   "salary": 82_000},
    {"name": "Carlos",  "dept": "HR",    "salary": 68_000},
    {"name": "Diana",   "dept": "HR",    "salary": 71_000},
    {"name": "Eve",     "dept": "Sales", "salary": 74_000},
    {"name": "Frank",   "dept": "Eng",   "salary": 105_000},
```

```
    {"name": "Grace",   "dept": "Sales", "salary": 88_000},
]

# ── Query 1: Names of engineers earning over $90k ──────────────────
high_earners = pipeline(
    where(lambda e: e["dept"] == "Eng" and e["salary"] > 90_000),
    transform(pluck("name")),
    sort_by(str.lower),
    to_list,
)

print("High-earning engineers:", high_earners(employees))
# ['Alice', 'Frank']


# ── Query 2: Average salary per department ─────────────────────────
avg_by_dept = pipeline(
    group_by(pluck("dept")),
)

groups = avg_by_dept(employees)
for dept, members in sorted(groups.items()):
    avg = sum(e["salary"] for e in members) / len(members)
    print(f"  {dept:<8}  avg ${avg:,.0f}")

# Eng       avg $94,000
# HR        avg $69,500
# Sales     avg $81,000


# ── Query 3: Top earner per department ─────────────────────────────
for dept, members in sorted(groups.items()):
    top = max(members, key=pluck("salary"))
    print(f"  {dept:<8}  top: {top['name']} (${top['salary']:,})")

# Eng       top: Frank ($105,000)
# HR        top: Diana ($71,000)
# Sales     top: Grace ($88,000)
```

25.10 What You Learned in This Chapter

- **Pure functions** always return the same output for the same input and cause no side effects. They're easy to test and reason about.
- **Immutability** — creating new data instead of modifying existing data — prevents a class of bugs and makes code predictable.
- `map(func, iterable)` — transform every element lazily.
- `filter(func, iterable)` — keep only elements where `func` returns truthy. `filter(None, iterable)` removes falsy values.
- `reduce(func, iterable, initial)` — fold a sequence into a single value.
- `functools.partial` — fix some arguments to create a specialized function.
- `functools.lru_cache` / `functools.cache` — memoize pure functions automatically.
- `functools.total_ordering` — define `__eq__` and `__lt__`, get all comparisons.
- `functools.singledispatch` — function overloading based on argument type.
- **Lambda functions** — short anonymous functions for use as inline arguments.
- **Function composition** — `pipe` and `compose` combine functions into processing pipelines.
- The **functional core, imperative shell** pattern keeps side effects isolated at the edges.

25.11 What's Next?

Chapter 26 covers **Advanced Python Types** — `typing` module annotations, `TypeVar`, `Generic`, `Protocol`, `NamedTuple`, `TypedDict`, and how type hints make large codebases maintainable. If you've ever wondered what `list[int]` or `Optional[str]` means in someone else's code, you're about to find out.

Chapter 26

Chapter 26: Type Hints and the `typing` Module

Python is dynamically typed — variables don't have fixed types, and Python doesn't check types at runtime. But large codebases written without type information are hard to maintain. You can't tell what a function expects just by reading its signature, editors can't autocomplete reliably, and bugs slip through that a type checker would have caught instantly.

Type hints fix this. They don't change how your program runs, but they tell humans and tools what types are expected. Your editor catches mistakes before you run the code. Your teammates understand your API without reading the implementation. Refactoring becomes safe.

This chapter teaches you how to read and write type hints, how to use the `typing` module, and how to run `mypy` to check your code statically.

26.1 The Basics

Type hints use the `:` syntax for variables and parameters, and `->` for return types:

```
# Variable annotations
name: str = "Alice"
age:  int = 30
score: float = 95.5
active: bool = True
```

```
# Without a value --- just declaring the type
user_id: int

# Function annotations
def greet(name: str) -> str:
    return f"Hello, {name}!"

def add(a: int, b: int) -> int:
    return a + b

def save_user(name: str, age: int) -> None:
    print(f"Saving {name}, age {age}")
```

Type hints are stored in `__annotations__` but Python does not enforce them at runtime:

```
def double(n: int) -> int:
    return n * 2

print(double(5))        # 10  --- works
print(double("ha"))     # 'haha' --- Python doesn't complain!
print(double.__annotations__)
# {'n': <class 'int'>, 'return': <class 'int'>}
```

Runtime enforcement requires a type checker like `mypy` or a library like `pydantic`. The hints themselves are documentation — very useful documentation.

26.2 Built-in Collection Types (Python 3.9+)

From Python 3.9 onwards, you can use built-in types directly as generic types:

```
# Python 3.9+
def process(items: list[int]) -> list[str]:
    return [str(i) for i in items]

def merge(a: dict[str, int], b: dict[str, int]) -> dict[str, int]:
    return {**a, **b}

def first_two(items: tuple[int, int]) -> int:
    return items[0] + items[1]

def unique(items: list[str]) -> set[str]:
    return set(items)
```

Before Python 3.9, you imported these from `typing`:

```python
# Python 3.7 / 3.8 --- use typing module
from typing import List, Dict, Tuple, Set

def process(items: List[int]) -> List[str]:
    return [str(i) for i in items]
```

From Python 3.10+, you can use `X | Y` instead of `Union[X, Y]`:

```python
# Python 3.10+
def parse(value: str | int | None) -> float:
    if value is None:
        return 0.0
    return float(value)
```

26.3 `Optional` — Values That Can Be `None`

`Optional[X]` means the value is either `X` or `None`. It's very common for function parameters with defaults and return values that might fail:

```python
from typing import Optional

def find_user(user_id: int) -> Optional[dict]:
    users = {1: {"name": "Alice"}, 2: {"name": "Bob"}}
    return users.get(user_id)    # returns dict or None


def greet_user(name: Optional[str] = None) -> str:
    if name is None:
        return "Hello, stranger!"
    return f"Hello, {name}!"


user = find_user(1)
if user is not None:
    print(user["name"])    # Alice

print(greet_user())          # Hello, stranger!
print(greet_user("Alice"))   # Hello, Alice!
```

In Python 3.10+, `Optional[X]` is equivalent to `X | None`:

```python
def find_user(user_id: int) -> dict | None:    # cleaner
    ...
```

26.4 `Union` — Multiple Possible Types

`Union[X, Y]` means the value can be either `X` or `Y`:

```
from typing import Union

def stringify(value: Union[int, float, bool]) -> str:
    return str(value)

# Python 3.10+ shorthand
def stringify(value: int | float | bool) -> str:
    return str(value)
```

26.5 `Any` — Opt Out of Type Checking

`Any` is compatible with every type. Use it when you genuinely can't or don't want to specify a type — but use it sparingly. Every `Any` is a gap in your type coverage.

```
from typing import Any

def log(value: Any) -> None:
    print(repr(value))

def passthrough(data: Any) -> Any:
    return data
```

26.6 `Callable` — Typing Functions

`Callable[[ArgTypes], ReturnType]` describes a callable (function, lambda, class with `__call__`):

```
from typing import Callable

def apply(func: Callable[[int], int], value: int) -> int:
    return func(value)

def apply_two(func: Callable[[int, int], str], a: int, b: int) -> str:
    return func(a, b)

# A callable with no arguments
def run(callback: Callable[[], None]) -> None:
```

```
    callback()

# Callable with any arguments (use ... as parameter list)
def call_anything(func: Callable[..., int]) -> int:
    return func()
```

26.7 `Sequence`, `Iterable`, `Mapping` — Accept More Types

Being specific about `list` means your function only works with lists. Using abstract types makes functions more flexible:

```
from typing import Sequence, Iterable, Mapping, MutableMapping

# Only works with lists
def sum_list(items: list[int]) -> int:
    return sum(items)

# Works with lists, tuples, strings, ranges --- anything iterable
def sum_any(items: Iterable[int]) -> int:
    return sum(items)

# Works with lists, tuples --- anything with len() and indexing
def first(items: Sequence[int]) -> int:
    return items[0]

# Read-only dict-like access
def lookup(data: Mapping[str, int], key: str) -> int:
    return data.get(key, 0)

# Writable dict-like access
def update(data: MutableMapping[str, int], key: str, value: int) -> None:
    data[key] = value
```

The hierarchy to remember: `Iterable` ☐ `Sequence` ☐ `list/tuple`. Accept the most general type that meets your needs; return the most specific type you produce.

26.8 `TypeVar` — Generic Functions

A `TypeVar` is a placeholder for "some type, I don't know which one yet." It lets you write functions that work on any type while still being type-safe:

```
from typing import TypeVar

T = TypeVar("T")

def first(items: list[T]) -> T:
    return items[0]

def last(items: list[T]) -> T:
    return items[-1]

def identity(value: T) -> T:
    return value
```

The type checker knows that if you call `first([1, 2, 3])`, the return type is `int`. If you call `first(["a", "b"])`, the return type is `str`. The `T` is inferred from what you pass in.

You can constrain a TypeVar to specific types:

```
from typing import TypeVar

Numeric = TypeVar("Numeric", int, float)

def add(a: Numeric, b: Numeric) -> Numeric:
    return a + b

print(add(1, 2))       # 3     (int)
print(add(1.5, 2.5))   # 4.0   (float)
# add("a", "b")  --- mypy would flag this
```

26.9 Generic Classes

You can make your own generic classes using `Generic[T]`:

```
from typing import TypeVar, Generic, Optional

T = TypeVar("T")

class Stack(Generic[T]):
    """A typed stack that holds items of type T."""
```

```
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        if not self._items:
            raise IndexError("Stack is empty.")
        return self._items.pop()

    def peek(self) -> Optional[T]:
        return self._items[-1] if self._items else None

    def __len__(self) -> int:
        return len(self._items)

    def __bool__(self) -> bool:
        return bool(self._items)

    def __repr__(self) -> str:
        return f"Stack({self._items!r})"


int_stack: Stack[int] = Stack()
int_stack.push(1)
int_stack.push(2)
int_stack.push(3)
print(int_stack.peek())    # 3
print(int_stack.pop())     # 3
print(len(int_stack))      # 2

str_stack: Stack[str] = Stack()
str_stack.push("hello")
str_stack.push("world")
print(str_stack.pop())     # world
```

26.10 `Protocol` — Structural Subtyping (Duck Typing, but Typed)

Python uses duck typing — if it has the right methods, it works. `Protocol` brings this into the type system. A class satisfies a Protocol if it has the required methods, without explicitly inheriting from it:

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Drawable(Protocol):
    def draw(self) -> str:
        ...


class Circle:
    def __init__(self, radius: float) -> None:
        self.radius = radius

    def draw(self) -> str:
        return f"○ Circle(r={self.radius})"


class Rectangle:
    def __init__(self, w: float, h: float) -> None:
        self.w, self.h = w, h

    def draw(self) -> str:
        return f"□ Rectangle({self.w}x{self.h})"


class Triangle:
    def draw(self) -> str:
        return "△ Triangle"


def render_all(shapes: list[Drawable]) -> None:
    for shape in shapes:
        print(shape.draw())


shapes = [Circle(5), Rectangle(3, 4), Triangle()]
render_all(shapes)
```

```
# Check at runtime (because of @runtime_checkable)
print(isinstance(Circle(1), Drawable))    # True
print(isinstance("not a shape", Drawable))  # False
```

`Circle`, `Rectangle`, and `Triangle` never inherit from `Drawable`. They just happen to have a `draw()` method. `Protocol` captures that relationship without forcing inheritance.

A richer protocol example:

```
from typing import Protocol

class Comparable(Protocol):
    def __lt__(self, other: "Comparable") -> bool: ...
    def __eq__(self, other: object) -> bool: ...

class Saveable(Protocol):
    def save(self) -> None: ...
    def load(self, path: str) -> None: ...

class Sizeable(Protocol):
    def __len__(self) -> int: ...
```

26.11 `NamedTuple` — Typed Tuples

`NamedTuple` gives you a tuple with named fields and type annotations — immutable, lightweight, and inspectable:

```
from typing import NamedTuple

class Point(NamedTuple):
    x: float
    y: float
    label: str = ""    # default value


class Employee(NamedTuple):
    name:   str
    dept:   str
    salary: float
    active: bool = True


p = Point(3.0, 4.0, "origin")
```

```
print(p)          # Point(x=3.0, y=4.0, label='origin')
print(p.x)        # 3.0
print(p[1])       # 4.0  --- still a tuple
print(p._asdict())    # {'x': 3.0, 'y': 4.0, 'label': 'origin'}

# Can unpack like a regular tuple
x, y, label = p

emp = Employee("Alice", "Eng", 95_000)
print(emp)        # Employee(name='Alice', dept='Eng', salary=95000, active=True)

# Works with sorted, min, max
employees = [
    Employee("Bob",    "HR",    68_000),
    Employee("Alice",  "Eng",   95_000),
    Employee("Carlos", "Sales", 74_000),
]
top = max(employees, key=lambda e: e.salary)
print(top.name)    # Alice
```

Use `NamedTuple` when you want a simple, immutable data carrier with field names. Use `@dataclass` when you need mutability, inheritance, or custom methods.

26.12 `TypedDict` — Typed Dictionaries

`TypedDict` describes a dictionary with specific string keys and typed values — useful for JSON-like data:

```
from typing import TypedDict, Required, NotRequired

class Address(TypedDict):
    street: str
    city:   str
    state:  str
    zip:    str

class User(TypedDict):
    id:      int
    name:    str
    email:   str
    address: Address
```

```python
# Optional keys (Python 3.11+)
class Config(TypedDict, total=False):
    debug:   bool       # optional
    timeout: int        # optional
    host:    str        # optional

# Mix required and optional (Python 3.11+)
class ServerConfig(TypedDict):
    host:    Required[str]
    port:    Required[int]
    debug:   NotRequired[bool]
    timeout: NotRequired[float]


def send_welcome(user: User) -> str:
    return f"Welcome, {user['name']}! Your ID is {user['id']}."


alice: User = {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com",
    "address": {
        "street": "123 Main St",
        "city": "Springfield",
        "state": "IL",
        "zip": "62701",
    },
}

print(send_welcome(alice))
```

`TypedDict` is a type-checking construct only — at runtime, it's just a plain dict. `mypy` uses the annotation to catch key typos and wrong value types.

26.13 Literal Types

`Literal` restricts a value to specific constants:

```python
from typing import Literal

Mode = Literal["r", "w", "a", "r+", "w+"]
```

```
def open_file(path: str, mode: Mode = "r") -> None:
    with open(path, mode) as f:
        pass

open_file("data.txt", "r")     # OK
open_file("data.txt", "x")     # mypy flags this --- "x" not in Literal
```

```
Direction = Literal["north", "south", "east", "west"]
Status    = Literal["pending", "active", "closed"]
LogLevel  = Literal["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]

def move(direction: Direction, steps: int) -> None:
    print(f"Moving {direction} {steps} step(s)")

move("north", 3)       # OK
move("up",    1)       # mypy: Argument 1 to "move" has incompatible type "Literal[
```

26.14 `Final` — Constants

`Final` marks a variable as constant — it cannot be reassigned:

```
from typing import Final

MAX_RETRIES: Final = 3
DEFAULT_HOST: Final[str] = "localhost"
PI: Final[float] = 3.14159265358979

MAX_RETRIES = 5   # mypy: Cannot assign to final name "MAX_RETRIES"
```

26.15 Running `mypy`

Install `mypy` and run it on your code:

```
pip install mypy
mypy my_script.py
```

A simple example to show what `mypy` catches:

```
# buggy.py
def greet(name: str) -> str:
    return f"Hello, {name}!"

def add(a: int, b: int) -> int:
```

```
    return a + b

result = greet(42)         # mypy: Argument 1 to "greet" has incompatible type "
total  = add("a", "b")     # mypy: Argument 1 to "add" has incompatible type "str
```

```
$ mypy buggy.py
buggy.py:7: error: Argument 1 to "greet" has incompatible type "int"; expected
buggy.py:8: error: Argument 1 to "add" has incompatible type "str"; expected "i
Found 2 errors in 1 file (checked 1 source file)
```

Common `mypy` configuration in `mypy.ini` or `pyproject.toml`:

```
# mypy.ini
[mypy]
python_version = 3.12
strict = True
ignore_missing_imports = True
```

`strict` enables the strictest checks: no implicit `Any`, no untyped function definitions, no unannotated variables. Good for new projects.

26.16 Type Aliases

Give complex types a readable name:

```
from typing import TypeAlias

# Simple alias
JSON: TypeAlias = dict[str, "JSON"] | list["JSON"] | str | int | float | bool |

# Function type
Predicate: TypeAlias = Callable[[int], bool]
Transformer: TypeAlias = Callable[[str], str]

# Nested types
Row: TypeAlias = dict[str, str | int | float | None]
Table: TypeAlias = list[Row]

def filter_table(table: Table, predicate: Callable[[Row], bool]) -> Table:
    return [row for row in table if predicate(row)]
```

In Python 3.12, the `type` statement makes aliases cleaner:

```
# Python 3.12+
type JSON = dict[str, JSON] | list[JSON] | str | int | float | bool | None
type Row  = dict[str, str | int | float | None]
```

26.17 Project: A Typed Data Pipeline

Let's build a small typed data pipeline that reads a CSV, validates the data, transforms it, and produces a report — fully annotated.

```
"""
pipeline.py --- A typed data processing pipeline.
"""
from __future__ import annotations   # allow forward references in annotations

import csv
from dataclasses import dataclass, field
from pathlib import Path
from typing import Callable, Iterator, TypeVar, NamedTuple

T = TypeVar("T")


# ── Data models ──────────────────────────────────────────────

class RawRecord(NamedTuple):
    name:   str
    dept:   str
    salary: str   # raw string from CSV
    active: str   # raw string from CSV


@dataclass(frozen=True)
class Employee:
    name:   str
    dept:   str
    salary: float
    active: bool

    @classmethod
    def from_raw(cls, raw: RawRecord) -> Employee:
        return cls(
            name   = raw.name.strip(),
            dept   = raw.dept.strip(),
            salary = float(raw.salary),
            active = raw.active.strip().lower() in ("true", "1", "yes"),
        )
```

```
@dataclass
class Report:
    dept:       str
    headcount:  int
    avg_salary: float
    max_salary: float
    min_salary: float


# ── Pipeline stages ──────────────────────────────────────────────

def read_csv(path: Path) -> Iterator[RawRecord]:
    with open(path, newline="") as f:
        reader = csv.DictReader(f)
        for row in reader:
            yield RawRecord(
                name   = row["name"],
                dept   = row["dept"],
                salary = row["salary"],
                active = row.get("active", "true"),
            )


def parse_employees(records: Iterator[RawRecord]) -> Iterator[Employee]:
    for record in records:
        try:
            yield Employee.from_raw(record)
        except (ValueError, KeyError) as e:
            print(f"  Skipping invalid record {record.name!r}: {e}")


def filter_employees(
    employees: Iterator[Employee],
    predicate: Callable[[Employee], bool],
) -> Iterator[Employee]:
    return (e for e in employees if predicate(e))


def group_by_dept(employees: Iterator[Employee]) -> dict[str, list[Employee]]:
    groups: dict[str, list[Employee]] = {}
    for emp in employees:
        groups.setdefault(emp.dept, []).append(emp)
```

```
    return groups


def make_report(dept: str, members: list[Employee]) -> Report:
    salaries = [e.salary for e in members]
    return Report(
        dept       = dept,
        headcount  = len(members),
        avg_salary = sum(salaries) / len(salaries),
        max_salary = max(salaries),
        min_salary = min(salaries),
    )


def print_report(reports: list[Report]) -> None:
    print(f"\n{'Department':<12} {'Count':>6} {'Avg Salary':>12} {'Min':>10} {'Max
    print("-" * 55)
    for r in sorted(reports, key=lambda r: r.dept):
        print(
            f"{r.dept:<12} {r.headcount:>6} "
            f"${r.avg_salary:>10,.0f} "
            f"${r.min_salary:>8,.0f} "
            f"${r.max_salary:>8,.0f}"
        )


# ── Main ──────────────────────────────────────────────────────────────

def run(path: Path, active_only: bool = True) -> list[Report]:
    raw       = read_csv(path)
    employees = parse_employees(raw)

    if active_only:
        employees = filter_employees(employees, lambda e: e.active)

    groups  = group_by_dept(employees)
    reports = [make_report(dept, members) for dept, members in groups.items()]
    print_report(reports)
    return reports


if __name__ == "__main__":
    run(Path("employees.csv"))
```

Every function has annotated parameters and return types. The type checker knows that `read_csv` yields `RawRecord`, `parse_employees` yields `Employee`, `group_by_dept` returns `dict[str, list[Employee]]`, and `make_report` returns a `Report`. If you wire them up wrong, `mypy` tells you before you run the code.

26.18 Type Hint Cheat Sheet

Syntax	Meaning
`x: int`	`x` is an integer
`x: str | None`	`x` is a string or `None`
`x: list[int]`	list of integers
`x: dict[str, int]`	dict mapping str keys to int values
`x: tuple[int, str]`	2-tuple: int then str
`x: tuple[int, ...]`	tuple of any number of ints
`x: set[str]`	set of strings
`x: frozenset[int]`	frozenset of integers
`Optional[X]`	`X | None`
`Union[X, Y]`	`X | Y`
`Any`	any type — opt out of checking
`Callable[[int, str], bool]`	function taking int and str, returning bool
`Callable[..., int]`	function with any args, returning int
`Iterable[X]`	anything you can loop over
`Sequence[X]`	anything with len and indexing
`Mapping[K, V]`	read-only dict-like
`TypeVar("T")`	generic type placeholder
`Generic[T]`	base for generic classes
`Protocol`	structural subtyping (duck typing)
`NamedTuple`	typed named tuple
`TypedDict`	typed dictionary
`Literal["a", "b"]`	only these specific values
`Final`	cannot be reassigned

26.19 What You Learned in This Chapter

- Type hints don't change runtime behavior — they're documentation for humans and type checkers.

- Use `->` for return types and `:` for parameter and variable types.
- Python 3.9+ supports `list[int]`, `dict[str, int]` directly; earlier versions use `typing.List`, `typing.Dict`.
- Python 3.10+ supports `X | Y` instead of `Union[X, Y]`.
- `Optional[X]` means `X | None` — very common for nullable values.
- `Callable`, `Iterable`, `Sequence`, `Mapping` describe abstract interfaces, not concrete types.
- `TypeVar` enables generic functions that preserve type information.
- `Generic[T]` creates generic classes.
- `Protocol` describes duck-typing contracts without inheritance.
- `NamedTuple` gives typed, immutable named tuples.
- `TypedDict` gives typed dictionaries for JSON-like data.
- `Literal` restricts values to specific constants; `Final` prevents reassignment.
- Run `mypy` to catch type errors before they become runtime bugs.

26.20 What's Next?

Chapter 27 covers **Concurrency** — `threading`, `multiprocessing`, and `asyncio`. Python's Global Interpreter Lock (GIL) limits threads for CPU work, but they shine for I/O. `asyncio` lets a single thread handle thousands of concurrent I/O operations. You'll learn when to use each and how to write both threaded and async code correctly.

Chapter 27

Chapter 27: Concurrency — Threading, Multiprocessing, and asyncio

Your computer has multiple cores. Most Python programs use exactly one of them. This chapter shows you how to use the rest — and how to handle situations where your program spends most of its time waiting rather than computing.

There are three tools:

- **Threading** — multiple threads in one process, good for I/O-bound work
- **Multiprocessing** — multiple processes with separate memory, good for CPU-bound work
- **asyncio** — a single thread handling thousands of I/O operations concurrently

Understanding which tool to reach for is more than half the battle. Let's start there.

27.1 The Two Types of Bottleneck

I/O-bound work spends most of its time waiting — for network responses, disk reads, database queries, user input. The CPU is mostly idle.

CPU-bound work spends most of its time computing — parsing files, running algorithms, processing images, training models. The CPU is pegged at 100%.

```
I/O-bound:  |——wait——|compute|——wait——|——wait——|compute|——wait——|
CPU-bound:  |compute|compute|compute|compute|compute|compute|

Solutions:
  I/O-bound  -> threading or asyncio  (overlap the waiting)
  CPU-bound  -> multiprocessing       (use multiple cores)
```

Threading does NOT help with CPU-bound work in Python — because of the **Global Interpreter Lock (GIL)**.

27.2 The GIL — Why Threads Don't Parallelize CPU Work

Python's interpreter uses a lock called the GIL (Global Interpreter Lock). Only one thread can execute Python bytecode at a time. For CPU-bound work, threads take turns — you get no speedup.

For I/O-bound work, the GIL is released during I/O operations (network calls, file reads, `time.sleep`). While one thread waits, another runs. This is why threading helps for I/O.

```
Without concurrency:
  request 1: |——wait 2s——|
  request 2:              |——wait 2s——|
  request 3:                          |——wait 2s——|
  Total: 6 seconds

With threading (I/O releases GIL):
  request 1: |——wait 2s——|
  request 2: |——wait 2s——|  (started immediately)
  request 3: |——wait 2s——|  (started immediately)
  Total: ~2 seconds
```

27.3 Threading

`threading.Thread` runs a function in a separate thread:

```
import threading
import time
```

```
def download(url, results, index):
    """Simulate downloading a URL."""
    print(f"  Starting: {url}")
    time.sleep(2)    # simulate network wait
    results[index] = f"Content of {url}"
    print(f"  Done:     {url}")


urls = [
    "https://example.com/page1",
    "https://example.com/page2",
    "https://example.com/page3",
]

results = [None] * len(urls)

# Sequential --- 6 seconds
start = time.perf_counter()
for i, url in enumerate(urls):
    download(url, results, i)
print(f"Sequential: {time.perf_counter() - start:.1f}s")

# Threaded --- ~2 seconds
results = [None] * len(urls)
threads = []

start = time.perf_counter()
for i, url in enumerate(urls):
    t = threading.Thread(target=download, args=(url, results, i))
    threads.append(t)
    t.start()

for t in threads:
    t.join()    # wait for all threads to finish

print(f"Threaded:   {time.perf_counter() - start:.1f}s")
print(results)
```

Output:

```
Sequential: 6.0s

  Starting: https://example.com/page1
  Starting: https://example.com/page2
```

```
  Starting: https://example.com/page3
  Done:     https://example.com/page1
  Done:     https://example.com/page2
  Done:     https://example.com/page3
Threaded:   2.0s
```

27.3.1 `ThreadPoolExecutor` — the cleaner way

`concurrent.futures.ThreadPoolExecutor` manages a pool of threads for you:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import time

def fetch(url):
    """Simulate fetching a URL. Returns (url, content)."""
    time.sleep(1.5)
    return url, f"<html>Content from {url}</html>"


urls = [f"https://example.com/page{i}" for i in range(8)]

start = time.perf_counter()

with ThreadPoolExecutor(max_workers=4) as executor:
    # Submit all tasks
    futures = {executor.submit(fetch, url): url for url in urls}

    # Process results as they complete
    for future in as_completed(futures):
        url, content = future.result()
        print(f"  Got {len(content)} bytes from {url}")

print(f"Total: {time.perf_counter() - start:.1f}s")
# 8 requests, 4 at a time, each takes 1.5s -> ~3s total
```

`as_completed` yields futures in the order they finish — not the order they were submitted. This is useful when you want to process results as soon as they're ready.

If you just want results in submission order:

```
with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(fetch, urls))
    # blocks until all are done, returns in order
```

27.3.2 Thread safety and locks

Threads share memory. When multiple threads modify the same data at the same time, you get **race conditions** — unpredictable, hard-to-reproduce bugs.

```
import threading

counter = 0

def increment(n):
    global counter
    for _ in range(n):
        counter += 1    # NOT thread-safe: read-modify-write


threads = [threading.Thread(target=increment, args=(100_000,)) for _ in range(5
for t in threads:
    t.start()
for t in threads:
    t.join()

print(f"Expected: 500000, Got: {counter}")
# Expected: 500000, Got: 423891  (varies each run --- race condition)
```

Fix with a `Lock`:

```
import threading

counter = 0
lock    = threading.Lock()

def increment_safe(n):
    global counter
    for _ in range(n):
        with lock:          # only one thread at a time
            counter += 1


threads = [threading.Thread(target=increment_safe, args=(100_000,)) for _ in ra
for t in threads:
    t.start()
for t in threads:
    t.join()

print(f"Expected: 500000, Got: {counter}")    # Expected: 500000, Got: 500000
```

Other synchronization primitives:

```
import threading

# RLock --- re-entrant lock (same thread can acquire multiple times)
rlock = threading.RLock()

# Semaphore --- limit concurrent access to N threads
sem = threading.Semaphore(3)    # max 3 threads at a time
with sem:
    pass    # do work

# Event --- signal between threads
event = threading.Event()
event.set()       # signal
event.clear()     # unsignal
event.wait()      # block until signaled

# Barrier --- wait for N threads to arrive before any proceed
barrier = threading.Barrier(4)
barrier.wait()    # block until 4 threads reach this line
```

27.4 Multiprocessing

For CPU-bound work, use `multiprocessing`. Each process has its own Python interpreter and its own GIL — true parallelism.

```
import multiprocessing
import time

def cpu_work(n):
    """A CPU-bound computation."""
    return sum(i * i for i in range(n))


data = [5_000_000] * 8    # 8 chunks of work

# Sequential
start = time.perf_counter()
results = [cpu_work(n) for n in data]
print(f"Sequential: {time.perf_counter() - start:.2f}s")

# Multiprocessing
```

```
start = time.perf_counter()
with multiprocessing.Pool() as pool:    # uses all CPU cores
    results = pool.map(cpu_work, data)
print(f"Parallel:   {time.perf_counter() - start:.2f}s")
```

On a 4-core machine, the parallel version is roughly 4x faster.

27.4.1 `ProcessPoolExecutor` — mirrors `ThreadPoolExecutor`

```
from concurrent.futures import ProcessPoolExecutor, as_completed
import time

def process_chunk(chunk):
    """CPU-intensive processing of a data chunk."""
    return sum(x ** 2 for x in chunk)


# Split work into chunks
data = list(range(10_000_000))
chunk_size = len(data) // 8
chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

start = time.perf_counter()

with ProcessPoolExecutor() as executor:
    results = list(executor.map(process_chunk, chunks))

total = sum(results)
print(f"Total: {total}")
print(f"Time:  {time.perf_counter() - start:.2f}s")
```

Important rules for multiprocessing: - Arguments and return values must be **picklable** (basic Python types, dataclasses, named tuples — but not lambdas, file handles, or locks). - Always protect the entry point with `if __name__ == "__main__":` on Windows. - Processes have separate memory — no shared state without explicit shared memory or queues.

```
# Always do this on Windows when using multiprocessing
if __name__ == "__main__":
    with ProcessPoolExecutor() as executor:
        results = list(executor.map(cpu_work, data))
```

27.4.2 Sharing data between processes

```
from multiprocessing import Process, Queue, Value, Array
import multiprocessing

# Queue --- safe for inter-process communication
def producer(queue, items):
    for item in items:
        queue.put(item)
    queue.put(None)    # sentinel value

def consumer(queue, results):
    while True:
        item = queue.get()
        if item is None:
            break
        results.append(item * 2)


# Shared memory --- for simple types
counter = multiprocessing.Value("i", 0)    # shared integer
array   = multiprocessing.Array("d", [1.0, 2.0, 3.0])  # shared float array

def increment_shared(counter, n):
    for _ in range(n):
        with counter.get_lock():
            counter.value += 1
```

For most use cases, prefer passing data via `pool.map()` arguments and return values, and avoid shared state entirely.

27.5 `asyncio` — Async/Await

`asyncio` is a fundamentally different model. Instead of multiple threads or processes, **one thread** manages many tasks by switching between them at `await` points.

When a task hits an `await`, it says: "I'm waiting for something. Run something else while I wait." The event loop picks another task and runs it. No OS threads, no locking, no race conditions on shared data (within a single async program).

```
import asyncio
import time
```

```
async def fetch(url, delay):
    """Simulate an async HTTP request."""
    print(f"  Fetching {url}...")
    await asyncio.sleep(delay)    # yields control to the event loop
    print(f"  Done:     {url}")
    return f"Content of {url}"


async def main():
    start = time.perf_counter()

    # Run all fetches concurrently
    results = await asyncio.gather(
        fetch("https://example.com/page1", 1.5),
        fetch("https://example.com/page2", 2.0),
        fetch("https://example.com/page3", 1.0),
    )

    elapsed = time.perf_counter() - start
    print(f"\nAll done in {elapsed:.2f}s")
    for r in results:
        print(f"  {r}")


asyncio.run(main())
```

Output:

```
  Fetching https://example.com/page1...
  Fetching https://example.com/page2...
  Fetching https://example.com/page3...
  Done:     https://example.com/page3
  Done:     https://example.com/page1
  Done:     https://example.com/page2

All done in 2.00s   <- the longest task, not the sum
```

Three 1-2 second fetches complete in 2 seconds total — the same as threading, but with zero threads.

27.5.1 `async` and `await` — the rules

An **async function** is declared with `async def`. It returns a **coroutine** when called — not the result. The result comes when you `await` it.

```python
async def greet(name: str) -> str:
    await asyncio.sleep(0)    # yield to the event loop (trivial here)
    return f"Hello, {name}!"

# This does NOT call greet --- it creates a coroutine object
coro = greet("Alice")

# This runs it and gets the result
result = await greet("Alice")    # only valid inside an async function

# Or run it as the top-level entry point
asyncio.run(greet("Alice"))
```

You can only `await` inside an `async def` function. You can only call `asyncio.run()` once, at the top level.

27.5.2 `asyncio.gather` — run tasks concurrently

```python
async def main():
    # All three run concurrently --- total time = max(individual times)
    a, b, c = await asyncio.gather(
        task_one(),
        task_two(),
        task_three(),
    )
```

27.5.3 `asyncio.create_task` — fire and forget

```python
async def main():
    # Start a task without immediately waiting for it
    task = asyncio.create_task(background_work())

    # Do other work while background_work runs
    await do_something_else()

    # Now wait for the background task
    result = await task
```

27.5.4 Async context managers and iterators

```python
import asyncio

class AsyncTimer:
    async def __aenter__(self):
        self.start = asyncio.get_event_loop().time()
```

```
        return self

    async def __aexit__(self, *args):
        elapsed = asyncio.get_event_loop().time() - self.start
        print(f"Elapsed: {elapsed:.3f}s")


async def async_range(n):
    for i in range(n):
        await asyncio.sleep(0)    # yield between each item
        yield i


async def main():
    # Async context manager
    async with AsyncTimer():
        await asyncio.sleep(0.5)

    # Async iterator
    async for i in async_range(5):
        print(i, end=" ")
    print()


asyncio.run(main())
```

27.5.5 Async with `aiohttp` — real HTTP requests

For real async HTTP in production, use `aiohttp` or `httpx`:

```
# pip install aiohttp
import asyncio
import aiohttp

async def fetch_url(session: aiohttp.ClientSession, url: str) -> str:
    async with session.get(url) as response:
        return await response.text()


async def fetch_all(urls: list[str]) -> list[str]:
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        return await asyncio.gather(*tasks)
```

```
urls = [
    "https://httpbin.org/delay/1",
    "https://httpbin.org/delay/1",
    "https://httpbin.org/delay/1",
]

results = asyncio.run(fetch_all(urls))
# 3 requests x 1s each -> ~1s total (not 3s)
```

27.6 Choosing the Right Tool

Situation	Tool	Why
Making many HTTP requests	`asyncio` + `aiohttp`	Most efficient for I/O
Reading many files at once	`ThreadPoolExecutor`	Simple, I/O releases GIL
Compressing images	`ProcessPoolExecutor`	CPU-bound, needs real parallelism
Parsing large CSV files	`ProcessPoolExecutor`	CPU-bound
Running a web server	`asyncio` (FastAPI/aiohttp)	Handles thousands of connections
Scraping websites	`asyncio` or threads	I/O-bound
Data science computation	`multiprocessing` or `numpy`	CPU-bound
Simple background task	`threading.Thread`	Simplest option

Quick decision tree:

```
Is your bottleneck I/O (network, disk)?
  ├── Yes, and you want maximum concurrency -> asyncio
  ├── Yes, and you want simplicity          -> ThreadPoolExecutor
  └── No (CPU is the bottleneck)            -> ProcessPoolExecutor
```

27.7 Project: Async Web Scraper with Rate Limiting

```
"""
scraper.py --- Async web scraper with rate limiting, retries, and timeout.
```

```
"""
import asyncio
import time
from dataclasses import dataclass, field
from typing import Optional
import random


@dataclass
class ScrapeResult:
    url:     str
    status:  int
    content: Optional[str]
    elapsed: float
    error:   Optional[str] = None


class RateLimiter:
    """Allow at most `rate` operations per second."""
    def __init__(self, rate: float):
        self.rate     = rate
        self.min_gap  = 1.0 / rate
        self._last    = 0.0
        self._lock    = asyncio.Lock()

    async def acquire(self):
        async with self._lock:
            now  = asyncio.get_event_loop().time()
            wait = self.min_gap - (now - self._last)
            if wait > 0:
                await asyncio.sleep(wait)
            self._last = asyncio.get_event_loop().time()


async def fetch_with_retry(
    url: str,
    limiter: RateLimiter,
    retries: int = 3,
    timeout: float = 5.0,
) -> ScrapeResult:
    """Fetch url respecting rate limit, with retries on failure."""
    await limiter.acquire()

    start = time.perf_counter()
```

```
    for attempt in range(1, retries + 1):
        try:
            # Simulate HTTP request (replace with aiohttp in production)
            await asyncio.wait_for(simulate_request(url), timeout=timeout)
            elapsed = time.perf_counter() - start
            return ScrapeResult(
                url     = url,
                status  = 200,
                content = f"<html>Content from {url}</html>",
                elapsed = elapsed,
            )
        except asyncio.TimeoutError:
            if attempt == retries:
                elapsed = time.perf_counter() - start
                return ScrapeResult(
                    url=url, status=408, content=None,
                    elapsed=elapsed, error="Timeout"
                )
            await asyncio.sleep(0.5 * attempt)    # exponential back-off
        except Exception as e:
            if attempt == retries:
                elapsed = time.perf_counter() - start
                return ScrapeResult(
                    url=url, status=500, content=None,
                    elapsed=elapsed, error=str(e)
                )
            await asyncio.sleep(0.5 * attempt)

    # Should not reach here
    return ScrapeResult(url=url, status=500, content=None, elapsed=0, error="Unkno


async def simulate_request(url: str) -> None:
    """Simulate a network request (0.1--1.5s, 15% failure rate)."""
    delay = random.uniform(0.1, 1.5)
    await asyncio.sleep(delay)
    if random.random() < 0.15:
        raise ConnectionError("Simulated connection error")


async def scrape_all(
    urls: list[str],
    max_concurrent: int = 10,
```

```
    rate_per_second: float = 5.0,
) -> list[ScrapeResult]:
    """Scrape all URLs with concurrency and rate limiting."""
    limiter   = RateLimiter(rate_per_second)
    semaphore = asyncio.Semaphore(max_concurrent)

    async def bounded_fetch(url):
        async with semaphore:
            return await fetch_with_retry(url, limiter)

    tasks   = [bounded_fetch(url) for url in urls]
    results = await asyncio.gather(*tasks)
    return list(results)


def print_summary(results: list[ScrapeResult]) -> None:
    ok      = [r for r in results if r.error is None]
    failed  = [r for r in results if r.error is not None]
    avg_ms  = sum(r.elapsed for r in ok) / len(ok) * 1000 if ok else 0

    print(f"\n{'URL':<35} {'Status':>6} {'Time':>8}  {'Error'}")
    print("-" * 65)
    for r in sorted(results, key=lambda r: r.url):
        status = str(r.status) if r.error is None else f"ERR"
        time_s = f"{r.elapsed*1000:.0f}ms"
        error  = r.error or ""
        print(f"  {r.url:<33} {status:>6} {time_s:>8}  {error}")

    print(f"\nSummary: {len(ok)}/{len(results)} succeeded, avg {avg_ms:.0f}ms p
    if failed:
        print(f"Failed:  {[r.url for r in failed]}")


if __name__ == "__main__":
    urls = [f"https://example.com/page{i:02d}" for i in range(20)]

    start = time.perf_counter()
    results = asyncio.run(scrape_all(urls, max_concurrent=5, rate_per_second=4.
    total = time.perf_counter() - start

    print_summary(results)
    print(f"\nTotal wall time: {total:.2f}s")
```

This scraper uses: - `asyncio.Semaphore` to limit concurrency (max 5 at a

time) - `RateLimiter` with an `asyncio.Lock` to cap requests per second - `asyncio.wait_for` for timeouts - Exponential back-off on retries - `asyncio.gather` to run all tasks concurrently

---

27.8 What You Learned in This Chapter

- **I/O-bound** programs wait for network/disk — threads and asyncio help by overlapping the waiting.
- **CPU-bound** programs compute constantly — only multiprocessing gives real speedup in Python.
- The **GIL** prevents threads from running Python code truly in parallel, but releases during I/O.
- `threading.Thread` and `ThreadPoolExecutor` run functions in threads — use for I/O-bound work.
- Use a `Lock` to protect shared data from race conditions.
- `ProcessPoolExecutor` and `multiprocessing.Pool` run functions in separate processes — use for CPU-bound work. Arguments must be picklable. Guard with `if __name__ == "__main__":`.
- `async def` and `await` write cooperative coroutines. The event loop switches between them at every `await`.
- `asyncio.gather` runs multiple coroutines concurrently.
- `asyncio.create_task` starts a coroutine without waiting for it immediately.
- `asyncio.Semaphore` limits how many coroutines run at once.
- Use `aiohttp` or `httpx` for real async HTTP requests.

27.9 What's Next?

Chapter 28 covers **Testing** — `unittest`, `pytest`, fixtures, parametrize, mocking, and test-driven development. Testing is how you prove your code works, catch regressions, and refactor with confidence. If you write tests, you can change anything. If you don't, every change is a gamble.

Chapter 28

Chapter 28: Testing with pytest

Code without tests is code you can't safely change. Every refactor becomes a gamble. Every bug fix might break something else. Tests are the net that catches you.

This chapter teaches you to write tests with `pytest` — Python's most popular testing framework. You'll go from zero tests to a full suite with fixtures, parametrize, mocking, and coverage. By the end you'll write tests as naturally as you write functions.

28.1 Why Test?

Tests give you three superpowers:

1. **Confidence** — you know the code does what you think it does.
2. **Safety** — when you change code, tests tell you immediately if something broke.
3. **Documentation** — tests show how code is supposed to be used, with real examples.

The best time to write a test is right after you write the function. The second best time is right now.

28.2 Installing and Running pytest

```
pip install pytest pytest-cov
```

Create a file that starts with `test_` or ends with `_test.py`. Put your test functions inside it, naming each one `test_something`:

```
my_project/
├── calculator.py
└── test_calculator.py
```

Run all tests:

```
pytest                      # run everything
pytest test_calculator.py   # run one file
pytest -v                   # verbose output (shows each test name)
pytest -k "add"             # run tests whose names contain "add"
pytest --tb=short           # shorter tracebacks
```

28.3 Your First Tests

Let's test a simple calculator module:

```
# calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero.")
    return a / b

def power(base, exp):
    return base ** exp
```

```
# test_calculator.py
from calculator import add, subtract, multiply, divide, power
```

```python
def test_add_integers():
    assert add(2, 3) == 5

def test_add_floats():
    assert add(1.5, 2.5) == 4.0

def test_add_negative():
    assert add(-1, -2) == -3

def test_add_zero():
    assert add(5, 0) == 5

def test_subtract():
    assert subtract(10, 3) == 7

def test_multiply():
    assert multiply(4, 5) == 20

def test_divide():
    assert divide(10, 2) == 5.0

def test_divide_by_zero():
    import pytest
    with pytest.raises(ZeroDivisionError):
        divide(5, 0)

def test_power():
    assert power(2, 10) == 1024
```

Run them:

```
$ pytest test_calculator.py -v

test_calculator.py::test_add_integers   PASSED
test_calculator.py::test_add_floats     PASSED
test_calculator.py::test_add_negative   PASSED
test_calculator.py::test_add_zero       PASSED
test_calculator.py::test_subtract       PASSED
test_calculator.py::test_multiply       PASSED
test_calculator.py::test_divide         PASSED
test_calculator.py::test_divide_by_zero PASSED
test_calculator.py::test_power          PASSED

9 passed in 0.05s
```

28.3.1 Reading a failure

Break a test on purpose:

```
def test_add_integers():
    assert add(2, 3) == 99    # wrong expected value
```

```
FAILED test_calculator.py::test_add_integers

    def test_add_integers():
>       assert add(2, 3) == 99
E       AssertionError: assert 5 == 99
E        +  where 5 = add(2, 3)
```

pytest shows you the exact expression that failed, the actual value, and how it was computed. No print statements needed.

28.4 `assert` — The Only Assertion You Need

pytest rewrites `assert` statements to give detailed error messages. Use plain `assert` for everything:

```
# Values
assert result == expected
assert result != unexpected
assert result > 0
assert result is None
assert result is not None

# Strings
assert "error" in message
assert message.startswith("Hello")

# Collections
assert len(items) == 3
assert item in collection
assert collection == [1, 2, 3]

# Floats --- never use == for floats
assert abs(result - expected) < 1e-9
# Or pytest's built-in approx:
import pytest
assert result == pytest.approx(expected)
assert result == pytest.approx(3.14, rel=1e-3)   # relative tolerance
assert result == pytest.approx(0.0,  abs=1e-6)   # absolute tolerance
```

```python
# Examples with approx
from calculator import divide
import pytest

def test_divide_result():
    assert divide(1, 3) == pytest.approx(0.3333, rel=1e-3)

def test_celsius_to_fahrenheit():
    def c_to_f(c):
        return c * 9 / 5 + 32

    assert c_to_f(100) == pytest.approx(212.0)
    assert c_to_f(0)   == pytest.approx(32.0)
    assert c_to_f(37)  == pytest.approx(98.6, rel=1e-3)
```

28.5 `pytest.raises` — Testing Exceptions

```python
import pytest
from calculator import divide

def test_divide_by_zero_raises():
    with pytest.raises(ZeroDivisionError):
        divide(10, 0)

def test_divide_by_zero_message():
    with pytest.raises(ZeroDivisionError, match="Cannot divide by zero"):
        divide(10, 0)

def test_raises_captures_exception():
    with pytest.raises(ValueError) as exc_info:
        int("not a number")

    assert "invalid literal" in str(exc_info.value)
    assert exc_info.type is ValueError
```

`match` takes a regex pattern — the test passes only if the exception message matches.

28.6 Parametrize — Test Many Inputs at Once

`@pytest.mark.parametrize` runs the same test function with different inputs. Instead of writing five nearly identical test functions, write one:

```
import pytest
from calculator import add, divide, power

@pytest.mark.parametrize("a, b, expected", [
    (2,   3,    5),
    (0,   0,    0),
    (-1, -2,   -3),
    (1.5, 2.5,  4.0),
    (100, -100,  0),
])
def test_add(a, b, expected):
    assert add(a, b) == expected


@pytest.mark.parametrize("base, exp, expected", [
    (2,  0,   1),
    (2,  1,   2),
    (2,  10,  1024),
    (3,  3,   27),
    (10, 3,   1000),
])
def test_power(base, exp, expected):
    assert power(base, exp) == expected


@pytest.mark.parametrize("a, b", [
    (1, 0),
    (0, 0),
    (-5, 0),
])
def test_divide_by_zero(a, b):
    with pytest.raises(ZeroDivisionError):
        divide(a, b)
```

pytest generates one test case per set of parameters — each is named and runs independently:

```
test_calculator.py::test_add[2-3-5]          PASSED
test_calculator.py::test_add[0-0-0]          PASSED
test_calculator.py::test_add[-1--2--3]        PASSED
test_calculator.py::test_add[1.5-2.5-4.0]    PASSED
```

```
test_calculator.py::test_add[100--100-0]      PASSED
```

28.7 Fixtures — Shared Setup and Teardown

A **fixture** is a function that prepares something your tests need. Fixtures replace repetitive `setUp`/`tearDown` code. pytest injects them by matching the parameter name:

```
import pytest
from pathlib import Path
import tempfile, os


@pytest.fixture
def sample_numbers():
    """Provide a list of numbers for tests."""
    return [1, 2, 3, 4, 5, 10, -3, 0]


@pytest.fixture
def temp_file(tmp_path):
    """Create a temporary file with some content. tmp_path is a pytest built-in
    file = tmp_path / "test_data.txt"
    file.write_text("line one\nline two\nline three\n")
    return file


def test_sum(sample_numbers):
    assert sum(sample_numbers) == 22

def test_min(sample_numbers):
    assert min(sample_numbers) == -3

def test_max(sample_numbers):
    assert max(sample_numbers) == 10

def test_file_exists(temp_file):
    assert temp_file.exists()

def test_file_content(temp_file):
    lines = temp_file.read_text().splitlines()
    assert len(lines) == 3
```

```
    assert lines[0] == "line one"
```

28.7.1 Fixture scope — how long it lives

By default, fixtures run fresh for every test. Change `scope` to share them:

```
@pytest.fixture(scope="session")    # once for the entire test session
def database_connection():
    conn = connect_to_db()
    yield conn      # yield instead of return when you need cleanup
    conn.close()

@pytest.fixture(scope="module")     # once per test file
def shared_config():
    return load_config("test_config.yaml")

@pytest.fixture(scope="function")   # default --- once per test
def fresh_list():
    return []
```

28.7.2 Fixtures with teardown

Use `yield` to run cleanup code after the test:

```
@pytest.fixture
def database(tmp_path):
    """Create a test database, yield it, then delete it."""
    import sqlite3
    db_path = tmp_path / "test.db"
    conn = sqlite3.connect(db_path)
    conn.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)")
    conn.commit()

    yield conn   # test runs here

    conn.close()
    # db_path is automatically cleaned up by tmp_path fixture


def test_insert_user(database):
    database.execute("INSERT INTO users (name) VALUES (?)", ("Alice",))
    database.commit()
    row = database.execute("SELECT name FROM users").fetchone()
    assert row[0] == "Alice"
```

```
def test_empty_database(database):
    rows = database.execute("SELECT * FROM users").fetchall()
    assert rows == []
```

28.7.3 Built-in fixtures

pytest ships with several useful fixtures:

```
def test_tmp_path(tmp_path):
    # tmp_path: a temporary directory unique to the test invocation
    file = tmp_path / "output.txt"
    file.write_text("hello")
    assert file.read_text() == "hello"

def test_capsys(capsys):
    # capsys: capture stdout and stderr
    print("Hello, pytest!")
    captured = capsys.readouterr()
    assert captured.out == "Hello, pytest!\n"

def test_monkeypatch(monkeypatch):
    # monkeypatch: temporarily replace functions, env vars, attributes
    monkeypatch.setenv("API_KEY", "test-key-123")
    import os
    assert os.environ["API_KEY"] == "test-key-123"
```

28.8 Mocking — Replace Real Dependencies

Tests should be fast, isolated, and repeatable. Network calls, databases, and the clock make tests slow and flaky. **Mocking** replaces real dependencies with fake ones you control.

```
from unittest.mock import Mock, MagicMock, patch, call
```

28.8.1 `Mock` — a fake object

```
from unittest.mock import Mock

# Create a mock
m = Mock()

# Call it --- returns a Mock
result = m(1, 2, 3)
```

```
print(type(result))   # <class 'unittest.mock.Mock'>

# Configure the return value
m.return_value = 42
assert m() == 42

# Configure side effects
m.side_effect = ValueError("boom")
try:
    m()
except ValueError as e:
    print(e)   # boom

# Check calls
m = Mock(return_value="ok")
m("hello", key="value")
m.assert_called_once_with("hello", key="value")
print(m.call_count)    # 1
print(m.call_args)     # call('hello', key='value')
```

28.8.2 `patch` — replace real objects temporarily

```
# email_sender.py
import smtplib

def send_email(to, subject, body):
    with smtplib.SMTP("smtp.gmail.com") as server:
        server.sendmail("from@example.com", to, f"Subject: {subject}\n\n{body}")
    return True
```

```
# test_email.py
from unittest.mock import patch, MagicMock

def test_send_email():
    with patch("smtplib.SMTP") as mock_smtp:
        # mock_smtp is called when smtplib.SMTP() is used
        mock_server = MagicMock()
        mock_smtp.return_value.__enter__.return_value = mock_server

        from email_sender import send_email
        result = send_email("alice@example.com", "Hello", "World")

        assert result is True
        mock_server.sendmail.assert_called_once_with(
            "from@example.com",
```

```
            "alice@example.com",
            "Subject: Hello\n\nWorld"
        )
```

28.8.3 `patch` as a decorator

```
from unittest.mock import patch
import pytest


class WeatherService:
    def get_temperature(self, city):
        # real implementation calls an external API
        raise NotImplementedError


def get_weather_report(city):
    service = WeatherService()
    temp = service.get_temperature(city)
    if temp < 0:
        return f"{city}: Freezing ({temp}°C)"
    elif temp < 15:
        return f"{city}: Cold ({temp}°C)"
    else:
        return f"{city}: Warm ({temp}°C)"


@patch.object(WeatherService, "get_temperature", return_value=25)
def test_warm_weather(mock_get_temp):
    report = get_weather_report("London")
    assert "Warm" in report
    assert "25" in report
    mock_get_temp.assert_called_once_with("London")


@patch.object(WeatherService, "get_temperature", return_value=-5)
def test_freezing_weather(mock_get_temp):
    report = get_weather_report("Oslo")
    assert "Freezing" in report
```

28.8.4 Mocking time and random

```
import pytest
from unittest.mock import patch
```

```python
import datetime


def get_greeting():
    hour = datetime.datetime.now().hour
    if hour < 12:
        return "Good morning!"
    elif hour < 18:
        return "Good afternoon!"
    else:
        return "Good evening!"


@patch("datetime.datetime")
def test_morning_greeting(mock_dt):
    mock_dt.now.return_value = datetime.datetime(2026, 1, 1, 9, 0, 0)
    assert get_greeting() == "Good morning!"


@patch("datetime.datetime")
def test_evening_greeting(mock_dt):
    mock_dt.now.return_value = datetime.datetime(2026, 1, 1, 20, 0, 0)
    assert get_greeting() == "Good evening!"
```

28.9 Testing Classes

```python
# bank.py
class InsufficientFundsError(Exception):
    pass

class BankAccount:
    def __init__(self, owner, balance=0.0):
        self.owner   = owner
        self._balance = balance
        self._history = []

    @property
    def balance(self):
        return self._balance

    def deposit(self, amount):
        if amount <= 0:
```

```
            raise ValueError("Deposit amount must be positive.")
        self._balance += amount
        self._history.append(("deposit", amount))
        return self._balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if amount > self._balance:
            raise InsufficientFundsError(
                f"Cannot withdraw {amount:.2f}, balance is {self._balance:.2f}."
            )
        self._balance -= amount
        self._history.append(("withdraw", amount))
        return self._balance

    def history(self):
        return list(self._history)


# test_bank.py
import pytest
from bank import BankAccount, InsufficientFundsError


@pytest.fixture
def account():
    """A fresh account with $100 balance."""
    return BankAccount("Alice", balance=100.0)


class TestDeposit:
    def test_deposit_increases_balance(self, account):
        account.deposit(50)
        assert account.balance == 150.0

    def test_deposit_returns_new_balance(self, account):
        result = account.deposit(50)
        assert result == 150.0

    def test_deposit_recorded_in_history(self, account):
        account.deposit(50)
        assert ("deposit", 50) in account.history()

    @pytest.mark.parametrize("amount", [0, -1, -100])
```

```
    def test_deposit_invalid_amount(self, account, amount):
        with pytest.raises(ValueError, match="positive"):
            account.deposit(amount)


class TestWithdraw:
    def test_withdraw_decreases_balance(self, account):
        account.withdraw(30)
        assert account.balance == 70.0

    def test_withdraw_exact_balance(self, account):
        account.withdraw(100)
        assert account.balance == 0.0

    def test_withdraw_overdraft(self, account):
        with pytest.raises(InsufficientFundsError, match="balance is 100.00"):
            account.withdraw(150)

    @pytest.mark.parametrize("amount", [0, -10])
    def test_withdraw_invalid_amount(self, account, amount):
        with pytest.raises(ValueError):
            account.withdraw(amount)

    def test_withdraw_recorded_in_history(self, account):
        account.withdraw(40)
        assert ("withdraw", 40) in account.history()


class TestHistory:
    def test_empty_history_for_new_account(self):
        acc = BankAccount("Bob")
        assert acc.history() == []

    def test_history_records_sequence(self, account):
        account.deposit(50)
        account.withdraw(30)
        account.deposit(10)
        assert account.history() == [
            ("deposit",  50),
            ("withdraw", 30),
            ("deposit",  10),
        ]

    def test_history_returns_copy(self, account):
```

```
    h = account.history()
    h.append(("fake", 999))
    assert ("fake", 999) not in account.history()
```

28.10 Test Coverage

Coverage tells you which lines of your code are exercised by tests:

```
pip install pytest-cov
pytest --cov=calculator --cov-report=term-missing
```

Output:

```
Name             Stmts   Miss  Cover   Missing
-----------------------------------------------
calculator.py       12      1    92%    15
-----------------------------------------------
TOTAL               12      1    92%
```

Line 15 is not covered — that's probably an edge case your tests don't reach yet. Generate an HTML report for a visual view:

```
pytest --cov=. --cov-report=html
# Open htmlcov/index.html in your browser
```

Coverage configuration in `pyproject.toml`:

```
[tool.pytest.ini_options]
testpaths = ["tests"]
addopts = "--cov=. --cov-report=term-missing --cov-fail-under=80"

[tool.coverage.run]
omit = ["tests/*", "setup.py"]
```

`--cov-fail-under=80` makes pytest exit with a failure if coverage drops below 80%. Good for CI pipelines.

28.11 Test Organization

```
my_project/
├── src/
│   └── my_project/
│       ├── __init__.py
│       ├── calculator.py
```

```
│       ├── bank.py
│       └── utils.py
├── tests/
│   ├── conftest.py          <- shared fixtures go here
│   ├── test_calculator.py
│   ├── test_bank.py
│   └── test_utils.py
├── pyproject.toml
└── README.md
```

`conftest.py` is automatically loaded by pytest. Put fixtures used by multiple test files here:

```
# tests/conftest.py
import pytest
import sqlite3


@pytest.fixture(scope="session")
def db_connection(tmp_path_factory):
    """One database for the entire test session."""
    db_path = tmp_path_factory.mktemp("data") / "test.db"
    conn    = sqlite3.connect(db_path)
    conn.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)")
    conn.commit()
    yield conn
    conn.close()


@pytest.fixture
def clean_db(db_connection):
    """Start each test with an empty database."""
    yield db_connection
    db_connection.execute("DELETE FROM users")
    db_connection.commit()
```

28.12 Marks — Categorizing Tests

pytest marks let you tag and selectively run tests:

```
import pytest

@pytest.mark.slow
def test_large_dataset():
```

```
    # takes 30 seconds
    ...

@pytest.mark.integration
def test_real_database():
    # requires actual database
    ...

@pytest.mark.skip(reason="Not implemented yet")
def test_future_feature():
    ...

@pytest.mark.skipif(
    condition=sys.platform == "win32",
    reason="Unix only"
)
def test_unix_permissions():
    ...

@pytest.mark.xfail(reason="Known bug, fix in progress")
def test_known_broken():
    assert 1 == 2   # expected to fail
```

Run only fast tests:

```
pytest -m "not slow"
pytest -m "not slow and not integration"
pytest -m slow
```

Register custom marks in `pyproject.toml` to avoid warnings:

```
[tool.pytest.ini_options]
markers = [
    "slow: marks tests as slow (deselect with '-m not slow')",
    "integration: marks tests requiring external services",
]
```

28.13 Test-Driven Development (TDD)

TDD means writing the test before the code. The cycle is:

1. **Red** — write a failing test for the feature you want
2. **Green** — write the minimum code to make it pass
3. **Refactor** — clean up the code; the test keeps it working

```
# Step 1 --- Red: write the test first
def test_fizzbuzz_three():
    assert fizzbuzz(3) == "Fizz"

def test_fizzbuzz_five():
    assert fizzbuzz(5) == "Buzz"

def test_fizzbuzz_fifteen():
    assert fizzbuzz(15) == "FizzBuzz"

def test_fizzbuzz_other():
    assert fizzbuzz(7) == "7"

# Tests fail because fizzbuzz doesn't exist yet

# Step 2 --- Green: write the minimum code
def fizzbuzz(n):
    if n % 15 == 0:
        return "FizzBuzz"
    if n % 3 == 0:
        return "Fizz"
    if n % 5 == 0:
        return "Buzz"
    return str(n)

# Tests pass

# Step 3 --- Refactor: maybe make it cleaner
def fizzbuzz(n):
    result = ""
    if n % 3 == 0:
        result += "Fizz"
    if n % 5 == 0:
        result += "Buzz"
    return result or str(n)

# Tests still pass --- refactor was safe
```

TDD's biggest benefit isn't the tests — it's that it forces you to think about the interface before the implementation.

28.14 Project: Test Suite for a Shopping Cart

```
# cart.py
from dataclasses import dataclass, field
from typing import Optional


class CartError(Exception):
    pass


@dataclass
class Product:
    name:  str
    price: float
    sku:   str


@dataclass
class CartItem:
    product:  Product
    quantity: int = 1

    @property
    def subtotal(self):
        return self.product.price * self.quantity


class ShoppingCart:
    def __init__(self, tax_rate: float = 0.0):
        self.tax_rate = tax_rate
        self._items: dict[str, CartItem] = {}

    def add(self, product: Product, quantity: int = 1) -> None:
        if quantity <= 0:
            raise CartError("Quantity must be positive.")
        if product.sku in self._items:
            self._items[product.sku].quantity += quantity
        else:
            self._items[product.sku] = CartItem(product, quantity)

    def remove(self, sku: str, quantity: Optional[int] = None) -> None:
        if sku not in self._items:
            raise CartError(f"Product {sku!r} not in cart.")
```

```
        if quantity is None or quantity >= self._items[sku].quantity:
            del self._items[sku]
        else:
            self._items[sku].quantity -= quantity

    def clear(self) -> None:
        self._items.clear()

    @property
    def subtotal(self) -> float:
        return sum(item.subtotal for item in self._items.values())

    @property
    def tax(self) -> float:
        return round(self.subtotal * self.tax_rate, 2)

    @property
    def total(self) -> float:
        return round(self.subtotal + self.tax, 2)

    @property
    def item_count(self) -> int:
        return sum(item.quantity for item in self._items.values())

    def items(self) -> list[CartItem]:
        return list(self._items.values())


# test_cart.py
import pytest
from cart import ShoppingCart, Product, CartItem, CartError


# ── Fixtures ──────────────────────────────────────────────────────────

@pytest.fixture
def book():
    return Product("Python Book", 29.99, "BOOK-001")

@pytest.fixture
def pen():
    return Product("Pen", 2.50, "PEN-001")

@pytest.fixture
def laptop():
```

```
    return Product("Laptop", 999.99, "LAP-001")

@pytest.fixture
def empty_cart():
    return ShoppingCart()

@pytest.fixture
def taxed_cart():
    return ShoppingCart(tax_rate=0.08)

@pytest.fixture
def cart_with_items(empty_cart, book, pen):
    empty_cart.add(book, quantity=2)
    empty_cart.add(pen, quantity=3)
    return empty_cart


# ── Adding items ──────────────────────────────────────────────

class TestAdd:
    def test_add_product(self, empty_cart, book):
        empty_cart.add(book)
        assert empty_cart.item_count == 1

    def test_add_same_product_twice_increases_quantity(self, empty_cart, book):
        empty_cart.add(book, 2)
        empty_cart.add(book, 3)
        assert empty_cart.item_count == 5

    def test_add_multiple_products(self, empty_cart, book, pen):
        empty_cart.add(book)
        empty_cart.add(pen)
        assert empty_cart.item_count == 2

    @pytest.mark.parametrize("quantity", [0, -1, -100])
    def test_add_invalid_quantity_raises(self, empty_cart, book, quantity):
        with pytest.raises(CartError, match="positive"):
            empty_cart.add(book, quantity)


# ── Removing items ────────────────────────────────────────────

class TestRemove:
    def test_remove_all(self, cart_with_items, book):
```

```
        cart_with_items.remove(book.sku)
        assert cart_with_items.item_count == 3   # only pens remain

    def test_remove_partial_quantity(self, cart_with_items, pen):
        cart_with_items.remove(pen.sku, quantity=1)
        remaining = next(
            i for i in cart_with_items.items() if i.product.sku == pen.sku
        )
        assert remaining.quantity == 2

    def test_remove_all_of_quantity(self, cart_with_items, book):
        cart_with_items.remove(book.sku, quantity=2)
        skus = [i.product.sku for i in cart_with_items.items()]
        assert book.sku not in skus

    def test_remove_nonexistent_raises(self, empty_cart):
        with pytest.raises(CartError, match="not in cart"):
            empty_cart.remove("FAKE-SKU")


# ── Totals ──────────────────────────────────────────────────────────

class TestTotals:
    def test_empty_cart_subtotal(self, empty_cart):
        assert empty_cart.subtotal == pytest.approx(0.0)

    def test_subtotal(self, cart_with_items):
        # 2 books at 29.99 + 3 pens at 2.50
        expected = 2 * 29.99 + 3 * 2.50
        assert cart_with_items.subtotal == pytest.approx(expected)

    def test_no_tax_by_default(self, cart_with_items):
        assert cart_with_items.tax == pytest.approx(0.0)

    def test_tax_calculation(self, taxed_cart, book):
        taxed_cart.add(book)
        expected_tax = round(book.price * 0.08, 2)
        assert taxed_cart.tax == pytest.approx(expected_tax)

    def test_total_with_tax(self, taxed_cart, book):
        taxed_cart.add(book, 2)
        subtotal = 2 * book.price
        expected = round(subtotal * 1.08, 2)
        assert taxed_cart.total == pytest.approx(expected)
```

```
    def test_clear_empties_cart(self, cart_with_items):
        cart_with_items.clear()
        assert cart_with_items.item_count == 0
        assert cart_with_items.subtotal == 0.0


# ── Edge cases ──────────────────────────────────────────────────────

class TestEdgeCases:
    def test_large_order(self, empty_cart, book):
        empty_cart.add(book, 1000)
        assert empty_cart.subtotal == pytest.approx(1000 * book.price)

    def test_cart_is_independent(self, book):
        cart1 = ShoppingCart()
        cart2 = ShoppingCart()
        cart1.add(book)
        assert cart2.item_count == 0    # cart2 unaffected

    def test_items_returns_copy(self, cart_with_items):
        items = cart_with_items.items()
        items.clear()
        assert cart_with_items.item_count > 0    # original unchanged
```

Run the full suite:

```
$ pytest test_cart.py -v --tb=short

test_cart.py::TestAdd::test_add_product                          PASSED
test_cart.py::TestAdd::test_add_same_product_twice_increases...  PASSED
...
test_cart.py::TestEdgeCases::test_items_returns_copy             PASSED

28 passed in 0.15s
```

28.15 What You Learned in This Chapter

- Tests give you confidence, safety, and living documentation.
- Name test files `test_*.py` and test functions `test_*`. Run with `pytest`.
- Use plain `assert` — pytest rewrites it for detailed failure messages.
- `pytest.approx` compares floats with tolerance.
- `pytest.raises` tests that the right exception is raised; `match=` checks the

message.

- `@pytest.mark.parametrize` runs one test function with many inputs.
- **Fixtures** provide shared setup/teardown, injected by parameter name. Use `yield` for cleanup.
- **Built-in fixtures:** `tmp_path`, `capsys`, `monkeypatch`.
- `unittest.mock.patch` replaces real dependencies with controllable fakes during tests.
- `pytest-cov` measures which lines your tests exercise. Aim for 80%+ on core logic.
- `conftest.py` holds fixtures shared across multiple test files.
- `@pytest.mark` categorizes tests; run subsets with `-m`.
- **TDD:** write the failing test first, then write the code, then refactor.

28.16 What's Next?

Chapter 29 covers **Debugging** — `pdb`, `breakpoint()`, `logging`, and the techniques experienced developers use to track down bugs systematically. Because even well-tested code has bugs — and knowing how to find them quickly is a skill on its own.

Chapter 29

Chapter 29: Debugging

Every programmer spends a significant portion of their time debugging. The difference between a junior and a senior developer isn't that the senior writes fewer bugs — it's that the senior finds and fixes them faster.

Debugging is a skill. It has techniques, tools, and a mindset. This chapter teaches all three.

29.1 The Debugging Mindset

Before touching any tool, adopt this mental model:

A bug is a gap between what you think the code does and what it actually does.

Your job is to find where your mental model is wrong. The code is always right — it does exactly what you told it to do. The bug is in your assumptions.

The process:

1. **Reproduce** — get a minimal, consistent way to trigger the bug.
2. **Isolate** — narrow down which part of the code is wrong.
3. **Understand** — figure out exactly why it's wrong.
4. **Fix** — change the code to match your intent.
5. **Verify** — confirm the fix works and didn't break anything else.

Skipping steps is how you waste hours. Especially step 1 — if you can't reliably reproduce a bug, you can't reliably fix it.

29.2 Print Debugging — Simple but Effective

The oldest debugging technique: print values to see what's happening.

```
def process_order(items, discount_code):
    subtotal = sum(item["price"] * item["qty"] for item in items)
    discount = apply_discount(subtotal, discount_code)
    tax      = subtotal * 0.08
    total    = subtotal - discount + tax
    return total


# Something is wrong. Add prints to see the values.
def process_order(items, discount_code):
    subtotal = sum(item["price"] * item["qty"] for item in items)
    print(f"DEBUG subtotal={subtotal}")
    discount = apply_discount(subtotal, discount_code)
    print(f"DEBUG discount={discount}")
    tax      = subtotal * 0.08
    print(f"DEBUG tax={tax}")
    total    = subtotal - discount + tax
    print(f"DEBUG total={total}")
    return total
```

Print debugging is fast for simple bugs. Its weakness: you have to remember to remove the prints, and it doesn't help with complex control flow.

Use `repr()` when the value might be ambiguous:

```
# Is it an empty string or None?
print(repr(value))   # '' vs None vs '  '
```

A slightly smarter approach — a debug print that includes the variable name:

```
def dbg(label, value):
    print(f"  [DEBUG] {label} = {value!r}")
    return value   # returns the value so you can wrap expressions

# Use it inline without restructuring code
total = dbg("total", subtotal - dbg("discount", apply_discount(subtotal, code)) +
```

Python 3.8+ f-strings have a built-in `=` specifier for this:

```
x = 42
y = "hello"
items = [1, 2, 3]

print(f"{x=}")       # x=42
```

```
print(f"{y=}")       # y='hello'
print(f"{items=}")   # items=[1, 2, 3]
print(f"{x + y=}")   # x + y=... (expression + result)

# Use in debugging:
def calculate(a, b):
    result = a ** 2 + b ** 2
    print(f"{a=} {b=} {result=}")
    return result
```

29.3 `breakpoint()` and `pdb`

`breakpoint()` drops you into Python's interactive debugger at that exact line. From there you can inspect variables, step through code line by line, and evaluate expressions.

```
def find_max(numbers):
    max_val = numbers[0]
    for i, n in enumerate(numbers):
        breakpoint()    # execution pauses here
        if n > max_val:
            max_val = n
    return max_val

find_max([3, 1, 4, 1, 5, 9, 2, 6])
```

When execution hits `breakpoint()`, you see the `(Pdb)` prompt. Essential commands:

Command	What it does
`h`	Help — list all commands
`n`	Next — run the current line, stop at the next one
`s`	Step — step into a function call
`c`	Continue — run until the next breakpoint
`q`	Quit — exit the debugger
`l`	List — show source code around the current line
`p expr`	Print — evaluate and print an expression
`pp expr`	Pretty-print — like `p` but formatted
`w`	Where — show the call stack
`u` / `d`	Up/down — move up/down the call stack
`b line`	Breakpoint — set a new breakpoint at a line number
`r`	Return — run until the current function returns
`a`	Args — print arguments of the current function

Command	What it does

```
> find_max([3, 1, 4, 1, 5, 9, 2, 6])

> /path/to/script.py(5)find_max()
-> if n > max_val:

(Pdb) p n
3
(Pdb) p max_val
3
(Pdb) p numbers
[3, 1, 4, 1, 5, 9, 2, 6]
(Pdb) n
> /path/to/script.py(6)find_max()
-> max_val = n
(Pdb) c
> /path/to/script.py(5)find_max()
-> if n > max_val:
(Pdb) p i, n, max_val
(1, 1, 3)
```

29.3.1 Post-mortem debugging

If your program crashes with an uncaught exception, you can inspect the crash state after the fact:

```
import pdb

try:
    buggy_function()
except Exception:
    pdb.post_mortem()    # drops into debugger at the crash point
```

Or from the command line — run with `pdb` and it automatically opens the debugger at any crash:

```
python -m pdb my_script.py
```

29.3.2 Conditional breakpoints

Break only when a specific condition is true — avoid stepping through 999 iterations to get to the interesting one:

```
for i, item in enumerate(large_list):
    if i == 500:
```

```
        breakpoint()    # only break at index 500
    process(item)
```

Or in pdb directly:

```
(Pdb) b 15, n > 100    # break at line 15 only when n > 100
```

29.4 `logging` — Structured Debug Output

`print()` is for users. `logging` is for developers. The key difference: logging has levels, and you can turn it on and off without changing code.

```
import logging

# Basic setup --- goes to stderr
logging.basicConfig(
    level   = logging.DEBUG,
    format = "%(asctime)s %(levelname)-8s %(name)s: %(message)s",
    datefmt= "%H:%M:%S",
)

log = logging.getLogger(__name__)

log.debug("This is debug info --- very verbose")
log.info("Normal operation message")
log.warning("Something unexpected but not fatal")
log.error("Something failed")
log.critical("System cannot continue")
```

Output:

```
10:23:45 DEBUG    __main__: This is debug info --- very verbose
10:23:45 INFO     __main__: Normal operation message
10:23:45 WARNING  __main__: Something unexpected but not fatal
10:23:45 ERROR    __main__: Something failed
10:23:45 CRITICAL __main__: System cannot continue
```

29.4.1 Log levels

Level	Value	When to use
`DEBUG`	10	Detailed diagnostic info, only during development

Level	Value	When to use
`INFO`	20	Confirmation that things are working as expected
`WARNING`	30	Something unexpected, but the program continues
`ERROR`	40	A serious problem, the function couldn't complete
`CRITICAL`	50	A fatal error, the program may not be able to continue

Setting the level to `WARNING` in production means `DEBUG` and `INFO` messages are silent — no code changes needed.

29.4.2 Named loggers

Use `logging.getLogger(__name__)` in every module. It creates a logger named after the module, which lets you control logging per module:

```
# calculator.py
import logging
log = logging.getLogger(__name__)    # logger name: "calculator"

def divide(a, b):
    log.debug(f"divide({a}, {b}) called")
    if b == 0:
        log.error("Division by zero attempted")
        raise ZeroDivisionError("Cannot divide by zero.")
    result = a / b
    log.debug(f"divide result: {result}")
    return result
```

```
# main.py
import logging
import calculator

logging.basicConfig(level=logging.WARNING)                    # show only warnings
logging.getLogger("calculator").setLevel(logging.DEBUG)       # but debug for calc

calculator.divide(10, 2)    # DEBUG messages print for calculator only
calculator.divide(5, 0)     # ERROR message prints
```

29.4.3 Logging to a file

```
import logging

logging.basicConfig(
    level    = logging.DEBUG,
    format   = "%(asctime)s %(levelname)-8s %(name)s: %(message)s",
    handlers = [
        logging.FileHandler("app.log"),    # write to file
        logging.StreamHandler(),            # also write to console
    ],
)
```

29.4.4 Structured logging with `extra`

```
log = logging.getLogger(__name__)

def process_request(request_id, user_id, endpoint):
    extra = {"request_id": request_id, "user_id": user_id}
    log.info("Request received: %s", endpoint, extra=extra)
    # ... process ...
    log.info("Request completed", extra=extra)
```

29.5 Reading Tracebacks

Python tracebacks are read **bottom up** — the last line is the actual error; the lines above show how you got there.

```
def a():
    return b()

def b():
    return c()

def c():
    return 1 / 0

a()
```

```
Traceback (most recent call last):
  File "script.py", line 9, in <module>
    a()
  File "script.py", line 2, in a
    return b()
```

```
  File "script.py", line 5, in b
    return c()
  File "script.py", line 8, in c
    return 1 / 0
ZeroDivisionError: division by zero
```

Reading it: 1. Start at the bottom: `ZeroDivisionError: division by zero` — the actual error. 2. The line above it: `return 1 / 0` in function `c` at line 8 — where it happened. 3. Work upward to understand the call chain: `a()` called `b()` called `c()`.

The line that caused the error is almost always at the bottom. Go there first.

29.5.1 Common exceptions and their usual causes

Exception	Common cause
`NameError`	Typo in a variable name, or used before defined
`TypeError`	Wrong type passed to a function, or unsupported operation
`ValueError`	Right type, wrong value (e.g. `int("hello")`)
`IndexError`	List index out of range
`KeyError`	Dictionary key doesn't exist
`AttributeError`	Object doesn't have that attribute/method (typo, or wrong type)
`ZeroDivisionError`	Divided by zero
`FileNotFoundError`	File path is wrong or file doesn't exist
`ImportError`	Module not found or not installed
`RecursionError`	Infinite recursion — missing base case
`StopIteration`	Calling `next()` on exhausted iterator
`MemoryError`	Out of memory — often from accidentally creating huge data

29.6 Common Bug Patterns and How to Find Them

29.6.1 Off-by-one errors

```
# Bug: loop runs one too many times
for i in range(1, len(items) + 1):    # should be range(len(items))
    print(items[i])                    # IndexError on last iteration

# Debug: print the index and length
for i in range(1, len(items) + 1):
    print(f"i={i}, len={len(items)}, accessing items[{i}]")
```

29.6.2 Mutating a list while iterating it

```
# Bug: removing items from a list you're looping over
numbers = [1, 2, 3, 4, 5, 6]
for n in numbers:
    if n % 2 == 0:
        numbers.remove(n)    # modifies the list mid-iteration --- skips items

print(numbers)    # [1, 3, 5] --- looks right but worked by accident

# The real danger:
items = [2, 4, 6, 8]
for item in items:
    if item % 2 == 0:
        items.remove(item)
print(items)    # [4, 8] --- wrong! skipped every other even number

# Fix: iterate a copy, or use a comprehension
items = [item for item in items if item % 2 != 0]
```

29.6.3 Mutable default arguments

```
# Bug: the list is created once, shared across all calls
def add_item(item, container=[]):    # default created at function definition ti
    container.append(item)
    return container

print(add_item("a"))    # ['a']
print(add_item("b"))    # ['a', 'b']  --- expected ['b']!
print(add_item("c"))    # ['a', 'b', 'c']  --- state persists between calls!
```

```
# Fix: use None as default, create fresh each call
def add_item(item, container=None):
    if container is None:
        container = []
    container.append(item)
    return container
```

29.6.4 `is` vs `==`

```
# Bug: comparing with 'is' when you mean '=='
a = 1000
b = 1000
print(a is b)     # False! (different objects)
print(a == b)     # True  (same value)

# 'is' checks identity (same object in memory)
# '==' checks equality (same value)

# Common mistakes:
x = None
if x is None:   # correct
    pass
if x == None:   # works but not idiomatic --- never do this
    pass

name = "Alice"
if name is "Alice":    # WRONG --- may or may not work (string interning)
    pass
if name == "Alice":    # correct
    pass
```

29.6.5 Variable shadowing

```
# Bug: accidentally reusing a name
list = [1, 2, 3]       # 'list' now shadows the built-in!
result = list([4, 5])  # TypeError: 'list' object is not callable

# Safer names: items, numbers, my_list
items = [1, 2, 3]

# Other commonly shadowed names:
# str, int, float, dict, set, input, print, type, id, min, max, sum
```

29.6.6 Integer division returning float, or vice versa

```
total  = 10
count  = 3
avg    = total / count    # 3.3333... (float division)
avg    = total // count   # 3 (integer division --- truncates)

# Bug: expected integer but got float
page   = 7
items_per_page = 3
pages  = page / items_per_page    # 2.333... --- should be 3 (ceiling)

import math
pages  = math.ceil(page / items_per_page)  # 3 --- correct
```

29.7 The `traceback` Module

For capturing and logging exception information programmatically:

```
import traceback
import logging

log = logging.getLogger(__name__)

def safe_process(data):
    try:
        return risky_operation(data)
    except Exception:
        log.error("Failed to process data:\n%s", traceback.format_exc())
        return None


# Get the traceback as a string
try:
    1 / 0
except ZeroDivisionError:
    tb_str = traceback.format_exc()
    print(tb_str)
    # Traceback (most recent call last):
    #   File "<stdin>", line 2, in <module>
    # ZeroDivisionError: division by zero
```

29.8 `assert` for Invariants

`assert` is not just for tests — it's a debugging tool. Use it to document and verify assumptions that must always be true:

```
def calculate_average(numbers):
    assert len(numbers) > 0, "Cannot average an empty list"
    assert all(isinstance(n, (int, float)) for n in numbers), \
        f"All items must be numeric, got: {numbers}"
    return sum(numbers) / len(numbers)


def binary_search(items, target):
    assert items == sorted(items), "binary_search requires a sorted list"
    # ... search ...
```

Run with `python -O script.py` (optimize flag) to disable all asserts in production — they have zero cost. Keep them in development where they catch bugs early.

29.9 Introspection Tools

Python lets you examine objects at runtime:

```
# What type is this?
print(type(obj))
print(isinstance(obj, list))

# What attributes/methods does it have?
print(dir(obj))

# What does this object look like?
print(repr(obj))

# Where is this function/class defined?
import inspect
print(inspect.getfile(MyClass))
print(inspect.getsource(my_function))

# What arguments does this function accept?
print(inspect.signature(my_function))

# What is the call stack right now?
import traceback
```

```
traceback.print_stack()

# What variables are in scope?
print(locals())
print(globals())

# A diagnostic helper --- print everything about an object
def inspect_object(obj, label="obj"):
    print(f"\n{'—'*40}")
    print(f"  {label}: {obj!r}")
    print(f"  type:   {type(obj).__name__}")
    print(f"  id:     {id(obj)}")
    if hasattr(obj, "__dict__"):
        print(f"  attrs:  {obj.__dict__}")
    print(f"{'—'*40}\n")
```

29.10 IDE Debugging

All modern Python editors have a visual debugger. In VS Code:

1. Click the gutter to set a breakpoint (red dot appears).
2. Press `F5` to start debugging.
3. The program pauses at your breakpoint.
4. Use the debug toolbar: Step Over (F10), Step Into (F11), Step Out (Shift+F11), Continue (F5).
5. Inspect variables in the Variables pane on the left.
6. Hover over any variable to see its value.
7. Use the Debug Console to evaluate expressions.

This is usually faster than `pdb` for complex bugs — you can see the full call stack, all local variables, and watch expressions simultaneously.

29.11 `icecream` — A Better Debug Print

```
pip install icecream
```

```
from icecream import ic

def process(items):
    total = ic(sum(items))        # prints: ic| sum(items): 15
    avg   = ic(total / len(items))  # prints: ic| total / len(items): 3.0
    return avg
```

```
process([1, 2, 3, 4, 5])
```

`ic()` prints the expression and its value, then returns the value so you can wrap it around any expression without restructuring code. It also includes the file, line number, and parent function by default.

Disable all `ic()` calls without removing them:

```
from icecream import ic
ic.disable()    # all ic() calls become no-ops
```

29.12 Systematic Debugging: The Bisect Method

For a hard bug — especially a regression (code that used to work and now doesn't) — use binary search on the code itself:

1. Find the last known good state (a commit, an input, a code path that worked).
2. Find the current broken state.
3. Test the midpoint. Is it broken?
    - Yes: the bug is in the first half.
    - No: the bug is in the second half.
4. Repeat until you've isolated the exact change that introduced the bug.

With git:

```
git bisect start
git bisect bad           # current state is broken
git bisect good v1.0.0   # v1.0.0 was working
# git checks out the midpoint
# test it, then:
git bisect good   # or: git bisect bad
# repeat until git identifies the offending commit
git bisect reset
```

29.13 Project: A Debug-Friendly Logger Class

```
"""
debug_utils.py --- Debugging utilities for development.
"""
import logging
```

```
import time
import functools
import traceback
import sys
from typing import Any, Callable, TypeVar
from contextlib import contextmanager

F = TypeVar("F", bound=Callable[..., Any])


def setup_logging(
    level: str = "DEBUG",
    log_file: str | None = None,
    fmt: str = "%(asctime)s %(levelname)-8s %(name)-20s %(message)s",
) -> None:
    """Configure root logger with console (and optional file) handler."""
    handlers: list[logging.Handler] = [logging.StreamHandler(sys.stdout)]
    if log_file:
        handlers.append(logging.FileHandler(log_file))

    logging.basicConfig(
        level    = getattr(logging, level.upper()),
        format   = fmt,
        datefmt = "%H:%M:%S",
        handlers= handlers,
        force    = True,    # override any existing configuration
    )


def trace(func: F) -> F:
    """Decorator: log every call with arguments and return value."""
    log = logging.getLogger(func.__module__)

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        args_repr = [repr(a) for a in args]
        kw_repr   = [f"{k}={v!r}" for k, v in kwargs.items()]
        signature = ", ".join(args_repr + kw_repr)
        log.debug(f"-> {func.__qualname__}({signature})")
        try:
            result = func(*args, **kwargs)
            log.debug(f"<- {func.__qualname__} = {result!r}")
            return result
        except Exception as e:
```

```
            log.error(f"[-] {func.__qualname__} raised {type(e).__name__}: {e}")
            raise
    return wrapper  # type: ignore


@contextmanager
def timer(label: str = "", level: int = logging.DEBUG):
    """Context manager: log how long a block takes."""
    log   = logging.getLogger(__name__)
    start = time.perf_counter()
    try:
        yield
    finally:
        ms = (time.perf_counter() - start) * 1000
        tag = f"[{label}] " if label else ""
        log.log(level, f"{tag}Elapsed: {ms:.2f}ms")


def catch_and_log(
    *exceptions: type[Exception],
    default: Any = None,
    reraise: bool = False,
) -> Callable:
    """
    Decorator: catch exceptions, log them, optionally return a default value.
    """
    def decorator(func: F) -> F:
        log = logging.getLogger(func.__module__)

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            try:
                return func(*args, **kwargs)
            except exceptions as e:
                log.error(
                    f"{func.__qualname__} failed: {type(e).__name__}: {e}\n"
                    f"{traceback.format_exc()}"
                )
                if reraise:
                    raise
                return default
        return wrapper  # type: ignore
    return decorator
```

```python
class DebugProxy:
    """
    Wrap any object and log every attribute access and method call.
    Useful for understanding how a third-party object is being used.
    """
    def __init__(self, obj: Any, name: str = ""):
        object.__setattr__(self, "_obj",  obj)
        object.__setattr__(self, "_name", name or type(obj).__name__)
        object.__setattr__(self, "_log",  logging.getLogger(__name__))

    def __getattr__(self, name: str) -> Any:
        obj  = object.__getattribute__(self, "_obj")
        log  = object.__getattribute__(self, "_log")
        label= object.__getattribute__(self, "_name")
        attr = getattr(obj, name)
        log.debug(f"{label}.{name} accessed -> {attr!r}")
        if callable(attr):
            def traced(*args, **kwargs):
                log.debug(f"{label}.{name}({args}, {kwargs})")
                result = attr(*args, **kwargs)
                log.debug(f"{label}.{name} returned {result!r}")
                return result
            return traced
        return attr


# ── Demo ──────────────────────────────────────────────────────────

if __name__ == "__main__":
    setup_logging("DEBUG")
    log = logging.getLogger(__name__)

    @trace
    def fibonacci(n: int) -> int:
        if n < 2:
            return n
        return fibonacci(n - 1) + fibonacci(n - 2)

    @catch_and_log(ValueError, ZeroDivisionError, default=0.0)
    def safe_divide(a: float, b: float) -> float:
        return a / b

    with timer("Fibonacci(10)"):
```

```
    result = fibonacci(10)
log.info(f"fibonacci(10) = {result}")

log.info(f"safe_divide(10, 2) = {safe_divide(10, 2)}")
log.info(f"safe_divide(10, 0) = {safe_divide(10, 0)}")  # logs error, returns

# DebugProxy wraps a list and logs every operation
proxy = DebugProxy([], "my_list")
proxy.append(1)
proxy.append(2)
proxy.extend([3, 4, 5])
log.info(f"List contents: {proxy._obj}")
```

29.14 Debugging Checklist

When you're stuck on a bug, go through this list:

- **Read the error message.** All of it. The last line first.
- **Read the traceback bottom-up.** Find the line that actually crashed.
- **Check your assumptions.** Print the value you think is there. Is it actually that?
- **Reproduce with minimal input.** Strip away everything not needed to trigger the bug.
- **Check recent changes.** What changed since it last worked? `git diff` is your friend.
- **Search for the error message.** Someone has hit this before.
- **Explain the code out loud.** Describe what each line does. The bug often reveals itself.
- **Take a break.** A fresh mind sees things a tired mind misses. Not a joke — it works.
- **Check the obvious things last.** Typos, wrong variable names, copy-paste errors.

29.15 What You Learned in This Chapter

- Debugging is finding the gap between what you think the code does and what it actually does.
- The five-step process: reproduce, isolate, understand, fix, verify.
- `print(f"{var=}")` (Python 3.8+) prints both name and value — fast and clean.

- `breakpoint()` drops into `pdb`. Learn six commands: `n`, `s`, `c`, `q`, `p`, `w`.
- `logging` is `print()` for professionals — has levels, can be silenced without code changes, writes to files, per-module control.
- Read tracebacks bottom-up. The last line is the error; the lines above are the call chain.
- Common bugs: off-by-one, mutating during iteration, mutable default arguments, `is` vs `==`, variable shadowing.
- `traceback.format_exc()` captures the exception traceback as a string for logging.
- `assert` documents and enforces invariants in development code.
- `inspect` module: `getsource`, `signature`, `getfile` for runtime introspection.
- `git bisect` finds the exact commit that introduced a regression.
- The `icecream` library gives you richer print-style debugging.

29.16 What's Next?

Chapter 30 covers **Performance and Profiling** — how to measure where your code is slow, and how to speed it up. `cProfile`, `line_profiler`, `timeit`, algorithmic complexity, and practical optimization patterns. Because fast code starts with measuring, not guessing.

Chapter 30

Chapter 30: Performance and Profiling

"My code is too slow." That's one of the most common complaints in software development. The wrong response is to start optimizing randomly — rewriting loops, switching data structures, adding caching — based on gut feeling.

The right response is to **measure first**. Find exactly where the time goes. Then fix that specific thing.

This chapter teaches you to measure performance precisely, understand algorithmic complexity, and apply targeted optimizations that actually matter.

30.1 The Golden Rule: Measure Before You Optimize

Donald Knuth said it in 1974: "Premature optimization is the root of all evil."

What he meant: optimizing code before you know what's slow wastes time, makes code harder to read, and often doesn't even help — because you optimized the wrong thing.

The process is always: 1. **Measure** — find the actual bottleneck 2. **Understand why** — is it algorithmic? I/O? Memory? 3. **Fix the bottleneck** — change the right thing 4. **Measure again** — verify the improvement

---

30.2 `timeit` — Measuring Small Code Snippets

`timeit` runs a snippet many times and tells you the average execution time. Use it to compare two approaches to the same problem.

```
import timeit

# Compare string concatenation methods
n = 10_000

# Method 1: += in a loop
def concat_plus():
    s = ""
    for i in range(n):
        s += str(i)
    return s

# Method 2: join
def concat_join():
    return "".join(str(i) for i in range(n))

# Method 3: f-string in list
def concat_list():
    parts = [str(i) for i in range(n)]
    return "".join(parts)


time_plus = timeit.timeit(concat_plus, number=1000)
time_join = timeit.timeit(concat_join, number=1000)
time_list = timeit.timeit(concat_list, number=1000)

print(f"concat_plus: {time_plus:.3f}s")
print(f"concat_join: {time_join:.3f}s")
print(f"concat_list: {time_list:.3f}s")
```

Output (approximate):

```
concat_plus: 2.841s
concat_join: 1.203s
concat_list: 0.891s
```

`join` is ~3x faster than `+=` for large strings. Good to know — but only worth doing if string building is actually your bottleneck.

30.2.1 `timeit` from the command line

```
python -m timeit "'-'.join(str(n) for n in range(100))"
# 100000 loops, best of 5: 12.2 usec per loop

python -m timeit "'-'.join([str(n) for n in range(100)])"
# 100000 loops, best of 5: 9.94 usec per loop
```

30.2.2 `%%timeit` in Jupyter notebooks

```
%%timeit
result = [x**2 for x in range(10_000)]
```

30.3 `cProfile` — Profiling Entire Programs

`timeit` measures snippets. `cProfile` profiles your whole program — it tells you how many times each function was called and how long each one took.

```
import cProfile
import pstats

def slow_function():
    total = 0
    for i in range(100_000):
        total += i ** 2
    return total

def another_function():
    data = [slow_function() for _ in range(3)]
    return sum(data)

def main():
    result = another_function()
    print(f"Result: {result}")


# Profile main()
profiler = cProfile.Profile()
profiler.enable()
main()
profiler.disable()

# Print sorted results
```

```
stats = pstats.Stats(profiler)
stats.strip_dirs()
stats.sort_stats("cumulative")     # sort by cumulative time
stats.print_stats(20)              # show top 20 functions
```

Output:

```
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    1.234    1.234 script.py:14(main)
     1   0.000   0.000   1.234   1.234 script.py:10(another_function)
      3   1.234   0.411   1.234   0.411 script.py:5(slow_function)
     300000       0.000       0.000       0.000       0.000  {built-
in method builtins.pow}
```

Columns explained: - `ncalls` — number of times the function was called - `tottime` — total time spent in this function (excluding sub-calls) - `percall` — tottime divided by ncalls - `cumtime` — cumulative time including all sub-calls - `percall` (second) — cumtime divided by ncalls

`slow_function` accounts for 1.234s — the clear bottleneck.

30.3.1 From the command line

```
python -m cProfile -s cumulative my_script.py
python -m cProfile -o profile.out my_script.py   # save to file
```

30.3.2 Visualizing with `snakeviz`

```
pip install snakeviz
python -m cProfile -o profile.out my_script.py
snakeviz profile.out   # opens an interactive flame graph in your browser
```

30.4 `line_profiler` — Line-by-Line Timing

`cProfile` tells you which function is slow. `line_profiler` tells you which *line* inside that function is slow.

```
pip install line_profiler
```

```
# profile_example.py
from line_profiler import LineProfiler

def process_data(data):
    cleaned  = [x for x in data if x > 0]          # filter
    squared  = [x ** 2 for x in cleaned]            # transform
```

```
    filtered = [x for x in squared if x % 2 == 0]   # filter again
    return sorted(filtered)                          # sort

data = list(range(-1000, 1000))

lp = LineProfiler()
lp.add_function(process_data)
lp.enable()
result = process_data(data)
lp.disable()
lp.print_stats()
```

Output:

```
Line #    Hits     Time  Per Hit   % Time  Line Contents
    5     1   142.0   142.0    8.5  cleaned = [x for x in data if x > 0]
    6     1   310.0   310.0   18.6  squared = [x ** 2 for x in cleaned]
    7     1   280.0   280.0   16.8  filtered = [x for x in squared if x % 2 == 0]
      8       1   934.0    934.0    56.1  return sorted(filtered)
```

The `sorted()` call takes 56% of the time. That's where to focus.

30.5 `memory_profiler` — Measuring Memory Usage

```
pip install memory_profiler
```

```
from memory_profiler import profile

@profile
def load_large_data():
    data  = list(range(1_000_000))          # list: ~8MB
    total = sum(data)                        # compute sum
    del data                                 # free memory
    result = [x ** 2 for x in range(10_000)]  # smaller list
    return total, result

load_large_data()
```

Output:

```
Line #    Mem usage    Increment  Line Contents
     4    45.3 MiB    45.3 MiB   def load_large_data():
     5    53.1 MiB     7.8 MiB   data  = list(range(1_000_000))
     6    53.1 MiB     0.0 MiB   total = sum(data)
```

```
     7    45.3 MiB     -7.8 MiB   del data
     8    45.6 MiB      0.3 MiB   result = [x**2 for x in range(10_000)]
```

Each line shows current memory usage and the change. `data` takes 7.8MB; `del data` releases it.

30.6 Algorithmic Complexity — The Real Performance Win

Tools are for measuring. But the biggest performance wins come from choosing better algorithms. A 10x faster computer running an O(n^2) algorithm is still slower than a slow computer running O(n log n) for large n.

Big O notation describes how an algorithm's runtime grows with input size n.

Complexity	Name	Example
O(1)	Constant	Dictionary lookup, list append
O(log n)	Logarithmic	Binary search, balanced tree operations
O(n)	Linear	Loop through a list once
O(n log n)	Linearithmic	`sorted()`, good sorting algorithms
O(n^2)	Quadratic	Nested loops over n items
$O(2^n)$	Exponential	Naive recursive Fibonacci
O(n!)	Factorial	Brute-force permutations

```
n = 1,000,000
O(1):       1 operation
O(log n):   20 operations
O(n):       1,000,000 operations
O(n log n): 20,000,000 operations
O(n^2):      1,000,000,000,000 operations  <- 1 trillion
```

The jump from O(n) to O(n^2) is the difference between milliseconds and hours.

30.6.1 Seeing it in practice

```
import time

def has_duplicate_quadratic(items):
    """O(n^2) --- nested loop"""
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
```

```
            if items[i] == items[j]:
                return True
    return False

def has_duplicate_linear(items):
    """O(n) --- set lookup"""
    seen = set()
    for item in items:
        if item in seen:
            return True
        seen.add(item)
    return False

def has_duplicate_oneliner(items):
    """O(n) --- even cleaner"""
    return len(items) != len(set(items))


sizes = [100, 1_000, 10_000, 100_000]
for n in sizes:
    data = list(range(n))    # no duplicates --- worst case

    t1 = timeit.timeit(lambda: has_duplicate_quadratic(data), number=1)
    t2 = timeit.timeit(lambda: has_duplicate_linear(data),    number=1)

    print(f"n={n:>7}: O(n^2)={t1:.4f}s  O(n)={t2:.6f}s  ratio={t1/t2:.0f}x")
```

Output:

```
n=    100: O(n^2)=0.0001s  O(n)=0.000010s  ratio=12x
n=  1,000: O(n^2)=0.0051s  O(n)=0.000089s  ratio=57x
n= 10,000: O(n^2)=0.4821s  O(n)=0.000891s  ratio=541x
n=100,000: O(n^2)=47.832s  O(n)=0.009103s  ratio=5254x
```

The same logic, 5000x faster at n=100,000 — just by choosing a set instead of a nested loop.

30.7 Choosing the Right Data Structure

The single most impactful optimization is usually choosing the right data structure.

```
import timeit
```

```python
# Membership testing
large_list = list(range(100_000))
large_set  = set(range(100_000))
large_dict = {i: True for i in range(100_000)}

target = 99_999   # worst case --- near the end

t_list = timeit.timeit(lambda: target in large_list, number=10_000)
t_set  = timeit.timeit(lambda: target in large_set,  number=10_000)
t_dict = timeit.timeit(lambda: target in large_dict, number=10_000)

print(f"list:  {t_list:.4f}s  O(n)")
print(f"set:   {t_set:.6f}s  O(1)")
print(f"dict:  {t_dict:.6f}s  O(1)")
```

Output:

```
list:  3.8412s  O(n)
set:   0.0003s  O(1)
dict:  0.0003s  O(1)
```

Set and dict lookups are 10,000x faster for large collections because they use hash tables.

30.7.1 Data structure complexity cheat sheet

Operation	list	set	dict	deque
Append (end)	O(1)	—	—	O(1)
Insert (middle)	O(n)	—	—	O(n)
Delete (middle)	O(n)	—	—	O(n)
Pop (end)	O(1)	—	—	O(1)
Pop (front)	O(n)	—	—	O(1)
Lookup by index	O(1)	—	—	O(n)
Membership test	O(n)	O(1)	O(1)	O(n)
Insert/delete	—	O(1)	O(1)	—
Sorted order	Yes (if sorted)	No	No (3.7+ insertion)	Yes (if sorted)

Use `collections.deque` when you need fast prepend (front insertion). Use

`set` or `dict` for fast membership testing.

```
from collections import deque

# Queue --- FIFO (First In, First Out)
queue = deque()
queue.append("first")
queue.append("second")
queue.append("third")
print(queue.popleft())    # "first" --- O(1)

# Stack --- LIFO (Last In, First Out)
stack = []
stack.append("first")
stack.append("second")
print(stack.pop())    # "second" --- O(1)

# deque as a sliding window (fixed-size buffer)
window = deque(maxlen=3)
for x in range(10):
    window.append(x)
    print(list(window))    # never longer than 3
```

30.8 Practical Optimization Techniques

30.8.1 1. Avoid repeated work — cache results

```
# Slow: recompute the same thing every iteration
def process(data):
    for item in data:
        if len(data) > 1000:    # len(data) called every iteration
            do_something(item)

# Fast: compute once
def process(data):
    n = len(data)
    for item in data:
        if n > 1000:
            do_something(item)
```

30.8.2 2. Move work out of inner loops

```
import re

# Slow: compile the regex on every call
def count_emails(texts):
    return sum(1 for t in texts if re.match(r"[^@]+@[^@]+\.[^@]+", t))

# Fast: compile once outside the loop (or at module level)
EMAIL_RE = re.compile(r"[^@]+@[^@]+\.[^@]+")

def count_emails(texts):
    return sum(1 for t in texts if EMAIL_RE.match(t))
```

30.8.3 3. Use built-ins — they're implemented in C

```
# Slow: Python loop
def sum_squares_loop(n):
    total = 0
    for i in range(n):
        total += i * i
    return total

# Fast: built-in sum with generator
def sum_squares_builtin(n):
    return sum(i * i for i in range(n))

# Even faster for math-heavy work: numpy
import numpy as np
def sum_squares_numpy(n):
    return int(np.sum(np.arange(n) ** 2))
```

```
import timeit
n = 1_000_000
t1 = timeit.timeit(lambda: sum_squares_loop(n),    number=5)
t2 = timeit.timeit(lambda: sum_squares_builtin(n), number=5)

print(f"Loop:    {t1:.3f}s")
print(f"Built-in:{t2:.3f}s")
```

30.8.4 4. List comprehensions over explicit loops

```
# Slower
result = []
for x in range(10_000):
```

```
    if x % 2 == 0:
        result.append(x ** 2)

# Faster (comprehensions are optimized in CPython)
result = [x ** 2 for x in range(10_000) if x % 2 == 0]
```

30.8.5 5. Use generators for large data

```
# Loads everything into memory
total = sum([x ** 2 for x in range(10_000_000)])

# Lazy --- one item at a time, much lower memory
total = sum(x ** 2 for x in range(10_000_000))
```

30.8.6 6. Local variable lookup is faster than global

```
import math

# Slow: looks up math.sqrt in global scope every call
def compute_slow(data):
    return [math.sqrt(x) for x in data]

# Fast: bind to a local variable
def compute_fast(data):
    sqrt = math.sqrt   # local lookup is faster
    return [sqrt(x) for x in data]
```

30.8.7 7. String formatting — f-strings are fastest

```
name, value = "Alice", 42

# Slowest
s = "Hello " + name + ", value is " + str(value)

# Fast
s = f"Hello {name}, value is {value}"

# Acceptable
s = "Hello {}, value is {}".format(name, value)
```

30.8.8 8. `functools.lru_cache` for expensive pure functions

```
from functools import lru_cache

@lru_cache(maxsize=None)
def expensive_lookup(key):
    # Simulate expensive computation or DB query
    import time
    time.sleep(0.1)
    return key * 2

# First call: 0.1s
# Subsequent calls: instant (cached)
```

30.9 `numpy` for Numerical Work

If you're doing numerical computation — matrix operations, statistics, signal processing — pure Python is the wrong tool. `numpy` operates in C on contiguous arrays and is 10-1000x faster.

```
import numpy as np
import timeit

n = 1_000_000

# Python list vs numpy array --- element-wise squaring
py_list  = list(range(n))
np_array = np.arange(n)

t_py = timeit.timeit(lambda: [x**2 for x in py_list], number=5)
t_np = timeit.timeit(lambda: np_array ** 2,            number=5)

print(f"Python list: {t_py:.3f}s")
print(f"Numpy array: {t_np:.4f}s")
print(f"Speedup: {t_py/t_np:.0f}x")
```

Output:

```
Python list: 2.341s
Numpy array: 0.024s
Speedup: 97x
```

```
# numpy operations are vectorized --- no Python loops needed
```

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])

print(a + b)            # [11 22 33 44 55]
print(a * b)            # [10 40 90 160 250]
print(np.dot(a, b))     # 550  (dot product)
print(np.sqrt(a))       # [1.  1.41 1.73 2.  2.24]

# 2D arrays (matrices)
matrix = np.arange(9).reshape(3, 3)
print(matrix)
# [[0 1 2]
#  [3 4 5]
#  [6 7 8]]
print(matrix.T)         # transpose
print(matrix @ matrix) # matrix multiplication
```

30.10 `__slots__` — Reducing Memory for Many Objects

By default, Python objects store their attributes in a dictionary (`__dict__`). For classes where you create millions of instances, this wastes memory. `__slots__` replaces the dict with a fixed set of attributes:

```
import sys

class PointDict:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

class PointSlots:
    __slots__ = ("x", "y", "z")

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


pd = PointDict(1.0, 2.0, 3.0)
```

```
ps = PointSlots(1.0, 2.0, 3.0)

print(f"With __dict__:  {sys.getsizeof(pd) + sys.getsizeof(pd.__dict__)} bytes")
print(f"With __slots__: {sys.getsizeof(ps)} bytes")

# With __dict__:  232 bytes
# With __slots__:  56 bytes   --- 4x smaller
```

For a million instances, that's 176MB vs 56MB. Use `__slots__` when you create many instances of a class with a fixed set of attributes.

30.11 Profiling in Practice: A Case Study

Let's profile a realistic slow function and fix it:

```
import cProfile
import pstats
import io
import timeit
from collections import defaultdict

# ── Version 1: Naive ──────────────────────────────────────────────

def word_frequency_v1(texts):
    """Count word frequencies across a list of strings."""
    frequencies = {}
    for text in texts:
        words = text.lower().split()
        for word in words:
            word = "".join(c for c in word if c.isalpha())  # strip punctuation
            if word:
                if word in frequencies:
                    frequencies[word] += 1
                else:
                    frequencies[word] = 1
    return dict(sorted(frequencies.items(), key=lambda x: x[1], reverse=True))


# ── Version 2: Optimized ──────────────────────────────────────────

import re
from collections import Counter
```

```python
WORD_RE = re.compile(r"[a-z]+")    # compiled once at module level

def word_frequency_v2(texts):
    """Faster version using Counter and compiled regex."""
    counter = Counter()
    for text in texts:
        counter.update(WORD_RE.findall(text.lower()))
    return dict(counter.most_common())


# ── Benchmark ──────────────────────────────────────────────────────

import random
import string

# Generate test data
words = ["python", "programming", "fast", "slow", "optimize", "measure",
         "performance", "data", "code", "function", "class", "list", "dict"]
texts = [
    " ".join(random.choices(words, k=100)) + "."
    for _ in range(1_000)
]

t1 = timeit.timeit(lambda: word_frequency_v1(texts), number=10)
t2 = timeit.timeit(lambda: word_frequency_v2(texts), number=10)

print(f"v1 (naive):     {t1:.3f}s")
print(f"v2 (optimized): {t2:.3f}s")
print(f"Speedup: {t1/t2:.1f}x")
```

Output:

```
v1 (naive):     4.821s
v2 (optimized): 0.312s
Speedup: 15.5x
```

The optimizations: 1. Compiled regex (`WORD_RE`) instead of `"".join(c for c in word if c.isalpha())` in a loop 2. `Counter` (implemented in C) instead of a dict with manual increment 3. `WORD_RE.findall()` processes the entire string at once instead of looping character by character

---

30.12 Performance Checklist

Before writing a single line of optimized code:

1. **Profile first.** Use `cProfile` to find the actual bottleneck.
2. **Check the algorithm.** Is there an O(n) solution to your O(n^2) problem?
3. **Check the data structure.** Replace `list` with `set` or `dict` for membership testing.
4. **Use built-ins.** `sum`, `max`, `sorted`, `Counter`, `defaultdict` — all implemented in C.
5. **Move work out of loops.** Compile regexes, bind methods, precompute constants.
6. **Use generators for large data.** Don't build a list if you only need to iterate once.
7. **Consider `numpy`.** For numerical arrays, it's 10-100x faster than Python loops.
8. **Cache expensive results.** `@lru_cache` for pure functions, `@functools.cache` (Python 3.9+).
9. **`__slots__`** for classes with many instances and fixed attributes.
10. **Measure the improvement.** Confirm the speedup with `timeit` before and after.

30.13 What You Learned in This Chapter

- Always measure before optimizing. Guessing wastes time and often makes things worse.
- `timeit.timeit()` benchmarks small code snippets accurately.
- `cProfile` profiles entire programs — shows call count, total time, cumulative time per function.
- `snakeviz` visualizes `cProfile` output as an interactive flame graph.
- `line_profiler` shows timing line by line inside a function.
- `memory_profiler` shows memory usage line by line.
- **Big O notation** describes how runtime scales with input size. O(n^2) vs O(n) can be the difference between seconds and hours.
- Choosing the right data structure is the highest-leverage optimization: `set` and `dict` membership is O(1), `list` is O(n).
- `collections.deque` for fast front/back operations. `collections.Counter` for counting. `collections.defaultdict` for grouped data.
- Practical wins: cache results, move work out of loops, compile regexes, use built-ins, use comprehensions, use generators for large data.
- `numpy` is 10-100x faster than Python loops for numerical operations on arrays.
- `__slots__` reduces memory by 4x for objects with many instances.

30.14 What's Next?

Chapter 31 covers **Working with Databases** — SQLite, PostgreSQL with `psycopg2`, and SQLAlchemy ORM. Data that outlives your program lives in a database, and knowing how to read, write, query, and migrate data is a fundamental skill for any Python developer.

Chapter 31

Chapter 31: Working with Databases

Files store data. But as soon as you need to search, filter, sort, or relate data across multiple collections, files become painful. Databases are purpose-built for exactly that.

This chapter covers three layers of database access in Python:

1. **SQLite** — built-in, zero setup, perfect for learning and small apps
2. **PostgreSQL with psycopg2** — production-grade relational database
3. **SQLAlchemy** — the standard Python ORM, lets you work with databases using Python objects instead of raw SQL

You'll understand SQL well enough to write real queries, and know when to use raw SQL vs an ORM.

31.1 Why Databases?

Consider storing 100,000 users in a CSV file. Finding a user by email means reading every line — O(n). Sorting by last name means loading everything into memory. Joining user data with their orders means matching records manually in Python.

A database does all of this efficiently: - Indexes make lookups O(log n) or O(1) - Queries happen in the database engine, not Python memory - Relationships are enforced by the schema - Multiple programs can read and write safely at the same time - Transactions ensure your data is always consistent

---

31.2 SQL Basics

SQL (Structured Query Language) is the language all relational databases use. The same core syntax works in SQLite, PostgreSQL, MySQL, and others.

The four fundamental operations — **CRUD**:

```
-- Create
INSERT INTO users (name, email, age) VALUES ('Alice', 'alice@example.com', 30);

-- Read
SELECT name, email FROM users WHERE age > 25 ORDER BY name;

-- Update
UPDATE users SET age = 31 WHERE email = 'alice@example.com';

-- Delete
DELETE FROM users WHERE email = 'alice@example.com';
```

Other essential SQL:

```
-- Create a table
CREATE TABLE users (
    id    INTEGER PRIMARY KEY AUTOINCREMENT,
    name  TEXT    NOT NULL,
    email TEXT    UNIQUE NOT NULL,
    age   INTEGER DEFAULT 0,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

-- Count rows
SELECT COUNT(*) FROM users;

-- Aggregate functions
SELECT AVG(age), MIN(age), MAX(age) FROM users;

-- Group and aggregate
SELECT age, COUNT(*) AS count FROM users GROUP BY age ORDER BY age;

-- Filter groups
SELECT age, COUNT(*) AS count FROM users
GROUP BY age
HAVING COUNT(*) > 1;

-- Join two tables
SELECT users.name, orders.total
```

```sql
FROM users
JOIN orders ON users.id = orders.user_id
WHERE orders.total > 100;

-- Left join --- include users with no orders
SELECT users.name, orders.total
FROM users
LEFT JOIN orders ON users.id = orders.user_id;
```

31.3 SQLite — The Built-in Database

SQLite is included with Python — no installation needed. Data lives in a single `.db` file. Perfect for prototypes, local apps, and learning.

31.3.1 Connecting and creating tables

```python
import sqlite3

# Connect --- creates the file if it doesn't exist
conn = sqlite3.connect("myapp.db")

# Use a context manager for automatic commit/rollback
with conn:
    conn.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id         INTEGER PRIMARY KEY AUTOINCREMENT,
            name       TEXT    NOT NULL,
            email      TEXT    UNIQUE NOT NULL,
            age        INTEGER DEFAULT 0,
            created_at TEXT    DEFAULT (datetime('now'))
        )
    """)

conn.close()
```

31.3.2 Inserting data safely

Always use parameterized queries — never format user input directly into SQL. Direct formatting opens the door to SQL injection attacks.

```python
conn = sqlite3.connect("myapp.db")

# WRONG --- SQL injection vulnerability
```

```
name = "Alice'; DROP TABLE users; --"
conn.execute(f"INSERT INTO users (name, email) VALUES ('{name}', 'a@b.com')")

# RIGHT --- parameterized query (? placeholders)
conn.execute(
    "INSERT INTO users (name, email, age) VALUES (?, ?, ?)",
    ("Alice", "alice@example.com", 30)
)
conn.commit()

# Insert many rows at once
users = [
    ("Bob",    "bob@example.com",    25),
    ("Carlos", "carlos@example.com", 35),
    ("Diana",  "diana@example.com",  28),
]
conn.executemany(
    "INSERT OR IGNORE INTO users (name, email, age) VALUES (?, ?, ?)",
    users
)
conn.commit()
conn.close()
```

31.3.3 Querying data

```
conn = sqlite3.connect("myapp.db")

# Make rows behave like dicts
conn.row_factory = sqlite3.Row

cursor = conn.execute("SELECT * FROM users ORDER BY name")

for row in cursor:
    print(f"{row['name']:<15} {row['email']:<30} age={row['age']}")

# Fetch methods
cursor = conn.execute("SELECT * FROM users WHERE age > ?", (27,))
one    = cursor.fetchone()    # first row or None
some   = cursor.fetchmany(3)  # list of up to 3 rows
all_   = cursor.fetchall()    # list of all remaining rows

# Parameterized query with named placeholders
cursor = conn.execute(
    "SELECT * FROM users WHERE name = :name AND age >= :min_age",
```

```
    {"name": "Alice", "min_age": 18}
)

conn.close()
```

31.3.4 Transactions

A **transaction** groups multiple statements into an atomic unit. Either all succeed or none do. SQLite uses transactions automatically, but you control them explicitly with `commit()` and `rollback()`:

```
conn = sqlite3.connect("myapp.db")

try:
    conn.execute("INSERT INTO accounts (user_id, balance) VALUES (1, 1000)")
    conn.execute("INSERT INTO accounts (user_id, balance) VALUES (2, 500)")

    # Transfer $200 from account 1 to account 2
    conn.execute("UPDATE accounts SET balance = balance - 200 WHERE user_id = 1
    conn.execute("UPDATE accounts SET balance = balance + 200 WHERE user_id = 2

    conn.commit()   # both updates succeed together
except Exception as e:
    conn.rollback()  # both updates undone if anything fails
    raise
finally:
    conn.close()
```

31.3.5 A full SQLite example

```
"""
contacts.py --- A simple contact book backed by SQLite.
"""
import sqlite3
from pathlib import Path
from typing import Optional


DB_PATH = Path("contacts.db")


def get_connection() -> sqlite3.Connection:
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row
    conn.execute("PRAGMA foreign_keys = ON")   # enforce FK constraints
```

```
    conn.execute("PRAGMA journal_mode = WAL")  # better concurrent access
    return conn


def setup_schema(conn: sqlite3.Connection) -> None:
    conn.executescript("""
        CREATE TABLE IF NOT EXISTS contacts (
            id         INTEGER PRIMARY KEY AUTOINCREMENT,
            name       TEXT    NOT NULL,
            email      TEXT    UNIQUE,
            phone      TEXT,
            notes      TEXT    DEFAULT '',
            created_at TEXT    DEFAULT (datetime('now'))
        );

        CREATE INDEX IF NOT EXISTS idx_contacts_name
            ON contacts(name COLLATE NOCASE);
    """)
    conn.commit()


def add_contact(conn, name: str, email: str = None,
                phone: str = None, notes: str = "") -> int:
    cursor = conn.execute(
        "INSERT INTO contacts (name, email, phone, notes) VALUES (?, ?, ?, ?)",
        (name, email, phone, notes)
    )
    conn.commit()
    return cursor.lastrowid   # ID of the newly inserted row


def find_by_name(conn, query: str) -> list:
    return conn.execute(
        "SELECT * FROM contacts WHERE name LIKE ? ORDER BY name",
        (f"%{query}%",)
    ).fetchall()


def find_by_id(conn, contact_id: int) -> Optional[sqlite3.Row]:
    return conn.execute(
        "SELECT * FROM contacts WHERE id = ?", (contact_id,)
    ).fetchone()
```

```python
def update_contact(conn, contact_id: int, **fields) -> bool:
    if not fields:
        return False
    allowed = {"name", "email", "phone", "notes"}
    updates = {k: v for k, v in fields.items() if k in allowed}
    if not updates:
        return False
    set_clause = ", ".join(f"{k} = ?" for k in updates)
    cursor = conn.execute(
        f"UPDATE contacts SET {set_clause} WHERE id = ?",
        [*updates.values(), contact_id]
    )
    conn.commit()
    return cursor.rowcount > 0


def delete_contact(conn, contact_id: int) -> bool:
    cursor = conn.execute("DELETE FROM contacts WHERE id = ?", (contact_id,))
    conn.commit()
    return cursor.rowcount > 0


def list_all(conn) -> list:
    return conn.execute("SELECT * FROM contacts ORDER BY name").fetchall()


def count(conn) -> int:
    return conn.execute("SELECT COUNT(*) FROM contacts").fetchone()[0]


# ── Demo ──────────────────────────────────────────────────────────

if __name__ == "__main__":
    conn = get_connection()
    setup_schema(conn)

    # Add contacts
    id1 = add_contact(conn, "Alice Smith",  "alice@example.com", "555-0101")
    id2 = add_contact(conn, "Bob Jones",    "bob@example.com",   "555-0102")
    id3 = add_contact(conn, "Alice Walker", "awalker@example.com")

    print(f"Added {count(conn)} contacts\n")

    # Search
```

```
    results = find_by_name(conn, "alice")
    print(f"Search 'alice': {len(results)} results")
    for r in results:
        print(f"  [{r['id']}] {r['name']} --- {r['email']}")

    # Update
    update_contact(conn, id1, phone="555-9999", notes="Updated phone")
    updated = find_by_id(conn, id1)
    print(f"\nUpdated: {updated['name']} --- {updated['phone']}")

    # Delete
    delete_contact(conn, id2)
    print(f"\nAfter delete: {count(conn)} contacts")

    conn.close()
```

31.4 PostgreSQL with psycopg2

For production applications, you want a full-featured database server. PostgreSQL is the best open-source relational database — it handles large datasets, concurrent users, advanced queries, and full ACID compliance.

```
pip install psycopg2-binary
```

```
import psycopg2
from psycopg2.extras import RealDictCursor

# Connect
conn = psycopg2.connect(
    host     = "localhost",
    port     = 5432,
    dbname   = "myapp",
    user     = "myuser",
    password = "mypassword",
)

# Use RealDictCursor so rows behave like dicts
with conn.cursor(cursor_factory=RealDictCursor) as cur:
    # Create table
    cur.execute("""
        CREATE TABLE IF NOT EXISTS products (
            id          SERIAL PRIMARY KEY,
            name        TEXT    NOT NULL,
```

```
            price       NUMERIC(10, 2) NOT NULL,
            stock       INTEGER DEFAULT 0,
            category    TEXT,
            created_at  TIMESTAMPTZ DEFAULT NOW()
        )
    """)

    # Insert --- psycopg2 uses %s placeholders
    cur.execute(
        "INSERT INTO products (name, price, stock, category) VALUES (%s, %s, %s
        ("Python Book", 29.99, 100, "Education")
    )

    # Insert many
    products = [
        ("Laptop",    999.99, 50,  "Electronics"),
        ("Keyboard",   49.99, 200, "Electronics"),
        ("Notebook",    4.99, 500, "Stationery"),
    ]
    cur.executemany(
        "INSERT INTO products (name, price, stock, category) VALUES (%s, %s, %s
        products
    )

    # Query
    cur.execute("SELECT * FROM products WHERE price < %s ORDER BY price", (100,
    for row in cur.fetchall():
        print(f"{row['name']}: ${row['price']}")

    conn.commit()

conn.close()
```

31.4.1 Connection pooling with `psycopg2.pool`

Opening a new database connection for every request is expensive — it takes ~100ms. Connection pooling maintains a pool of open connections and reuses them:

```
from psycopg2 import pool

connection_pool = pool.ThreadedConnectionPool(
    minconn = 2,
    maxconn = 10,
    host    = "localhost",
```

```
    dbname   = "myapp",
    user     = "myuser",
    password= "mypassword",
)

def get_user(user_id: int) -> dict:
    conn = connection_pool.getconn()
    try:
        with conn.cursor(cursor_factory=RealDictCursor) as cur:
            cur.execute("SELECT * FROM users WHERE id = %s", (user_id,))
            return cur.fetchone()
    finally:
        connection_pool.putconn(conn)    # return to pool, not close
```

31.5 SQLAlchemy — The Python ORM

Writing raw SQL for every query works, but it's verbose and error-prone for complex apps. **SQLAlchemy** is Python's most popular ORM (Object-Relational Mapper). It lets you define your data model as Python classes and query using Python syntax.

```
pip install sqlalchemy
```

31.5.1 Defining models

```
from sqlalchemy import (
    create_engine, Column, Integer, String, Float, Boolean,
    ForeignKey, DateTime, Text, func
)
from sqlalchemy.orm import DeclarativeBase, relationship, Session
from datetime import datetime


class Base(DeclarativeBase):
    pass


class User(Base):
    __tablename__ = "users"

    id         = Column(Integer, primary_key=True, autoincrement=True)
    name       = Column(String(100), nullable=False)
    email      = Column(String(200), unique=True, nullable=False)
```

```
    age        = Column(Integer, default=0)
    active     = Column(Boolean, default=True)
    created_at = Column(DateTime, default=datetime.utcnow)

    # Relationship --- one user has many orders
    orders = relationship("Order", back_populates="user", cascade="all, delete-

    def __repr__(self):
        return f"User(id={self.id}, name={self.name!r}, email={self.email!r})"


class Order(Base):
    __tablename__ = "orders"

    id         = Column(Integer, primary_key=True, autoincrement=True)
    user_id    = Column(Integer, ForeignKey("users.id"), nullable=False)
    total      = Column(Float,   nullable=False)
    status     = Column(String(20), default="pending")
    created_at = Column(DateTime, default=datetime.utcnow)
    notes      = Column(Text, default="")

    # Relationship --- each order belongs to one user
    user  = relationship("User", back_populates="orders")

    def __repr__(self):
        return f"Order(id={self.id}, total={self.total}, status={self.status!r}
```

31.5.2 Creating the database and session

```
# SQLite for this example (change the URL for PostgreSQL)
engine = create_engine(
    "sqlite:///shop.db",
    echo=False,   # set True to print all SQL
)

# Create all tables
Base.metadata.create_all(engine)

# Session factory
from sqlalchemy.orm import sessionmaker
SessionLocal = sessionmaker(bind=engine)
```

31.5.3 CRUD with SQLAlchemy

```
# ── Create ──────────────────────────────────────────────────────────

with SessionLocal() as session:
    alice = User(name="Alice", email="alice@example.com", age=30)
    bob   = User(name="Bob",   email="bob@example.com",   age=25)
    session.add_all([alice, bob])
    session.flush()   # assigns IDs without committing

    # Add orders for Alice
    session.add(Order(user_id=alice.id, total=59.98, status="completed"))
    session.add(Order(user_id=alice.id, total=129.99, status="pending"))
    session.add(Order(user_id=bob.id,   total=19.99,  status="completed"))

    session.commit()
    print(f"Created: {alice}, {bob}")


# ── Read ────────────────────────────────────────────────────────────

with SessionLocal() as session:
    # Get by primary key
    user = session.get(User, 1)
    print(user)

    # Filter
    active_users = session.query(User).filter(User.active == True).all()
    young_users  = session.query(User).filter(User.age < 30).all()
    alice        = session.query(User).filter_by(email="alice@example.com").first(

    # Order and limit
    top_users = session.query(User).order_by(User.name).limit(10).all()

    # Access related objects (lazy loaded by default)
    for user in session.query(User).all():
        print(f"{user.name}: {len(user.orders)} orders")


# ── Update ──────────────────────────────────────────────────────────

with SessionLocal() as session:
    user = session.query(User).filter_by(email="alice@example.com").first()
    if user:
```

```
        user.age = 31
        user.name = "Alice Smith"
        session.commit()
        print(f"Updated: {user}")


# ── Delete ──────────────────────────────────────────────────────────────

with SessionLocal() as session:
    user = session.get(User, 2)
    if user:
        session.delete(user)    # cascade deletes orders too
        session.commit()
        print("Deleted Bob and all his orders")
```

31.5.4 Querying with joins and aggregates

```
from sqlalchemy import func, and_, or_

with SessionLocal() as session:
    # Join users and orders
    results = (
        session.query(User.name, Order.total, Order.status)
        .join(Order, User.id == Order.user_id)
        .filter(Order.total > 50)
        .order_by(Order.total.desc())
        .all()
    )
    for name, total, status in results:
        print(f"{name}: ${total:.2f} ({status})")

    # Aggregate --- total spent per user
    spending = (
        session.query(
            User.name,
            func.count(Order.id).label("order_count"),
            func.sum(Order.total).label("total_spent"),
            func.avg(Order.total).label("avg_order"),
        )
        .join(Order, isouter=True)    # LEFT JOIN --- include users with no orde
        .group_by(User.id)
        .order_by(func.sum(Order.total).desc())
        .all()
    )
```

```
    print("\nSpending by user:")
    for row in spending:
        print(f"  {row.name}: {row.order_count} orders, "
              f"${row.total_spent or 0:.2f} total")

    # Complex filter
    high_value_pending = (
        session.query(Order)
        .filter(
            and_(
                Order.status == "pending",
                Order.total > 100,
            )
        )
        .all()
    )
```

31.5.5 SQLAlchemy 2.0 style (the modern way)

SQLAlchemy 2.0 introduced a cleaner query syntax using `select()`:

```
from sqlalchemy import select

with SessionLocal() as session:
    # Simple select
    stmt = select(User).where(User.age > 25).order_by(User.name)
    users = session.scalars(stmt).all()

    # Join
    stmt = (
        select(User, Order)
        .join(Order, User.id == Order.user_id)
        .where(Order.status == "pending")
    )
    rows = session.execute(stmt).all()

    # Update in bulk
    from sqlalchemy import update
    session.execute(
        update(Order)
        .where(Order.status == "pending")
        .values(status="processing")
    )
    session.commit()
```

31.6 Database Migrations with Alembic

When your database schema needs to change — adding a column, renaming a table, creating an index — you need **migrations**. Migrations are version-controlled SQL scripts that upgrade and downgrade your schema safely.

```
pip install alembic
alembic init migrations    # creates alembic.ini and migrations/ folder
```

Configure `alembic.ini`:

```
sqlalchemy.url = sqlite:///myapp.db
```

Configure `migrations/env.py` to point to your models:

```
from myapp.models import Base
target_metadata = Base.metadata
```

Generate a migration automatically from model changes:

```
alembic revision --autogenerate -m "add phone to users"
```

This produces a file like `migrations/versions/abc123_add_phone_to_users.py`:

```
def upgrade():
    op.add_column("users", sa.Column("phone", sa.String(20), nullable=True))

def downgrade():
    op.drop_column("users", "phone")
```

Apply all pending migrations:

```
alembic upgrade head
```

Roll back one migration:

```
alembic downgrade -1
```

Check current version:

```
alembic current
alembic history
```

31.7 Project: A Complete User + Order System

```
"""
shop_db.py --- A complete shop database with SQLAlchemy.
```

```
"""
from __future__ import annotations

from dataclasses import dataclass
from datetime import datetime
from typing import Optional

from sqlalchemy import (
    Boolean, Column, DateTime, Float, ForeignKey,
    Integer, String, Text, create_engine, func
)
from sqlalchemy.orm import (
    DeclarativeBase, Session, relationship, sessionmaker
)


# ── Models ──────────────────────────────────────────────────────────

class Base(DeclarativeBase):
    pass


class User(Base):
    __tablename__ = "users"

    id         = Column(Integer, primary_key=True)
    name       = Column(String(100), nullable=False)
    email      = Column(String(200), unique=True, nullable=False)
    active     = Column(Boolean, default=True)
    created_at = Column(DateTime, default=datetime.utcnow)

    orders = relationship("Order", back_populates="user",
                          cascade="all, delete-orphan")

    @property
    def total_spent(self) -> float:
        return sum(o.total for o in self.orders if o.status == "completed")

    def __repr__(self):
        return f"User({self.id}, {self.name!r})"


class Product(Base):
    __tablename__ = "products"
```

```
    id       = Column(Integer, primary_key=True)
    name     = Column(String(200), nullable=False)
    price    = Column(Float, nullable=False)
    stock    = Column(Integer, default=0)
    category = Column(String(50))

    order_items = relationship("OrderItem", back_populates="product")

    def __repr__(self):
        return f"Product({self.id}, {self.name!r}, ${self.price:.2f})"


class Order(Base):
    __tablename__ = "orders"

    id         = Column(Integer, primary_key=True)
    user_id    = Column(Integer, ForeignKey("users.id"), nullable=False)
    status     = Column(String(20), default="pending")
    created_at = Column(DateTime, default=datetime.utcnow)
    notes      = Column(Text, default="")

    user  = relationship("User",      back_populates="orders")
    items = relationship("OrderItem", back_populates="order",
                         cascade="all, delete-orphan")

    @property
    def total(self) -> float:
        return sum(item.subtotal for item in self.items)

    def __repr__(self):
        return f"Order({self.id}, user={self.user_id}, ${self.total:.2f})"


class OrderItem(Base):
    __tablename__ = "order_items"

    id         = Column(Integer, primary_key=True)
    order_id   = Column(Integer, ForeignKey("orders.id"), nullable=False)
    product_id = Column(Integer, ForeignKey("products.id"), nullable=False)
    quantity   = Column(Integer, default=1)
    unit_price = Column(Float, nullable=False)

    order   = relationship("Order",   back_populates="items")
```

```
    product = relationship("Product", back_populates="order_items")

    @property
    def subtotal(self) -> float:
        return self.unit_price * self.quantity


# ── Database setup ──────────────────────────────────────────────

engine       = create_engine("sqlite:///shop.db", echo=False)
Base.metadata.create_all(engine)
SessionLocal = sessionmaker(bind=engine)


# ── Service layer ───────────────────────────────────────────────

class ShopService:
    def __init__(self, session: Session):
        self.session = session

    # Users
    def create_user(self, name: str, email: str) -> User:
        user = User(name=name, email=email)
        self.session.add(user)
        self.session.commit()
        return user

    def get_user(self, user_id: int) -> Optional[User]:
        return self.session.get(User, user_id)

    def list_users(self, active_only: bool = True) -> list[User]:
        q = self.session.query(User)
        if active_only:
            q = q.filter(User.active == True)
        return q.order_by(User.name).all()

    # Products
    def create_product(self, name: str, price: float,
                       stock: int, category: str = "") -> Product:
        p = Product(name=name, price=price, stock=stock, category=category)
        self.session.add(p)
        self.session.commit()
        return p
```

```
    def list_products(self, category: str = None) -> list[Product]:
        q = self.session.query(Product).filter(Product.stock > 0)
        if category:
            q = q.filter(Product.category == category)
        return q.order_by(Product.name).all()

    # Orders
    def create_order(self, user_id: int, items: list[dict]) -> Order:
        """
        items: list of {"product_id": int, "quantity": int}
        """
        order = Order(user_id=user_id)
        self.session.add(order)
        self.session.flush()    # get order.id

        for item_data in items:
            product = self.session.get(Product, item_data["product_id"])
            if not product:
                raise ValueError(f"Product {item_data['product_id']} not found.
            qty = item_data["quantity"]
            if product.stock < qty:
                raise ValueError(
                    f"Insufficient stock for {product.name}: "
                    f"{product.stock} available, {qty} requested."
                )
            product.stock -= qty
            order_item = OrderItem(
                order_id   = order.id,
                product_id = product.id,
                quantity   = qty,
                unit_price = product.price,
            )
            self.session.add(order_item)

        self.session.commit()
        return order

    def complete_order(self, order_id: int) -> Order:
        order = self.session.get(Order, order_id)
        if not order:
            raise ValueError(f"Order {order_id} not found.")
        order.status = "completed"
        self.session.commit()
        return order
```

```
    # Reports
    def sales_report(self) -> list[dict]:
        rows = (
            self.session.query(
                User.name.label("customer"),
                func.count(Order.id).label("orders"),
                func.sum(OrderItem.unit_price * OrderItem.quantity).label("spent")
            )
            .join(Order, User.id == Order.user_id)
            .join(OrderItem, Order.id == OrderItem.order_id)
            .filter(Order.status == "completed")
            .group_by(User.id)
            .order_by(func.sum(OrderItem.unit_price * OrderItem.quantity).desc())
            .all()
        )
        return [{"customer": r.customer, "orders": r.orders,
                 "spent": float(r.spent)} for r in rows]


# ── Demo ──────────────────────────────────────────────────────────────

if __name__ == "__main__":
    with SessionLocal() as session:
        svc = ShopService(session)

        # Seed
        alice   = svc.create_user("Alice", "alice@example.com")
        bob     = svc.create_user("Bob",   "bob@example.com")

        book    = svc.create_product("Python Book",  29.99, 100, "Education")
        laptop  = svc.create_product("Laptop",       999.99, 20, "Electronics")
        pen     = svc.create_product("Pen",            2.50, 500, "Stationery")

        # Place orders
        order1 = svc.create_order(alice.id, [
            {"product_id": book.id,   "quantity": 2},
            {"product_id": pen.id,    "quantity": 5},
        ])
        order2 = svc.create_order(bob.id, [
            {"product_id": laptop.id, "quantity": 1},
        ])

        svc.complete_order(order1.id)
```

```
svc.complete_order(order2.id)

# Report
print("\n── Sales Report ──")
for row in svc.sales_report():
    print(f"  {row['customer']:<15} {row['orders']} orders  "
          f"${row['spent']:.2f} spent")

# Product stock
print("\n── Current Stock ──")
for p in svc.list_products():
    print(f"  {p.name:<20} ${p.price:>8.2f}  stock={p.stock}")
```

31.8 SQLite vs PostgreSQL — When to Use Which

Feature	SQLite	PostgreSQL
Setup	Zero — built into Python	Server required
Concurrent writes	Limited (one writer at a time)	Full concurrency
Data size	Up to ~140TB	Effectively unlimited
Performance	Excellent for reads	Excellent for reads and writes
Data types	Flexible (dynamic types)	Strict (typed)
JSON support	Basic	Full `jsonb` with indexing
Full-text search	Basic	Built-in, powerful
Best for	Local apps, prototyping, testing	Production apps, multi-user

Rule of thumb: Start with SQLite. Switch to PostgreSQL when you need multiple concurrent writers, a separate database server, or advanced SQL features.

31.9 What You Learned in This Chapter

- Databases provide efficient search, sort, filter, and relate operations that files cannot.
- **SQL CRUD:** `INSERT`, `SELECT`, `UPDATE`, `DELETE`. Always use parameterized queries — never format user input into SQL.
- **Transactions** group statements atomically — all succeed or all roll back.
- **SQLite** is built into Python, stores data in a single file, and is perfect for learning and small apps. `conn.row_factory = sqlite3.Row` makes rows behave like dicts.
- `cursor.lastrowid` gives the ID of the last inserted row.
- **PostgreSQL** handles production workloads. Use `psycopg2` for raw access, connection pooling for performance.
- **SQLAlchemy ORM** defines data models as Python classes, generates SQL for you, and manages relationships with `relationship()`.
- `session.add()`, `session.commit()`, `session.query()`, `session.get()` are the core ORM operations.
- **Alembic** manages schema migrations — version-controlled changes to your database structure.
- Use SQLite for prototypes, PostgreSQL for production.

31.10 What's Next?

Chapter 32 covers **Web Development with Flask and FastAPI** — building HTTP APIs, handling requests and responses, authentication, and serving data from your database over the web. Everything you've learned about Python, OOP, databases, and type hints comes together.

Chapter 32

Chapter 32: Web Development with Flask and FastAPI

Every web application — from a simple API to a complex platform — boils down to the same loop: receive a request, do some work, send a response. Python has excellent tools for this. Two dominate the ecosystem:

- **Flask** — lightweight, explicit, minimal magic. You control everything. Great for learning and small-to-medium apps.
- **FastAPI** — modern, fast, built on type hints and async. Generates API documentation automatically. The choice for new production APIs in 2026.

This chapter builds real applications with both. By the end you'll have a working REST API backed by a database with authentication.

32.1 HTTP in 60 Seconds

Every web request has: - A **method** — what you want to do (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`) - A **URL** — where to do it (`/users/42`) - **Headers** — metadata (`Content-Type: application/json`, `Authorization: Bearer token`) - A **body** — data sent with the request (only for `POST`, `PUT`, `PATCH`)

Every response has: - A **status code** — what happened (`200 OK`, `201 Created`, `400 Bad Request`, `404 Not Found`, `500 Internal Server Error`) - **Headers** — metadata about the response - A **body** — the data (usually JSON for APIs)

REST API conventions:

Method	URL	Action
`GET`	`/users`	List all users
`GET`	`/users/42`	Get user 42
`POST`	`/users`	Create a new user
`PUT`	`/users/42`	Replace user 42 entirely
`PATCH`	`/users/42`	Update user 42 partially
`DELETE`	`/users/42`	Delete user 42

32.2 Flask

```
pip install flask
```

32.2.1 Hello, Flask

```
# app.py
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, World!"

@app.route("/hello/<name>")
def hello(name):
    return f"Hello, {name}!"

if __name__ == "__main__":
    app.run(debug=True)
```

```
python app.py
# * Running on http://127.0.0.1:5000
```

Visit `http://localhost:5000/hello/Alice` -> `Hello, Alice!`

`@app.route` registers a URL pattern. `<name>` is a variable part of the URL, passed as a parameter to the function.

32.2.2 Returning JSON

```
from flask import Flask, jsonify, request, abort

app = Flask(__name__)

# In-memory store for this example
users = {
    1: {"id": 1, "name": "Alice", "email": "alice@example.com"},
    2: {"id": 2, "name": "Bob",   "email": "bob@example.com"},
}
next_id = 3


@app.route("/users", methods=["GET"])
def list_users():
    return jsonify(list(users.values()))


@app.route("/users/<int:user_id>", methods=["GET"])
def get_user(user_id):
    user = users.get(user_id)
    if user is None:
        abort(404)    # raises 404 Not Found
    return jsonify(user)


@app.route("/users", methods=["POST"])
def create_user():
    global next_id
    data = request.get_json()
    if not data or "name" not in data or "email" not in data:
        abort(400)    # Bad Request

    user = {"id": next_id, "name": data["name"], "email": data["email"]}
    users[next_id] = user
    next_id += 1
    return jsonify(user), 201    # 201 Created


@app.route("/users/<int:user_id>", methods=["PUT"])
def update_user(user_id):
    if user_id not in users:
        abort(404)
```

```
    data = request.get_json()
    if not data:
        abort(400)
    users[user_id].update(data)
    return jsonify(users[user_id])


@app.route("/users/<int:user_id>", methods=["DELETE"])
def delete_user(user_id):
    if user_id not in users:
        abort(404)
    del users[user_id]
    return "", 204   # 204 No Content
```

32.2.3 Error handlers

```
@app.errorhandler(404)
def not_found(e):
    return jsonify({"error": "Not found"}), 404

@app.errorhandler(400)
def bad_request(e):
    return jsonify({"error": "Bad request"}), 400

@app.errorhandler(500)
def server_error(e):
    return jsonify({"error": "Internal server error"}), 500
```

32.2.4 Request object

```
from flask import request

@app.route("/search")
def search():
    # Query parameters: /search?q=python&limit=10
    query = request.args.get("q", "")
    limit = request.args.get("limit", 10, type=int)

    # Headers
    auth  = request.headers.get("Authorization", "")
    ct    = request.headers.get("Content-Type", "")

    # JSON body
    data  = request.get_json()
```

```
    # Form data
    field = request.form.get("username")

    return jsonify({"query": query, "limit": limit})
```

32.2.5 Blueprints — organizing a larger app

```
# users/routes.py
from flask import Blueprint, jsonify

users_bp = Blueprint("users", __name__, url_prefix="/users")

@users_bp.route("/")
def list_users():
    return jsonify([])

@users_bp.route("/<int:user_id>")
def get_user(user_id):
    return jsonify({"id": user_id})


# orders/routes.py
from flask import Blueprint

orders_bp = Blueprint("orders", __name__, url_prefix="/orders")

@orders_bp.route("/")
def list_orders():
    return jsonify([])


# app.py
from flask import Flask
from users.routes import users_bp
from orders.routes import orders_bp

app = Flask(__name__)
app.register_blueprint(users_bp)
app.register_blueprint(orders_bp)
```

32.2.6 A complete Flask API with SQLite

```
"""
flask_api.py --- A REST API for a task manager, backed by SQLite.
"""
import sqlite3
from flask import Flask, jsonify, request, abort, g
from pathlib import Path

app     = Flask(__name__)
DB_PATH = Path("tasks.db")


# ── Database helpers ──────────────────────────────────────────────

def get_db():
    """Return a database connection, reusing the one for this request."""
    if "db" not in g:
        g.db = sqlite3.connect(DB_PATH)
        g.db.row_factory = sqlite3.Row
        g.db.execute("PRAGMA journal_mode=WAL")
    return g.db

@app.teardown_appcontext
def close_db(error):
    db = g.pop("db", None)
    if db is not None:
        db.close()

def init_db():
    with app.app_context():
        db = get_db()
        db.executescript("""
            CREATE TABLE IF NOT EXISTS tasks (
                id          INTEGER PRIMARY KEY AUTOINCREMENT,
                title       TEXT    NOT NULL,
                description TEXT    DEFAULT '',
                done        INTEGER DEFAULT 0,
                created_at  TEXT    DEFAULT (datetime('now'))
            );
        """)
        db.commit()
```

```python
# ── Routes ──────────────────────────────────────────────────────────

@app.route("/tasks", methods=["GET"])
def list_tasks():
    done  = request.args.get("done")
    query = "SELECT * FROM tasks"
    if done is not None:
        query += f" WHERE done = {1 if done == 'true' else 0}"
    query += " ORDER BY created_at DESC"
    tasks = get_db().execute(query).fetchall()
    return jsonify([dict(t) for t in tasks])


@app.route("/tasks/<int:task_id>", methods=["GET"])
def get_task(task_id):
    task = get_db().execute(
        "SELECT * FROM tasks WHERE id = ?", (task_id,)
    ).fetchone()
    if task is None:
        abort(404)
    return jsonify(dict(task))


@app.route("/tasks", methods=["POST"])
def create_task():
    data = request.get_json(silent=True) or {}
    title = data.get("title", "").strip()
    if not title:
        return jsonify({"error": "title is required"}), 400

    db     = get_db()
    cursor = db.execute(
        "INSERT INTO tasks (title, description) VALUES (?, ?)",
        (title, data.get("description", ""))
    )
    db.commit()
    task = db.execute(
        "SELECT * FROM tasks WHERE id = ?", (cursor.lastrowid,)
    ).fetchone()
    return jsonify(dict(task)), 201


@app.route("/tasks/<int:task_id>", methods=["PATCH"])
def update_task(task_id):
```

```
    task = get_db().execute(
        "SELECT id FROM tasks WHERE id = ?", (task_id,)
    ).fetchone()
    if task is None:
        abort(404)

    data    = request.get_json(silent=True) or {}
    allowed = {"title", "description", "done"}
    updates = {k: v for k, v in data.items() if k in allowed}
    if not updates:
        return jsonify({"error": "No valid fields to update"}), 400

    set_clause = ", ".join(f"{k} = ?" for k in updates)
    db = get_db()
    db.execute(
        f"UPDATE tasks SET {set_clause} WHERE id = ?",
        [*updates.values(), task_id]
    )
    db.commit()
    updated = db.execute(
        "SELECT * FROM tasks WHERE id = ?", (task_id,)
    ).fetchone()
    return jsonify(dict(updated))


@app.route("/tasks/<int:task_id>", methods=["DELETE"])
def delete_task(task_id):
    db     = get_db()
    cursor = db.execute("DELETE FROM tasks WHERE id = ?", (task_id,))
    db.commit()
    if cursor.rowcount == 0:
        abort(404)
    return "", 204


@app.errorhandler(404)
def not_found(e):
    return jsonify({"error": "Not found"}), 404

@app.errorhandler(400)
def bad_request(e):
    return jsonify({"error": "Bad request"}), 400
```

```
if __name__ == "__main__":
    init_db()
    app.run(debug=True)
```

Test it with `curl`:

```
# Create
curl -X POST http://localhost:5000/tasks \
     -H "Content-Type: application/json" \
     -d '{"title": "Learn FastAPI", "description": "Read the docs"}'

# List
curl http://localhost:5000/tasks

# Get one
curl http://localhost:5000/tasks/1

# Update
curl -X PATCH http://localhost:5000/tasks/1 \
     -H "Content-Type: application/json" \
     -d '{"done": 1}'

# Delete
curl -X DELETE http://localhost:5000/tasks/1
```

32.3 FastAPI

FastAPI is built on type hints, Pydantic models, and `asyncio`. It auto-generates interactive API documentation, validates request and response data automatically, and is one of the fastest Python frameworks available.

```
pip install fastapi uvicorn[standard]
```

32.3.1 Hello, FastAPI

```
# main.py
from fastapi import FastAPI

app = FastAPI(title="My API", version="1.0.0")

@app.get("/")
def home():
    return {"message": "Hello, World!"}
```

```
@app.get("/hello/{name}")
def hello(name: str):
    return {"message": f"Hello, {name}!"}

uvicorn main:app --reload
# Open http://localhost:8000/docs  <- interactive Swagger UI, auto-generated
# Open http://localhost:8000/redoc <- ReDoc documentation
```

32.3.2 Pydantic models for validation

FastAPI uses **Pydantic** for data validation. Define a model and FastAPI validates incoming JSON automatically:

```
from fastapi import FastAPI, HTTPException, status
from pydantic import BaseModel, EmailStr, Field, field_validator
from typing import Optional
from datetime import datetime

app = FastAPI()


class UserCreate(BaseModel):
    name:  str          = Field(..., min_length=1, max_length=100)
    email: str          = Field(..., pattern=r"[^@]+@[^@]+\.[^@]+")
    age:   int          = Field(..., ge=0, le=150)
    bio:   Optional[str] = None

    @field_validator("name")
    @classmethod
    def name_must_not_be_blank(cls, v: str) -> str:
        if not v.strip():
            raise ValueError("Name cannot be blank.")
        return v.strip()


class UserResponse(BaseModel):
    id:         int
    name:       str
    email:      str
    age:        int
    created_at: datetime

    model_config = {"from_attributes": True}  # allow ORM objects
```

```python
# In-memory store
users: dict[int, dict] = {}
next_id = 1


@app.get("/users", response_model=list[UserResponse])
def list_users():
    return list(users.values())


@app.get("/users/{user_id}", response_model=UserResponse)
def get_user(user_id: int):
    if user_id not in users:
        raise HTTPException(status_code=404, detail="User not found")
    return users[user_id]


@app.post("/users", response_model=UserResponse, status_code=201)
def create_user(user: UserCreate):
    global next_id
    new_user = {
        "id":         next_id,
        "name":       user.name,
        "email":      user.email,
        "age":        user.age,
        "created_at": datetime.utcnow(),
    }
    users[next_id] = new_user
    next_id += 1
    return new_user


@app.delete("/users/{user_id}", status_code=204)
def delete_user(user_id: int):
    if user_id not in users:
        raise HTTPException(status_code=404, detail="User not found")
    del users[user_id]
```

If you POST invalid data — a negative age, a malformed email — FastAPI returns a detailed `422 Unprocessable Entity` response automatically, with the exact field and error message. No validation code needed in your route handlers.

32.3.3 Query parameters and filters

```
from fastapi import Query

@app.get("/users")
def list_users(
    skip:   int = Query(0, ge=0),
    limit:  int = Query(10, ge=1, le=100),
    active: Optional[bool] = None,
    search: Optional[str]  = None,
):
    result = list(users.values())
    if active is not None:
        result = [u for u in result if u.get("active") == active]
    if search:
        result = [u for u in result if search.lower() in u["name"].lower()]
    return result[skip : skip + limit]
```

32.3.4 Path parameters with validation

```
from fastapi import Path

@app.get("/users/{user_id}")
def get_user(user_id: int = Path(..., ge=1)):
    ...

@app.get("/items/{item_name}")
def get_item(item_name: str = Path(..., min_length=1, max_length=50)):
    ...
```

32.3.5 Dependency injection

FastAPI's dependency injection system is one of its most powerful features. Dependencies are functions that run before your route handler and provide resources:

```
from fastapi import Depends
import sqlite3

def get_db():
    conn = sqlite3.connect("app.db")
    conn.row_factory = sqlite3.Row
    try:
        yield conn
    finally:
```

```
        conn.close()


@app.get("/tasks")
def list_tasks(db: sqlite3.Connection = Depends(get_db)):
    return db.execute("SELECT * FROM tasks").fetchall()

@app.post("/tasks")
def create_task(task: TaskCreate, db: sqlite3.Connection = Depends(get_db)):
    cursor = db.execute(
        "INSERT INTO tasks (title) VALUES (?)", (task.title,)
    )
    db.commit()
    return {"id": cursor.lastrowid, "title": task.title}
```

32.3.6 Async routes

FastAPI supports both sync and async route handlers. Use `async def` when calling async I/O (databases, HTTP, file I/O):

```
import asyncio
import httpx

@app.get("/weather/{city}")
async def get_weather(city: str):
    async with httpx.AsyncClient() as client:
        response = await client.get(
            f"https://api.open-meteo.com/v1/forecast",
            params={"latitude": 51.5, "longitude": -0.1, "current_weather": Tru
        )
        return response.json()
```

32.3.7 A complete FastAPI app with SQLAlchemy

```
"""
fastapi_app.py --- A production-ready task API with FastAPI + SQLAlchemy.
"""
from __future__ import annotations

from datetime import datetime
from typing import Optional

import uvicorn
from fastapi import Depends, FastAPI, HTTPException, Query, status
from pydantic import BaseModel, Field
```

```
from sqlalchemy import Boolean, Column, DateTime, Integer, String, Text, create_en
from sqlalchemy.orm import DeclarativeBase, Session, sessionmaker


# ── Database ──────────────────────────────────────────────────────────

class Base(DeclarativeBase):
    pass

class TaskModel(Base):
    __tablename__ = "tasks"
    id          = Column(Integer, primary_key=True, index=True)
    title       = Column(String(200), nullable=False)
    description = Column(Text, default="")
    done        = Column(Boolean, default=False)
    created_at  = Column(DateTime, default=datetime.utcnow)
    updated_at  = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcn


engine       = create_engine("sqlite:///tasks.db", connect_args={"check_same_threa
Base.metadata.create_all(engine)
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()


# ── Pydantic schemas ──────────────────────────────────────────────────

class TaskCreate(BaseModel):
    title:       str = Field(..., min_length=1, max_length=200)
    description: str = Field("", max_length=2000)

class TaskUpdate(BaseModel):
    title:       Optional[str]  = Field(None, min_length=1, max_length=200)
    description: Optional[str]  = Field(None, max_length=2000)
    done:        Optional[bool] = None

class TaskOut(BaseModel):
```

```
    id:          int
    title:       str
    description: str
    done:        bool
    created_at:  datetime
    updated_at:  datetime

    model_config = {"from_attributes": True}


# ── App ──────────────────────────────────────────────────────────

app = FastAPI(
    title       = "Task Manager API",
    description = "A production-ready task manager.",
    version     = "1.0.0",
)


@app.get("/tasks", response_model=list[TaskOut], tags=["Tasks"])
def list_tasks(
    skip:   int           = Query(0, ge=0),
    limit:  int           = Query(20, ge=1, le=100),
    done:   Optional[bool]= Query(None),
    search: Optional[str] = Query(None, min_length=1),
    db:     Session       = Depends(get_db),
):
    """List tasks with optional filtering and pagination."""
    q = db.query(TaskModel)
    if done is not None:
        q = q.filter(TaskModel.done == done)
    if search:
        q = q.filter(TaskModel.title.ilike(f"%{search}%"))
    return q.order_by(TaskModel.created_at.desc()).offset(skip).limit(limit).al


@app.get("/tasks/{task_id}", response_model=TaskOut, tags=["Tasks"])
def get_task(task_id: int, db: Session = Depends(get_db)):
    """Get a single task by ID."""
    task = db.get(TaskModel, task_id)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    return task
```

```python
@app.post("/tasks", response_model=TaskOut, status_code=201, tags=["Tasks"])
def create_task(payload: TaskCreate, db: Session = Depends(get_db)):
    """Create a new task."""
    task = TaskModel(**payload.model_dump())
    db.add(task)
    db.commit()
    db.refresh(task)
    return task


@app.patch("/tasks/{task_id}", response_model=TaskOut, tags=["Tasks"])
def update_task(task_id: int, payload: TaskUpdate, db: Session = Depends(get_db)):
    """Partially update a task."""
    task = db.get(TaskModel, task_id)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    for field, value in payload.model_dump(exclude_unset=True).items():
        setattr(task, field, value)
    task.updated_at = datetime.utcnow()
    db.commit()
    db.refresh(task)
    return task


@app.delete("/tasks/{task_id}", status_code=204, tags=["Tasks"])
def delete_task(task_id: int, db: Session = Depends(get_db)):
    """Delete a task."""
    task = db.get(TaskModel, task_id)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    db.delete(task)
    db.commit()


@app.get("/stats", tags=["Stats"])
def stats(db: Session = Depends(get_db)):
    """Return task statistics."""
    total   = db.query(TaskModel).count()
    done    = db.query(TaskModel).filter(TaskModel.done == True).count()
    pending = total - done
    return {"total": total, "done": done, "pending": pending}
```

```
if __name__ == "__main__":
    uvicorn.run("fastapi_app:app", host="0.0.0.0", port=8000, reload=True)
```

Run it, then visit `http://localhost:8000/docs` for the auto-generated Swagger UI where you can test every endpoint interactively, right in the browser.

32.4 Adding Authentication

The most common pattern for API authentication is **JWT (JSON Web Tokens)**:

```
pip install python-jose[cryptography] passlib[bcrypt]
```

```
from datetime import datetime, timedelta
from typing import Optional

from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel

SECRET_KEY  = "your-secret-key-store-in-env-not-in-code"
ALGORITHM   = "HS256"
TOKEN_EXPIRY_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/token")


def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain: str, hashed: str) -> bool:
    return pwd_context.verify(plain, hashed)

def create_token(data: dict, expires_delta: Optional[timedelta] = None) -> str:
    payload = data.copy()
    expire  = datetime.utcnow() + (expires_delta or timedelta(minutes=15))
    payload.update({"exp": expire})
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def decode_token(token: str) -> dict:
    try:
```

```
        return jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
    except JWTError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid or expired token",
            headers={"WWW-Authenticate": "Bearer"},
        )


# Fake user database
fake_users = {
    "alice": {
        "username": "alice",
        "hashed_password": hash_password("secret"),
        "email": "alice@example.com",
    }
}


@app.post("/auth/token", tags=["Auth"])
def login(form: OAuth2PasswordRequestForm = Depends()):
    user = fake_users.get(form.username)
    if not user or not verify_password(form.password, user["hashed_password"]):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
        )
    token = create_token({"sub": form.username},
                         timedelta(minutes=TOKEN_EXPIRY_MINUTES))
    return {"access_token": token, "token_type": "bearer"}


def get_current_user(token: str = Depends(oauth2_scheme)) -> dict:
    payload = decode_token(token)
    username = payload.get("sub")
    user = fake_users.get(username)
    if not user:
        raise HTTPException(status_code=401, detail="User not found")
    return user


# Protect a route
@app.get("/me", tags=["Auth"])
def get_me(current_user: dict = Depends(get_current_user)):
```

```
    return {"username": current_user["username"], "email": current_user["email"
}

@app.get("/tasks/mine", tags=["Tasks"])
def my_tasks(
    current_user: dict = Depends(get_current_user),
    db: Session        = Depends(get_db),
):
    # Filter tasks by the current user
    return db.query(TaskModel).filter(
        TaskModel.owner == current_user["username"]
    ).all()
```

32.5 Flask vs FastAPI — When to Use Which

Feature	Flask	FastAPI
Learning curve	Lower	Slightly higher
Auto-generated docs	No (use Flask-RESTX)	Yes — Swagger + ReDoc
Data validation	Manual	Automatic (Pydantic)
Type safety	Optional	First-class
Async support	Limited (Flask 2+)	Native
Performance	Good	Excellent (one of the fastest)
Ecosystem maturity	Very mature (since 2010)	Growing fast (2018+)
Best for	Prototypes, full-stack, legacy	New APIs, production REST

Rule of thumb: If you're building a new REST API in 2026, use FastAPI. If you're maintaining an existing Flask app or building a full-stack app with Jinja2 templates, use Flask.

32.6 Project Structure for a Real API

```
my_api/
├── app/
│   ├── __init__.py
│   ├── main.py           <- FastAPI app creation
│   ├── config.py         <- Settings from environment variables
```

```
│   ├── database.py       <- Engine, SessionLocal, get_db
│   ├── models/
│   │   ├── __init__.py
│   │   ├── user.py
│   │   └── task.py
│   ├── schemas/
│   │   ├── __init__.py
│   │   ├── user.py       <- Pydantic models for User
│   │   └── task.py       <- Pydantic models for Task
│   ├── routers/
│   │   ├── __init__.py
│   │   ├── users.py
│   │   ├── tasks.py
│   │   └── auth.py
│   ├── services/
│   │   ├── user_service.py
│   │   └── task_service.py
│   └── dependencies.py   <- get_db, get_current_user
├── tests/
│   ├── conftest.py
│   ├── test_users.py
│   └── test_tasks.py
├── alembic/              <- Database migrations
├── requirements.txt
├── .env            <- SECRET_KEY, DATABASE_URL (never commit this)
└── README.md
```

```python
# app/config.py --- settings from environment variables
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    database_url:    str = "sqlite:///./app.db"
    secret_key:      str = "change-me-in-production"
    token_expire_min:int = 30
    debug:           bool = False

    class Config:
        env_file = ".env"

settings = Settings()
```

```python
# app/routers/tasks.py --- a proper router module
from fastapi import APIRouter, Depends, HTTPException, Query
from sqlalchemy.orm import Session
from typing import Optional
```

```
from ..database    import get_db
from ..models.task import TaskModel
from ..schemas.task import TaskCreate, TaskUpdate, TaskOut
from ..dependencies import get_current_user

router = APIRouter(prefix="/tasks", tags=["Tasks"])

@router.get("/", response_model=list[TaskOut])
def list_tasks(
    skip:   int  = Query(0,  ge=0),
    limit:  int  = Query(20, ge=1, le=100),
    db: Session  = Depends(get_db),
):
    return db.query(TaskModel).offset(skip).limit(limit).all()

# ... other endpoints
```

```
# app/main.py
from fastapi import FastAPI
from .routers import tasks, users, auth

app = FastAPI(title="My API")
app.include_router(auth.router)
app.include_router(users.router)
app.include_router(tasks.router)
```

32.7 What You Learned in This Chapter

- HTTP has methods (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`), status codes (`200`, `201`, `400`, `404`, `500`), and a URL + body structure. REST APIs follow a standard URL naming convention.
- **Flask** registers routes with `@app.route`. `request.get_json()` reads the body. `jsonify()` returns JSON. `abort(404)` raises HTTP errors. Blueprints organize routes into separate files.
- **FastAPI** uses type hints and Pydantic models for automatic request validation and response serialization. Bad data returns a detailed `422` error automatically.
- `Field(...)` sets validation rules — `min_length`, `ge`, `le`, `pattern`, etc.
- `Query(...)` and `Path(...)` validate query parameters and path parameters.
- `Depends(get_db)` injects a database session using FastAPI's dependency

injection.

- `async def` routes handle async I/O natively in FastAPI.
- JWT tokens with `python-jose` and `passlib` implement secure authentication. `OAuth2PasswordBearer` wires it into FastAPI's docs UI.
- Run FastAPI with `uvicorn main:app --reload`. Docs appear at `/docs` and `/redoc`.
- Structure a real API into `models/`, `schemas/`, `routers/`, `services/`, and load settings from `.env` with `pydantic-settings`.

32.8 What's Next?

Chapter 33 covers **Data Science with Python** — `numpy`, `pandas`, `matplotlib`, and `scikit-learn`. Whether you're analyzing sales data, visualizing trends, or training a machine learning model, Python's data science stack is unmatched.

Chapter 33

Chapter 33: Data Science with Python

Python became the language of data science for one reason: the ecosystem. `numpy` for fast array math. `pandas` for working with tabular data. `matplotlib` and `seaborn` for visualization. `scikit-learn` for machine learning. These four libraries, used together, let you go from raw data to trained model to chart in a few hundred lines of code.

This chapter teaches you each one, then shows how they work together in a complete data analysis workflow.

33.1 The Data Science Workflow

Every data science project follows the same steps:

```
1. Load data          --- read CSV, database, API, JSON
2. Explore         --- understand shape, types, missing values, distributions
3. Clean          --- fix missing values, wrong types, duplicates, outliers
4. Analyze         --- compute statistics, find patterns, test hypotheses
5. Visualize          --- charts that reveal what numbers hide
6. Model            --- train a machine learning model (if needed)
7. Evaluate           --- measure how good the model is
8. Communicate        --- share findings clearly
```

33.2 numpy — Fast Array Mathematics

`numpy` is the foundation of the Python data science stack. It provides the `ndarray` — a typed, fixed-size, multidimensional array stored in contiguous memory. Operations on numpy arrays run in C — 10-100x faster than Python loops.

```
pip install numpy
```

33.2.1 Creating arrays

```
import numpy as np

# From a list
a = np.array([1, 2, 3, 4, 5])
print(a)            # [1 2 3 4 5]
print(a.dtype)      # int64
print(a.shape)      # (5,)
print(a.ndim)       # 1

# Specify dtype
b = np.array([1.0, 2.0, 3.0], dtype=np.float32)

# Built-in constructors
zeros   = np.zeros(5)                   # [0. 0. 0. 0. 0.]
ones    = np.ones((3, 4))               # 3x4 matrix of 1s
empty   = np.empty((2, 2))              # uninitialized (fast)
rng     = np.arange(0, 10, 2)           # [0 2 4 6 8]
linsp   = np.linspace(0, 1, 5)          # [0.   0.25 0.5  0.75 1.  ]
eye     = np.eye(3)                     # 3x3 identity matrix
rand    = np.random.rand(3, 3)          # 3x3 random floats [0, 1)
randn   = np.random.randn(3, 3)         # 3x3 standard normal
randint= np.random.randint(0, 10, (3, 3))  # 3x3 random ints
```

33.2.2 Vectorized operations

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])

# Element-wise --- no loops needed
print(a + b)        # [11 22 33 44 55]
print(a * b)        # [10 40 90 160 250]
print(a ** 2)       # [ 1  4  9 16 25]
print(np.sqrt(a))   # [1.   1.41 1.73 2.   2.24]

# Broadcasting --- operate on arrays of different shapes
```

```
matrix = np.ones((3, 4))
row    = np.array([1, 2, 3, 4])
print(matrix + row)   # adds row to each row of matrix

col = np.array([[1], [2], [3]])
print(matrix + col)   # adds col to each column
```

33.2.3 Indexing, slicing, and masking

```
a = np.array([10, 20, 30, 40, 50])

print(a[0])       # 10
print(a[-1])      # 50
print(a[1:4])     # [20 30 40]
print(a[::2])     # [10 30 50]

# 2D indexing
m = np.arange(12).reshape(3, 4)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

print(m[1, 2])    # 6   (row 1, col 2)
print(m[:, 1])    # [1 5 9]  (all rows, col 1)
print(m[0, :])    # [0 1 2 3]  (row 0, all cols)
print(m[1:, 2:])  # [[6 7] [10 11]]  (submatrix)

# Boolean masking
a = np.array([1, -2, 3, -4, 5])
mask = a > 0
print(mask)    # [ True False  True False  True]
print(a[mask]) # [1 3 5]
print(a[a > 0]) # same thing --- inline mask
a[a < 0] = 0   # set negatives to zero in-place
```

33.2.4 Aggregation and statistics

```
data = np.array([4, 7, 13, 2, 9, 1, 5, 8])

print(np.sum(data))     # 49
print(np.mean(data))    # 6.125
print(np.median(data))  # 6.0
print(np.std(data))     # 3.48
print(np.var(data))     # 12.11
```

```
print(np.min(data))      # 1
print(np.max(data))      # 13
print(np.argmin(data))   # 5  (index of minimum)
print(np.argmax(data))   # 2  (index of maximum)
print(np.percentile(data, [25, 50, 75]))  # quartiles

# Axis-wise for 2D
m = np.array([[1, 2, 3], [4, 5, 6]])
print(np.sum(m, axis=0))    # [5 7 9]  --- sum each column
print(np.sum(m, axis=1))    # [ 6 15]  --- sum each row
```

33.2.5 Linear algebra

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(A @ B)           # matrix multiplication
print(A.T)             # transpose
print(np.linalg.det(A))          # determinant: -2.0
print(np.linalg.inv(A))          # inverse
eigenvalues, eigenvectors = np.linalg.eig(A)
```

33.3 pandas — Tabular Data

`pandas` adds two data structures on top of numpy: `Series` (1D) and `DataFrame` (2D table). A DataFrame is like a spreadsheet or database table — columns have names and types, rows have an index.

```
pip install pandas
```

33.3.1 Creating DataFrames

```
import pandas as pd

# From a dict
df = pd.DataFrame({
    "name":   ["Alice", "Bob", "Carlos", "Diana"],
    "age":    [30, 25, 35, 28],
    "salary": [95000, 82000, 110000, 78000],
    "dept":   ["Eng", "HR", "Eng", "Sales"],
})

print(df)
```

```
#      name  age  salary   dept
# 0   Alice   30   95000    Eng
# 1     Bob   25   82000     HR
# 2  Carlos   35  110000    Eng
# 3   Diana   28   78000  Sales

print(df.shape)   # (4, 4)
print(df.dtypes)
# name       object
# age         int64
# salary      int64
# dept       object
```

33.3.2 Loading data from files

```
# CSV --- most common
df = pd.read_csv("data.csv")
df = pd.read_csv("data.csv", sep=";", encoding="utf-8", index_col="id")

# Excel
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")

# JSON
df = pd.read_json("data.json")

# From a SQL query
import sqlite3
conn = sqlite3.connect("myapp.db")
df   = pd.read_sql("SELECT * FROM users", conn)

# Save back to file
df.to_csv("output.csv", index=False)
df.to_excel("output.xlsx", index=False)
df.to_json("output.json", orient="records")
```

33.3.3 Exploring data

```
df = pd.read_csv("sales.csv")

print(df.head())          # first 5 rows
print(df.tail(3))         # last 3 rows
print(df.shape)           # (rows, columns)
print(df.columns.tolist()) # column names
print(df.dtypes)           # column types
```

```
print(df.info())            # shape + types + null counts
print(df.describe())        # count, mean, std, min, quartiles, max

# Missing values
print(df.isnull().sum())    # null count per column
print(df.isnull().mean())   # null fraction per column
```

33.3.4 Selecting data

```
# Select a column --- returns a Series
names = df["name"]

# Select multiple columns --- returns a DataFrame
subset = df[["name", "salary"]]

# Select rows by position (like list slicing)
first_three = df.iloc[0:3]
last_row    = df.iloc[-1]
cell        = df.iloc[0, 1]     # row 0, column 1

# Select rows by label/condition (loc)
row         = df.loc[0]         # row with index 0
eng_staff   = df.loc[df["dept"] == "Eng"]

# Boolean filtering
high_earners = df[df["salary"] > 90000]
eng_seniors  = df[(df["dept"] == "Eng") & (df["age"] > 30)]
hr_or_sales  = df[df["dept"].isin(["HR", "Sales"])]
```

33.3.5 Cleaning data

```
# Drop rows with any null values
df_clean = df.dropna()

# Drop rows where specific column is null
df_clean = df.dropna(subset=["email"])

# Fill null values
df["age"].fillna(df["age"].median(), inplace=True)
df["notes"].fillna("", inplace=True)

# Drop duplicate rows
df = df.drop_duplicates()
df = df.drop_duplicates(subset=["email"])    # deduplicate by email
```

```python
# Rename columns
df = df.rename(columns={"dept": "department", "salary": "annual_salary"})

# Change data types
df["age"]    = df["age"].astype(int)
df["date"]   = pd.to_datetime(df["date"])
df["salary"] = pd.to_numeric(df["salary"], errors="coerce")  # NaN for bad valu

# Strip whitespace from string columns
df["name"] = df["name"].str.strip()
df["name"] = df["name"].str.title()   # Title Case

# Remove outliers (beyond 3 standard deviations)
col  = df["salary"]
mask = (col - col.mean()).abs() <= 3 * col.std()
df   = df[mask]
```

33.3.6 Transforming data

```python
# Add a computed column
df["monthly_salary"] = df["salary"] / 12
df["senior"]         = df["age"] >= 35

# Apply a function to a column
df["name_upper"] = df["name"].str.upper()
df["tax"]        = df["salary"].apply(lambda s: s * 0.3 if s > 100000 else s *

# Apply a function to every cell
df_normalized = df[["age", "salary"]].apply(
    lambda col: (col - col.min()) / (col.max() - col.min())
)

# Map values
dept_map = {"Eng": "Engineering", "HR": "Human Resources", "Sales": "Sales"}
df["dept_full"] = df["dept"].map(dept_map)
```

33.3.7 Grouping and aggregation

```python
# Group by one column
dept_stats = df.groupby("dept").agg({
    "salary": ["mean", "min", "max", "count"],
    "age":    "mean",
})
```

```
print(dept_stats)

# Rename the multi-level columns
dept_stats.columns = ["avg_salary", "min_salary", "max_salary", "headcount", "avg_

# Group by multiple columns
df.groupby(["dept", "senior"])["salary"].mean()

# Pivot table
pivot = df.pivot_table(
    values  = "salary",
    index   = "dept",
    columns = "senior",
    aggfunc = "mean",
)
```

33.3.8 Sorting and ranking

```
df_sorted = df.sort_values("salary", ascending=False)
df_sorted = df.sort_values(["dept", "salary"], ascending=[True, False])

df["salary_rank"] = df["salary"].rank(ascending=False).astype(int)
```

33.3.9 Merging DataFrames

```
users  = pd.DataFrame({"id": [1, 2, 3], "name": ["Alice", "Bob", "Carlos"]})
orders = pd.DataFrame({"user_id": [1, 1, 2], "amount": [50, 30, 120]})

# Inner join --- only rows that match in both
merged = pd.merge(users, orders, left_on="id", right_on="user_id")

# Left join --- all rows from left, NaN where no match in right
merged = pd.merge(users, orders, left_on="id", right_on="user_id", how="left")

# Stack DataFrames vertically
combined = pd.concat([df1, df2], ignore_index=True)
```

33.4 matplotlib — Data Visualization

`matplotlib` is Python's core plotting library. Every other visualization library builds on top of it.

```
pip install matplotlib seaborn
```

```
import matplotlib.pyplot as plt
import numpy as np

# Line plot
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

plt.figure(figsize=(10, 4))
plt.plot(x, y, color="steelblue", linewidth=2, label="sin(x)")
plt.plot(x, np.cos(x), color="coral", linestyle="--", label="cos(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Sine and Cosine")
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("trig.png", dpi=150)
plt.show()
```

33.4.1 Common chart types

```
fig, axes = plt.subplots(2, 3, figsize=(15, 8))

# Bar chart
depts  = ["Eng", "HR", "Sales"]
counts = [15, 8, 12]
axes[0, 0].bar(depts, counts, color="steelblue")
axes[0, 0].set_title("Headcount by Department")

# Horizontal bar
axes[0, 1].barh(depts, counts, color="coral")
axes[0, 1].set_title("Headcount (horizontal)")

# Histogram
data = np.random.normal(70000, 15000, 500)
axes[0, 2].hist(data, bins=30, color="mediumpurple", edgecolor="white")
axes[0, 2].set_title("Salary Distribution")
axes[0, 2].set_xlabel("Salary ($)")

# Scatter plot
x = np.random.rand(100)
y = x * 2 + np.random.randn(100) * 0.3
axes[1, 0].scatter(x, y, alpha=0.6, c="steelblue", edgecolors="white")
axes[1, 0].set_title("Scatter Plot")
```

```
# Pie chart
sizes  = [45, 25, 30]
labels = ["Eng", "HR", "Sales"]
axes[1, 1].pie(sizes, labels=labels, autopct="%1.1f%%", startangle=90)
axes[1, 1].set_title("Department Share")

# Box plot
data_by_group = [np.random.normal(loc, 5, 50) for loc in [60, 70, 80]]
axes[1, 2].boxplot(data_by_group, labels=["Q1", "Q2", "Q3"])
axes[1, 2].set_title("Score by Quarter")

plt.tight_layout()
plt.savefig("charts.png", dpi=150)
plt.show()
```

33.4.2 `seaborn` — statistical visualization

`seaborn` builds on matplotlib with a better default style and higher-level statistical charts:

```
import seaborn as sns
import pandas as pd
import numpy as np

# Use the built-in tips dataset for demonstration
tips = sns.load_dataset("tips")

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Distribution plot
sns.histplot(tips["total_bill"], kde=True, ax=axes[0, 0])
axes[0, 0].set_title("Bill Distribution")

# Scatter with regression line
sns.regplot(x="total_bill", y="tip", data=tips, ax=axes[0, 1])
axes[0, 1].set_title("Tip vs Bill")

# Box plot by category
sns.boxplot(x="day", y="total_bill", data=tips, ax=axes[1, 0])
axes[1, 0].set_title("Bills by Day")

# Heatmap --- correlation matrix
numeric_cols = tips.select_dtypes(include="number")
corr = numeric_cols.corr()
```

```
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm", ax=axes[1, 1])
axes[1, 1].set_title("Correlation Matrix")

plt.tight_layout()
plt.savefig("seaborn_charts.png", dpi=150)
plt.show()
```

33.5 scikit-learn — Machine Learning

`scikit-learn` provides a consistent API for dozens of machine learning algorithms. Every model follows the same pattern: `fit()`, `predict()`, `score()`.

```
pip install scikit-learn
```

33.5.1 The scikit-learn API

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing   import StandardScaler
from sklearn.linear_model    import LinearRegression
from sklearn.metrics         import mean_squared_error, r2_score
import numpy as np

# 1. Prepare data
X = np.array([[1], [2], [3], [4], [5], [6], [7], [8]])
y = np.array([2.1, 3.9, 6.2, 7.8, 10.1, 12.0, 14.2, 15.9])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, rando

# 2. Train model
model = LinearRegression()
model.fit(X_train, y_train)

# 3. Predict
y_pred = model.predict(X_test)

# 4. Evaluate
mse  = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2   = r2_score(y_test, y_pred)

print(f"RMSE: {rmse:.3f}")
print(f"R^2:   {r2:.3f}")
```

```
print(f"Coefficient: {model.coef_[0]:.3f}")
print(f"Intercept:   {model.intercept_:.3f}")
```

33.5.2 A classification example

```
from sklearn.datasets        import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing   import StandardScaler
from sklearn.ensemble        import RandomForestClassifier
from sklearn.metrics         import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Load built-in dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Scale features
scaler  = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)    # use same scaler --- never fit on test data

# Train
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Cross-validation --- more robust estimate of performance
cv_scores = cross_val_score(model, X, y, cv=5, scoring="accuracy")
print(f"CV accuracy: {cv_scores.mean():.3f} ± {cv_scores.std():.3f}")

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=iris.target_names,
            yticklabels=iris.target_names)
```

```
plt.title("Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.tight_layout()
plt.savefig("confusion_matrix.png")
```

33.5.3 Pipelines — the right way to combine steps

A `Pipeline` chains preprocessing and modeling steps, preventing data leakage and making deployment clean:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV

# Suppose our features have mixed types
numeric_features     = ["age", "salary"]
categorical_features = ["dept", "location"]

# Preprocessing for each type
numeric_transformer     = Pipeline([("scaler", StandardScaler())])
categorical_transformer = Pipeline([("onehot", OneHotEncoder(handle_unknown="ig

preprocessor = ColumnTransformer([
    ("num", numeric_transformer,     numeric_features),
    ("cat", categorical_transformer, categorical_features),
])

# Full pipeline: preprocess -> model
full_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier",   GradientBoostingClassifier(random_state=42)),
])

# Hyperparameter search over the whole pipeline
param_grid = {
    "classifier__n_estimators":  [50, 100, 200],
    "classifier__learning_rate": [0.05, 0.1, 0.2],
    "classifier__max_depth":     [3, 4, 5],
}

grid_search = GridSearchCV(
    full_pipeline,
```

```
    param_grid,
    cv=5,
    scoring="accuracy",
    n_jobs=-1,      # use all CPU cores
    verbose=1,
)

# grid_search.fit(X_train, y_train)
# print(f"Best params: {grid_search.best_params_}")
# print(f"Best score:  {grid_search.best_score_:.3f}")
```

33.6 Project: End-to-End Sales Analysis

Let's tie everything together in a complete analysis workflow:

```
"""
sales_analysis.py --- End-to-end data analysis of sales data.
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import seaborn as sns
from pathlib import Path


# ── 1. Generate synthetic data ──────────────────────────────────────

np.random.seed(42)
rng = pd.date_range("2024-01-01", "2024-12-31", freq="D")

n = 500
data = pd.DataFrame({
    "date":       np.random.choice(rng, n),
    "product":    np.random.choice(["Laptop", "Phone", "Tablet", "Watch"], n,
                                   p=[0.2, 0.4, 0.3, 0.1]),
    "region":     np.random.choice(["North", "South", "East", "West"], n),
    "units":      np.random.randint(1, 20, n),
    "unit_price":np.random.choice([999.99, 699.99, 499.99, 299.99], n,
                                   p=[0.2, 0.4, 0.3, 0.1]),
    "rep":        np.random.choice([f"Rep{i:02d}" for i in range(1, 11)], n),
})
```

```
# Inject some missing values and noise
data.loc[np.random.choice(n, 20), "unit_price"] = np.nan
data.loc[np.random.choice(n, 5),  "units"]      = -1   # bad data


# ── 2. Clean ──────────────────────────────────────────────────────────

# Remove impossible values
data = data[data["units"] > 0]

# Fill missing prices with median per product
data["unit_price"] = data.groupby("product")["unit_price"].transform(
    lambda x: x.fillna(x.median())
)

# Add computed columns
data["revenue"] = data["units"] * data["unit_price"]
data["month"]   = pd.to_datetime(data["date"]).dt.month
data["quarter"] = pd.to_datetime(data["date"]).dt.quarter

print(f"Loaded {len(data):,} records after cleaning")
print(f"Total revenue: ${data['revenue'].sum():,.0f}\n")


# ── 3. Analyze ────────────────────────────────────────────────────────

# Revenue by product
by_product = (
    data.groupby("product")["revenue"]
    .agg(["sum", "mean", "count"])
    .rename(columns={"sum": "total", "mean": "avg_order", "count": "orders"})
    .sort_values("total", ascending=False)
)
print("Revenue by product:")
for product, row in by_product.iterrows():
    print(f"  {product:<8}  ${row['total']:>10,.0f}  ({row['orders']} orders)")

# Revenue by region
by_region = data.groupby("region")["revenue"].sum().sort_values(ascending=False

# Monthly trend
monthly = data.groupby("month")["revenue"].sum()
```

```
# Top sales reps
top_reps = data.groupby("rep")["revenue"].sum().nlargest(5)


# ── 4. Visualize ──────────────────────────────────────────────────────────

sns.set_theme(style="whitegrid", palette="muted")
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle("2024 Sales Dashboard", fontsize=16, fontweight="bold")

# Revenue by product (bar)
by_product["total"].plot(kind="bar", ax=axes[0, 0], color="steelblue", edgecolor="
axes[0, 0].set_title("Total Revenue by Product")
axes[0, 0].set_xlabel("")
axes[0, 0].yaxis.set_major_formatter(mticker.FuncFormatter(lambda x, _: f"${x/1e3:
axes[0, 0].tick_params(axis="x", rotation=0)

# Revenue by region (horizontal bar)
by_region.plot(kind="barh", ax=axes[0, 1], color="coral", edgecolor="white")
axes[0, 1].set_title("Revenue by Region")
axes[0, 1].xaxis.set_major_formatter(mticker.FuncFormatter(lambda x, _: f"${x/1e3:

# Monthly revenue trend (line)
monthly.plot(kind="line", ax=axes[1, 0], marker="o", color="mediumpurple", linewid
axes[1, 0].set_title("Monthly Revenue")
axes[1, 0].set_xlabel("Month")
axes[1, 0].yaxis.set_major_formatter(mticker.FuncFormatter(lambda x, _: f"${x/1e3:

# Top reps (bar)
top_reps.plot(kind="bar", ax=axes[1, 1], color="mediumseagreen", edgecolor="white"
axes[1, 1].set_title("Top 5 Sales Reps")
axes[1, 1].set_xlabel("")
axes[1, 1].yaxis.set_major_formatter(mticker.FuncFormatter(lambda x, _: f"${x/1e3:
axes[1, 1].tick_params(axis="x", rotation=30)

plt.tight_layout()
plt.savefig("sales_dashboard.png", dpi=150, bbox_inches="tight")
plt.show()
print("Dashboard saved to sales_dashboard.png")


# ── 5. Simple forecast with scikit-learn ──────────────────────────────────

from sklearn.linear_model    import LinearRegression
```

```
from sklearn.metrics          import mean_absolute_percentage_error

# Use month as a feature to predict revenue
X = monthly.index.values.reshape(-1, 1)
y = monthly.values

model = LinearRegression()
model.fit(X, y)

# Predict next 3 months (months 13, 14, 15)
next_months = np.array([[13], [14], [15]])
forecast    = model.predict(next_months)

print("\nForecast for next 3 months:")
for month, pred in zip([13, 14, 15], forecast):
    print(f"  Month {month}: ${pred:,.0f}")

# MAPE on training data
mape = mean_absolute_percentage_error(y, model.predict(X))
print(f"\nTraining MAPE: {mape:.1%}")
```

33.7 The Data Science Toolkit Summary

Library	Purpose	Key objects
`numpy`	Fast array math	`ndarray`
`pandas`	Tabular data	`DataFrame`, `Series`
`matplotlib`	Low-level plotting	`Figure`, `Axes`
`seaborn`	Statistical visualization	High-level wrappers
`scikit-learn`	Machine learning	`Pipeline`, `fit/predict/score`
`scipy`	Scientific computing	Statistics, optimization, signal
`statsmodels`	Statistical modeling	Regression, time series
`plotly`	Interactive charts	Web-ready visualizations
`jupyter`	Interactive notebooks	`.ipynb` files

33.8 What You Learned in This Chapter

- The data science workflow: load -> explore -> clean -> analyze -> visualize -> model -> evaluate.

- **numpy** provides the `ndarray` — vectorized operations run in C, 10-100x faster than Python loops. Key tools: broadcasting, boolean masking, `reshape`, `argmin/argmax`, `np.linalg`.
- **pandas** `DataFrame` is the workhorse for tabular data. `read_csv/to_csv`, `head/info/describe`, `iloc/loc`, boolean filtering, `fillna/dropna/drop_duplicates`, `groupby/agg`, `merge/concat`, `apply/map`.
- Never `fit` a scaler or encoder on test data — only on training data.
- **matplotlib** — `plt.figure`, `plt.plot/bar/hist/scatter/pie`, `axes.set_title/xlabel/ylabel`, `plt.savefig`.
- **seaborn** — `histplot`, `regplot`, `boxplot`, `heatmap` — higher-level, better defaults.
- **scikit-learn** — `train_test_split`, `StandardScaler`, `fit/predict/score`, `classification_report`, `cross_val_score`, `Pipeline`, `GridSearchCV`.
- A `Pipeline` chains preprocessing and modeling — prevents data leakage and makes deployment clean.

33.9 What's Next?

Chapter 34 covers **Packaging and Distribution** — `pyproject.toml`, building installable packages, publishing to PyPI, and setting up a professional Python project that others can install with `pip install yourpackage`.

Chapter 34

Chapter 34: Packaging and Distribution

You've written useful code. Now you want to share it — with a colleague, with your team, or with the world on PyPI so anyone can install it with `pip install yourpackage`. This chapter shows you how.

Packaging is also important even for internal projects. A properly packaged project is easier to install, version, test, and maintain than a loose collection of scripts.

34.1 What a Package Actually Is

When someone runs `pip install requests`, pip downloads a **wheel** (or source distribution) from PyPI, unpacks it, and copies files into their Python's `site-packages` directory. After that, `import requests` works anywhere in that environment.

A Python package is fundamentally: 1. A directory with an `__init__.py` 2. A `pyproject.toml` that describes the package (name, version, dependencies, entry points) 3. Optionally, a build backend that converts those files into a distributable artifact

---

34.2 Modern Project Structure

The modern, recommended layout for a Python package:

```
my_package/                    <- project root (git repo)
├── src/
│   └── my_package/          <- the actual package (importable code)
│       ├── __init__.py
│       ├── core.py
│       ├── utils.py
│       └── cli.py
├── tests/
│   ├── __init__.py
│   ├── test_core.py
│   └── test_utils.py
├── docs/
│   └── index.md
├── pyproject.toml         <- the single source of truth for everything
├── README.md
├── LICENSE
└── .gitignore
```

The `src/` layout (putting your package inside a `src/` directory) prevents the common mistake where Python imports your local source instead of your installed package during testing.

34.3 `pyproject.toml` — The Modern Configuration File

`pyproject.toml` replaced `setup.py`, `setup.cfg`, and `requirements.txt` in one file. It's the standard for every new Python project.

```toml
[build-system]
requires      = ["hatchling"]
build-backend = "hatchling.build"


[project]
name         = "my-awesome-package"
version      = "0.1.0"
description = "A short description of what this package does"
readme       = "README.md"
license      = { file = "LICENSE" }
requires-python = ">=3.10"
authors = [
    { name = "Alice Smith", email = "alice@example.com" },
]
```

```
keywords = ["python", "utility", "example"]
classifiers = [
    "Development Status :: 3 - Alpha",
    "Intended Audience :: Developers",
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.10",
    "Programming Language :: Python :: 3.11",
    "Programming Language :: Python :: 3.12",
]

# Runtime dependencies --- what your package needs to work
dependencies = [
    "requests>=2.28",
    "pydantic>=2.0",
]


[project.optional-dependencies]
# pip install my-awesome-package[dev]
dev = [
    "pytest>=7",
    "pytest-cov",
    "mypy",
    "ruff",
]
# pip install my-awesome-package[docs]
docs = [
    "mkdocs",
    "mkdocs-material",
]


[project.urls]
Homepage      = "https://github.com/alice/my-awesome-package"
Documentation = "https://my-awesome-package.readthedocs.io"
Repository    = "https://github.com/alice/my-awesome-package"
"Bug Tracker" = "https://github.com/alice/my-awesome-package/issues"


# Command-line scripts --- creates `my-tool` command after pip install
[project.scripts]
my-tool = "my_package.cli:main"
```

```toml
# Tool configuration lives here too --- one file for everything
[tool.pytest.ini_options]
testpaths  = ["tests"]
addopts    = "--cov=src --cov-report=term-missing"

[tool.mypy]
python_version        = "3.12"
strict                = true
ignore_missing_imports = true

[tool.ruff]
line-length = 88
target-version = "py310"
select  = ["E", "F", "I", "N", "W"]
ignore  = ["E501"]

[tool.coverage.run]
source = ["src"]
omit   = ["tests/*"]
```

34.3.1 Choosing a build backend

Backend	Install	Best for
`hatchling`	`pip install hatch`	Most projects — simple, modern
`setuptools`	Pre-installed	Legacy projects, C extensions
`flit`	`pip install flit`	Pure Python, minimal config
`poetry`	Separate install	Integrated dependency management

For most new projects, use `hatchling`. It's fast, simple, and well-maintained.

34.4 Building a Real Package

Let's build a package called `texttools` — a collection of text processing utilities.

34.4.1 Directory structure

```
texttools/
├── src/
│   └── texttools/
```

```
│       ├── __init__.py
│       ├── analysis.py
│       ├── transform.py
│       └── cli.py
├── tests/
│   ├── test_analysis.py
│   └── test_transform.py
├── pyproject.toml
├── README.md
└── LICENSE
```

34.4.2 The package code

```
# src/texttools/__init__.py
"""
texttools --- A collection of text processing utilities.
"""
from .analysis  import word_count, char_frequency, readability_score
from .transform import clean, titlecase, truncate, slug

__version__ = "1.0.0"
__all__     = [
    "word_count", "char_frequency", "readability_score",
    "clean", "titlecase", "truncate", "slug",
]
```

```
# src/texttools/analysis.py
"""Text analysis utilities."""
import re
from collections import Counter
from typing import Counter as CounterType


def word_count(text: str) -> int:
    """Return the number of words in text."""
    return len(text.split())


def char_frequency(text: str, ignore_case: bool = True) -> CounterType[str]:
    """Return a Counter of character frequencies, excluding whitespace."""
    if ignore_case:
        text = text.lower()
    return Counter(c for c in text if not c.isspace())
```

```
def readability_score(text: str) -> float:
    """
    Flesch Reading Ease score.
    90-100: Very easy. 0-30: Very difficult.
    """
    sentences = max(1, len(re.findall(r"[.!?]+", text)))
    words     = max(1, word_count(text))
    syllables = max(1, sum(_count_syllables(w) for w in text.split()))

    return 206.835 - 1.015 * (words / sentences) - 84.6 * (syllables / words)


def _count_syllables(word: str) -> int:
    """Rough syllable counter."""
    word    = word.lower().strip(".,!?;:")
    vowels  = "aeiou"
    count   = 0
    prev_v  = False
    for ch in word:
        is_v = ch in vowels
        if is_v and not prev_v:
            count += 1
        prev_v = is_v
    return max(1, count)


# src/texttools/transform.py
"""Text transformation utilities."""
import re
import unicodedata


def clean(text: str) -> str:
    """Strip leading/trailing whitespace and normalize internal spaces."""
    return " ".join(text.split())


def titlecase(text: str) -> str:
    """Title case, but don't capitalize short words like 'a', 'the', 'of'."""
    small = {"a", "an", "and", "as", "at", "but", "by", "for",
             "if", "in", "nor", "of", "on", "or", "so", "the",
             "to", "up", "yet"}
    words  = text.lower().split()
    result = []
    for i, word in enumerate(words):
```

```
        if i == 0 or word not in small:
            result.append(word.capitalize())
        else:
            result.append(word)
    return " ".join(result)


def truncate(text: str, max_length: int, suffix: str = "...") -> str:
    """Truncate text to max_length characters, appending suffix if truncated."""
    if len(text) <= max_length:
        return text
    return text[: max_length - len(suffix)].rstrip() + suffix


def slug(text: str) -> str:
    """Convert text to a URL-friendly slug."""
    text = unicodedata.normalize("NFKD", text)
    text = text.encode("ascii", "ignore").decode("ascii")
    text = text.lower()
    text = re.sub(r"[^\w\s-]", "", text)
    text = re.sub(r"[\s_-]+", "-", text)
    return text.strip("-")


# src/texttools/cli.py
"""Command-line interface for texttools."""
import sys
import argparse
from . import word_count, readability_score, slug, truncate


def main() -> None:
    parser = argparse.ArgumentParser(
        prog        = "texttools",
        description = "Text processing utilities",
    )
    subparsers = parser.add_subparsers(dest="command", required=True)

    # texttools count <text>
    count_p = subparsers.add_parser("count", help="Count words in text")
    count_p.add_argument("text", nargs="?", help="Text to analyze (or stdin)")

    # texttools score <text>
    score_p = subparsers.add_parser("score", help="Flesch readability score")
    score_p.add_argument("text", nargs="?")
```

```
    # texttools slug <text>
    slug_p = subparsers.add_parser("slug", help="Convert to URL slug")
    slug_p.add_argument("text", nargs="?")

    # texttools truncate <text> --length 50
    trunc_p = subparsers.add_parser("truncate", help="Truncate text")
    trunc_p.add_argument("text",   nargs="?")
    trunc_p.add_argument("--length", type=int, default=80)
    trunc_p.add_argument("--suffix", default="...")

    args = parser.parse_args()

    text = args.text or sys.stdin.read().strip()
    if not text:
        parser.error("Provide text as an argument or via stdin")

    if args.command == "count":
        print(word_count(text))
    elif args.command == "score":
        score = readability_score(text)
        print(f"{score:.1f}")
    elif args.command == "slug":
        print(slug(text))
    elif args.command == "truncate":
        print(truncate(text, args.length, args.suffix))


if __name__ == "__main__":
    main()
```

34.4.3 `pyproject.toml` for texttools

```
[build-system]
requires      = ["hatchling"]
build-backend = "hatchling.build"

[project]
name            = "texttools"
version         = "1.0.0"
description     = "A collection of text processing utilities"
readme          = "README.md"
requires-python = ">=3.10"
license         = { text = "MIT" }
```

```
authors         = [{ name = "Alice Smith", email = "alice@example.com" }]
keywords        = ["text", "nlp", "utilities"]
classifiers     = [
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Topic :: Text Processing",
]
dependencies    = []   # no external dependencies

[project.optional-dependencies]
dev = ["pytest", "pytest-cov", "mypy", "ruff"]

[project.scripts]
texttools = "texttools.cli:main"

[project.urls]
Repository = "https://github.com/alice/texttools"

[tool.hatch.build.targets.wheel]
packages = ["src/texttools"]

[tool.pytest.ini_options]
testpaths = ["tests"]
addopts   = "-v --cov=src/texttools --cov-report=term-missing"
```

34.5 Installing in Development Mode

While developing, install the package in **editable mode** — changes to source files are reflected immediately without reinstalling:

```
# From the project root
pip install -e .

# With dev dependencies
pip install -e ".[dev]"
```

After this, you can run:

```
python -c "import texttools; print(texttools.word_count('hello world'))"
# 2

texttools count "The quick brown fox jumps over the lazy dog"
# 9
```

```
texttools slug "Hello World! This is a URL slug."
# hello-world-this-is-a-url-slug

texttools score "Python is easy to learn. It has simple syntax. Beginners love it.
# 74.2
```

34.6 Writing a Good README

A README is the first thing people see. It should answer five questions in order:

```
# texttools

A collection of text processing utilities for Python.

## Installation

```bash
pip install texttools
```

34.7 Quick Start

```
import texttools

print(texttools.word_count("Hello world"))        # 2
print(texttools.slug("Hello World!"))             # hello-world
print(texttools.truncate("Long text...", 10))     # Long te...
```

34.8 CLI

```
echo "Hello World" | texttools slug
# hello-world

texttools score "Python is a great language for beginners."
# 68.4
```

34.9 API Reference

34.9.1 `word_count(text: str) -> int`

Count the number of words in `text`.
```

34.9.2 `slug(text: str) -> str`

Convert `text` to a URL-friendly slug.

34.10 License

MIT

```
Keep it short. Show the install command. Show a working example. Link to full docs.

---

## Versioning --- Semantic Versioning

Use **SemVer**: `MAJOR.MINOR.PATCH`

- **PATCH** (`1.0.0` -> `1.0.1`) --- bug fix, no API change
- **MINOR** (`1.0.0` -> `1.1.0`) --- new feature, backwards compatible
- **MAJOR** (`1.0.0` -> `2.0.0`) --- breaking change

```toml
# pyproject.toml
[project]
version = "1.2.3"
```

Some projects automate versioning from git tags:

```toml
[build-system]
requires = ["hatchling", "hatch-vcs"]
build-backend = "hatchling.build"

[tool.hatch.version]
source = "vcs"   # version comes from git tags

# Then tag a release:
# git tag v1.2.3
# git push --tags
```

34.11 Building Distribution Files

```
pip install build
python -m build
```
```

This creates two files in `dist/`:

```
dist/
├── texttools-1.0.0-py3-none-any.whl   <- wheel (binary distribution)
└── texttools-1.0.0.tar.gz             <- sdist (source distribution)
```

- **Wheel** (`.whl`) — the preferred format. pip installs it directly without building.
- **Source distribution** (`.tar.gz`) — a tarball of the source. Used as a fallback, or when the wheel isn't available for a platform.

34.12 Publishing to PyPI

TestPyPI — a sandbox for practicing without polluting the real index:

```
pip install twine

# Upload to TestPyPI first
twine upload --repository testpypi dist/*

# Install from TestPyPI to verify
pip install --index-url https://test.pypi.org/simple/ texttools
```

Once you're confident, upload to the real PyPI:

```
twine upload dist/*
```

You need a PyPI account. Create one at pypi.org and use an API token (not your password):

```
# Configure once in ~/.pypirc
[pypi]
username = __token__
password = pypi-AgEIcH...your-token-here...
```

Or set environment variables:

```
export TWINE_USERNAME=__token__
export TWINE_PASSWORD=pypi-AgEIcH...
twine upload dist/*
```

After publishing, anyone can install your package:

```
pip install texttools
```

34.13 Automating Releases with GitHub Actions

A CI/CD pipeline that tests your code, builds the package, and publishes to PyPI automatically when you push a tag:

```yaml
# .github/workflows/publish.yml
name: Publish to PyPI

on:
  push:
    tags:
      - "v*"   # trigger on any tag like v1.0.0

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.10", "3.11", "3.12"]

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}
      - run: pip install -e ".[dev]"
      - run: pytest

  publish:
    needs: test   # only publish if tests pass
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install build twine
      - run: python -m build
      - run: twine upload dist/*
        env:
          TWINE_USERNAME: __token__
          TWINE_PASSWORD: ${{ secrets.PYPI_API_TOKEN }}
```

Add your PyPI token to GitHub repository secrets as `PYPI_API_TOKEN`. Now every `git tag v1.2.3 && git push --tags` runs your tests and publishes automatically.

34.14 Code Quality Tools

A modern Python project uses these tools, all configured in `pyproject.toml`:

34.14.1 `ruff` — fast linter + formatter

```
pip install ruff
```

```
ruff check .        # lint
ruff check . --fix  # auto-fix what's possible
ruff format .       # format (like Black)
```

```
[tool.ruff]
line-length    = 88
target-version = "py310"

[tool.ruff.lint]
select = [
    "E",   # pycodestyle errors
    "F",   # pyflakes
    "I",   # isort (import sorting)
    "N",   # pep8-naming
    "UP",  # pyupgrade (modernize syntax)
]
```

34.14.2 `mypy` — static type checking

```
mypy src/
```

```
[tool.mypy]
python_version         = "3.12"
strict                 = true
ignore_missing_imports = true
```

34.14.3 `pre-commit` — run checks before every commit

```
pip install pre-commit
```

```
# .pre-commit-config.yaml
repos:
```

```
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.3.0
    hooks:
      - id: ruff
        args: [--fix]
      - id: ruff-format

  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.9.0
    hooks:
      - id: mypy
        additional_dependencies: [pydantic]
```

```
pre-commit install          # hook into git
pre-commit run --all-files  # run manually
```

Now `git commit` automatically runs ruff and mypy. Bad code can't accidentally be committed.

34.15 Dependency Management Best Practices

34.15.1 Pinning dependencies

Your `pyproject.toml` lists **minimum versions** for dependencies (`requests>=2.28`). This keeps the package flexible for users.

For **applications** (not libraries), pin exact versions to ensure reproducibility:

```
pip freeze > requirements.txt
# or
pip-compile pyproject.toml -o requirements.txt
```

34.15.2 `pip-tools` for reproducible installs

```
pip install pip-tools

# Generate pinned requirements from pyproject.toml
pip-compile pyproject.toml -o requirements.txt
pip-compile --extra dev -o requirements-dev.txt

# Install exactly these versions
pip-sync requirements.txt requirements-dev.txt
```

34.15.3 Using `uv` — the modern, fast alternative

`uv` is a new package manager written in Rust — 10-100x faster than pip:

```
pip install uv

uv venv                             # create virtual environment
uv pip install -e ".[dev]"          # install
uv pip compile pyproject.toml       # generate lockfile
uv run pytest                       # run in the venv
```

34.16 Complete Workflow Summary

Here is the complete workflow for a new Python package from zero to published:

```
# 1. Create the project
mkdir texttools && cd texttools
python -m venv .venv
source .venv/bin/activate   # Windows: .venv\Scripts\activate

# 2. Create structure
mkdir -p src/texttools tests
touch src/texttools/__init__.py
touch src/texttools/core.py
touch tests/test_core.py
touch pyproject.toml README.md LICENSE

# 3. Write pyproject.toml (see above)

# 4. Install in editable mode
pip install -e ".[dev]"

# 5. Write code and tests
# ... (the fun part)

# 6. Run tests and quality checks
pytest
mypy src/
ruff check .

# 7. Commit and tag
git init && git add .
```

```
git commit -m "Initial release"
git tag v1.0.0

# 8. Build
python -m build

# 9. Publish
twine upload --repository testpypi dist/*   # test first
twine upload dist/*                          # then real PyPI
```

34.17 What You Learned in This Chapter

- A Python package is a directory with `__init__.py` described by `pyproject.toml`.
- Use the `src/` layout to prevent importing local source during testing.
- `pyproject.toml` contains package metadata, dependencies, optional dependency groups, entry points (CLI commands), and tool configuration.
- Choose `hatchling` as your build backend for new projects.
- `pip install -e .` installs in editable mode — source changes are reflected immediately.
- Semantic versioning: `MAJOR.MINOR.PATCH`. Automate from git tags with `hatch-vcs`.
- `python -m build` creates a wheel and source distribution in `dist/`.
- `twine upload dist/*` publishes to PyPI. Use TestPyPI first.
- GitHub Actions automates testing and publishing on every git tag.
- `ruff` lints and formats. `mypy` type-checks. `pre-commit` enforces both before every commit.
- Pin exact versions in applications (`pip freeze`), use ranges in libraries (`requests>=2.28`).
- `uv` is the modern, fast alternative to pip.

34.18 What's Next?

Chapter 35 covers **Professional Python Practices** — code style, documentation with docstrings and Sphinx, `git` workflow, code review, and the habits that separate professional Python developers from hobbyists. This is the final chapter on professional practices before the advanced topics in Part 10.

Chapter 35

Chapter 35: Professional Python Practices

There is a gap between code that works and code that is professional. Both produce correct output. But professional code is readable by others, maintainable six months later, testable, consistently styled, and documented clearly. This chapter closes that gap.

These aren't rules for their own sake. Every practice here exists because it makes real projects easier to work on — alone or in a team.

35.1 Code Style — PEP 8 and Why It Matters

PEP 8 is Python's official style guide. The reason for following it isn't aesthetics — it's that consistent style lets your brain stop parsing formatting and focus on logic.

The key rules:

```
# ── Naming ──────────────────────────────────────────────────────────────

# Variables and functions: snake_case
user_name = "Alice"
total_price = 99.99

def calculate_tax(price, rate):
    return price * rate

# Classes: PascalCase
```

```python
class BankAccount:
    pass

# Constants: UPPER_SNAKE_CASE
MAX_RETRIES = 3
DEFAULT_TIMEOUT = 30.0

# Private (by convention): leading underscore
_internal_helper = "not for public use"
__name_mangled   = "class-private"

# ── Spacing ──────────────────────────────────────────────────────────

# Two blank lines between top-level definitions
def first_function():
    pass


def second_function():
    pass


class MyClass:
    # One blank line between methods
    def method_one(self):
        pass

    def method_two(self):
        pass

# Spaces around operators
x = 1 + 2
y = x * 3

# No spaces around = for default arguments
def connect(host, port=8080, timeout=30):
    pass

# Spaces after commas
items = [1, 2, 3]
result = function(a, b, c)

# ── Line length ──────────────────────────────────────────────────────
```

```python
# Keep lines under 88 characters (Black's default, slightly more than PEP 8's 79)
# Break long lines with implicit continuation inside brackets
result = some_function(
    argument_one,
    argument_two,
    argument_three,
)

# Or with backslash (less preferred)
total = (first_value
         + second_value
         + third_value)

# ── Imports ──────────────────────────────────────────────────────────

# Standard library first, then third-party, then local --- blank line between grou
import os
import sys
from pathlib import Path

import requests
from pydantic import BaseModel

from mypackage import utils
from mypackage.models import User
```

35.1.1 Let tools enforce style — don't do it manually

Don't spend mental energy on formatting. Let `ruff format` (or `black`) handle it automatically:

```
ruff format .           # format all files
ruff format --check .   # check without changing (for CI)
```

Configure your editor to format on save. After that, style stops being a decision you make.

35.2 Writing Good Docstrings

Docstrings are the official documentation for your code. They're accessible via `help()`, shown in editors as you type, and used by documentation generators.

35.2.1 The Google style (most common in modern Python)

```
def calculate_shipping(
    weight_kg: float,
    distance_km: float,
    express: bool = False,
) -> float:
    """
    Calculate shipping cost based on weight, distance, and service level.

    Args:
        weight_kg:   Weight of the package in kilograms.
        distance_km: Shipping distance in kilometers.
        express:     If True, apply the express multiplier (2x).

    Returns:
        Shipping cost in USD, rounded to two decimal places.

    Raises:
        ValueError: If weight_kg or distance_km is negative.

    Examples:
        >>> calculate_shipping(2.5, 100)
        12.5
        >>> calculate_shipping(2.5, 100, express=True)
        25.0
    """
    if weight_kg < 0 or distance_km < 0:
        raise ValueError("Weight and distance must be non-negative.")
    base_cost = weight_kg * 0.05 * distance_km
    if express:
        base_cost *= 2
    return round(base_cost, 2)
```

35.2.2 Class docstrings

```
class PaymentProcessor:
    """
    Process payments through multiple payment gateways.

    Supports Stripe, PayPal, and Square. Automatically retries
    on transient failures and logs all transactions.
```

```
    Attributes:
        gateway:  The active payment gateway name.
        currency: ISO 4217 currency code (default: "USD").

    Examples:
        >>> processor = PaymentProcessor("stripe", currency="EUR")
        >>> processor.charge(amount=99.99, card_token="tok_visa")
        Transaction(id="ch_abc", status="succeeded", amount=99.99)
    """

    def __init__(self, gateway: str, currency: str = "USD") -> None:
        self.gateway  = gateway
        self.currency = currency
```

35.2.3 Module docstrings

```
"""
payments.py --- Payment processing utilities.

This module provides a unified interface for multiple payment gateways.
All monetary values are in the smallest currency unit (cents for USD).

Typical usage:
    from payments import PaymentProcessor
    processor = PaymentProcessor("stripe")
    result = processor.charge(amount=9999, card_token=token)
"""
```

35.2.4 `doctest` — runnable examples in docstrings

The `Examples:` section above isn't just documentation — it can run as a test:

```
def add(a: int, b: int) -> int:
    """
    Add two integers.

    >>> add(2, 3)
    5
    >>> add(-1, 1)
    0
    >>> add(0, 0)
    0
    """
    return a + b
```

```
python -m doctest my_module.py -v    # run all docstring examples as tests
pytest --doctest-modules             # run doctests with pytest
```

Doctests serve as both documentation and light test coverage. Use them for simple, illustrative examples.

35.3 The Zen of Python Applied

```
import this
```

The Zen of Python has 19 aphorisms. Here are the seven that matter most in practice:

"Beautiful is better than ugly." Code is read far more than it's written. Spend time making it readable.

"Explicit is better than implicit."

```
# Implicit --- what does True mean here?
connect(host, True)

# Explicit
connect(host, secure=True)
```

"Simple is better than complex."

```
# Complex
def get_items(data):
    return list(filter(lambda x: x is not None,
                       map(lambda x: x.get("value"), data)))

# Simple
def get_items(data):
    return [item["value"] for item in data if item.get("value") is not None]
```

"Flat is better than nested."

```
# Nested --- hard to follow
def process(data):
    if data:
        if isinstance(data, list):
            if len(data) > 0:
                return data[0]

# Flat --- early return pattern
def process(data):
    if not data:
```

```
        return None
    if not isinstance(data, list):
        return None
    return data[0] if data else None
```

"Errors should never pass silently."

```
# Bad --- swallows all exceptions
try:
    result = risky()
except:
    pass

# Good --- catch specific exceptions, log them
try:
    result = risky()
except ValueError as e:
    logger.warning(f"Invalid value: {e}")
    result = default_value
```

"If the implementation is hard to explain, it's a bad idea." If you struggle to explain what your code does, refactor until you can.

"There should be one obvious way to do it." When choosing between several approaches, the simplest one that clearly expresses intent is usually right.

35.4 Code Review Practices

Code review is how teams catch bugs, share knowledge, and maintain standards. Whether you're reviewing others' code or preparing your own for review, these habits matter.

35.4.1 Writing reviewable code

- **Keep pull requests small.** A PR with 50 lines is reviewed carefully. One with 500 lines is rubber-stamped.
- **One logical change per PR.** Mixing a refactor with a bug fix with a new feature makes review impossible.
- **Write a clear PR description.** Explain *why*, not just *what*. Link to the issue. Show before/after if the behavior changed.
- **Test everything.** Reviewers shouldn't have to run your code to verify correctness.

```
## What this does
Adds retry logic to the payment processor for transient 5xx errors.
```

```
## Why
We're seeing intermittent 503s from Stripe during peak hours.
Retrying with exponential backoff reduces payment failures by ~40%.

## Changes
- `payments.py`: Added `@retry(times=3, delay=1.0)` to `charge()`
- `test_payments.py`: Added tests for retry behavior with mocked failures

## Notes
The retry count (3) and initial delay (1.0s) are configurable via env vars.
```

35.4.2 Reviewing others' code

Ask questions rather than making demands:

```
# Less collaborative
"This is wrong. Use a dict instead."

# More collaborative
"Would a dict here give us O(1) lookup? The current list search
 might be slow if users_to_check grows large."
```

Look for: - Correctness (does it do what the description says?) - Edge cases (what happens with empty input? None? Very large input?) - Tests (are they testing behavior, not implementation?) - Readability (can you follow the logic without reading the commit message?) - Security (any SQL injection risk? Hardcoded credentials? Unvalidated user input?)

35.5 Git Workflow for Python Projects

35.5.1 `.gitignore`

```
# Python
__pycache__/
*.py[cod]
*.pyo
*.pyd
.Python
*.egg-info/
dist/
build/
*.egg
```

```
# Virtual environments
.venv/
venv/
env/

# Testing
.pytest_cache/
.coverage
htmlcov/
.mypy_cache/
.ruff_cache/

# IDE
.vscode/settings.json
.idea/
*.swp

# Secrets --- NEVER commit these
.env
*.pem
*.key
secrets.json
```

35.5.2 Commit message convention

Follow the **Conventional Commits** format — makes changelogs automatic and history readable:

```
<type>(<scope>): <short description>

[optional body]

[optional footer]
```

Types: `feat`, `fix`, `docs`, `style`, `refactor`, `test`, `chore`, `perf`, `ci`

```
feat(auth): add JWT token refresh

Tokens now auto-refresh 5 minutes before expiry.
Fixes intermittent 401 errors for long-running sessions.

Closes #142

---
```

```
fix(payments): retry on transient 503 errors

---

docs(readme): update installation instructions for Python 3.12

---

test(auth): add tests for token expiry edge cases

---

refactor(database): replace raw SQL with SQLAlchemy ORM
```

35.5.3 Branch strategy

```
main          <- production-ready, tagged releases
└── develop   <- integration branch (optional)
    ├── feature/add-payment-retry
    ├── fix/auth-token-expiry
    └── chore/update-dependencies
```

Simple projects can skip `develop` and branch directly off `main`.

```
# Start a feature
git checkout -b feature/add-payment-retry

# Work, commit often
git add -p                                  # stage interactively --- review each c
git commit -m "feat(payments): add retry logic for 503 errors"

# Keep branch up to date
git fetch origin
git rebase origin/main

# Push and open a PR
git push -u origin feature/add-payment-retry
```

Use `git add -p` (patch mode) to review each change before staging. It forces you to think about what you're committing.

35.6 Environment Variables and Configuration

Never hardcode secrets, URLs, or environment-specific values in your code. Read them from environment variables:

```
# config.py
import os
from pathlib import Path


class Config:
    # Database
    DATABASE_URL: str     = os.environ.get("DATABASE_URL", "sqlite:///dev.db")
    DATABASE_POOL_SIZE: int = int(os.environ.get("DATABASE_POOL_SIZE", "5"))

    # Auth
    SECRET_KEY: str       = os.environ.get("SECRET_KEY", "")
    TOKEN_EXPIRE_MIN: int= int(os.environ.get("TOKEN_EXPIRE_MIN", "30"))

    # External APIs
    STRIPE_API_KEY: str  = os.environ.get("STRIPE_API_KEY", "")
    SENDGRID_API_KEY: str= os.environ.get("SENDGRID_API_KEY", "")

    # App
    DEBUG: bool           = os.environ.get("DEBUG", "false").lower() == "true"
    LOG_LEVEL: str        = os.environ.get("LOG_LEVEL", "INFO")

    def validate(self) -> None:
        """Raise on startup if required secrets are missing."""
        required = {"SECRET_KEY", "STRIPE_API_KEY"}
        missing  = [k for k in required if not getattr(self, k)]
        if missing:
            raise RuntimeError(
                f"Missing required environment variables: {', '.join(missing)}"
            )


config = Config()
```

Use a `.env` file in development:

```
# .env --- never commit this file
DATABASE_URL=postgresql://user:password@localhost/myapp
SECRET_KEY=dev-secret-key-change-in-production
```

```
STRIPE_API_KEY=sk_test_...
DEBUG=true
LOG_LEVEL=DEBUG
```

Load it with `python-dotenv`:

```
from dotenv import load_dotenv
load_dotenv()    # reads .env into os.environ

from config import config
config.validate()
```

Or with `pydantic-settings` (cleaner, with type coercion):

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_file=".env", env_file_encoding="utf-8

    database_url:     str   = "sqlite:///dev.db"
    secret_key:       str   = ""
    stripe_api_key:   str   = ""
    debug:            bool  = False
    log_level:        str   = "INFO"
    token_expire_min: int   = 30

settings = Settings()
```

35.7 Logging in Production

Good production logging is structured, levelled, and goes to the right place.

```
import logging
import sys
from pathlib import Path


def setup_logging(level: str = "INFO", log_file: Path | None = None) -> None:
    """Configure structured logging for production."""
    fmt = "%(asctime)s %(levelname)-8s %(name)-30s %(message)s"

    handlers: list[logging.Handler] = [
        logging.StreamHandler(sys.stdout),
    ]
    if log_file:
```

```
        handlers.append(
            logging.handlers.RotatingFileHandler(
                log_file,
                maxBytes    = 10 * 1024 * 1024,  # 10MB
                backupCount = 5,
            )
        )

    logging.basicConfig(
        level    = getattr(logging, level.upper()),
        format   = fmt,
        datefmt  = "%Y-%m-%d %H:%M:%S",
        handlers = handlers,
        force    = True,
    )

    # Silence noisy third-party loggers
    logging.getLogger("urllib3").setLevel(logging.WARNING)
    logging.getLogger("sqlalchemy.engine").setLevel(logging.WARNING)


# Get a logger per module --- never use the root logger directly
log = logging.getLogger(__name__)

# Use structured context with extra
log.info("User logged in", extra={"user_id": 42, "ip": "192.168.1.1"})
log.error("Payment failed", extra={"order_id": 99, "amount": 49.99})
```

For high-volume production logging, consider `structlog` for JSON-formatted structured logs that work well with log aggregators like Datadog, Splunk, or Elasticsearch.

35.8 Writing Clean Code — Practical Rules

35.8.1 Functions should do one thing

```
# Bad --- three responsibilities in one function
def process_user(user_id):
    user = db.query(f"SELECT * FROM users WHERE id = {user_id}")  # SQL injection!
    if user:
        user["email"] = user["email"].lower()
        send_email(user["email"], "Welcome!")
        db.query(f"UPDATE users SET processed=1 WHERE id = {user_id}")
```

```python
# Good --- one responsibility each
def get_user(db, user_id: int) -> dict | None:
    return db.execute("SELECT * FROM users WHERE id = ?", (user_id,)).fetchone(

def normalize_user(user: dict) -> dict:
    return {**user, "email": user["email"].lower()}

def send_welcome_email(email: str) -> None:
    send_email(email, "Welcome!")

def mark_user_processed(db, user_id: int) -> None:
    db.execute("UPDATE users SET processed=1 WHERE id = ?", (user_id,))
    db.commit()
```

35.8.2 Avoid magic numbers

```python
# Bad --- what are these numbers?
if score > 0.85:
    tier = "gold"
elif score > 0.65:
    tier = "silver"
else:
    tier = "bronze"

# Good --- named constants
GOLD_THRESHOLD   = 0.85
SILVER_THRESHOLD = 0.65

if score > GOLD_THRESHOLD:
    tier = "gold"
elif score > SILVER_THRESHOLD:
    tier = "silver"
else:
    tier = "bronze"
```

35.8.3 Return early, don't nest deeply

```python
# Bad --- deep nesting
def process_order(order):
    if order is not None:
        if order["status"] == "pending":
            if order["total"] > 0:
                if has_inventory(order):
```

```
                return fulfill(order)

# Good --- guard clauses
def process_order(order):
    if order is None:
        raise ValueError("Order cannot be None")
    if order["status"] != "pending":
        raise ValueError(f"Cannot process order in status {order['status']!r}")
    if order["total"] <= 0:
        raise ValueError("Order total must be positive")
    if not has_inventory(order):
        raise InsufficientInventoryError(order["id"])
    return fulfill(order)
```

35.8.4 Name things clearly

```
# Bad names
def f(d, n):
    r = []
    for i in d:
        if i > n:
            r.append(i)
    return r

# Good names
def filter_above_threshold(values: list[float], threshold: float) -> list[float]:
    return [v for v in values if v > threshold]
```

The function with good names doesn't need a comment. The code IS the documentation.

35.8.5 Avoid comments that explain what — write code that shows what

```
# Bad --- comment restates the code
# Increment counter by 1
counter += 1

# Bad --- comment explains code that should be clearer
# Check if user is an adult
if user.age >= 18:

# Good --- comment explains WHY, not WHAT
# Stripe requires amounts in the smallest currency unit (cents)
amount_cents = int(amount_dollars * 100)
```

```
# RFC 3339 format required by the external API
timestamp = datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ")
```

35.9 The Habits of Professional Python Developers

Beyond specific practices, professionals have habits:

They read error messages fully. Every line, not just the last one.

They write tests as they write code. Not after. The test is how you know you're done.

They commit small and often. A commit is a checkpoint. Frequent checkpoints mean less work lost.

They know their tools. Shell shortcuts, editor keybindings, debugger commands. Time saved per day adds up to weeks per year.

They ask "what could go wrong?" Every function that accepts input — what if it's None? Empty? Malformed? Extremely large?

They document decisions, not mechanics. "We use Redis here because PostgreSQL locks caused 200ms latency spikes under this query pattern" is valuable. "# connect to database" is noise.

They favor boring solutions. The newest, most clever approach is usually not the right one. Battle-tested libraries, simple data structures, and readable code win over novelty.

They read others' code. The Python standard library is beautifully written. So is `requests`, `flask`, `pydantic`. Reading excellent code is the fastest way to improve your own.

They refactor without fear. Because they have tests. Without tests, every refactor is a gamble. With tests, it's just work.

35.10 Project: A Professional Python Module

Let's put everything together in one module that demonstrates every practice from this chapter:

```
"""
ratelimiter.py --- Thread-safe rate limiter with configurable strategies.
```

```
Provides a token bucket rate limiter for controlling how frequently
a resource can be accessed. Thread-safe for concurrent use.

Typical usage::

    limiter = RateLimiter(rate=10, per=1.0)  # 10 requests per second

    @limiter.limit
    def call_api(endpoint):
        return requests.get(endpoint)

    # Or as a context manager
    with limiter:
        call_api("/users")
"""
from __future__ import annotations

import logging
import threading
import time
from dataclasses import dataclass, field
from functools import wraps
from typing import Callable, TypeVar

log = logging.getLogger(__name__)

F = TypeVar("F", bound=Callable)

# The minimum wait time we bother sleeping for (below this, just spin)
_MIN_SLEEP_SECONDS = 0.001


@dataclass
class RateLimiterStats:
    """Runtime statistics for a RateLimiter instance."""
    total_calls:    int   = field(default=0)
    total_waited:   float = field(default=0.0)
    total_rejected: int   = field(default=0)

    @property
    def avg_wait_ms(self) -> float:
        """Average wait time per call in milliseconds."""
        if self.total_calls == 0:
```

```
            return 0.0
        return (self.total_waited / self.total_calls) * 1000


class RateLimitExceeded(Exception):
    """Raised when a non-blocking rate limiter cannot acquire a token."""


class RateLimiter:
    """
    Token bucket rate limiter --- thread-safe.

    Allows up to `rate` operations per `per` seconds. When the bucket
    is empty, callers either wait (blocking mode) or raise
    `RateLimitExceeded` (non-blocking mode).

    Args:
        rate:     Maximum number of operations allowed per interval.
        per:      Interval in seconds (default: 1.0 second).
        blocking: If True (default), callers wait when the bucket is empty.
                  If False, raises RateLimitExceeded immediately.

    Examples:
        >>> limiter = RateLimiter(rate=5, per=1.0)
        >>> with limiter:
        ...     print("allowed")
        allowed

        >>> @limiter.limit
        ... def my_function():
        ...     return "result"
        >>> my_function()
        'result'
    """

    def __init__(
        self,
        rate:     float,
        per:      float = 1.0,
        blocking: bool  = True,
    ) -> None:
        if rate <= 0:
            raise ValueError(f"rate must be positive, got {rate}")
        if per <= 0:
```

```
            raise ValueError(f"per must be positive, got {per}")

        self._rate     = rate
        self._per      = per
        self._blocking = blocking
        self._tokens   = rate           # start full
        self._last     = time.monotonic()
        self._lock     = threading.Lock()
        self._stats    = RateLimiterStats()

        log.debug(
            "RateLimiter created: rate=%.1f per=%.1fs blocking=%s",
            rate, per, blocking,
        )

    # ── Public interface ──────────────────────────────────────────────

    def acquire(self) -> float:
        """
        Acquire one token, waiting if necessary (blocking mode).

        Returns:
            The number of seconds waited (0.0 if no wait was needed).

        Raises:
            RateLimitExceeded: If non-blocking and no token is available.
        """
        with self._lock:
            return self._acquire_locked()

    def limit(self, func: F) -> F:
        """Decorator: apply rate limiting to a function."""
        @wraps(func)
        def wrapper(*args, **kwargs):
            self.acquire()
            return func(*args, **kwargs)
        return wrapper  # type: ignore

    @property
    def stats(self) -> RateLimiterStats:
        """Return a snapshot of runtime statistics."""
        return self._stats

    def reset(self) -> None:
```

```
    """Reset the bucket to full capacity and clear statistics."""
    with self._lock:
        self._tokens = self._rate
        self._last   = time.monotonic()
        self._stats  = RateLimiterStats()
    log.debug("RateLimiter reset")

# ── Context manager ──────────────────────────────────────────────────────

def __enter__(self) -> "RateLimiter":
    self.acquire()
    return self

def __exit__(self, *args: object) -> None:
    pass

# ── Internal helpers ─────────────────────────────────────────────────────

def _acquire_locked(self) -> float:
    """Must be called with self._lock held."""
    now     = time.monotonic()
    elapsed = now - self._last
    self._tokens = min(self._rate, self._tokens + elapsed * (self._rate / s
    self._last   = now

    if self._tokens >= 1:
        self._tokens -= 1
        self._stats.total_calls += 1
        return 0.0

    if not self._blocking:
        self._stats.total_rejected += 1
        raise RateLimitExceeded(
            f"Rate limit exceeded: {self._rate:.0f} per {self._per:.1f}s"
        )

    # Calculate wait time and sleep outside the lock
    wait = (1 - self._tokens) * (self._per / self._rate)
    self._lock.release()
    try:
        if wait > _MIN_SLEEP_SECONDS:
            time.sleep(wait)
    finally:
        self._lock.acquire()
```

```
            # Now take the token
            self._tokens = max(0.0, self._tokens - 1)
            self._stats.total_calls  += 1
            self._stats.total_waited += wait
            log.debug("Rate limiter waited %.3fs", wait)
            return wait

        def __repr__(self) -> str:
            return (
                f"RateLimiter(rate={self._rate}, per={self._per}, "
                f"blocking={self._blocking})"
            )


    # ── Demo ──────────────────────────────────────────────────────────

    if __name__ == "__main__":
        logging.basicConfig(level=logging.DEBUG, format="%(levelname)s %(message)s")

        limiter = RateLimiter(rate=3, per=1.0)

        @limiter.limit
        def simulate_api_call(n: int) -> str:
            return f"Response #{n}"

        print("Making 9 API calls at 3/second...")
        start = time.perf_counter()
        for i in range(9):
            result = simulate_api_call(i + 1)
            print(f"  {result} at {time.perf_counter() - start:.2f}s")

        s = limiter.stats
        print(f"\nStats: {s.total_calls} calls, "
              f"avg wait {s.avg_wait_ms:.1f}ms, "
              f"{s.total_rejected} rejected")
```

This module demonstrates every professional practice from this chapter: - Google-style docstrings on the module, class, and every public method - Named constants (`_MIN_SLEEP_SECONDS`) - Single-responsibility methods - Guard clauses with clear error messages - Type hints throughout - Structured logging with `log.debug` - A custom exception class (`RateLimitExceeded`) - `__repr__` for easy debugging - A `stats` property for introspection - Context manager protocol (`__enter__`/`__exit__`) - Decorator support (`.limit`) -

Thread safety with a `Lock` - Early returns where appropriate

35.11 What You Learned in This Chapter

- **PEP 8** defines naming (`snake_case`, `PascalCase`, `UPPER_SNAKE_CASE`), spacing, and line length. Let `ruff format` enforce it automatically.
- **Docstrings** document what a function does, its args, return value, exceptions, and examples. Use Google style. Make examples runnable with `doctest`.
- **The Zen of Python** in practice: explicit over implicit, simple over complex, flat over nested, errors never silent, one obvious way.
- **Code review:** small PRs, one logical change, clear descriptions. Ask questions rather than make demands when reviewing.
- **`.gitignore`** excludes `__pycache__`, virtual environments, `.env`, dist, and coverage files.
- **Conventional Commits** (`feat:`, `fix:`, `docs:`, `refactor:`) make history readable and changelogs automatic.
- **Environment variables** for all secrets and environment-specific config. Use `pydantic-settings` for typed, validated settings.
- **Production logging:** levelled, named loggers per module, `RotatingFileHandler`, silence noisy third-party loggers.
- **Clean code rules:** one responsibility per function, named constants instead of magic numbers, early returns instead of deep nesting, names that explain intent.
- **Professional habits:** read error messages fully, write tests while writing code, commit small and often, document decisions not mechanics, favor boring solutions, refactor fearlessly with test coverage.

35.12 What's Next?

Chapter 36 covers **Advanced Python Internals** — how Python actually works under the hood: the CPython interpreter, bytecode, the GIL in depth, memory management, reference counting, and the descriptor protocol. Understanding internals makes you a more effective Python developer — you'll know *why* things work the way they do, not just *that* they do.

Chapter 36

Chapter 36: Advanced Python Internals

Most developers use Python for years without understanding how it works internally. That's fine — the abstraction holds. But understanding what happens under the hood makes you a better Python developer. You'll know why some things are fast and others slow, why the GIL exists, how memory is managed, and why certain patterns work the way they do.

This chapter goes deep. Take it slowly.

36.1 How Python Runs Your Code

When you run `python script.py`, several things happen:

```
1. Lexing    --- the source text is tokenized into tokens
2. Parsing   --- tokens are turned into an Abstract Syntax Tree (AST)
3. Compiling --- the AST is compiled to bytecode (.pyc files)
4. Executing --- the CPython interpreter executes the bytecode
```

You can inspect every stage:

```
# Stage 1: Tokens
import tokenize
import io

source = "x = 1 + 2"
tokens = list(tokenize.generate_tokens(io.StringIO(source).readline))
for tok in tokens:
```

```
    print(tok)
# TokenInfo(type=1 (NAME),  string='x', ...)
# TokenInfo(type=54 (OP),   string='=', ...)
# TokenInfo(type=2 (NUMBER),string='1', ...)
# TokenInfo(type=54 (OP),   string='+', ...)
# TokenInfo(type=2 (NUMBER),string='2', ...)

# Stage 2: Abstract Syntax Tree
import ast

source = "x = 1 + 2"
tree   = ast.parse(source)
print(ast.dump(tree, indent=2))
```

Output (simplified):

```
Module(
  body=[
    Assign(
      targets=[Name(id='x')],
      value=BinOp(
        left=Constant(value=1),
        op=Add(),
        right=Constant(value=2)
      )
    )
  ]
)
```

```
# Stage 3: Bytecode
import dis

def add(a, b):
    return a + b

dis.dis(add)
```

Output:

```
  2           0 RESUME           0

  3           2 LOAD_FAST        0 (a)
              4 LOAD_FAST        1 (b)
              6 BINARY_OP        0 (+)
             10 RETURN_VALUE
```

Each line is a **bytecode instruction** — a simple operation the Python virtual machine executes. `LOAD_FAST` pushes a local variable onto the stack. `BI-`

`NARY_OP` pops two values, adds them, pushes the result. `RETURN_VALUE` pops the top of the stack and returns it.

Python is a **stack machine** — instructions operate on a stack of values.

36.1.1 Inspecting bytecode of any function

```
import dis

def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

dis.dis(fibonacci)
print()
print(f"Constants: {fibonacci.__code__.co_consts}")
print(f"Names:     {fibonacci.__code__.co_names}")
print(f"Varnames:  {fibonacci.__code__.co_varnames}")
print(f"Stack size:{fibonacci.__code__.co_stacksize}")
```

The `__code__` object contains everything about the compiled function: its bytecode (`co_code`), local variable names, constants, free variables, and more.

36.2 The CPython Virtual Machine

CPython is the reference implementation of Python — the one you download from python.org. It's written in C.

The core execution loop is in `ceval.c` — a giant `switch` statement that dispatches on bytecode opcodes. For every `LOAD_FAST`, `BINARY_OP`, `CALL_FUNCTION`, there's a case in that switch that executes the corresponding C code.

This is why Python is slower than compiled languages like C or Go — each Python operation involves multiple C operations, plus the overhead of the dispatch loop. The GIL (covered next) adds more overhead for multi-threaded programs.

36.2.1 Code objects

Every function, class, and module has a **code object** (`types.CodeType`):

```
import types
```

```
def outer():
    x = 10
    def inner():
        return x
    return inner

print(type(outer.__code__))    # <class 'code'>

# Code object attributes
code = outer.__code__
print(f"co_name:      {code.co_name}")         # outer
print(f"co_filename:  {code.co_filename}")     # script.py
print(f"co_firstlineno:{code.co_firstlineno}")# 1
print(f"co_varnames:  {code.co_varnames}")     # ('x', 'inner')
print(f"co_freevars:  {code.co_freevars}")     # () --- outer has no free vars
print(f"co_cellvars:  {code.co_cellvars}")     # ('x',) --- x is captured by inn

inner_code = outer.__code__.co_consts[1]       # the inner function's code objec
print(f"inner freevars: {inner_code.co_freevars}")  # ('x',) --- captured from
```

36.2.2 Frame objects

When a function is called, Python creates a **frame object** — a runtime snapshot of the execution state:

```
import sys

def show_frame():
    frame = sys._getframe()      # the current frame
    print(f"Function:  {frame.f_code.co_name}")
    print(f"File:      {frame.f_code.co_filename}")
    print(f"Line:      {frame.f_lineno}")
    print(f"Locals:    {frame.f_locals}")
    print(f"Globals:   {list(frame.f_globals.keys())[:5]}")

    caller = frame.f_back        # the frame that called us
    if caller:
        print(f"Called by: {caller.f_code.co_name}")

def main():
    x = 42
    show_frame()

main()
```

The call stack is a linked list of frame objects — each frame has `f_back` pointing to its caller.

```
# Walk the entire call stack
import traceback

def deep():
    for frame_info in traceback.extract_stack():
        print(f"  {frame_info.filename}:{frame_info.lineno} in {frame_info.name}")

def middle():
    deep()

def top():
    middle()

top()
```

36.3 The GIL — Global Interpreter Lock in Depth

The GIL is a mutex in CPython that ensures only one thread executes Python bytecode at a time. It's the most controversial aspect of CPython.

36.3.1 Why the GIL exists

CPython's memory management is not thread-safe by design. It uses **reference counting** — every object has a counter of how many references point to it. When the counter reaches zero, the object is freed. Without the GIL, two threads could simultaneously modify an object's reference count, causing corruption.

The GIL is a coarse solution: instead of fine-grained locks on every object, one global lock protects everything. This is simpler and faster for single-threaded programs — and most programs are mostly single-threaded.

36.3.2 The GIL releases during I/O

The GIL is released before any I/O operation and re-acquired after. This is why threading helps for I/O-bound work:

```
import threading
import time
```

```
def io_bound():
    time.sleep(1)    # GIL is released during sleep

def cpu_bound():
    sum(range(10_000_000))    # GIL is held throughout

# I/O --- threads help (GIL releases during sleep)
start = time.perf_counter()
threads = [threading.Thread(target=io_bound) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
print(f"I/O 4 threads: {time.perf_counter() - start:.2f}s")    # ~1.0s

# CPU --- threads don't help (GIL prevents true parallelism)
start = time.perf_counter()
threads = [threading.Thread(target=cpu_bound) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
print(f"CPU 4 threads: {time.perf_counter() - start:.2f}s")    # ~4.0s (no speed
```

36.3.3 The GIL check interval

The GIL doesn't hold forever. Python checks every `sys.getswitchinterval()` seconds (default: 5ms) whether another thread wants the GIL. At that point, the current thread drops it, and another thread can acquire it.

```
import sys
print(sys.getswitchinterval())    # 0.005  (5 milliseconds)
sys.setswitchinterval(0.01)       # change to 10ms
```

36.3.4 Python 3.13 — the no-GIL build

Python 3.13 introduced an experimental no-GIL build (`--disable-gil`). True multi-threading for CPU-bound work is coming to CPython — but the default build still has the GIL for backwards compatibility. Watch this space.

36.4 Memory Management

36.4.1 Reference counting

Every Python object has a reference count — an integer tracking how many names, containers, or other objects hold a reference to it.

```
import sys

x = [1, 2, 3]
print(sys.getrefcount(x))    # 2 (x itself, and the argument to getrefcount)

y = x
print(sys.getrefcount(x))    # 3 (x, y, and getrefcount's argument)

del y
print(sys.getrefcount(x))    # 2
```

When the reference count hits zero, CPython immediately deallocates the object — no waiting for a garbage collection cycle.

```
class MyObject:
    def __init__(self, name):
        self.name = name
        print(f"Created  {name}")

    def __del__(self):
        print(f"Destroyed {self.name}")


obj = MyObject("A")
print("About to delete")
del obj                  # reference count -> 0, __del__ called immediately
print("After delete")
```

Output:

```
Created  A
About to delete
Destroyed A    <- immediate
After delete
```

36.4.2 Cyclic garbage collector

Reference counting has one weakness: **reference cycles**. Two objects that reference each other will never reach zero, even when no outside code references either.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next  = None

# Create a cycle
```

```
a = Node(1)
b = Node(2)
a.next = b
b.next = a    # cycle: a -> b -> a

del a
del b
# Reference counts are now 1 (each object references the other)
# Not zero --- they're never freed by reference counting alone
```

Python's **cyclic garbage collector** (in the `gc` module) detects and breaks these cycles:

```
import gc

gc.enable()                    # on by default
gc.collect()                   # force a collection
print(gc.get_count())          # (generation 0, 1, 2 counts)
print(gc.get_threshold())      # (700, 10, 10) --- collection thresholds

# Inspect what's tracked
gc.set_debug(gc.DEBUG_LEAK)    # print objects that can't be freed
```

The cyclic GC runs automatically in three generations. Most objects are young (generation 0) and die young. Objects that survive multiple collections move to older generations and are collected less frequently.

36.4.3 Object internment and small integer caching

CPython caches small integers (-5 to 256) and short strings. This is why `is` comparisons work for small integers but not large ones:

```
a = 256
b = 256
print(a is b)     # True --- same cached object

a = 257
b = 257
print(a is b)     # False (may vary) --- different objects

# String interning
a = "hello"
b = "hello"
print(a is b)     # True --- interned (short strings matching identifier rules)

a = "hello world"
```

```
b = "hello world"
print(a is b)     # False (may vary) --- not interned
```

This is why you must always use `==` for value comparison, never `is`.

36.4.4 Memory pools — `pymalloc`

CPython doesn't call `malloc`/`free` directly for small objects. It uses its own memory allocator (`pymalloc`) that maintains pools of fixed-size blocks. This dramatically reduces memory fragmentation and allocation overhead for the many small, short-lived objects Python creates.

```
# See memory usage
import tracemalloc

tracemalloc.start()

# Your code here
data = [i ** 2 for i in range(100_000)]

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics("lineno")

print("Top 5 memory allocations:")
for stat in top_stats[:5]:
    print(f"  {stat}")

tracemalloc.stop()
```

36.5 The Descriptor Protocol

Descriptors are one of Python's most powerful mechanisms. They're how `@property`, `@classmethod`, `@staticmethod`, and `super()` work internally.

A descriptor is any object that defines `__get__`, `__set__`, or `__delete__`. When you access an attribute on an object, Python checks whether the class has a descriptor with that name.

```
class Descriptor:
    def __set_name__(self, owner, name):
        self.name = name          # called when class is defined

    def __get__(self, obj, objtype=None):
        if obj is None:
```

```
            return self           # accessed from the class, not an instance
        return obj.__dict__.get(self.name)

    def __set__(self, obj, value):
        obj.__dict__[self.name] = value

    def __delete__(self, obj):
        del obj.__dict__[self.name]


class MyClass:
    x = Descriptor()
    y = Descriptor()


obj = MyClass()
obj.x = 42
print(obj.x)     # 42 --- goes through Descriptor.__get__
del obj.x        # goes through Descriptor.__delete__
```

36.5.1 Building a typed attribute descriptor

```
class TypedAttribute:
    """Descriptor that enforces a type constraint."""
    def __set_name__(self, owner, name):
        self.public_name  = name
        self.private_name = f"_{name}"

    def __init__(self, expected_type, default=None):
        self.expected_type = expected_type
        self.default       = default

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return getattr(obj, self.private_name, self.default)

    def __set__(self, obj, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(
                f"{self.public_name} must be {self.expected_type.__name__}, "
                f"got {type(value).__name__}"
            )
        setattr(obj, self.private_name, value)
```

```
class Person:
    name  = TypedAttribute(str)
    age   = TypedAttribute(int, default=0)
    score = TypedAttribute(float, default=0.0)


p = Person()
p.name  = "Alice"
p.age   = 30
p.score = 95.5

print(p.name, p.age, p.score)    # Alice 30 95.5

try:
    p.age = "thirty"     # TypeError: age must be int, got str
except TypeError as e:
    print(e)
```

36.5.2 How `@property` is implemented

`property` is a built-in descriptor. Here's a simplified version:

```
class property_:
    """A simplified version of the built-in property descriptor."""
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc or (fget.__doc__ if fget else None)

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
```

```
            if self.fdel is None:
                raise AttributeError("can't delete attribute")
            self.fdel(obj)

        def getter(self, fget):
            return type(self)(fget, self.fset, self.fdel, self.__doc__)

        def setter(self, fset):
            return type(self)(self.fget, fset, self.fdel, self.__doc__)

        def deleter(self, fdel):
            return type(self)(self.fget, self.fset, fdel, self.__doc__)


    class Temperature:
        def __init__(self, celsius=0.0):
            self._celsius = celsius

        @property_()
        def celsius(self):
            return self._celsius

        @celsius.setter
        def celsius(self, value):
            if value < -273.15:
                raise ValueError("Temperature below absolute zero")
            self._celsius = value

        @property_()
        def fahrenheit(self):
            return self._celsius * 9 / 5 + 32


    t = Temperature(100)
    print(t.celsius)      # 100
    print(t.fahrenheit)   # 212.0
    t.celsius = -5
    print(t.celsius)      # -5
```

36.6 Metaclasses

A metaclass is a class whose instances are classes. The default metaclass for all Python classes is `type`. When you write `class Foo:`, Python calls `type("Foo", (object,), {...})` under the hood.

```
# These are equivalent
class MyClass:
    x = 42

MyClass = type("MyClass", (object,), {"x": 42})

print(type(MyClass))    # <class 'type'>
print(MyClass.x)        # 42
```

36.6.1 Writing a metaclass

```
class SingletonMeta(type):
    """Metaclass that makes a class a Singleton."""
    _instances: dict = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]


class Database(metaclass=SingletonMeta):
    def __init__(self, url: str):
        self.url = url
        print(f"Connected to {url}")


db1 = Database("postgresql://localhost/myapp")    # Connected to ...
db2 = Database("postgresql://localhost/other")    # NOT printed --- returns existin
print(db1 is db2)    # True --- same object
print(db2.url)       # postgresql://localhost/myapp
```

36.6.2 Metaclass `__new__` and `__init__`

```
class ValidatedMeta(type):
    """Metaclass that validates class definitions at creation time."""

    def __new__(mcs, name, bases, namespace):
        # Check that every method has type hints
```

```
        for attr_name, value in namespace.items():
            if callable(value) and not attr_name.startswith("_"):
                if not hasattr(value, "__annotations__"):
                    raise TypeError(
                        f"{name}.{attr_name} must have type annotations"
                    )
        return super().__new__(mcs, name, bases, namespace)


class StrictAPI(metaclass=ValidatedMeta):
    def get_user(self, user_id: int) -> dict:   # OK --- has annotations
        return {}

# class BadAPI(metaclass=ValidatedMeta):
#     def fetch():   # TypeError --- no annotations
#         pass
```

36.6.3 `__init_subclass__` — the modern alternative to metaclasses

Metaclasses are powerful but complex. For most use cases, `__init_subclass__` is simpler:

```
class Plugin:
    _registry: dict[str, type] = {}

    def __init_subclass__(cls, name: str = "", **kwargs):
        super().__init_subclass__(**kwargs)
        if name:
            Plugin._registry[name] = cls
            print(f"Registered plugin: {name!r}")


class JSONPlugin(Plugin, name="json"):
    def serialize(self, data):
        import json
        return json.dumps(data)


class CSVPlugin(Plugin, name="csv"):
    def serialize(self, data):
        return ",".join(str(v) for v in data)
```

```
print(Plugin._registry)
# {'json': <class 'JSONPlugin'>, 'csv': <class 'CSVPlugin'>}

plugin = Plugin._registry["json"]()
print(plugin.serialize({"hello": "world"}))
```

`__init_subclass__` is called automatically when a class inherits from `Plugin`. It's the preferred pattern for plugin systems, registries, and framework hooks.

36.7 `__slots__` Internals

We used `__slots__` in Chapter 30 for memory reduction. Now let's understand why it works.

Without `__slots__`, every instance has a `__dict__` — a hash table that can hold any attribute. A `dict` takes at least 232 bytes.

With `__slots__`, Python creates **slot descriptors** in the class — fixed memory offsets in the instance structure. No `__dict__` is created. The instance is a small C struct with named slots.

```
import sys

class WithDict:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class WithSlots:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y


d = WithDict(1, 2)
s = WithSlots(1, 2)

print(f"WithDict:  {sys.getsizeof(d)} + {sys.getsizeof(d.__dict__)} = "
      f"{sys.getsizeof(d) + sys.getsizeof(d.__dict__)} bytes")
print(f"WithSlots: {sys.getsizeof(s)} bytes")

# WithDict:  48 + 232 = 280 bytes
# WithSlots: 56 bytes   --- 5x smaller
```

```
# Verify: WithSlots has no __dict__
print(hasattr(d, "__dict__"))    # True
print(hasattr(s, "__dict__"))    # False
```

36.8 `__getattr__` and `__getattribute__`

These two methods control attribute access:

- `__getattribute__` — called for **every** attribute access, even if the attribute exists
- `__getattr__` — called only when normal lookup **fails**

```
class TrackedAccess:
    """Log every attribute access."""
    def __init__(self):
        object.__setattr__(self, "_accessed", set())

    def __getattribute__(self, name):
        if not name.startswith("_"):
            accessed = object.__getattribute__(self, "_accessed")
            accessed.add(name)
        return object.__getattribute__(self, name)

    def accessed_attrs(self):
        return self._accessed


class LazyLoader:
    """Load attributes from a dict only when accessed."""
    def __init__(self, data: dict):
        self._data = data

    def __getattr__(self, name):
        # Only called when normal lookup fails
        if name in self._data:
            value = self._data[name]
            setattr(self, name, value)    # cache it so __getattr__ isn't called
            return value
        raise AttributeError(f"{type(self).__name__!r} has no attribute {name!r}


config = LazyLoader({
```

```
    "host":     "localhost",
    "port":     5432,
    "database": "myapp",
})

print(config.host)      # localhost --- loaded from _data, cached on instance
print(config.port)      # 5432
print(config.database)  # myapp
```

36.9 AST Manipulation

You can parse, inspect, and even modify Python's AST — the internal representation of your code before compilation:

```
import ast

source = """
def greet(name):
    return f"Hello, {name}!"
"""

tree = ast.parse(source)

# Walk all nodes
for node in ast.walk(tree):
    if isinstance(node, ast.FunctionDef):
        print(f"Function: {node.name}, line {node.lineno}")
    elif isinstance(node, ast.Return):
        print(f"Return statement at line {node.lineno}")


# Transform the AST --- add a print to every function
class AddLogging(ast.NodeTransformer):
    def visit_FunctionDef(self, node):
        self.generic_visit(node)    # process children first

        # Add: print(f"Calling {node.name}")
        log_stmt = ast.Expr(
            value=ast.Call(
                func=ast.Name(id="print", ctx=ast.Load()),
                args=[ast.Constant(value=f"Calling {node.name}")],
                keywords=[],
```

```
            )
        )
        node.body.insert(0, log_stmt)
        return node


transformed = AddLogging().visit(tree)
ast.fix_missing_locations(transformed)

code = compile(transformed, "<ast>", "exec")
exec(code)

greet("Alice")
# Calling greet      <- injected by AST transformation
# Hello, Alice!      <- original behavior
```

This is how many Python tools work: `pytest` rewrites `assert` statements, `dataclasses` generates `__init__` code, `attrs` builds constructors — all by manipulating the AST.

36.10 `sys` Module Internals

```
import sys

# Python implementation details
print(sys.implementation.name)     # cpython
print(sys.version)                 # 3.12.0 (...)
print(sys.version_info)            # sys.version_info(major=3, minor=12, ...)

# Memory
print(sys.getsizeof([]))           # 56 bytes (empty list)
print(sys.getsizeof({}))           # 64 bytes (empty dict)
print(sys.getsizeof(""))           # 49 bytes (empty string)
print(sys.getsizeof(0))            # 28 bytes (int)

# Recursion
print(sys.getrecursionlimit())     # 1000
sys.setrecursionlimit(5000)        # increase for deep recursion

# Reference counting
x = [1, 2, 3]
print(sys.getrefcount(x))         # 2
```

```
# All loaded modules
print(list(sys.modules.keys())[:10])

# Platform
print(sys.platform)   # 'win32', 'darwin', 'linux'

# Intern strings (force string to be shared)
a = sys.intern("hello world")
b = sys.intern("hello world")
print(a is b)   # True --- both refer to the same object
```

36.11 Project: A Simple Bytecode Analyzer

```
"""
bytecode_analyzer.py --- Inspect and report on Python bytecode.
"""
import dis
import sys
import types
from collections import Counter


def analyze(func) -> dict:
    """
    Analyze a function's bytecode and return a report.

    Args:
        func: Any Python function or method.

    Returns:
        A dict with keys: name, instructions, opcode_counts,
        constants, locals, stack_size, complexity_estimate.
    """
    code  = func.__code__
    instrs= list(dis.get_instructions(code))

    opcode_counts = Counter(i.opname for i in instrs)

    # Estimate cyclomatic complexity: 1 + number of branches
    branch_ops    = {"POP_JUMP_IF_TRUE", "POP_JUMP_IF_FALSE",
                     "JUMP_IF_TRUE_OR_POP", "JUMP_IF_FALSE_OR_POP",
```

```
                    "FOR_ITER"}
    complexity    = 1 + sum(1 for i in instrs if i.opname in branch_ops)

    return {
        "name":               code.co_name,
        "file":               code.co_filename,
        "line":               code.co_firstlineno,
        "instruction_count":  len(instrs),
        "opcode_counts":      dict(opcode_counts.most_common(5)),
        "constants":          [c for c in code.co_consts if c is not None],
        "local_variables":    list(code.co_varnames),
        "free_variables":     list(code.co_freevars),
        "stack_size":         code.co_stacksize,
        "complexity_estimate":complexity,
    }


def print_report(func) -> None:
    """Print a human-readable bytecode analysis report."""
    report = analyze(func)

    print(f"\n{'□'*50}")
    print(f"  Function: {report['name']}")
    print(f"  File:     {report['file']}:{report['line']}")
    print(f"{'□'*50}")
    print(f"  Instructions:    {report['instruction_count']}")
    print(f"  Stack depth:     {report['stack_size']}")
    print(f"  Complexity est.: {report['complexity_estimate']}")

    if report["constants"]:
        print(f"  Constants:       {report['constants']}")
    if report["local_variables"]:
        print(f"  Locals:          {report['local_variables']}")
    if report["free_variables"]:
        print(f"  Free vars:       {report['free_variables']}")

    print(f"\n  Top opcodes:")
    for opname, count in report["opcode_counts"].items():
        bar = "" * count
        print(f"    {opname:<30} {count:>3}  {bar}")

    print()
    print("  Disassembly:")
    dis.dis(func)
```

```
# ── Demo ──────────────────────────────────────────────────────────────

def simple(x, y):
    return x + y

def with_branch(x):
    if x > 0:
        return "positive"
    elif x < 0:
        return "negative"
    else:
        return "zero"

def with_loop(items):
    total = 0
    for item in items:
        if item > 0:
            total += item
    return total

def with_closure():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter


if __name__ == "__main__":
    for func in [simple, with_branch, with_loop]:
        print_report(func)
```

36.12 What You Learned in This Chapter

- Python source goes through **lexing -> parsing -> compilation -> execution**. `tokenize`, `ast`, and `dis` let you inspect each stage.
- CPython is a **stack machine** — bytecode instructions push and pop values on a stack.
- Every function has a **code object** (`__code__`) containing bytecode, con-

stants, variable names, and free variables.
- Every function call creates a **frame object** (`f_locals`, `f_globals`, `f_back`, `f_lineno`) — the call stack is a linked list of frames.
- The **GIL** is a coarse mutex protecting CPython's non-thread-safe reference counting. It releases during I/O, preventing true CPU parallelism in threads.
- **Reference counting** frees objects immediately when their count reaches zero. The **cyclic GC** handles reference cycles in three generations.
- Small integers (-5 to 256) and short strings are **interned** — always the same object. Use `==`, never `is`, for value comparison.
- `tracemalloc` measures memory allocation per line.
- A **descriptor** defines `__get__`, `__set__`, or `__delete__`. `property`, `classmethod`, and `staticmethod` are all descriptors.
- A **metaclass** is a class whose instances are classes. `type` is the default metaclass. Use `__init_subclass__` for most plugin/registry patterns instead.
- `__slots__` replaces `__dict__` with fixed C struct slots — 4-5x smaller per instance.
- `__getattribute__` intercepts every access; `__getattr__` only fires when normal lookup fails.
- Python's AST is manipulable — parse, inspect, transform, compile, and execute.

36.13 What's Next?

Chapter 37 covers **Metaprogramming** — class factories, dynamic class creation, `__class_getitem__`, runtime code generation, and `exec`/`eval`. You've seen the internals; now you'll use them to write code that writes code.

Chapter 37

Chapter 37: Metaprogramming

Metaprogramming means writing code that operates on code. Instead of writing ten similar classes by hand, you write one function that generates all ten. Instead of repeating validation logic across twenty methods, you write a decorator that injects it everywhere at once. Instead of maintaining boilerplate, you eliminate it.

Python's metaprogramming tools are unusually accessible. You've already used some — decorators, `dataclasses`, `@property` — without realizing they're metaprogramming. This chapter goes further: class factories, dynamic attribute generation, `exec`/`eval`, and runtime code generation.

37.1 What Metaprogramming Is (and Isn't)

Metaprogramming is not magic or clever tricks. It's a tool for reducing repetition at the structural level — when copy-paste would multiply across classes or modules, not just loops.

Use it when: - You have N similar classes that differ only in a few parameters - You need to enforce a pattern across many functions or methods - You're building a framework or library, not a single application - Boilerplate is so repetitive that maintaining it manually is error-prone

Don't use it when: - A simple function or loop would do - The generated code would be harder to debug than the repetition - Your team won't understand it without significant explanation

37.2 `type()` as a Class Factory

In Chapter 36 you saw that `type("Name", bases, namespace)` creates a class dynamically. This is the core of class factories:

```
# Statically defined
class Dog:
    sound = "Woof"
    def speak(self):
        return f"{type(self).__name__} says {self.sound}"

# Dynamically equivalent
Dog = type("Dog", (object,), {
    "sound": "Woof",
    "speak": lambda self: f"{type(self).__name__} says {self.sound}",
})

print(Dog().speak())    # Dog says Woof
```

37.2.1 Building an animal factory

```
def make_animal(name: str, sound: str, legs: int):
    """Create an animal class dynamically."""
    def speak(self):
        return f"{self.name} says {self.sound}!"

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"{type(self).__name__}({self.name!r})"

    return type(name, (object,), {
        "sound":    sound,
        "legs":     legs,
        "__init__": __init__,
        "speak":    speak,
        "__repr__": __repr__,
    })


Dog  = make_animal("Dog",  "Woof", 4)
```

```
Cat  = make_animal("Cat",  "Meow", 4)
Bird = make_animal("Bird", "Tweet", 2)

fido = Dog("Fido")
whiskers = Cat("Whiskers")

print(fido.speak())       # Fido says Woof!
print(whiskers.speak())   # Whiskers says Meow!
print(Bird.legs)          # 2
print(isinstance(fido, Dog))   # True
print(isinstance(fido, Cat))   # False
```

37.2.2 A model factory for database tables

```
def make_model(table_name: str, fields: dict[str, type]):
    """
    Create a simple model class for a database table.
    fields: {"column_name": python_type}
    """
    def __init__(self, **kwargs):
        for name, expected_type in fields.items():
            value = kwargs.get(name)
            if value is not None and not isinstance(value, expected_type):
                raise TypeError(
                    f"{name} must be {expected_type.__name__}, "
                    f"got {type(value).__name__}"
                )
            setattr(self, name, value)

    def __repr__(self):
        attrs = ", ".join(f"{k}={getattr(self, k)!r}" for k in fields)
        return f"{type(self).__name__}({attrs})"

    def to_dict(self):
        return {k: getattr(self, k) for k in fields}

    @classmethod
    def from_dict(cls, data: dict):
        return cls(**{k: data.get(k) for k in fields})

    namespace = {
        "__init__":    __init__,
        "__repr__":    __repr__,
        "to_dict":     to_dict,
```

```
        "from_dict":   from_dict,
        "_table_name": table_name,
        "_fields":     fields,
    }
    return type(table_name.capitalize(), (object,), namespace)


# Create model classes without repeating yourself
User    = make_model("users",    {"name": str,  "email": str, "age": int})
Product = make_model("products", {"name": str,  "price": float, "stock": int})
Order   = make_model("orders",   {"user_id": int, "total": float})

alice = User(name="Alice", email="alice@example.com", age=30)
book  = Product(name="Python Book", price=29.99, stock=100)
order = Order(user_id=1, total=59.98)

print(alice)
print(book)
print(alice.to_dict())
```

Output:

```
Users(name='Alice', email='alice@example.com', age=30)
Products(name='Python Book', price=29.99, stock=100)
{'name': 'Alice', 'email': 'alice@example.com', 'age': 30}
```

37.3 Class Decorators

Just as function decorators wrap functions, class decorators wrap classes. A class decorator takes a class and returns a modified class:

```
def add_repr(cls):
    """Add a __repr__ based on __init__ parameter names."""
    import inspect
    params = list(inspect.signature(cls.__init__).parameters.keys())[1:]  # ski

    def __repr__(self):
        attrs = ", ".join(f"{p}={getattr(self, p)!r}" for p in params
                          if hasattr(self, p))
        return f"{type(self).__name__}({attrs})"

    cls.__repr__ = __repr__
    return cls
```

```
def add_eq(cls):
    """Add __eq__ based on all instance attributes."""
    def __eq__(self, other):
        if type(self) is not type(other):
            return NotImplemented
        return self.__dict__ == other.__dict__

    cls.__eq__ = __eq__
    return cls


def frozen(cls):
    """Make instances immutable after __init__."""
    original_setattr = cls.__setattr__ if hasattr(cls, "__setattr__") else object.
    original_init    = cls.__init__

    def __init__(self, *args, **kwargs):
        object.__setattr__(self, "_initialized", False)
        original_init(self, *args, **kwargs)
        object.__setattr__(self, "_initialized", True)

    def __setattr__(self, name, value):
        if object.__getattribute__(self, "_initialized"):
            raise AttributeError(f"{type(self).__name__} instances are immutable")
        object.__setattr__(self, name, value)

    cls.__init__    = __init__
    cls.__setattr__ = __setattr__
    return cls


@add_repr
@add_eq
@frozen
class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y


p1 = Point(1.0, 2.0)
p2 = Point(1.0, 2.0)
p3 = Point(3.0, 4.0)
```

```
print(p1)            # Point(x=1.0, y=2.0)
print(p1 == p2)      # True
print(p1 == p3)      # False

try:
    p1.x = 99        # AttributeError: Point instances are immutable
except AttributeError as e:
    print(e)
```

37.4 `__init_subclass__` — Framework Hooks

`__init_subclass__` is called whenever a subclass is defined. It's one of the cleanest ways to build registration systems and enforce constraints on subclasses:

```
class Serializable:
    """
    Mix-in that requires subclasses to declare _fields
    and auto-generates to_dict / from_dict.
    """
    _fields: tuple[str, ...] = ()

    def __init_subclass__(cls, fields: tuple[str, ...] = (), **kwargs):
        super().__init_subclass__(**kwargs)
        if fields:
            cls._fields = fields
        elif not cls._fields:
            raise TypeError(
                f"{cls.__name__} must declare fields=(...) or set _fields"
            )

        def to_dict(self) -> dict:
            return {f: getattr(self, f) for f in cls._fields}

        @classmethod
        def from_dict(klass, data: dict):
            return klass(**{f: data[f] for f in klass._fields})

        cls.to_dict   = to_dict
        cls.from_dict = from_dict
```

```
class User(Serializable, fields=("name", "email", "age")):
    def __init__(self, name, email, age):
        self.name  = name
        self.email = email
        self.age   = age


class Product(Serializable, fields=("name", "price", "stock")):
    def __init__(self, name, price, stock):
        self.name  = name
        self.price = price
        self.stock = stock


alice = User("Alice", "alice@example.com", 30)
print(alice.to_dict())
# {'name': 'Alice', 'email': 'alice@example.com', 'age': 30}

alice2 = User.from_dict({"name": "Alice", "email": "alice@example.com", "age": 30})
print(alice2.name)    # Alice

book = Product("Python Book", 29.99, 100)
print(book.to_dict())
# {'name': 'Python Book', 'price': 29.99, 'stock': 100}
```

37.5 Dynamic Attribute Generation

Sometimes attributes don't exist statically — they're computed or looked up at access time:

```
class DynamicConfig:
    """
    Config object where any attribute is read from a dict.
    Nested dicts become nested DynamicConfig objects.
    """
    def __init__(self, data: dict):
        object.__setattr__(self, "_data", data)

    def __getattr__(self, name: str):
        data = object.__getattribute__(self, "_data")
        if name not in data:
```

```
            raise AttributeError(f"No config key {name!r}")
        value = data[name]
        if isinstance(value, dict):
            return DynamicConfig(value)
        return value

    def __setattr__(self, name: str, value):
        data = object.__getattribute__(self, "_data")
        data[name] = value

    def __repr__(self):
        data = object.__getattribute__(self, "_data")
        return f"DynamicConfig({data!r})"


config = DynamicConfig({
    "database": {
        "host":     "localhost",
        "port":     5432,
        "name":     "myapp",
    },
    "cache": {
        "backend":  "redis",
        "timeout":  300,
    },
    "debug": True,
})

print(config.debug)                 # True
print(config.database.host)         # localhost
print(config.database.port)         # 5432
print(config.cache.backend)         # redis

config.debug = False
print(config.debug)                 # False
```

37.5.1 Auto-generating methods

```
class AutoProperty:
    """
    Metaclass-free way to generate properties from a list of field names.
    """

    def __init_subclass__(cls, fields=(), **kwargs):
```

```
        super().__init_subclass__(**kwargs)

        for field in fields:
            private = f"_{field}"

            def make_property(priv):
                @property
                def prop(self):
                    return getattr(self, priv, None)

                @prop.setter
                def prop(self, value):
                    setattr(self, priv, value)

                return prop

            setattr(cls, field, make_property(private))


class Point(AutoProperty, fields=("x", "y", "z")):
    def __init__(self, x, y, z=0):
        self._x = x
        self._y = y
        self._z = z

    def __repr__(self):
        return f"Point({self.x}, {self.y}, {self.z})"


p = Point(1, 2, 3)
print(p)        # Point(1, 2, 3)
p.x = 10
print(p)        # Point(10, 2, 3)
```

37.6 `exec` and `eval` — Runtime Code Execution

`eval` evaluates a single expression. `exec` executes statements. Both should be used carefully — arbitrary code execution is a security risk if the input comes from untrusted sources.

```
# eval --- evaluate an expression, returns a value
result = eval("2 ** 10")
```

```
print(result)   # 1024

# With a custom namespace
ns = {"x": 5, "y": 3}
print(eval("x ** y", ns))   # 125

# exec --- execute statements (returns None)
code = """
def greet(name):
    return f"Hello, {name}!"
"""
namespace = {}
exec(code, namespace)
print(namespace["greet"]("Alice"))   # Hello, Alice!
```

37.6.1 Safe use: controlled namespace

```
# Only expose specific names --- block access to builtins
safe_ns = {
    "__builtins__": {},   # no built-ins
    "abs":   abs,
    "round": round,
    "min":   min,
    "max":   max,
}

formula = "max(abs(-5), round(3.7))"
result  = eval(formula, safe_ns)
print(result)   # 5

# Dangerous --- never do this with untrusted input
# eval("__import__('os').system('rm -rf /')")
```

37.6.2 Generating code at runtime

```
def make_validator_class(rules: dict[str, tuple]):
    """
    Generate a validator class from a rules dict.
    rules: {"field": (type, min_val, max_val)}
    """
    method_lines = [
        "class GeneratedValidator:",
        "    def validate(self, data):",
        "        errors = []",
```

```
    ]

    for field, (expected_type, min_val, max_val) in rules.items():
        method_lines += [
            f"        if '{field}' not in data:",
            f"            errors.append('{field} is required')",
            f"        elif not isinstance(data['{field}'], {expected_type.__name__}):",
            f"            errors.append('{field} must be {expected_type.__name__}')",
        ]
        if min_val is not None:
            method_lines += [
                f"        elif data['{field}'] < {min_val}:",
                f"            errors.append('{field} must be >= {min_val}')",
            ]
        if max_val is not None:
            method_lines += [
                f"        elif data['{field}'] > {max_val}:",
                f"            errors.append('{field} must be <= {max_val}')",
            ]

    method_lines += [
        "        return errors",
    ]

    source = "\n".join(method_lines)
    ns: dict = {}
    exec(source, ns)
    return ns["GeneratedValidator"]


UserValidator = make_validator_class({
    "age":    (int,   0,   150),
    "score":  (float, 0.0, 100.0),
    "name":   (str,   None, None),
})

v = UserValidator()
print(v.validate({"name": "Alice", "age": 30, "score": 95.5}))   # []
print(v.validate({"age": -1, "score": 200.0}))
# ["name is required", "age must be >= 0", "score must be <= 100.0"]
```

37.7 `__class_getitem__` — Making Classes Generic

`__class_getitem__` is called when you write `MyClass[SomeType]`. This is how `list[int]`, `dict[str, int]`, and custom generics work:

```
from typing import Generic, TypeVar

T = TypeVar("T")

class TypedList(Generic[T]):
    """A list that enforces a single element type."""

    def __init__(self, item_type: type):
        self._type  = item_type
        self._items: list = []

    def __class_getitem__(cls, item_type):
        """Called when you write TypedList[int]."""
        instance = cls.__new__(cls)
        instance.__init__(item_type)
        return instance

    def append(self, item):
        if not isinstance(item, self._type):
            raise TypeError(
                f"Expected {self._type.__name__}, got {type(item).__name__}"
            )
        self._items.append(item)

    def __iter__(self):
        return iter(self._items)

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return f"TypedList[{self._type.__name__}]({self._items!r})"


int_list  = TypedList[int]
str_list  = TypedList[str]

int_list.append(1)
```

```
int_list.append(2)
int_list.append(3)
print(int_list)    # TypedList[int]([1, 2, 3])

try:
    int_list.append("hello")    # TypeError
except TypeError as e:
    print(e)
```

37.8 `__set_name__` — Descriptor Self-Registration

`__set_name__` is called on a descriptor when the class that contains it is defined. It tells the descriptor its own name:

```
class Validated:
    """A descriptor that validates on set using a provided validator function."""
    def __init__(self, validator, doc=""):
        self.validator = validator
        self.__doc__   = doc
        self.name      = None      # set by __set_name__

    def __set_name__(self, owner, name):
        self.name = name
        # Register this descriptor with the owner class
        if not hasattr(owner, "_validated_fields"):
            owner._validated_fields = []
        owner._validated_fields.append(name)

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name)

    def __set__(self, obj, value):
        error = self.validator(value)
        if error:
            raise ValueError(f"{self.name}: {error}")
        obj.__dict__[self.name] = value


# Validator functions --- return error string or None
def positive(v):
```

```
    return "must be positive" if v <= 0 else None

def non_empty_string(v):
    if not isinstance(v, str):
        return "must be a string"
    if not v.strip():
        return "must not be empty"
    return None

def valid_email(v):
    import re
    if not re.match(r"[^@]+@[^@]+\.[^@]+", v):
        return "must be a valid email"
    return None


class User:
    name  = Validated(non_empty_string, "User's display name")
    email = Validated(valid_email,      "User's email address")
    age   = Validated(positive,         "User's age in years")

    def __init__(self, name, email, age):
        self.name  = name
        self.email = email
        self.age   = age

    def __repr__(self):
        return f"User({self.name!r}, {self.email!r}, age={self.age})"


alice = User("Alice", "alice@example.com", 30)
print(alice)
print(f"Validated fields: {User._validated_fields}")

try:
    bad = User("", "not-an-email", -5)
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
User('Alice', 'alice@example.com', age=30)
Validated fields: ['name', 'email', 'age']
Error: name: must not be empty
```

37.9 Building a Mini ORM with Metaprogramming

Let's put it all together — a small ORM (Object-Relational Mapper) that uses metaprogramming to map Python classes to database tables:

```
"""
mini_orm.py --- A small ORM built with metaprogramming techniques.
"""
import sqlite3
from typing import Any, Optional


# ── Field descriptors ──────────────────────────────────────────

class Field:
    """Base field descriptor for ORM models."""
    sql_type = "TEXT"

    def __init__(self, required: bool = True, default: Any = None, unique: bool =
        self.required = required
        self.default  = default
        self.unique   = unique
        self.name     = None   # set by __set_name__

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name, self.default)

    def __set__(self, obj, value):
        if value is None and self.required:
            raise ValueError(f"{self.name} is required")
        obj.__dict__[self.name] = value

    def validate(self, value):
        pass

    def column_def(self) -> str:
        parts = [self.name, self.sql_type]
        if self.unique:
```

```
            parts.append("UNIQUE")
        return " ".join(parts)


class IntField(Field):
    sql_type = "INTEGER"

    def __set__(self, obj, value):
        if value is not None and not isinstance(value, int):
            raise TypeError(f"{self.name} must be int, got {type(value).__name_
        super().__set__(obj, value)


class FloatField(Field):
    sql_type = "REAL"

    def __set__(self, obj, value):
        if value is not None and not isinstance(value, (int, float)):
            raise TypeError(f"{self.name} must be numeric")
        super().__set__(obj, float(value) if value is not None else None)


class TextField(Field):
    sql_type = "TEXT"

    def __init__(self, max_length: int = 0, **kwargs):
        super().__init__(**kwargs)
        self.max_length = max_length

    def __set__(self, obj, value):
        if value is not None:
            if not isinstance(value, str):
                raise TypeError(f"{self.name} must be str")
            if self.max_length and len(value) > self.max_length:
                raise ValueError(f"{self.name} too long (max {self.max_length})
        super().__set__(obj, value)


# ── Model metaclass ──────────────────────────────────────────────────

class ModelMeta(type):
    """Metaclass that collects Field descriptors and generates SQL helpers."""

    def __new__(mcs, name, bases, namespace):
```

```
        fields = {}

        # Inherit fields from base models
        for base in bases:
            if hasattr(base, "_fields"):
                fields.update(base._fields)

        # Collect Field instances from this class's namespace
        for attr_name, value in list(namespace.items()):
            if isinstance(value, Field):
                fields[attr_name] = value

        namespace["_fields"] = fields
        namespace["_table"]  = name.lower() + "s"    # simple pluralization

        cls = super().__new__(mcs, name, bases, namespace)
        return cls


# ── Base model ──────────────────────────────────────────────────────

class Model(metaclass=ModelMeta):
    """Base class for all ORM models."""

    id = IntField(required=False)

    def __init__(self, **kwargs):
        for name, field in self._fields.items():
            value = kwargs.get(name, field.default)
            setattr(self, name, value)

    def __repr__(self):
        attrs = " ".join(
            f"{k}={getattr(self, k)!r}"
            for k in self._fields
            if k != "id" or getattr(self, k) is not None
        )
        return f"{type(self).__name__}({attrs})"

    # ── Schema ──────────────────────────────────────────────────────

    @classmethod
    def create_table(cls, conn: sqlite3.Connection) -> None:
        columns = ["id INTEGER PRIMARY KEY AUTOINCREMENT"]
```

```
        for name, field in cls._fields.items():
            if name == "id":
                continue
            columns.append(field.column_def())
        sql = f"CREATE TABLE IF NOT EXISTS {cls._table} ({', '.join(columns)})"
        conn.execute(sql)
        conn.commit()

    # ── CRUD ──────────────────────────────────────────────────────────

    def save(self, conn: sqlite3.Connection) -> "Model":
        data = {k: getattr(self, k) for k in self._fields if k != "id"}
        if self.id is None:
            # INSERT
            cols   = ", ".join(data.keys())
            placeholders = ", ".join("?" for _ in data)
            cursor = conn.execute(
                f"INSERT INTO {self._table} ({cols}) VALUES ({placeholders})",
                list(data.values()),
            )
            self.id = cursor.lastrowid
        else:
            # UPDATE
            set_clause = ", ".join(f"{k} = ?" for k in data)
            conn.execute(
                f"UPDATE {self._table} SET {set_clause} WHERE id = ?",
                [*data.values(), self.id],
            )
        conn.commit()
        return self

    @classmethod
    def get(cls, conn: sqlite3.Connection, id: int) -> Optional["Model"]:
        row = conn.execute(
            f"SELECT * FROM {cls._table} WHERE id = ?", (id,)
        ).fetchone()
        if row is None:
            return None
        return cls._from_row(row)

    @classmethod
    def all(cls, conn: sqlite3.Connection) -> list["Model"]:
        rows = conn.execute(f"SELECT * FROM {cls._table}").fetchall()
        return [cls._from_row(r) for r in rows]
```

```
    @classmethod
    def filter(cls, conn: sqlite3.Connection, **conditions) -> list["Model"]:
        where  = " AND ".join(f"{k} = ?" for k in conditions)
        values = list(conditions.values())
        rows   = conn.execute(
            f"SELECT * FROM {cls._table} WHERE {where}", values
        ).fetchall()
        return [cls._from_row(r) for r in rows]

    def delete(self, conn: sqlite3.Connection) -> None:
        if self.id is None:
            raise ValueError("Cannot delete unsaved model")
        conn.execute(f"DELETE FROM {self._table} WHERE id = ?", (self.id,))
        conn.commit()
        self.id = None

    @classmethod
    def _from_row(cls, row: tuple) -> "Model":
        """Create an instance from a DB row tuple."""
        col_names = ["id"] + [k for k in cls._fields if k != "id"]
        data = dict(zip(col_names, row))
        return cls(**data)


# ── Define models ──────────────────────────────────────────────

class User(Model):
    name  = TextField(max_length=100)
    email = TextField(unique=True)
    age   = IntField(required=False, default=0)


class Post(Model):
    title   = TextField(max_length=200)
    content = TextField(required=False, default="")
    user_id = IntField()
    views   = IntField(required=False, default=0)


# ── Demo ───────────────────────────────────────────────────────

if __name__ == "__main__":
    conn = sqlite3.connect(":memory:")
```

```
    User.create_table(conn)
    Post.create_table(conn)

    # Create users
    alice = User(name="Alice", email="alice@example.com", age=30).save(conn)
    bob   = User(name="Bob",   email="bob@example.com",   age=25).save(conn)

    print(f"Created: {alice}")
    print(f"Created: {bob}")

    # Create posts
    p1 = Post(title="Hello World", content="My first post!", user_id=alice.id).
    p2 = Post(title="Python Tips", content="Use list comprehensions.", user_id=
    p3 = Post(title="Databases",   content="SQL is powerful.", user_id=bob.id).

    # Query
    all_users = User.all(conn)
    print(f"\nAll users: {all_users}")

    alice_posts = Post.filter(conn, user_id=alice.id)
    print(f"\nAlice's posts: {alice_posts}")

    # Update
    alice.age = 31
    alice.save(conn)
    refetched = User.get(conn, alice.id)
    print(f"\nUpdated Alice: {refetched}")

    # Delete
    p3.delete(conn)
    print(f"\nPosts after delete: {Post.all(conn)}")

    conn.close()
```

Output:

```
Created: User(name='Alice' email='alice@example.com' age=30)
Created: User(name='Bob' email='bob@example.com' age=25)

All users: [User(id=1 name='Alice' ...), User(id=2 name='Bob' ...)]

Alice's posts: [Post(id=1 title='Hello World' ...), Post(id=2 title='Python Tips'

Updated Alice: User(id=1 name='Alice' email='alice@example.com' age=31)
```

```
Posts after delete: [Post(id=1 ...), Post(id=2 ...)]
```

This ORM used: - `__set_name__` on descriptors to auto-capture field names - `ModelMeta` metaclass to collect all Field instances into `_fields` - `type()` introspection to generate `CREATE TABLE` SQL automatically - Dynamic SQL generation for `INSERT`, `UPDATE`, `SELECT`, `DELETE` - Inheritance — `Model` provides all CRUD, subclasses just declare fields

---

37.10 What You Learned in This Chapter

- Metaprogramming reduces structural repetition — use it when N similar classes or N similar method patterns would otherwise require copy-paste.
- `type(name, bases, namespace)` creates classes dynamically at runtime.
- Class decorators modify a class after definition — the same pattern as function decorators but applied to the class object.
- `__init_subclass__` runs when a subclass is defined — use it for registration, validation, and automatic method generation without metaclasses.
- Dynamic attribute access with `__getattr__` / `__getattribute__` enables lazy loading, proxies, and configuration objects.
- `exec` executes arbitrary Python statements; `eval` evaluates expressions. Both accept a custom namespace for safety. Never run untrusted input.
- `__class_getitem__` implements `MyClass[Type]` syntax for custom generic classes.
- `__set_name__` tells descriptors their own attribute name and enables self-registration.
- Combining these tools — descriptors + metaclass + `__init_subclass__` + dynamic SQL — you can build a full ORM in ~200 lines.

37.11 What's Next?

Chapter 38 is the final chapter — **The Capstone Project**. You'll build a complete, production-quality Python application from scratch: a REST API with authentication, a database, a background task queue, structured logging, tests, and a Dockerfile. Everything you've learned in this book comes together in one project.

Chapter 38

Chapter 38: The Capstone Project — Build a Production Task Manager API

You've traveled a long way. Variables and loops in Chapter 1. Now metaclasses, async programming, type hints, databases, and deployment. This final chapter ties everything together.

We're building **TaskFlow** — a production-quality task management REST API. Not a toy. A real application you could deploy today.

Here's what it includes: - FastAPI REST API with full CRUD - SQLAlchemy ORM with Alembic migrations - JWT authentication with bcrypt password hashing - Background task processing with a simple job queue - Structured logging with rotation - pytest test suite with 90%+ coverage - Docker containerization - GitHub Actions CI/CD pipeline

Every technique from this book appears in this project. Let's build it.

38.1 Project Structure

```
taskflow/
├── src/
│   └── taskflow/
│       ├── __init__.py
```

```
│       ├── main.py          <- FastAPI app entry point
│       ├── config.py        <- Settings from environment
│       ├── database.py      <- SQLAlchemy setup
│       ├── logging_config.py<- Structured logging
│       ├── models/
│       │   ├── __init__.py
│       │   ├── user.py
│       │   └── task.py
│       ├── schemas/
│       │   ├── __init__.py
│       │   ├── user.py
│       │   └── task.py
│       ├── routers/
│       │   ├── __init__.py
│       │   ├── auth.py
│       │   ├── users.py
│       │   └── tasks.py
│       ├── services/
│       │   ├── __init__.py
│       │   ├── auth_service.py
│       │   ├── task_service.py
│       │   └── queue.py
│       └── dependencies.py
├── tests/
│   ├── conftest.py
│   ├── test_auth.py
│   ├── test_tasks.py
│   └── test_users.py
├── alembic/
│   ├── env.py
│   └── versions/
├── pyproject.toml
├── .env.example
├── Dockerfile
├── docker-compose.yml
└── .github/
    └── workflows/
        └── ci.yml
```

38.2 Configuration

```
# src/taskflow/config.py
from __future__ import annotations
from functools import lru_cache
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file         = ".env",
        env_file_encoding = "utf-8",
        case_sensitive   = False,
    )

    # Application
    app_name:        str  = "TaskFlow"
    app_version:     str  = "1.0.0"
    debug:           bool = False

    # Database
    database_url:    str  = "sqlite:///./taskflow.db"

    # Security
    secret_key:      str  = "change-me-in-production-use-a-long-random-string"
    algorithm:       str  = "HS256"
    access_token_expire_minutes: int = 30

    # Logging
    log_level:       str  = "INFO"
    log_file:        str  = "logs/taskflow.log"

    # Queue
    max_workers:     int  = 4

    def is_production(self) -> bool:
        return not self.debug

    def validate_secrets(self) -> None:
        if self.is_production() and self.secret_key == "change-me-in-production
            raise RuntimeError("SECRET_KEY must be changed in production.")


@lru_cache
```

```python
def get_settings() -> Settings:
    """Return cached settings --- called once, reused everywhere."""
    return Settings()
```

38.3 Logging

```python
# src/taskflow/logging_config.py
import logging
import logging.handlers
import sys
from pathlib import Path
from taskflow.config import get_settings


def setup_logging() -> None:
    settings = get_settings()
    log_path = Path(settings.log_file)
    log_path.parent.mkdir(parents=True, exist_ok=True)

    fmt = "%(asctime)s %(levelname)-8s %(name)-30s %(message)s"

    handlers: list[logging.Handler] = [
        logging.StreamHandler(sys.stdout),
        logging.handlers.RotatingFileHandler(
            log_path,
            maxBytes    = 10 * 1024 * 1024,  # 10 MB
            backupCount = 5,
            encoding    = "utf-8",
        ),
    ]

    logging.basicConfig(
        level    = getattr(logging, settings.log_level.upper()),
        format   = fmt,
        datefmt  = "%Y-%m-%d %H:%M:%S",
        handlers = handlers,
        force    = True,
    )

    # Reduce noise from libraries
    for noisy in ("uvicorn.access", "sqlalchemy.engine", "passlib"):
        logging.getLogger(noisy).setLevel(logging.WARNING)
```

```
log = logging.getLogger(__name__)
```

38.4 Database

```
# src/taskflow/database.py
from sqlalchemy import create_engine, event
from sqlalchemy.orm import DeclarativeBase, sessionmaker, Session
from taskflow.config import get_settings

settings = get_settings()

connect_args = {}
if settings.database_url.startswith("sqlite"):
    connect_args["check_same_thread"] = False

engine = create_engine(
    settings.database_url,
    connect_args = connect_args,
    pool_pre_ping = True,   # detect stale connections
    echo          = settings.debug,
)

# Enable WAL mode for SQLite (better concurrent reads)
if settings.database_url.startswith("sqlite"):
    @event.listens_for(engine, "connect")
    def set_sqlite_pragma(dbapi_conn, _):
        cursor = dbapi_conn.cursor()
        cursor.execute("PRAGMA journal_mode=WAL")
        cursor.execute("PRAGMA foreign_keys=ON")
        cursor.close()

SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)


class Base(DeclarativeBase):
    pass


def get_db() -> Session:
    """FastAPI dependency --- yields a database session."""
```

```
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

38.5 Models

```
# src/taskflow/models/user.py
from __future__ import annotations
from datetime import datetime
from sqlalchemy import Boolean, Column, DateTime, Integer, String
from sqlalchemy.orm import relationship
from taskflow.database import Base


class User(Base):
    __tablename__ = "users"

    id           = Column(Integer, primary_key=True, index=True)
    username     = Column(String(50),  unique=True, nullable=False, index=True)
    email        = Column(String(200), unique=True, nullable=False, index=True)
    hashed_password = Column(String(200), nullable=False)
    full_name    = Column(String(200), default="")
    is_active    = Column(Boolean, default=True)
    is_admin     = Column(Boolean, default=False)
    created_at   = Column(DateTime, default=datetime.utcnow)
    updated_at   = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utc

    tasks = relationship("Task", back_populates="owner",
                         cascade="all, delete-orphan")

    def __repr__(self) -> str:
        return f"User(id={self.id}, username={self.username!r})"

# src/taskflow/models/task.py
from __future__ import annotations
from datetime import datetime
from sqlalchemy import (
    Boolean, Column, DateTime, Enum, ForeignKey,
    Integer, String, Text
)
```

```
from sqlalchemy.orm import relationship
import enum
from taskflow.database import Base


class Priority(str, enum.Enum):
    low    = "low"
    medium = "medium"
    high   = "high"
    urgent = "urgent"


class Status(str, enum.Enum):
    todo        = "todo"
    in_progress = "in_progress"
    done        = "done"
    cancelled   = "cancelled"


class Task(Base):
    __tablename__ = "tasks"

    id          = Column(Integer, primary_key=True, index=True)
    title       = Column(String(200), nullable=False)
    description = Column(Text, default="")
    priority    = Column(Enum(Priority), default=Priority.medium, nullable=Fals
    status      = Column(Enum(Status),   default=Status.todo,    nullable=False
    due_date    = Column(DateTime, nullable=True)
    completed_at= Column(DateTime, nullable=True)
    owner_id    = Column(Integer, ForeignKey("users.id"), nullable=False)
    created_at  = Column(DateTime, default=datetime.utcnow)
    updated_at  = Column(DateTime, default=datetime.utcnow, onupdate=datetime.u

    owner = relationship("User", back_populates="tasks")

    def __repr__(self) -> str:
        return f"Task(id={self.id}, title={self.title!r}, status={self.status})
```

38.6 Schemas (Pydantic)

```
# src/taskflow/schemas/user.py
from __future__ import annotations
```

```
from datetime import datetime
from pydantic import BaseModel, EmailStr, Field, field_validator


class UserCreate(BaseModel):
    username:  str = Field(..., min_length=3, max_length=50,
                           pattern=r"^[a-zA-Z0-9_]+$")
    email:     EmailStr
    password:  str = Field(..., min_length=8)
    full_name: str = Field("", max_length=200)

    @field_validator("password")
    @classmethod
    def password_strength(cls, v: str) -> str:
        if not any(c.isupper() for c in v):
            raise ValueError("Password must contain an uppercase letter.")
        if not any(c.isdigit() for c in v):
            raise ValueError("Password must contain a digit.")
        return v


class UserUpdate(BaseModel):
    full_name: str | None = Field(None, max_length=200)
    email:     EmailStr | None = None


class UserOut(BaseModel):
    id:         int
    username:   str
    email:      str
    full_name:  str
    is_active:  bool
    created_at: datetime

    model_config = {"from_attributes": True}


class UserOutWithStats(UserOut):
    task_count: int = 0

# src/taskflow/schemas/task.py
from __future__ import annotations
from datetime import datetime
from typing import Optional
```

```python
from pydantic import BaseModel, Field
from taskflow.models.task import Priority, Status


class TaskCreate(BaseModel):
    title:       str      = Field(..., min_length=1, max_length=200)
    description: str      = Field("", max_length=5000)
    priority:    Priority = Priority.medium
    due_date:    Optional[datetime] = None


class TaskUpdate(BaseModel):
    title:       Optional[str]      = Field(None, min_length=1, max_length=200)
    description: Optional[str]      = Field(None, max_length=5000)
    priority:    Optional[Priority] = None
    status:      Optional[Status]   = None
    due_date:    Optional[datetime] = None


class TaskOut(BaseModel):
    id:           int
    title:        str
    description:  str
    priority:     Priority
    status:       Status
    due_date:     Optional[datetime]
    completed_at: Optional[datetime]
    owner_id:     int
    created_at:   datetime
    updated_at:   datetime

    model_config = {"from_attributes": True}
```

38.7 Services

```python
# src/taskflow/services/auth_service.py
from __future__ import annotations
from datetime import datetime, timedelta
import logging
from jose import JWTError, jwt
from passlib.context import CryptContext
from sqlalchemy.orm import Session
```

```
from taskflow.config import get_settings
from taskflow.models.user import User

log      = logging.getLogger(__name__)
settings = get_settings()
pwd_ctx  = CryptContext(schemes=["bcrypt"], deprecated="auto")


def hash_password(password: str) -> str:
    return pwd_ctx.hash(password)


def verify_password(plain: str, hashed: str) -> bool:
    return pwd_ctx.verify(plain, hashed)


def create_access_token(data: dict, expires_delta: timedelta | None = None) -> str
    payload = data.copy()
    expire  = datetime.utcnow() + (
        expires_delta or timedelta(minutes=settings.access_token_expire_minutes)
    )
    payload["exp"] = expire
    return jwt.encode(payload, settings.secret_key, algorithm=settings.algorithm)


def decode_token(token: str) -> dict:
    try:
        return jwt.decode(token, settings.secret_key, algorithms=[settings.algorit
    except JWTError as e:
        log.warning("Token decode failed: %s", e)
        raise


def authenticate_user(db: Session, username: str, password: str) -> User | None:
    user = db.query(User).filter(User.username == username).first()
    if not user or not verify_password(password, user.hashed_password):
        log.warning("Failed login attempt for username=%r", username)
        return None
    log.info("User authenticated: %s", username)
    return user


def create_user(db: Session, username: str, email: str,
                password: str, full_name: str = "") -> User:
```

```
    user = User(
        username        = username,
        email           = email,
        hashed_password = hash_password(password),
        full_name       = full_name,
    )
    db.add(user)
    db.commit()
    db.refresh(user)
    log.info("User created: %s (id=%d)", username, user.id)
    return user


# src/taskflow/services/task_service.py
from __future__ import annotations
from datetime import datetime
import logging
from typing import Optional
from sqlalchemy.orm import Session
from taskflow.models.task import Task, Status
from taskflow.schemas.task import TaskCreate, TaskUpdate

log = logging.getLogger(__name__)


def create_task(db: Session, payload: TaskCreate, owner_id: int) -> Task:
    task = Task(**payload.model_dump(), owner_id=owner_id)
    db.add(task)
    db.commit()
    db.refresh(task)
    log.info("Task created: id=%d owner=%d", task.id, owner_id)
    return task


def get_task(db: Session, task_id: int, owner_id: int) -> Optional[Task]:
    return db.query(Task).filter(
        Task.id == task_id,
        Task.owner_id == owner_id,
    ).first()


def list_tasks(
    db:       Session,
    owner_id: int,
    status:   Optional[Status] = None,
```

```
    priority: Optional[str]    = None,
    skip:     int = 0,
    limit:    int = 50,
) -> list[Task]:
    q = db.query(Task).filter(Task.owner_id == owner_id)
    if status:
        q = q.filter(Task.status == status)
    if priority:
        q = q.filter(Task.priority == priority)
    return q.order_by(Task.created_at.desc()).offset(skip).limit(limit).all()


def update_task(db: Session, task: Task, payload: TaskUpdate) -> Task:
    updates = payload.model_dump(exclude_unset=True)
    for field, value in updates.items():
        setattr(task, field, value)
    # Auto-set completed_at when status becomes "done"
    if updates.get("status") == Status.done and task.completed_at is None:
        task.completed_at = datetime.utcnow()
    elif updates.get("status") and updates["status"] != Status.done:
        task.completed_at = None
    task.updated_at = datetime.utcnow()
    db.commit()
    db.refresh(task)
    log.info("Task updated: id=%d", task.id)
    return task


def delete_task(db: Session, task: Task) -> None:
    db.delete(task)
    db.commit()
    log.info("Task deleted: id=%d", task.id)


def get_stats(db: Session, owner_id: int) -> dict:
    from sqlalchemy import func
    from taskflow.models.task import Priority

    rows = (
        db.query(Task.status, Task.priority, func.count(Task.id))
        .filter(Task.owner_id == owner_id)
        .group_by(Task.status, Task.priority)
        .all()
    )
```

```
    stats: dict = {"total": 0, "by_status": {}, "by_priority": {}}
    for status, priority, count in rows:
        stats["total"] += count
        stats["by_status"][status]     = stats["by_status"].get(status, 0) + co
        stats["by_priority"][priority] = stats["by_priority"].get(priority, 0)
    return stats
```

38.8 Dependencies

```
# src/taskflow/dependencies.py
from __future__ import annotations
import logging
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError
from sqlalchemy.orm import Session
from taskflow.database import get_db
from taskflow.models.user import User
from taskflow.services.auth_service import decode_token

log           = logging.getLogger(__name__)
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/token")


def get_current_user(
    token: str     = Depends(oauth2_scheme),
    db:    Session = Depends(get_db),
) -> User:
    credentials_exception = HTTPException(
        status_code = status.HTTP_401_UNAUTHORIZED,
        detail      = "Could not validate credentials",
        headers     = {"WWW-Authenticate": "Bearer"},
    )
    try:
        payload  = decode_token(token)
        username = payload.get("sub")
        if not username:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = db.query(User).filter(User.username == username).first()
```

```
    if not user:
        raise credentials_exception
    if not user.is_active:
        raise HTTPException(status_code=400, detail="Inactive user")
    return user


def get_admin_user(current_user: User = Depends(get_current_user)) -> User:
    if not current_user.is_admin:
        raise HTTPException(status_code=403, detail="Admin access required")
    return current_user
```

38.9 Routers

```
# src/taskflow/routers/auth.py
from __future__ import annotations
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from taskflow.database import get_db
from taskflow.schemas.user import UserCreate, UserOut
from taskflow.services import auth_service

router = APIRouter(prefix="/auth", tags=["Authentication"])


@router.post("/register", response_model=UserOut, status_code=201)
def register(payload: UserCreate, db: Session = Depends(get_db)):
    """Register a new user account."""
    if db.query(auth_service.User).filter_by(username=payload.username).first():
        raise HTTPException(status_code=400, detail="Username already taken")
    if db.query(auth_service.User).filter_by(email=payload.email).first():
        raise HTTPException(status_code=400, detail="Email already registered")
    user = auth_service.create_user(
        db, payload.username, payload.email,
        payload.password, payload.full_name
    )
    return user


@router.post("/token")
def login(form: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_d
```

```
    """Obtain a JWT access token."""
    user = auth_service.authenticate_user(db, form.username, form.password)
    if not user:
        raise HTTPException(
            status_code = status.HTTP_401_UNAUTHORIZED,
            detail      = "Incorrect username or password",
            headers     = {"WWW-Authenticate": "Bearer"},
        )
    token = auth_service.create_access_token({"sub": user.username})
    return {"access_token": token, "token_type": "bearer"}


# src/taskflow/routers/tasks.py
from __future__ import annotations
from typing import Optional
from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlalchemy.orm import Session
from taskflow.database import get_db
from taskflow.dependencies import get_current_user
from taskflow.models.task import Priority, Status
from taskflow.models.user import User
from taskflow.schemas.task import TaskCreate, TaskOut, TaskUpdate
from taskflow.services import task_service

router = APIRouter(prefix="/tasks", tags=["Tasks"])


@router.get("/", response_model=list[TaskOut])
def list_tasks(
    task_status: Optional[Status]   = Query(None, alias="status"),
    priority:    Optional[Priority] = None,
    skip:        int                = Query(0, ge=0),
    limit:       int                = Query(50, ge=1, le=200),
    current_user: User              = Depends(get_current_user),
    db:           Session           = Depends(get_db),
):
    """List all tasks for the current user."""
    return task_service.list_tasks(
        db, current_user.id, task_status, priority, skip, limit
    )


@router.get("/stats")
def task_stats(
    current_user: User    = Depends(get_current_user),
```

```
    db:           Session = Depends(get_db),
):
    """Return task statistics for the current user."""
    return task_service.get_stats(db, current_user.id)


@router.get("/{task_id}", response_model=TaskOut)
def get_task(
    task_id:      int,
    current_user: User    = Depends(get_current_user),
    db:           Session = Depends(get_db),
):
    """Get a specific task by ID."""
    task = task_service.get_task(db, task_id, current_user.id)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    return task


@router.post("/", response_model=TaskOut, status_code=201)
def create_task(
    payload:      TaskCreate,
    current_user: User    = Depends(get_current_user),
    db:           Session = Depends(get_db),
):
    """Create a new task."""
    return task_service.create_task(db, payload, current_user.id)


@router.patch("/{task_id}", response_model=TaskOut)
def update_task(
    task_id:      int,
    payload:      TaskUpdate,
    current_user: User    = Depends(get_current_user),
    db:           Session = Depends(get_db),
):
    """Partially update a task."""
    task = task_service.get_task(db, task_id, current_user.id)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    return task_service.update_task(db, task, payload)


@router.delete("/{task_id}", status_code=204)
```

```
def delete_task(
    task_id:      int,
    current_user: User    = Depends(get_current_user),
    db:           Session = Depends(get_db),
):
    """Delete a task."""
    task = task_service.get_task(db, task_id, current_user.id)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    task_service.delete_task(db, task)
```

38.10 Background Queue

```
# src/taskflow/services/queue.py
"""
A simple in-process background task queue using threading.
For production, replace with Celery + Redis or RQ.
"""
from __future__ import annotations
import logging
import queue
import threading
from dataclasses import dataclass, field
from datetime import datetime
from typing import Any, Callable
from uuid import uuid4

log = logging.getLogger(__name__)


@dataclass
class Job:
    id:         str      = field(default_factory=lambda: str(uuid4()))
    func:       Callable = field(repr=False)
    args:       tuple    = ()
    kwargs:     dict     = field(default_factory=dict)
    created_at: datetime = field(default_factory=datetime.utcnow)
    status:     str      = "pending"
    result:     Any      = None
    error:      str | None = None
```

```
class TaskQueue:
    """Thread-pool backed task queue."""

    def __init__(self, workers: int = 4):
        self._queue   = queue.Queue()
        self._jobs: dict[str, Job] = {}
        self._lock    = threading.Lock()
        self._workers = [
            threading.Thread(target=self._worker, daemon=True, name=f"worker-{i}")
            for i in range(workers)
        ]
        for w in self._workers:
            w.start()
        log.info("TaskQueue started with %d workers", workers)

    def enqueue(self, func: Callable, *args, **kwargs) -> str:
        job = Job(func=func, args=args, kwargs=kwargs)
        with self._lock:
            self._jobs[job.id] = job
        self._queue.put(job)
        log.debug("Enqueued job %s: %s", job.id, func.__name__)
        return job.id

    def get_job(self, job_id: str) -> Job | None:
        with self._lock:
            return self._jobs.get(job_id)

    def _worker(self) -> None:
        while True:
            job: Job = self._queue.get()
            log.debug("Worker %s processing job %s", threading.current_thread().na
            try:
                with self._lock:
                    job.status = "running"
                result = job.func(*job.args, **job.kwargs)
                with self._lock:
                    job.status = "done"
                    job.result = result
                log.info("Job %s completed", job.id)
            except Exception as e:
                with self._lock:
                    job.status = "failed"
                    job.error  = str(e)
                log.error("Job %s failed: %s", job.id, e)
```

```
            finally:
                self._queue.task_done()


# Singleton queue --- shared across the app
_queue: TaskQueue | None = None


def get_queue() -> TaskQueue:
    global _queue
    if _queue is None:
        from taskflow.config import get_settings
        _queue = TaskQueue(workers=get_settings().max_workers)
    return _queue
```

38.11 Main Application

```
# src/taskflow/main.py
from __future__ import annotations
import logging
from contextlib import asynccontextmanager
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
from taskflow.config import get_settings
from taskflow.database import Base, engine
from taskflow.logging_config import setup_logging
from taskflow.routers import auth, tasks
from taskflow.services.queue import get_queue

setup_logging()
log      = logging.getLogger(__name__)
settings = get_settings()


@asynccontextmanager
async def lifespan(app: FastAPI):
    """Run on startup and shutdown."""
    log.info("Starting %s v%s", settings.app_name, settings.app_version)
    Base.metadata.create_all(bind=engine)
    get_queue()    # initialize background queue
    yield
```

```
    log.info("Shutting down %s", settings.app_name)


app = FastAPI(
    title       = settings.app_name,
    version     = settings.app_version,
    description = "A production-ready task management API.",
    lifespan    = lifespan,
    docs_url    = "/docs",
    redoc_url   = "/redoc",
)

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins     = ["*"],
    allow_credentials = True,
    allow_methods     = ["*"],
    allow_headers     = ["*"],
)

# Routers
app.include_router(auth.router)
app.include_router(tasks.router)


@app.get("/", tags=["Health"])
def root():
    return {"name": settings.app_name, "version": settings.app_version, "status":


@app.get("/health", tags=["Health"])
def health():
    return {"status": "healthy"}


@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    log.error("Unhandled exception on %s: %s", request.url, exc, exc_info=True)
    return JSONResponse(status_code=500, content={"detail": "Internal server error
```

38.12 Tests

```
# tests/conftest.py
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from taskflow.database import Base, get_db
from taskflow.main import app
from taskflow.services.auth_service import create_user

TEST_DB_URL = "sqlite:///./test.db"
engine      = create_engine(TEST_DB_URL, connect_args={"check_same_thread": Fal
TestSession = sessionmaker(bind=engine)


@pytest.fixture(autouse=True)
def setup_db():
    Base.metadata.create_all(engine)
    yield
    Base.metadata.drop_all(engine)


@pytest.fixture
def db():
    session = TestSession()
    try:
        yield session
    finally:
        session.close()


@pytest.fixture
def client(db):
    def override_get_db():
        try:
            yield db
        finally:
            pass
    app.dependency_overrides[get_db] = override_get_db
    with TestClient(app) as c:
        yield c
    app.dependency_overrides.clear()
```

```python
@pytest.fixture
def test_user(db):
    return create_user(db, "testuser", "test@example.com", "Password1", "Test User

@pytest.fixture
def auth_headers(client, test_user):
    response = client.post("/auth/token", data={
        "username": "testuser",
        "password": "Password1",
    })
    token = response.json()["access_token"]
    return {"Authorization": f"Bearer {token}"}

# tests/test_auth.py
import pytest

def test_register_success(client):
    r = client.post("/auth/register", json={
        "username": "alice",
        "email":    "alice@example.com",
        "password": "Password1",
    })
    assert r.status_code == 201
    data = r.json()
    assert data["username"] == "alice"
    assert "hashed_password" not in data

def test_register_duplicate_username(client, test_user):
    r = client.post("/auth/register", json={
        "username": "testuser",
        "email":    "other@example.com",
        "password": "Password1",
    })
    assert r.status_code == 400

def test_login_success(client, test_user):
    r = client.post("/auth/token", data={
        "username": "testuser",
        "password": "Password1",
```

```
    })
    assert r.status_code == 200
    assert "access_token" in r.json()
    assert r.json()["token_type"] == "bearer"


def test_login_wrong_password(client, test_user):
    r = client.post("/auth/token", data={
        "username": "testuser",
        "password": "WrongPassword1",
    })
    assert r.status_code == 401


def test_protected_route_without_token(client):
    r = client.get("/tasks/")
    assert r.status_code == 401


def test_protected_route_with_token(client, auth_headers):
    r = client.get("/tasks/", headers=auth_headers)
    assert r.status_code == 200
```

```
# tests/test_tasks.py
import pytest


@pytest.fixture
def created_task(client, auth_headers):
    r = client.post("/tasks/", headers=auth_headers, json={
        "title":    "Write tests",
        "priority": "high",
    })
    assert r.status_code == 201
    return r.json()


def test_create_task(client, auth_headers):
    r = client.post("/tasks/", headers=auth_headers, json={
        "title":       "Learn Python",
        "description": "Read the book",
        "priority":    "high",
    })
    assert r.status_code == 201
```

```
    data = r.json()
    assert data["title"]    == "Learn Python"
    assert data["priority"] == "high"
    assert data["status"]   == "todo"


def test_list_tasks_empty(client, auth_headers):
    r = client.get("/tasks/", headers=auth_headers)
    assert r.status_code == 200
    assert r.json() == []


def test_list_tasks_with_filter(client, auth_headers, created_task):
    r = client.get("/tasks/?status=todo", headers=auth_headers)
    assert r.status_code == 200
    assert len(r.json()) == 1

    r = client.get("/tasks/?status=done", headers=auth_headers)
    assert r.status_code == 200
    assert len(r.json()) == 0


def test_get_task(client, auth_headers, created_task):
    r = client.get(f"/tasks/{created_task['id']}", headers=auth_headers)
    assert r.status_code == 200
    assert r.json()["id"] == created_task["id"]


def test_get_task_not_found(client, auth_headers):
    r = client.get("/tasks/99999", headers=auth_headers)
    assert r.status_code == 404


def test_update_task_status(client, auth_headers, created_task):
    r = client.patch(
        f"/tasks/{created_task['id']}",
        headers=auth_headers,
        json={"status": "done"},
    )
    assert r.status_code == 200
    data = r.json()
    assert data["status"]       == "done"
    assert data["completed_at"] is not None
```

```python
def test_delete_task(client, auth_headers, created_task):
    r = client.delete(f"/tasks/{created_task['id']}", headers=auth_headers)
    assert r.status_code == 204

    r = client.get(f"/tasks/{created_task['id']}", headers=auth_headers)
    assert r.status_code == 404


def test_task_stats(client, auth_headers):
    client.post("/tasks/", headers=auth_headers, json={"title": "Task A", "prio
    client.post("/tasks/", headers=auth_headers, json={"title": "Task B", "prio
    client.post("/tasks/", headers=auth_headers, json={"title": "Task C", "prio

    r = client.get("/tasks/stats", headers=auth_headers)
    assert r.status_code == 200
    stats = r.json()
    assert stats["total"] == 3
    assert stats["by_status"]["todo"] == 3
    assert stats["by_priority"]["high"] == 2
```

38.13 Dockerfile

```dockerfile
# Dockerfile
FROM python:3.12-slim

WORKDIR /app

# Install dependencies first (cached layer)
COPY pyproject.toml .
RUN pip install --no-cache-dir build && \
    pip install --no-cache-dir -e ".[prod]"

# Copy application code
COPY src/ src/

# Non-root user for security
RUN useradd -m -u 1000 appuser && chown -R appuser:appuser /app
USER appuser

EXPOSE 8000
```

```
CMD ["uvicorn", "taskflow.main:app", "--host", "0.0.0.0", "--port", "8000"]

# docker-compose.yml
version: "3.9"
services:
  api:
    build: .
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=sqlite:///./taskflow.db
      - SECRET_KEY=${SECRET_KEY}
      - LOG_LEVEL=INFO
    volumes:
      - ./data:/app/data
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

38.14 GitHub Actions CI/CD

```
# .github/workflows/ci.yml
name: CI

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.11", "3.12"]

    steps:
      - uses: actions/checkout@v4
```

```yaml
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}

      - name: Install dependencies
        run: pip install -e ".[dev]"

      - name: Lint
        run: ruff check src/ tests/

      - name: Type check
        run: mypy src/taskflow

      - name: Test
        run: pytest tests/ --cov=src/taskflow --cov-report=xml --cov-fail-under

      - name: Upload coverage
        uses: codecov/codecov-action@v4
        with:
          file: ./coverage.xml

  docker:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - uses: actions/checkout@v4

      - name: Build Docker image
        run: docker build -t taskflow:latest .

      - name: Smoke test
        run: |
          docker run -d -p 8000:8000 --name taskflow_test \
            -e SECRET_KEY=test-secret-key taskflow:latest
          sleep 5
          curl -f http://localhost:8000/health
          docker stop taskflow_test
```

38.15 `pyproject.toml`

```
[build-system]
requires      = ["hatchling"]
build-backend = "hatchling.build"

[project]
name            = "taskflow"
version         = "1.0.0"
description     = "Production-ready task management API"
requires-python = ">=3.11"
dependencies    = [
    "fastapi>=0.110",
    "uvicorn[standard]>=0.29",
    "sqlalchemy>=2.0",
    "alembic>=1.13",
    "pydantic>=2.6",
    "pydantic-settings>=2.2",
    "python-jose[cryptography]>=3.3",
    "passlib[bcrypt]>=1.7",
    "python-multipart>=0.0.9",
]

[project.optional-dependencies]
dev = [
    "pytest>=8",
    "pytest-cov>=5",
    "httpx>=0.27",
    "mypy>=1.9",
    "ruff>=0.3",
]
prod = [
    "gunicorn>=22",
]

[project.scripts]
taskflow = "taskflow.main:app"

[tool.hatch.build.targets.wheel]
packages = ["src/taskflow"]

[tool.pytest.ini_options]
testpaths = ["tests"]
addopts   = "-v --tb=short"
```

```
[tool.mypy]
python_version         = "3.12"
strict                 = true
ignore_missing_imports = true

[tool.ruff.lint]
select = ["E", "F", "I", "N", "UP"]
```

38.16 Running the Application

```
# 1. Clone and set up
git clone https://github.com/you/taskflow
cd taskflow
python -m venv .venv
source .venv/bin/activate          # Windows: .venv\Scripts\activate
pip install -e ".[dev]"

# 2. Configure
cp .env.example .env
# Edit .env --- set SECRET_KEY to a long random string

# 3. Run
uvicorn taskflow.main:app --reload
# -> http://localhost:8000/docs  (Swagger UI)
# -> http://localhost:8000/redoc (ReDoc)

# 4. Test
pytest tests/ -v --cov=src/taskflow

# 5. Docker
docker-compose up --build
```

Visit `http://localhost:8000/docs` — you'll see the full interactive API:

1. Click `POST /auth/register` -> register a user
2. Click `POST /auth/token` -> log in, copy the token
3. Click the **Authorize** button -> paste the token
4. Now all task endpoints are accessible

38.17 What This Project Used

Every major topic from this book appeared in TaskFlow:

Chapter	Used in TaskFlow
Variables, types, f-strings	Throughout
Conditionals, loops	Services, queue worker
Functions, scope	All service functions
Classes, OOP	Models, schemas, services
Inheritance, polymorphism	SQLAlchemy model hierarchy
Special methods	`__repr__` on all models
Error handling	All routers, auth service
Comprehensions	Stats aggregation, list filtering
Modules, packages	Full package structure
Standard library	`logging`, `threading`, `queue`, `datetime`, `uuid`
Type hints	Every function signature
Dataclasses	`Job` dataclass in queue
Decorators	`@lru_cache`, `@field_validator`, `@asynccontextmanager`
Generators	SQLAlchemy `yield` sessions
Context managers	`get_db()` yield, `lifespan()`
Functional programming	`model_dump(exclude_unset=True)`
Concurrency	`threading.Thread` worker pool
Testing	Full pytest suite with fixtures
Databases	SQLAlchemy ORM + SQLite
Web development	FastAPI with Pydantic
Packaging	`pyproject.toml`
Professional practices	Logging, env vars, clean code

38.18 Where to Go From Here

You've finished the book. Here's what to do next:

Deploy TaskFlow. Take this exact project, set `DATABASE_URL` to a real PostgreSQL instance (free tier on Supabase or Railway), add a `SECRET_KEY`, and run `docker-compose up`. You'll have a live API in 10 minutes.

Add features. Tags on tasks. Task assignments (multiple users). Email notifications when tasks are due. A dashboard page with charts using Chart.js. All of these use skills from this book.

Read excellent code. The best way to improve is to read code written by expert Python developers. Study the source of `fastapi`, `pydantic`, `httpx`, and `rich`. Each one teaches you something new.

Contribute to open source. Find a project you use, fix a bug, improve the docs, add a test. Your first PR is the hardest. After that, it becomes normal.

Build something real. Every skill in this book was taught through building — and building is how you retain it. Pick one idea you genuinely care about and build it completely. Not a tutorial project. Something you'd actually use.

Python is a tool. What matters is what you build with it.

Go build something worth building.

38.19 Final Summary: What You Now Know

You started with `print("Hello, World!")`. You ended with a production API, complete with authentication, a background queue, structured logging, a test suite, and Docker deployment.

Along the way you learned:

- **The fundamentals** — variables, operators, conditionals, loops, functions, files, error handling
- **Data structures** — lists, tuples, sets, dictionaries, strings — and when to use each
- **Object-oriented programming** — classes, inheritance, polymorphism, encapsulation, special methods, properties, dataclasses
- **Functional programming** — pure functions, closures, decorators, generators, context managers, `map`/`filter`/`reduce`, `functools`
- **Advanced types** — type hints, `TypeVar`, `Generic`, `Protocol`, `NamedTuple`, `TypedDict`
- **Concurrency** — threading, multiprocessing, asyncio — and when to use each
- **Testing** — pytest, fixtures, parametrize, mocking, coverage
- **Debugging** — `pdb`, `breakpoint()`, `logging`, tracebacks, profiling
- **Performance** — `timeit`, `cProfile`, algorithmic complexity, numpy
- **Databases** — SQLite, PostgreSQL, SQLAlchemy, migrations
- **Web development** — Flask, FastAPI, REST, JWT authentication
- **Data science** — numpy, pandas, matplotlib, scikit-learn
- **Packaging** — `pyproject.toml`, building, publishing to PyPI
- **Professional practices** — style, docstrings, code review, git, environment variables

- **Internals** — bytecode, the GIL, memory management, descriptors, metaclasses
- **Metaprogramming** — class factories, `exec`/`eval`, dynamic attribute generation

You are no longer a beginner. You are a Python developer.

The language is the same. What you do with it — that's up to you.

Chapter 39

Chapter 39: Web Scraping

The web is the largest database ever built. Billions of pages of prices, news, job listings, sports scores, research data — most of it accessible to anyone with a browser. Web scraping is the skill of collecting that data automatically, at scale, with Python.

This chapter teaches you to scrape the right way: respectfully, robustly, and legally.

39.1 How the Web Works (What You Need to Know)

When you visit a website, your browser: 1. Sends an HTTP `GET` request to the server 2. Receives HTML — plain text that describes the page structure 3. Parses that HTML and renders it visually

Web scraping replaces steps 1 and 3 with Python. You send the same HTTP request and parse the HTML yourself — extracting just the data you need.

```
Browser:  GET https://example.com/products
Server:   -> 200 OK + HTML page
Browser:  Parse HTML -> render page visually

Python:   GET https://example.com/products
Server:   -> 200 OK + HTML page
Python:   Parse HTML -> extract product names and prices
```

39.2 Before You Scrape — Check These Things

1. **`robots.txt`** — Every site has `https://site.com/robots.txt`. It specifies which paths bots may visit. Respect it.
2. **Terms of Service** — Some sites explicitly prohibit scraping. Read the ToS.
3. **Rate limits** — Don't hammer servers. Add delays between requests.
4. **APIs first** — Many sites have official APIs that are easier, more reliable, and explicitly allowed.

```
import requests

# Always check robots.txt first
r = requests.get("https://books.toscrape.com/robots.txt")
print(r.text)
```

39.3 `requests` — Fetching Pages

```
pip install requests beautifulsoup4 lxml
```

```
import requests

# Basic GET request
response = requests.get("https://books.toscrape.com/")

print(response.status_code)     # 200
print(response.headers["Content-Type"])  # text/html; charset=utf-8
print(len(response.text))       # length of HTML
```

39.3.1 Setting headers to look like a browser

Some servers block requests that don't look like a browser. Add a `User-Agent` header:

```
headers = {
    "User-Agent": (
        "Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
        "AppleWebKit/537.36 (KHTML, like Gecko) "
        "Chrome/120.0.0.0 Safari/537.36"
    )
}

response = requests.get("https://books.toscrape.com/", headers=headers)
```

39.3.2 Handling errors

```
import requests
from requests.exceptions import RequestException

def fetch_page(url: str, timeout: int = 10) -> str | None:
    """Fetch a URL and return its HTML, or None on failure."""
    try:
        response = requests.get(url, timeout=timeout, headers={
            "User-Agent": "MyBot/1.0 (educational purposes)"
        })
        response.raise_for_status()    # raises HTTPError for 4xx/5xx
        return response.text
    except requests.HTTPError as e:
        print(f"HTTP error {e.response.status_code}: {url}")
    except requests.ConnectionError:
        print(f"Cannot connect: {url}")
    except requests.Timeout:
        print(f"Timeout: {url}")
    except RequestException as e:
        print(f"Request failed: {e}")
    return None
```

39.4 `BeautifulSoup` — Parsing HTML

BeautifulSoup parses HTML and lets you navigate and search the element tree:

```
from bs4 import BeautifulSoup

html = """
<html>
  <body>
    <h1 class="title">Python Books</h1>
    <ul id="book-list">
      <li class="book" data-price="29.99">
        <a href="/book/1">Learn Python</a>
      </li>
      <li class="book" data-price="39.99">
        <a href="/book/2">Python Tricks</a>
      </li>
    </ul>
  </body>
</html>
```

```
"""

soup = BeautifulSoup(html, "lxml")
```

39.4.1 Finding elements

```
# Find the first matching element
h1 = soup.find("h1")
print(h1.text)               # Python Books
print(h1["class"])           # ['title']

# Find all matching elements
books = soup.find_all("li", class_="book")
print(len(books))            # 2

# CSS selectors (very powerful)
books = soup.select("ul#book-list li.book")
titles = soup.select("li.book a")
first  = soup.select_one("li.book")

# Navigate the tree
ul    = soup.find("ul")
first_li = ul.find("li")       # first child li
all_li   = ul.find_all("li")   # all child lis
parent   = first_li.parent     # the ul element
siblings = list(first_li.next_siblings)
```

39.4.2 Extracting data

```
for book in soup.find_all("li", class_="book"):
    # Text content
    title = book.find("a").text.strip()

    # Attributes
    price = book["data-price"]
    href  = book.find("a")["href"]

    print(f"{title}: ${price} --- {href}")
```

Output:

```
Learn Python: $29.99 --- /book/1
Python Tricks: $39.99 --- /book/2
```

39.5 A Real Scraper — Books to Scrape

`books.toscrape.com` is a practice site built specifically for learning web scraping. Let's scrape its book catalogue:

```
import requests
import time
from bs4 import BeautifulSoup
from dataclasses import dataclass


@dataclass
class Book:
    title:  str
    price:  float
    rating: int
    url:    str


RATING_MAP = {"One": 1, "Two": 2, "Three": 3, "Four": 4, "Five": 5}
BASE_URL   = "https://books.toscrape.com"


def parse_book(article) -> Book:
    """Extract a Book from a catalogue <article> element."""
    title  = article.find("h3").find("a")["title"]
    price  = float(article.find("p", class_="price_color").text.strip("£Â"))
    rating = RATING_MAP.get(article.find("p", class_="star-rating")["class"][1]
    href   = article.find("h3").find("a")["href"]
    url    = f"{BASE_URL}/catalogue/{href.replace('../', '')}"
    return Book(title=title, price=price, rating=rating, url=url)


def scrape_page(url: str) -> tuple[list[Book], str | None]:
    """
    Scrape one catalogue page.
    Returns (books_on_page, next_page_url or None).
    """
    response = requests.get(url, timeout=10)
    response.raise_for_status()
    soup = BeautifulSoup(response.text, "lxml")

    books = [parse_book(a) for a in soup.select("article.product_pod")]

    # Get the "next" button URL if it exists
```

```
    next_btn = soup.select_one("li.next a")
    next_url = None
    if next_btn:
        current_path = url.rsplit("/", 1)[0]
        next_url = f"{current_path}/{next_btn['href']}"

    return books, next_url


def scrape_catalogue(
    start_url: str = f"{BASE_URL}/catalogue/page-1.html",
    max_pages:  int = 5,
    delay:      float = 1.0,
) -> list[Book]:
    """Scrape up to max_pages of the catalogue."""
    all_books = []
    url       = start_url
    page      = 1

    while url and page <= max_pages:
        print(f"Scraping page {page}: {url}")
        books, next_url = scrape_page(url)
        all_books.extend(books)
        print(f"  Found {len(books)} books (total: {len(all_books)})")

        url   = next_url
        page += 1
        if url:
            time.sleep(delay)    # be polite --- don't hammer the server

    return all_books


def print_summary(books: list[Book]) -> None:
    if not books:
        print("No books found.")
        return

    prices  = [b.price for b in books]
    ratings = [b.rating for b in books]

    print(f"\n{'—'*50}")
    print(f"Total books:    {len(books)}")
    print(f"Avg price:      £{sum(prices)/len(prices):.2f}")
```

```
    print(f"Cheapest:       £{min(prices):.2f}")
    print(f"Most expensive: £{max(prices):.2f}")
    print(f"Avg rating:     {sum(ratings)/len(ratings):.1f}/5")

    print("\nTop 5 highest-rated books:")
    top = sorted(books, key=lambda b: (-b.rating, b.price))[:5]
    for book in top:
        print(f"  {'[*]'*book.rating:<5} £{book.price:.2f}  {book.title[:50]}")


if __name__ == "__main__":
    books = scrape_catalogue(max_pages=3)
    print_summary(books)
```

Output:

```
Scraping page 1: https://books.toscrape.com/catalogue/page-1.html
  Found 20 books (total: 20)
Scraping page 2: ...
  Found 20 books (total: 40)
Scraping page 3: ...
  Found 20 books (total: 60)

──────────────────────────────────────────────────

Total books:    60
Avg price:      £35.14
Cheapest:       £10.00
Most expensive: £59.69
Avg rating:     2.9/5

Top 5 highest-rated books:
  [*][*][*][*][*] £17.46  It's Only the Himalayas
  [*][*][*][*][*] £45.17  Full Moon over Noah's Ark
  ...
```

39.6 Handling Pagination

The pattern above — extract data, find the "next" link, repeat — handles any paginated site:

```
def paginate(start_url: str, get_next_url, scrape_page_fn, max_pages=None):
    """
    Generic pagination driver.
```

```
    Args:
        start_url:     First page URL.
        get_next_url:  soup -> next URL or None.
        scrape_page_fn:soup -> list of items.
        max_pages:     Stop after this many pages (None = all).
    """
    url    = start_url
    page   = 0
    items  = []

    while url:
        page += 1
        if max_pages and page > max_pages:
            break

        response = requests.get(url, timeout=10)
        soup     = BeautifulSoup(response.text, "lxml")

        page_items = scrape_page_fn(soup)
        items.extend(page_items)

        url = get_next_url(soup)
        if url:
            time.sleep(1.0)

    return items
```

39.7 Storing Scraped Data

```
import csv
import json
from pathlib import Path
from dataclasses import asdict

def save_to_csv(books: list[Book], path: str = "books.csv") -> None:
    with open(path, "w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=["title", "price", "rating", "url"])
        writer.writeheader()
        writer.writerows(asdict(b) for b in books)
    print(f"Saved {len(books)} books to {path}")
```

```
def save_to_json(books: list[Book], path: str = "books.json") -> None:
    data = [asdict(b) for b in books]
    Path(path).write_text(json.dumps(data, indent=2), encoding="utf-8")
    print(f"Saved {len(books)} books to {path}")


def save_to_sqlite(books: list[Book], db_path: str = "books.db") -> None:
    import sqlite3
    conn = sqlite3.connect(db_path)
    conn.execute("""
        CREATE TABLE IF NOT EXISTS books (
            title  TEXT,
            price  REAL,
            rating INTEGER,
            url    TEXT UNIQUE
        )
    """)
    conn.executemany(
        "INSERT OR IGNORE INTO books VALUES (?, ?, ?, ?)",
        [(b.title, b.price, b.rating, b.url) for b in books]
    )
    conn.commit()
    conn.close()
    print(f"Saved {len(books)} books to {db_path}")
```

39.8 Scraping with Sessions and Cookies

Some sites require you to log in first. Use a `requests.Session` to persist cookies:

```
import requests

session = requests.Session()
session.headers.update({
    "User-Agent": "Mozilla/5.0 ..."
})

# Log in
login_response = session.post("https://example.com/login", data={
    "username": "myuser",
    "password": "mypassword",
})
```

```
login_response.raise_for_status()

# All subsequent requests use the session (cookies maintained)
page = session.get("https://example.com/dashboard")
```

39.9 Dynamic Pages with Selenium

Some sites load content with JavaScript — the initial HTML is nearly empty, and a script populates it after the page loads. `requests` + `BeautifulSoup` can't handle this. You need a real browser.

```
pip install selenium
# Also download ChromeDriver: https://chromedriver.chromium.org/
```

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.chrome.options import Options
import time

# Run headless --- no visible browser window
options = Options()
options.add_argument("--headless")
options.add_argument("--no-sandbox")
options.add_argument("--disable-dev-shm-usage")

driver = webdriver.Chrome(options=options)

try:
    driver.get("https://quotes.toscrape.com/js/")

    # Wait for the quotes to load (up to 10 seconds)
    WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.CLASS_NAME, "quote"))
    )

    quotes = driver.find_elements(By.CLASS_NAME, "quote")
    for quote in quotes:
        text   = quote.find_element(By.CLASS_NAME, "text").text
        author = quote.find_element(By.CLASS_NAME, "author").text
        print(f"{author}: {text[:60]}")
```

```
finally:
    driver.quit()    # always close the browser
```

Use Selenium only when necessary — it's slower and more complex than `requests` + `BeautifulSoup`. For most sites, static scraping is enough.

A lighter alternative is `playwright`:

```
pip install playwright
playwright install chromium
```

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page    = browser.new_page()
    page.goto("https://quotes.toscrape.com/js/")
    page.wait_for_selector(".quote")

    quotes = page.query_selector_all(".quote")
    for q in quotes:
        text   = q.query_selector(".text").inner_text()
        author = q.query_selector(".author").inner_text()
        print(f"{author}: {text[:60]}")

    browser.close()
```

39.10 Rate Limiting and Politeness

A good scraper is a polite scraper:

```
import time
import random

def polite_get(url: str, min_delay: float = 1.0, max_delay: float = 3.0) -> req
    """Fetch a URL with a random delay to avoid overwhelming the server."""
    time.sleep(random.uniform(min_delay, max_delay))
    return requests.get(url, timeout=10, headers={
        "User-Agent": "ResearchBot/1.0 (contact: you@example.com)"
    })
```

Rules to follow: - **Delay between requests.** 1-3 seconds is reasonable. Never fire requests as fast as possible. - **Identify your bot.** Put a contact email in the `User-Agent` so site owners can reach you. - **Honor `Retry-After` headers.** If the server says "wait 60 seconds," wait 60 seconds. - **Cache responses.** Don't

re-fetch pages you already have. - **Scrape off-peak hours.** Less load on their servers.

39.11 Project: Job Listing Scraper

```
"""
job_scraper.py --- Scrape job listings from a practice site.
Uses: https://realpython.github.io/fake-jobs/
"""
import csv
import time
import requests
from bs4 import BeautifulSoup
from dataclasses import dataclass, asdict, field
from datetime import datetime
from pathlib import Path


BASE_URL = "https://realpython.github.io/fake-jobs/"


@dataclass
class Job:
    title:    str
    company:  str
    location: str
    date:     str
    url:      str = ""
    scraped_at: str = field(default_factory=lambda: datetime.utcnow().isoformat())


def scrape_jobs(url: str = BASE_URL) -> list[Job]:
    """Scrape all job listings from the fake-jobs site."""
    print(f"Fetching: {url}")
    response = requests.get(url, timeout=10)
    response.raise_for_status()

    soup = BeautifulSoup(response.text, "lxml")
    jobs = []

    for card in soup.select("div.card"):
        title    = card.select_one("h2.title").text.strip()
```

```
            company  = card.select_one("h3.subtitle").text.strip()
            location = card.select_one("p.location").text.strip()
            date     = card.select_one("time")["datetime"] if card.select_one("time
            link     = card.select_one("a[href]")
            job_url  = link["href"] if link else ""

            jobs.append(Job(
                title    = title,
                company  = company,
                location = location,
                date     = date,
                url      = job_url,
            ))

    print(f"Found {len(jobs)} jobs")
    return jobs


def filter_jobs(
    jobs:     list[Job],
    keywords: list[str] = (),
    location: str = "",
) -> list[Job]:
    """Filter jobs by keyword in title or location."""
    result = jobs
    if keywords:
        kw_lower = [k.lower() for k in keywords]
        result   = [j for j in result
                    if any(k in j.title.lower() for k in kw_lower)]
    if location:
        result   = [j for j in result
                    if location.lower() in j.location.lower()]
    return result


def save_jobs(jobs: list[Job], path: str = "jobs.csv") -> None:
    with open(path, "w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=list(Job.__dataclass_fields__))
        writer.writeheader()
        writer.writerows(asdict(j) for j in jobs)
    print(f"Saved to {path}")


if __name__ == "__main__":
```

```
all_jobs = scrape_jobs()

# Filter for Python-related jobs
python_jobs = filter_jobs(all_jobs, keywords=["python", "django", "flask"])
print(f"\nPython-related jobs: {len(python_jobs)}")
for job in python_jobs[:5]:
    print(f"  {job.title} at {job.company} --- {job.location}")

save_jobs(all_jobs, "all_jobs.csv")
save_jobs(python_jobs, "python_jobs.csv")
```

39.12 What You Learned in This Chapter

- Web scraping replaces your browser — Python sends HTTP requests and parses the HTML response.
- Always check `robots.txt` and the site's Terms of Service before scraping.
- `requests.get(url)` fetches a page. `.raise_for_status()` raises on HTTP errors. Use a `User-Agent` header.
- `BeautifulSoup(html, "lxml")` parses HTML. `.find()`, `.find_all()`, `.select()`, `.select_one()` navigate the tree. `.text` extracts text. `["attr"]` extracts attributes.
- Paginate by finding the "next" link and looping.
- Store scraped data in CSV, JSON, or SQLite.
- Use `requests.Session` to persist cookies across requests (login sessions).
- Use `selenium` or `playwright` for JavaScript-rendered pages.
- Be polite: add delays, identify your bot, respect `robots.txt`.

39.13 What's Next?

Chapter 40 covers **Web Development with Flask** — building a full server-side web application with routes, HTML templates (Jinja2), forms, and a database. You've scraped data off the web; now you'll serve your own.

Chapter 40

Chapter 40: Web Development with Flask

Flask is a web framework for Python. It lets you turn Python functions into web pages and APIs with just a few lines of code.

You already know Python. With Flask, that's enough to build a real website.

40.1 What is Flask?

Flask is called a "micro-framework." Micro doesn't mean small or weak — it means Flask gives you the essentials and lets you add only what you need. No mandatory ORM, no required project structure, no magic configuration files.

Compare that to Django, which comes with everything built-in. Flask is lighter, easier to learn, and perfect for APIs and small-to-medium apps.

```
pip install flask
```

40.2 Your First Web App

```
# app.py
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def home():
    return "Hello, World!"


@app.route("/about")
def about():
    return "This is the about page."


if __name__ == "__main__":
    app.run(debug=True)
```

Run it:

```
python app.py
```

Open `http://127.0.0.1:5000/` in your browser. You'll see "Hello, World!".

That's it. That's a web server.

`@app.route("/")` maps the URL path `/` to the `home` function. When someone visits that URL, Flask calls your function and sends the return value back as the response.

`debug=True` enables auto-reload (no need to restart when you edit code) and a browser debugger on errors. Never use it in production.

40.3 Routes and URL Parameters

40.3.1 Static routes

```
@app.route("/")
def home():
    return "Home page"

@app.route("/contact")
def contact():
    return "Contact us at hello@example.com"
```

40.3.2 Dynamic routes — path parameters

```
@app.route("/user/<username>")
def profile(username):
    return f"Profile of {username}"
```

```
@app.route("/product/<int:product_id>")
def product(product_id):
    # Flask converts product_id to int for you
    return f"Product #{product_id}"

@app.route("/price/<float:amount>")
def price(amount):
    return f"Price: ${amount:.2f}"
```

Visit `/user/alice` -> “Profile of alice” Visit `/product/42` -> “Product #42”

Type converters: `string` (default), `int`, `float`, `path` (slashes allowed), `uuid`.

40.3.3 HTTP methods

By default, routes only accept `GET`. Specify `methods` to allow others:

```
from flask import Flask, request

@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        return f"Logging in as {username}"
    # GET --- show the login form
    return """
        <form method="POST">
            <input name="username" placeholder="Username">
            <input name="password" type="password" placeholder="Password">
            <button type="submit">Log in</button>
        </form>
    """
```

40.4 The `request` Object

When someone calls your route, Flask puts all request data in `request`:

```
from flask import request

@app.route("/search")
def search():
    # Query string: /search?q=python&page=2
    query = request.args.get("q", "")       # "python"
```

```
    page  = request.args.get("page", 1, type=int)  # 2
    return f"Search: '{query}', page {page}"


@app.route("/submit", methods=["POST"])
def submit():
    # Form data (application/x-www-form-urlencoded)
    name  = request.form.get("name")
    email = request.form.get("email")

    # JSON body (application/json)
    data  = request.get_json()

    # Headers
    auth  = request.headers.get("Authorization")

    # Uploaded files
    file  = request.files.get("avatar")

    return "OK"
```

40.5 HTML Templates with Jinja2

Returning HTML strings from functions gets messy fast. Flask uses **Jinja2** — a template engine that keeps HTML in separate files.

Create a `templates/` folder next to `app.py`:

```
my_app/
├── app.py
└── templates/
    ├── base.html
    ├── index.html
    └── users.html
```

`templates/base.html` — the layout every page inherits:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}My Site{% endblock %}</title>
    <style>
        body { font-family: sans-serif; max-width: 800px; margin: 0 auto; padding:
```

```
        nav a { margin-right: 1rem; }
    </style>
</head>
<body>
    <nav>
        <a href="/">Home</a>
        <a href="/users">Users</a>
    </nav>
    <hr>
    {% block content %}{% endblock %}
</body>
</html>
```

`templates/index.html` — extends the base:

```
{% extends "base.html" %}

{% block title %}Home{% endblock %}

{% block content %}
  <h1>Welcome, {{ name }}!</h1>
  <p>The date is {{ date }}.</p>

  {% if items %}
    <ul>
      {% for item in items %}
        <li>{{ item }}</li>
      {% endfor %}
    </ul>
  {% else %}
    <p>No items found.</p>
  {% endif %}
{% endblock %}
```

`app.py` — render the template:

```
from flask import Flask, render_template
from datetime import date

app = Flask(__name__)


@app.route("/")
def home():
    return render_template(
        "index.html",
```

```
        name="Alice",
        date=date.today(),
        items=["Python", "Flask", "Jinja2"],
    )
```

Jinja2 syntax: - `{{ variable }}` — output a value - `{% if condition %}...{% endif %}` — conditionals - `{% for item in list %}...{% endfor %}` — loops - `{% extends "base.html" %}` — template inheritance - `{% block name %}...{% endblock %}` — override sections - `{{ value | filter }}` — filters: `upper`, `lower`, `length`, `default("none")`, `safe`

40.6 Static Files — CSS and Images

Put static files (CSS, JS, images) in a `static/` folder:

```
my_app/
├── app.py
├── templates/
└── static/
    ├── style.css
    └── logo.png
```

Reference them in templates using `url_for`:

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<img src="{{ url_for('static', filename='logo.png') }}" alt="Logo">
```

`url_for` generates the correct URL regardless of where you deploy the app.

40.7 Forms — Reading POST Data

templates/contact.html:

```
{% extends "base.html" %}
{% block title %}Contact{% endblock %}
{% block content %}
<h1>Contact Us</h1>

{% if message %}
  <p style="color: green;">{{ message }}</p>
{% endif %}

{% for error in errors %}
```

```
  <p style="color: red;">{{ error }}</p>
{% endfor %}

<form method="POST">
  <label>Name<br>
    <input type="text" name="name" value="{{ form_data.name or '' }}">
  </label><br><br>
  <label>Email<br>
    <input type="email" name="email" value="{{ form_data.email or '' }}">
  </label><br><br>
  <label>Message<br>
    <textarea name="body">{{ form_data.body or '' }}</textarea>
  </label><br><br>
  <button type="submit">Send</button>
</form>
{% endblock %}
```

app.py:

```
@app.route("/contact", methods=["GET", "POST"])
def contact():
    message   = ""
    errors    = []
    form_data = {}

    if request.method == "POST":
        name  = request.form.get("name", "").strip()
        email = request.form.get("email", "").strip()
        body  = request.form.get("body", "").strip()
        form_data = {"name": name, "email": email, "body": body}

        if not name:
            errors.append("Name is required.")
        if not email or "@" not in email:
            errors.append("A valid email is required.")
        if not body:
            errors.append("Message cannot be empty.")

        if not errors:
            # In a real app: send an email, save to DB, etc.
            message = f"Thanks, {name}! We'll be in touch."
            form_data = {}

    return render_template(
        "contact.html",
```

```
        message=message,
        errors=errors,
        form_data=form_data,
    )
```

40.8 Flask with SQLite — A Real Task App

Let's build a task manager that stores tasks in SQLite.

```
task_app/
├── app.py
└── templates/
    ├── base.html
    └── tasks.html
```

app.py:

```
import sqlite3
from flask import Flask, render_template, request, redirect, url_for, flash
from pathlib import Path

app = Flask(__name__)
app.secret_key = "change-this-in-production"  # required for flash messages

DB_PATH = "tasks.db"


def get_db():
    """Open a database connection."""
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row   # rows behave like dicts
    return conn


def init_db():
    """Create tables if they don't exist."""
    with get_db() as conn:
        conn.execute("""
            CREATE TABLE IF NOT EXISTS tasks (
                id        INTEGER PRIMARY KEY AUTOINCREMENT,
                title     TEXT    NOT NULL,
                done      BOOLEAN NOT NULL DEFAULT 0,
                created   TEXT    NOT NULL DEFAULT (datetime('now'))
            )
```

```
        """)


@app.route("/")
def task_list():
    with get_db() as conn:
        tasks = conn.execute(
            "SELECT * FROM tasks ORDER BY done, created DESC"
        ).fetchall()
    return render_template("tasks.html", tasks=tasks)


@app.route("/add", methods=["POST"])
def add_task():
    title = request.form.get("title", "").strip()
    if not title:
        flash("Task title cannot be empty.", "error")
    else:
        with get_db() as conn:
            conn.execute("INSERT INTO tasks (title) VALUES (?)", (title,))
        flash(f"Task '{title}' added.", "success")
    return redirect(url_for("task_list"))


@app.route("/toggle/<int:task_id>", methods=["POST"])
def toggle_task(task_id):
    with get_db() as conn:
        conn.execute(
            "UPDATE tasks SET done = NOT done WHERE id = ?", (task_id,)
        )
    return redirect(url_for("task_list"))


@app.route("/delete/<int:task_id>", methods=["POST"])
def delete_task(task_id):
    with get_db() as conn:
        conn.execute("DELETE FROM tasks WHERE id = ?", (task_id,))
    flash("Task deleted.", "info")
    return redirect(url_for("task_list"))


if __name__ == "__main__":
    init_db()
    app.run(debug=True)
```

templates/tasks.html:

```
{% extends "base.html" %}
{% block title %}Tasks{% endblock %}
{% block content %}

<h1>My Tasks</h1>

{% with messages = get_flashed_messages(with_categories=true) %}
  {% for category, msg in messages %}
    <p style="color: {{ 'green' if category == 'success' else 'red' if category ==
      {{ msg }}
    </p>
  {% endfor %}
{% endwith %}

<form method="POST" action="/add">
  <input type="text" name="title" placeholder="New task..." required>
  <button type="submit">Add</button>
</form>

<ul style="list-style: none; padding: 0; margin-top: 1rem;">
  {% for task in tasks %}
  <li style="margin: 0.5rem 0; {{ 'opacity: 0.5;' if task.done }}">
    <form method="POST" action="/toggle/{{ task.id }}" style="display: inline;">
      <button type="submit" title="{{ 'Mark undone' if task.done else 'Mark done'
        {{ '[x]' if task.done else '[ ]' }}
      </button>
    </form>
    <span style="{{ 'text-decoration: line-through;' if task.done }}">
      {{ task.title }}
    </span>
    <form method="POST" action="/delete/{{ task.id }}" style="display: inline;">
      <button type="submit" onclick="return confirm('Delete this task?')"
              style="margin-left: 0.5rem; color: red;">[-]</button>
    </form>
  </li>
  {% endfor %}
</ul>

{% if not tasks %}
  <p>No tasks yet. Add one above!</p>
{% endif %}
```

```
{% endblock %}
```

40.9 Blueprints — Organizing a Larger App

When your app grows, split it into **Blueprints**:

```
my_app/
├── app.py
├── auth/
│   ├── __init__.py
│   └── routes.py
└── tasks/
    ├── __init__.py
    └── routes.py
```

auth/routes.py:

```python
from flask import Blueprint, render_template, request, redirect, url_for

auth_bp = Blueprint("auth", __name__, url_prefix="/auth")


@auth_bp.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        # handle login
        return redirect(url_for("tasks.task_list"))
    return render_template("auth/login.html")


@auth_bp.route("/logout")
def logout():
    return redirect(url_for("auth.login"))
```

app.py:

```python
from flask import Flask
from auth.routes import auth_bp
from tasks.routes import tasks_bp

app = Flask(__name__)
app.secret_key = "your-secret-key"

app.register_blueprint(auth_bp)
app.register_blueprint(tasks_bp)
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

Each blueprint has its own routes, templates, and URL prefix. `url_for("auth.login")` generates `/auth/login`.

40.10 Building a JSON API with Flask

Flask also works great as a pure API server:

```
from flask import Flask, jsonify, request, abort

app = Flask(__name__)

# In-memory store (use a database in production)
tasks = [
    {"id": 1, "title": "Buy milk",  "done": False},
    {"id": 2, "title": "Walk dog",  "done": True},
]
next_id = 3


@app.route("/api/tasks", methods=["GET"])
def get_tasks():
    done = request.args.get("done")
    if done is not None:
        done_bool = done.lower() == "true"
        filtered  = [t for t in tasks if t["done"] == done_bool]
        return jsonify(filtered)
    return jsonify(tasks)


@app.route("/api/tasks/<int:task_id>", methods=["GET"])
def get_task(task_id):
    task = next((t for t in tasks if t["id"] == task_id), None)
    if task is None:
        abort(404)
    return jsonify(task)


@app.route("/api/tasks", methods=["POST"])
def create_task():
```

```
    global next_id
    data = request.get_json()
    if not data or "title" not in data:
        abort(400, description="'title' is required")
    task = {"id": next_id, "title": data["title"], "done": False}
    next_id += 1
    tasks.append(task)
    return jsonify(task), 201


@app.route("/api/tasks/<int:task_id>", methods=["PATCH"])
def update_task(task_id):
    task = next((t for t in tasks if t["id"] == task_id), None)
    if task is None:
        abort(404)
    data = request.get_json() or {}
    task.update({k: v for k, v in data.items() if k in ("title", "done")})
    return jsonify(task)


@app.route("/api/tasks/<int:task_id>", methods=["DELETE"])
def delete_task(task_id):
    global tasks
    task = next((t for t in tasks if t["id"] == task_id), None)
    if task is None:
        abort(404)
    tasks = [t for t in tasks if t["id"] != task_id]
    return "", 204


@app.errorhandler(404)
def not_found(e):
    return jsonify(error=str(e)), 404


@app.errorhandler(400)
def bad_request(e):
    return jsonify(error=str(e)), 400
```

Test it:

```
# Get all tasks
curl http://127.0.0.1:5000/api/tasks

# Create a task
```

```
curl -X POST http://127.0.0.1:5000/api/tasks \
     -H "Content-Type: application/json" \
     -d '{"title": "Learn Flask"}'

# Update a task
curl -X PATCH http://127.0.0.1:5000/api/tasks/1 \
     -H "Content-Type: application/json" \
     -d '{"done": true}'

# Delete a task
curl -X DELETE http://127.0.0.1:5000/api/tasks/1
```

40.11 Error Handlers

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template("404.html"), 404

@app.errorhandler(500)
def internal_error(e):
    return render_template("500.html"), 500
```

40.12 Deploying Flask

40.12.1 Option 1: Render (free tier)

1. Push your code to GitHub
2. Go to render.com -> New Web Service
3. Connect your repo
4. Set start command: `gunicorn app:app`
5. Done — Render gives you a public URL

```
pip install gunicorn
```

Never use `app.run()` (Flask's dev server) in production. `gunicorn` is the production server.

40.12.2 Option 2: PythonAnywhere (free tier)

1. Upload your files
2. Go to the Web tab -> Add new web app

3. Choose Flask
4. Configure WSGI to point to your `app.py`

40.12.3 `requirements.txt` for deployment

```
pip freeze > requirements.txt
```

Include at minimum:

```
flask
gunicorn
```

40.13 What You Learned in This Chapter

- `Flask(__name__)` creates an app. `@app.route("/path")` maps URLs to functions.
- Return a string for plain text, `render_template()` for HTML, `jsonify()` for JSON.
- `<int:id>` in the route captures a typed URL parameter.
- `request.args` for query strings, `request.form` for POST form data, `request.get_json()` for JSON bodies.
- Jinja2 templates live in `templates/`. Use `{{ }}` for values, `{% %}` for logic, `{% extends %}` and `{% block %}` for layouts.
- Static files live in `static/`. Reference them with `url_for('static', filename='...')`.
- `flash()` + `get_flashed_messages()` for one-time user notifications.
- `redirect(url_for("function_name"))` sends users to another route.
- Blueprints split a large app into independent modules.
- Use `gunicorn` to deploy — never Flask's dev server.

40.14 What's Next?

Chapter 41 covers **FastAPI** — a modern, high-performance API framework with automatic validation, type hints, and interactive documentation. If Flask is the classic choice, FastAPI is where the industry is heading.

Chapter 41

Chapter 41: Web Development with FastAPI

FastAPI is the fastest-growing Python web framework. It's built for APIs, runs on async Python, validates data automatically using type hints, and generates interactive documentation for free.

Companies like Uber, Netflix, and Microsoft use it. If you're building an API in 2026, FastAPI is the default choice.

41.1 What Makes FastAPI Different?

Feature	Flask	FastAPI
Speed	Fast	Very fast (Starlette + async)
Data validation	Manual	Automatic (Pydantic)
API docs	Manual	Auto-generated (Swagger + ReDoc)
Type hints	Optional	Built-in, used for validation
Async support	Limited	First-class
Learning curve	Lower	Slightly higher

You already know Flask from Chapter 40. FastAPI feels familiar but more powerful.

41.2 Setup

```
pip install fastapi uvicorn[standard] pydantic
```

`uvicorn` is the ASGI server that runs FastAPI. Think of it like `gunicorn` for Flask, but async.

41.3 Your First FastAPI App

```python
# main.py
from fastapi import FastAPI

app = FastAPI(title="My API", version="1.0.0")


@app.get("/")
def home():
    return {"message": "Hello, World!"}


@app.get("/about")
def about():
    return {"app": "My API", "author": "Alice"}
```

Run it:

```
uvicorn main:app --reload
```

Open `http://127.0.0.1:8000/` — you'll see the JSON response.

Now open `http://127.0.0.1:8000/docs` — you get a full interactive Swagger UI. Every endpoint is documented, testable, and shows request/response schemas. No extra code required.

`--reload` restarts the server on code changes, like `debug=True` in Flask.

41.4 Path Parameters

```python
@app.get("/users/{user_id}")
def get_user(user_id: int):
    # FastAPI validates that user_id is an int
    # If you pass /users/abc -> automatic 422 error
    return {"user_id": user_id}
```

```
@app.get("/items/{item_name}")
def get_item(item_name: str):
    return {"item": item_name}
```

The type hint `user_id: int` does two things at once: 1. Converts the string from the URL to an integer 2. Returns a `422 Unprocessable Entity` error if it's not a valid integer

---

41.5 Query Parameters

```
from typing import Optional


@app.get("/search")
def search(
    q:        str,
    page:     int = 1,
    per_page: int = 10,
    active:   Optional[bool] = None,
):
    # /search?q=python
    # /search?q=python&page=2&per_page=20
    # /search?q=python&active=true
    return {
        "query":    q,
        "page":     page,
        "per_page": per_page,
        "active":   active,
    }
```

Parameters with default values are optional query params. Parameters without defaults are required.

41.6 Request Bodies with Pydantic

This is where FastAPI gets powerful. Define a Pydantic model and FastAPI uses it for: - Automatic JSON parsing - Validation - Error messages - Documentation

```python
from pydantic import BaseModel, Field, EmailStr
from typing import Optional


class UserCreate(BaseModel):
    name:     str             = Field(..., min_length=1, max_length=100)
    email:    str             = Field(..., description="User's email address")
    age:      int             = Field(..., ge=0, le=150)
    bio:      Optional[str]   = None


@app.post("/users", status_code=201)
def create_user(user: UserCreate):
    # user is already validated --- if we get here, it's clean data
    return {
        "id":    42,
        "name":  user.name,
        "email": user.email,
        "age":   user.age,
    }
```

If you send this:

```
{"name": "", "email": "not-an-email", "age": -5}
```

You get back:

```
{
  "detail": [
    {"loc": ["body", "name"], "msg": "ensure this value has at least 1 characte
    {"loc": ["body", "age"],  "msg": "ensure this value is greater than or equa
  ]
}
```

Zero code written for validation. Pydantic handles everything.

41.6.1 Field validators

```python
from pydantic import BaseModel, field_validator
import re


class UserCreate(BaseModel):
    username: str
    password: str
    email:    str
```

```
    @field_validator("username")
    @classmethod
    def username_alphanumeric(cls, v: str) -> str:
        if not v.isalnum():
            raise ValueError("Username must be alphanumeric")
        return v.lower()

    @field_validator("password")
    @classmethod
    def password_strong(cls, v: str) -> str:
        if len(v) < 8:
            raise ValueError("Password must be at least 8 characters")
        return v

    @field_validator("email")
    @classmethod
    def email_valid(cls, v: str) -> str:
        if "@" not in v:
            raise ValueError("Invalid email address")
        return v.lower()
```

41.7 Response Models

Define what the API returns separately from what it accepts. This lets you hide sensitive fields (like passwords) from responses:

```
from pydantic import BaseModel
from datetime import datetime


class UserCreate(BaseModel):
    name:     str
    email:    str
    password: str   # accepted in request


class UserOut(BaseModel):
    id:         int
    name:       str
    email:      str
    created_at: datetime
    # password is NOT here --- never returned
```

```python
@app.post("/users", response_model=UserOut, status_code=201)
def create_user(user: UserCreate):
    # In a real app: hash password, save to DB
    return UserOut(
        id=1,
        name=user.name,
        email=user.email,
        created_at=datetime.utcnow(),
    )
```

FastAPI filters the response through `UserOut` automatically — even if you return extra fields, they won't appear in the output.

41.8 A Full CRUD API

```python
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel, Field
from typing import Optional
from datetime import datetime

app = FastAPI(title="Task API", version="1.0.0")


# ── Models ──────────────────────────────────────────────────────────

class TaskCreate(BaseModel):
    title:       str            = Field(..., min_length=1, max_length=200)
    description: Optional[str]  = None
    priority:    int            = Field(1, ge=1, le=5)


class TaskUpdate(BaseModel):
    title:       Optional[str]  = Field(None, min_length=1, max_length=200)
    description: Optional[str]  = None
    priority:    Optional[int]  = Field(None, ge=1, le=5)
    done:        Optional[bool] = None


class TaskOut(BaseModel):
    id:          int
```

```
    title:       str
    description: Optional[str]
    priority:    int
    done:        bool
    created_at:  datetime


# ── In-memory store ──────────────────────────────────────────

tasks:   dict[int, TaskOut] = {}
next_id: int = 1


# ── Endpoints ────────────────────────────────────────────────

@app.get("/tasks", response_model=list[TaskOut])
def list_tasks(
    done:     Optional[bool] = Query(None, description="Filter by done status"),
    priority: Optional[int]  = Query(None, ge=1, le=5),
    limit:    int             = Query(20, ge=1, le=100),
    offset:   int             = Query(0, ge=0),
):
    result = list(tasks.values())
    if done is not None:
        result = [t for t in result if t.done == done]
    if priority is not None:
        result = [t for t in result if t.priority == priority]
    return result[offset : offset + limit]


@app.post("/tasks", response_model=TaskOut, status_code=201)
def create_task(task: TaskCreate):
    global next_id
    new_task = TaskOut(
        id=next_id,
        title=task.title,
        description=task.description,
        priority=task.priority,
        done=False,
        created_at=datetime.utcnow(),
    )
    tasks[next_id] = new_task
    next_id += 1
    return new_task
```

```python
@app.get("/tasks/{task_id}", response_model=TaskOut)
def get_task(task_id: int):
    task = tasks.get(task_id)
    if task is None:
        raise HTTPException(status_code=404, detail=f"Task {task_id} not found"
    return task


@app.patch("/tasks/{task_id}", response_model=TaskOut)
def update_task(task_id: int, updates: TaskUpdate):
    task = tasks.get(task_id)
    if task is None:
        raise HTTPException(status_code=404, detail=f"Task {task_id} not found"

    task_data = task.model_dump()
    update_data = updates.model_dump(exclude_unset=True)  # only changed fields
    task_data.update(update_data)
    tasks[task_id] = TaskOut(**task_data)
    return tasks[task_id]


@app.delete("/tasks/{task_id}", status_code=204)
def delete_task(task_id: int):
    if task_id not in tasks:
        raise HTTPException(status_code=404, detail=f"Task {task_id} not found"
    del tasks[task_id]
```

Visit `http://127.0.0.1:8000/docs` to test every endpoint interactively.

41.9 Async Endpoints

FastAPI is built on `asyncio`. Use `async def` for I/O-bound routes:

```python
import asyncio
import httpx   # async HTTP client


@app.get("/weather/{city}")
async def get_weather(city: str):
    async with httpx.AsyncClient() as client:
        response = await client.get(
```

```
            f"https://wttr.in/{city}?format=j1",
            timeout=10,
        )
        response.raise_for_status()
        data = response.json()

    return {
        "city":        city,
        "temp_c":      data["current_condition"][0]["temp_C"],
        "description": data["current_condition"][0]["weatherDesc"][0]["value"],
    }
```

```
pip install httpx
```

Use `async def` when your endpoint does I/O (database queries, HTTP calls, file reads). Use regular `def` for CPU-bound or synchronous code. FastAPI handles both correctly.

41.10 Dependency Injection

Dependencies let you share logic across endpoints — database connections, authentication, pagination params:

```
from fastapi import Depends


# ── A reusable "pagination" dependency ──────────────────────────

class Pagination:
    def __init__(self, limit: int = 20, offset: int = 0):
        self.limit  = limit
        self.offset = offset


@app.get("/products")
def list_products(pagination: Pagination = Depends()):
    # pagination.limit and pagination.offset are set automatically
    return {"limit": pagination.limit, "offset": pagination.offset}


# ── A reusable "current user" dependency ────────────────────────

from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
```

```
security = HTTPBearer()


def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security),
) -> dict:
    token = credentials.credentials
    # In a real app: decode JWT, look up user in DB
    if token != "valid-token":
        raise HTTPException(status_code=401, detail="Invalid token")
    return {"id": 1, "name": "Alice"}


@app.get("/me")
def get_me(current_user: dict = Depends(get_current_user)):
    return current_user


@app.get("/dashboard")
def dashboard(current_user: dict = Depends(get_current_user)):
    return {"welcome": current_user["name"]}
```

Both `/me` and `/dashboard` are now protected without repeating the auth logic.

41.11 Authentication with JWT

A realistic authentication flow:

```
pip install python-jose[cryptography] passlib[bcrypt]
```

```
from datetime import datetime, timedelta
from typing import Optional
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel

SECRET_KEY  = "super-secret-key-change-this"
ALGORITHM   = "HS256"
TOKEN_EXPIRY = timedelta(hours=24)

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
```

```
app = FastAPI()

# Fake user database
USERS_DB = {
    "alice": {
        "username": "alice",
        "hashed_password": pwd_context.hash("password123"),
        "email": "alice@example.com",
    }
}


class Token(BaseModel):
    access_token: str
    token_type:   str


class UserOut(BaseModel):
    username: str
    email:    str


def create_access_token(data: dict) -> str:
    payload = data.copy()
    payload["exp"] = datetime.utcnow() + TOKEN_EXPIRY
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)


def get_current_user(token: str = Depends(oauth2_scheme)) -> dict:
    try:
        payload  = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username = payload.get("sub")
        if username is None:
            raise HTTPException(status_code=401, detail="Invalid token")
    except JWTError:
        raise HTTPException(status_code=401, detail="Invalid token")

    user = USERS_DB.get(username)
    if user is None:
        raise HTTPException(status_code=401, detail="User not found")
    return user
```

```python
@app.post("/token", response_model=Token)
def login(form: OAuth2PasswordRequestForm = Depends()):
    user = USERS_DB.get(form.username)
    if not user or not pwd_context.verify(form.password, user["hashed_password"
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    token = create_access_token({"sub": user["username"]})
    return {"access_token": token, "token_type": "bearer"}


@app.get("/me", response_model=UserOut)
def get_me(current_user: dict = Depends(get_current_user)):
    return current_user
```

Test it in `/docs` — click "Authorize," log in as `alice`/`password123`, then call `/me`.

41.12 Lifespan Events — Startup and Shutdown

Use `lifespan` to run code once at startup and shutdown (connect to DB, warm up caches):

```python
from contextlib import asynccontextmanager
from fastapi import FastAPI


@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup
    print("Starting up --- connecting to database...")
    app.state.db = {}    # replace with real DB connection

    yield   # app is running here

    # Shutdown
    print("Shutting down --- closing connections...")


app = FastAPI(lifespan=lifespan)
```

```
@app.get("/")
def home():
    return {"status": "running"}
```

41.13 Middleware

Middleware runs before and after every request:

```
import time
from fastapi import Request


@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    start    = time.time()
    response = await call_next(request)
    duration = time.time() - start
    response.headers["X-Process-Time"] = f"{duration:.4f}s"
    return response
```

Add CORS support (required for browser-based frontends):

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "https://myapp.com"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

41.14 Background Tasks

Run slow tasks (sending emails, processing files) after returning the response:

```
from fastapi import BackgroundTasks


def send_welcome_email(email: str, name: str):
```

```
    import time
    time.sleep(2)    # simulate slow email sending
    print(f"Email sent to {email}: Welcome, {name}!")


@app.post("/register")
def register(
    name:             str,
    email:            str,
    background_tasks: BackgroundTasks,
):
    # This returns immediately
    background_tasks.add_task(send_welcome_email, email, name)
    return {"message": f"Registered {name}. Welcome email on its way!"}
```

41.15 Deploying FastAPI

41.15.1 Option 1: Render

```
# Procfile or render.yaml start command:
uvicorn main:app --host 0.0.0.0 --port $PORT
```

41.15.2 Option 2: Docker

```
FROM python:3.12-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
docker build -t myapi .
docker run -p 8000:8000 myapi
```

41.15.3 Option 3: Multiple workers with Gunicorn

```
pip install gunicorn
gunicorn main:app -w 4 -k uvicorn.workers.UvicornWorker
```

41.16 Flask vs FastAPI — When to Use Which

Use Flask when: - Building a server-rendered website with HTML templates - You need a simple API with no special requirements - Your team already knows Flask - You need the massive Flask ecosystem (extensions)

Use FastAPI when: - Building a REST or GraphQL API - You want automatic validation and docs - Performance matters - You're using async Python - You're starting a new project today

---

41.17 What You Learned in This Chapter

- `FastAPI()` creates the app. `uvicorn main:app --reload` runs it.
- Type hints on function parameters do triple duty: URL parsing, validation, and docs generation.
- `response_model=` controls what fields are returned — use it to hide sensitive data.
- `HTTPException(status_code=..., detail=...)` returns clean error responses.
- Pydantic `BaseModel` subclasses define request and response shapes. `Field(...)` adds validation rules. `@field_validator` adds custom checks.
- `Depends()` injects shared logic (auth, DB, pagination) into endpoints.
- `async def` routes handle I/O-bound work efficiently.
- JWT auth: `OAuth2PasswordBearer` + `python-jose` + `passlib`.
- `lifespan` context manager handles startup/shutdown.
- `BackgroundTasks` runs work after the response is sent.
- Auto-generated docs at `/docs` (Swagger) and `/redoc` (ReDoc).

41.18 What's Next?

Chapter 42 covers **NumPy** — the numerical computing library at the heart of Python's data science ecosystem. You'll learn how to work with arrays, perform vectorized math, and process numerical data orders of magnitude faster than plain Python lists.

Chapter 42

Chapter 42: Data Science

Python lists are convenient. But when you need to do math on a million numbers, they're too slow.

NumPy fixes that. It gives you arrays — like lists, but stored as compact blocks of memory and operated on at the speed of C. One NumPy operation replaces a Python `for` loop and runs 10 to 100 times faster.

Every data science library in Python — pandas, scikit-learn, matplotlib, TensorFlow — is built on NumPy. Learn NumPy and everything else clicks into place.

42.1 Setup

```
pip install numpy
```

```
import numpy as np   # np is the universal alias --- always use it
```

42.2 Arrays — The Core Concept

A NumPy **array** (`ndarray`) is a grid of values, all the same type, stored in contiguous memory.

```
import numpy as np

# 1D array --- like a list
a = np.array([1, 2, 3, 4, 5])
print(a)         # [1 2 3 4 5]
```

```
print(type(a))  # <class 'numpy.ndarray'>

# 2D array --- like a list of lists (a matrix)
m = np.array([[1, 2, 3],
              [4, 5, 6]])
print(m)
# [[1 2 3]
#  [4 5 6]]
```

42.2.1 Key properties

```
a = np.array([[1, 2, 3],
              [4, 5, 6]])

print(a.shape)     # (2, 3) --- 2 rows, 3 columns
print(a.ndim)      # 2 --- number of dimensions
print(a.size)      # 6 --- total number of elements
print(a.dtype)     # int64 --- data type of elements
```

42.3 Creating Arrays

```
# All zeros
np.zeros((3, 4))          # 3x4 matrix of 0.0

# All ones
np.ones((2, 3))           # 2x3 matrix of 1.0

# Filled with a value
np.full((2, 2), 7)        # [[7 7] [7 7]]

# Identity matrix
np.eye(3)                 # 3x3 identity

# Evenly spaced values (like range)
np.arange(0, 10, 2)       # [0 2 4 6 8]
np.arange(5)              # [0 1 2 3 4]

# Evenly spaced values (by count)
np.linspace(0, 1, 5)      # [0.   0.25 0.5  0.75 1.  ]

# Random numbers
np.random.seed(42)                  # for reproducibility
```

```
np.random.rand(3, 3)                # uniform [0, 1)
np.random.randn(3, 3)               # standard normal
np.random.randint(0, 10, size=(3, 3))  # random ints

# From a Python list --- with explicit dtype
np.array([1.0, 2.5, 3.7])           # float64
np.array([1, 2, 3], dtype=np.float32)
```

42.4 Data Types

```
a = np.array([1, 2, 3])
print(a.dtype)       # int64 (on most 64-bit systems)

b = np.array([1.0, 2.0])
print(b.dtype)       # float64

# Common dtypes:
# np.int8, np.int16, np.int32, np.int64
# np.float32, np.float64
# np.bool_
# np.complex128

# Convert dtype
a_float = a.astype(np.float32)
```

Choosing smaller dtypes (like `float32` instead of `float64`) halves memory usage — important for large datasets.

42.5 Indexing and Slicing

42.5.1 1D arrays — just like lists

```
a = np.array([10, 20, 30, 40, 50])

a[0]         # 10
a[-1]        # 50
a[1:4]       # [20 30 40]
a[::2]       # [10 30 50]
```

42.5.2 2D arrays

```
m = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

m[0]         # [1 2 3] --- first row
m[0, 1]      # 2 --- row 0, column 1
m[:, 1]      # [2 5 8] --- all rows, column 1
m[1:, :2]    # [[4 5] [7 8]] --- rows 1+, columns 0-1
```

42.5.3 Boolean indexing — filter without a loop

```
a = np.array([10, 25, 3, 47, 8, 30])

mask = a > 20
print(mask)          # [False  True False  True False  True]
print(a[mask])       # [25 47 30]

# One-liner
print(a[a > 20])     # [25 47 30]

# Set all values below 10 to 0
a[a < 10] = 0
print(a)             # [10 25  0 47  0 30]
```

42.5.4 Fancy indexing — select by index list

```
a = np.array([100, 200, 300, 400, 500])
indices = [0, 2, 4]
print(a[indices])    # [100 300 500]
```

42.6 Vectorized Operations — No Loops Needed

This is the key insight: NumPy operations apply to the whole array at once, without a Python `for` loop.

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])

# Element-wise operations
a + b        # [11 22 33 44 55]
```

```
a * b        # [10 40 90 160 250]
a ** 2       # [ 1  4  9 16 25]
np.sqrt(a)   # [1. 1.41 1.73 2.  2.24]

# Scalar operations --- broadcast to every element
a + 100      # [101 102 103 104 105]
a * 2        # [  2   4   6   8  10]

# Comparison
a > 3        # [False False False  True  True]
```

42.6.1 Speed comparison

```
import time
import numpy as np

n = 10_000_000

# Python list
lst = list(range(n))
start = time.time()
result = [x * 2 for x in lst]
print(f"Python list:  {time.time() - start:.3f}s")

# NumPy array
arr = np.arange(n)
start = time.time()
result = arr * 2
print(f"NumPy array:  {time.time() - start:.3f}s")
```

Output:

```
Python list:  0.847s
NumPy array:  0.008s
```

NumPy is ~100x faster for this operation.

42.7 Broadcasting

Broadcasting lets NumPy operate on arrays of different shapes without making copies.

```
# Add a scalar to a 2D array
m = np.array([[1, 2, 3],
```

```
              [4, 5, 6]])
m + 10
# [[11 12 13]
#  [14 15 16]]

# Add a 1D array to each row of a 2D array
row = np.array([1, 2, 3])
m + row
# [[ 2  4  6]
#  [ 5  7  9]]

# Add a column vector to each column
col = np.array([[10], [20]])  # shape (2, 1)
m + col
# [[11 12 13]
#  [24 25 26]]
```

Broadcasting rules (simplified): dimensions are compared right-to-left. A dimension of 1 is stretched to match the other.

42.8 Mathematical Functions

```
a = np.array([1, 4, 9, 16, 25])

np.sqrt(a)       # [1. 2. 3. 4. 5.]
np.log(a)        # natural log
np.log2(a)
np.log10(a)
np.exp(a)        # e^x
np.abs(a)        # absolute value
np.sin(a)
np.cos(a)
np.floor(a)
np.ceil(a)
np.round(a, 2)
```

42.9 Aggregation — Summary Statistics

```
a = np.array([4, 7, 2, 9, 1, 5, 8, 3, 6])
```

```
np.sum(a)        # 45
np.mean(a)       # 5.0
np.median(a)     # 5.0
np.std(a)        # standard deviation: 2.58
np.var(a)        # variance: 6.67
np.min(a)        # 1
np.max(a)        # 9
np.argmin(a)     # 4 --- index of minimum
np.argmax(a)     # 3 --- index of maximum
np.cumsum(a)     # running total
np.sort(a)       # sorted copy

# 2D aggregation --- specify axis
m = np.array([[1, 2, 3],
              [4, 5, 6]])

np.sum(m)            # 21 --- total
np.sum(m, axis=0)    # [5 7 9] --- sum each column
np.sum(m, axis=1)    # [ 6 15] --- sum each row
np.mean(m, axis=0)   # [2.5 3.5 4.5] --- mean of each column
```

42.10 Reshaping and Combining

```
a = np.arange(12)        # [0 1 2 3 4 5 6 7 8 9 10 11]

# Reshape --- total elements must be the same
a.reshape(3, 4)          # 3 rows, 4 columns
a.reshape(2, 2, 3)       # 3D: 2x2x3
a.reshape(-1, 3)         # -1 means "figure it out" -> (4, 3)

# Flatten back to 1D
m = a.reshape(3, 4)
m.flatten()              # copy
m.ravel()                # view (faster, shares memory)

# Transpose
m.T                      # swap rows and columns

# Stack arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
np.vstack([a, b])        # [[1 2 3] [4 5 6]] --- stack vertically
```

```
np.hstack([a, b])        # [1 2 3 4 5 6]     --- stack horizontally
np.concatenate([a, b])   # [1 2 3 4 5 6]

# Split
np.split(a, 3)           # [array([1]), array([2]), array([3])]
```

42.11 Linear Algebra

```
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6],
              [7, 8]])

# Matrix multiplication
A @ B                # [[19 22] [43 50]]
np.dot(A, B)         # same

# Determinant
np.linalg.det(A)     # -2.0

# Inverse
np.linalg.inv(A)     # [[-2.   1. ] [ 1.5 -0.5]]

# Eigenvalues and eigenvectors
vals, vecs = np.linalg.eig(A)

# Solve Ax = b
b = np.array([1, 2])
x = np.linalg.solve(A, b)  # solution to Ax = b
```

42.12 Working with Real Data

```
import numpy as np

# Load from a text file
data = np.loadtxt("data.csv", delimiter=",", skiprows=1)

# Load from a binary file (much faster for large data)
np.save("data.npy", data)
loaded = np.load("data.npy")
```

```
# Generate sample sensor data: 1000 readings with noise
np.random.seed(42)
time_points = np.linspace(0, 10, 1000)          # 0 to 10 seconds
signal      = np.sin(2 * np.pi * time_points)   # clean sine wave
noise       = np.random.randn(1000) * 0.1       # random noise
readings    = signal + noise

# Find peaks
peaks = np.where(readings > 0.9)[0]
print(f"Found {len(peaks)} peaks above 0.9")
print(f"Mean reading: {readings.mean():.4f}")
print(f"Std deviation: {readings.std():.4f}")
```

42.13 Practical Project — Grade Analysis

```
"""
grade_analysis.py

Analyse student exam scores using NumPy.
"""
import numpy as np


def letter_grade(score: float) -> str:
    if score >= 90: return "A"
    if score >= 80: return "B"
    if score >= 70: return "C"
    if score >= 60: return "D"
    return "F"


def analyse_grades(scores: np.ndarray) -> dict:
    """Return summary statistics for an array of scores."""
    return {
        "count":   len(scores),
        "mean":    round(float(scores.mean()), 2),
        "median":  round(float(np.median(scores)), 2),
        "std":     round(float(scores.std()), 2),
        "min":     float(scores.min()),
        "max":     float(scores.max()),
        "passing": int((scores >= 60).sum()),
```

```
        "failing": int((scores < 60).sum()),
    }


def grade_distribution(scores: np.ndarray) -> dict:
    """Count scores in each letter-grade bucket."""
    thresholds = [90, 80, 70, 60, 0]
    labels     = ["A", "B", "C", "D", "F"]
    dist       = {}
    for label, low, high in zip(
        labels, thresholds[:-1] + [0], [100] + thresholds[:-1]
    ):
        dist[label] = int(((scores >= low) & (scores < high)).sum())
    return dist


def normalise(scores: np.ndarray) -> np.ndarray:
    """Scale scores to [0, 1]."""
    min_s, max_s = scores.min(), scores.max()
    if max_s == min_s:
        return np.zeros_like(scores, dtype=float)
    return (scores - min_s) / (max_s - min_s)


def curve(scores: np.ndarray, target_mean: float = 75.0) -> np.ndarray:
    """Add a curve so the mean hits target_mean."""
    shift   = target_mean - scores.mean()
    curved  = scores + shift
    return np.clip(curved, 0, 100)   # cap at 0--100


# ── Demo ──────────────────────────────────────────────────────────────

if __name__ == "__main__":
    np.random.seed(99)

    # Simulate 30 students, 4 exams
    raw = np.random.normal(loc=68, scale=15, size=(30, 4))
    raw = np.clip(raw, 0, 100).round(1)

    # Final score = weighted average (exams weighted equally here)
    weights = np.array([0.2, 0.2, 0.25, 0.35])
    finals  = raw @ weights     # matrix multiply: (30,4) @ (4,) -> (30,)
```

```
    stats = analyse_grades(finals)
    dist  = grade_distribution(finals)

    print("=== Grade Analysis ===")
    print(f"Students: {stats['count']}")
    print(f"Mean:     {stats['mean']:.1f}   Median: {stats['median']:.1f}")
    print(f"Std dev:  {stats['std']:.1f}")
    print(f"Min: {stats['min']:.1f}   Max: {stats['max']:.1f}")
    print(f"Passing: {stats['passing']}   Failing: {stats['failing']}")

    print("\nGrade Distribution:")
    for grade, count in dist.items():
        bar = "" * count
        print(f"  {grade}: {bar} ({count})")

    # Apply curve if mean is below 70
    if stats["mean"] < 70:
        curved = curve(finals, target_mean=72)
        print(f"\nCurve applied: mean {stats['mean']:.1f} -> {curved.mean():.1f
```

Output:

```
=== Grade Analysis ===
Students: 30
Mean:     67.8   Median: 68.4
Std dev:  13.2
Min: 38.1   Max: 94.7
Passing: 22   Failing: 8

Grade Distribution:
  A:  (2)
  B:  (4)
  C:  (8)
  D:  (8)
  F:  (8)

Curve applied: mean 67.8 -> 72.0
```

42.14 Common Pitfalls

42.14.1 Views vs copies

```
a = np.array([1, 2, 3, 4, 5])

# Slices return VIEWS --- modifying them modifies the original
b = a[1:4]
b[0] = 99
print(a)     # [ 1 99  3  4  5] <- a is changed!

# Make a copy if you don't want this
c = a[1:4].copy()
c[0] = 0
print(a)     # [ 1 99  3  4  5] <- a is unchanged
```

42.14.2 Integer division

```
a = np.array([1, 2, 3])
print(a / 2)       # [0.5 1.  1.5] --- true division, float result
print(a // 2)      # [0 1 1]         --- floor division, int result
```

42.14.3 Shape mismatches

```
a = np.array([1, 2, 3])        # shape (3,)
b = np.array([[1], [2], [3]]) # shape (3, 1)
# a + b -> shape (3, 3) due to broadcasting --- often unexpected!
```

When in doubt, use `.reshape()` to make shapes explicit.

42.15 What You Learned in This Chapter

- NumPy arrays (`ndarray`) store homogeneous data in contiguous memory, making math 10–100x faster than Python lists.
- `.shape`, `.dtype`, `.size`, `.ndim` describe an array.
- `np.zeros`, `np.ones`, `np.arange`, `np.linspace`, `np.random.*` create arrays.
- Indexing with `[row, col]`, slicing with `[start:stop:step]`, and boolean indexing `a[a > 5]` all work without loops.
- All arithmetic operators (`+`, `-`, `*`, `/`, `**`) and math functions (`np.sqrt`, `np.log`, `np.sin`) work element-wise.
- Broadcasting applies operations between arrays of compatible shapes without copying data.

- `np.sum`, `np.mean`, `np.std`, `np.min`, `np.max`, `np.argmin`, `np.argmax` summarise arrays. Use `axis=` for row/column operations.
- `reshape`, `flatten`, `vstack`, `hstack`, `concatenate` restructure arrays.
- `@` or `np.dot` multiplies matrices. `np.linalg` handles determinants, inverses, and solvers.
- Slices are *views* — modifying them modifies the original. Use `.copy()` to avoid this.

42.16 What's Next?

Chapter 43 covers **pandas** — the library built on top of NumPy that adds labels, DataFrames, and powerful tools for reading, cleaning, and transforming real-world tabular data.

Chapter 43

Chapter 43: Data Science with Pandas

NumPy is great for numbers. But real-world data has labels, mixed types, missing values, and dates. That's what pandas is built for.

pandas gives you the **DataFrame** — a table with named columns and rows. Think of it as a Python-powered spreadsheet that you control entirely with code. You can load a CSV, clean it, filter it, group it, and visualise it in fewer than twenty lines.

43.1 Setup

```
pip install pandas openpyxl xlrd
```

```
import pandas as pd
import numpy as np
```

`openpyxl` reads `.xlsx` files. `xlrd` reads older `.xls` files.

43.2 The Two Core Objects

43.2.1 Series — a labelled 1D array

```
# From a list
s = pd.Series([10, 20, 30, 40])
print(s)
```

```
# 0    10
# 1    20
# 2    30
# 3    40
# dtype: int64

# With custom index
s = pd.Series([10, 20, 30], index=["a", "b", "c"])
print(s["b"])    # 20

# From a dict
s = pd.Series({"jan": 100, "feb": 85, "mar": 92})
print(s.index)   # Index(['jan', 'feb', 'mar'], dtype='object')
```

43.2.2 DataFrame — a labelled 2D table

```
# From a dict of lists
df = pd.DataFrame({
    "name":   ["Alice", "Bob", "Carol", "Dave"],
    "age":    [28, 34, 22, 45],
    "score":  [88.5, 72.0, 95.3, 61.8],
    "passed": [True, True, True, False],
})

print(df)
#     name  age  score  passed
# 0  Alice   28   88.5    True
# 1    Bob   34   72.0    True
# 2  Carol   22   95.3    True
# 3   Dave   45   61.8   False
```

Every column is a `Series`. The `df` itself is a collection of aligned Series sharing the same index.

43.3 Reading and Writing Data

43.3.1 CSV

```
# Read
df = pd.read_csv("sales.csv")
df = pd.read_csv("sales.csv", index_col="id")    # use a column as the index
df = pd.read_csv("sales.csv", parse_dates=["date"])  # parse date columns
```

```
# Write
df.to_csv("output.csv", index=False)    # index=False omits the row numbers
```

43.3.2 Excel

```
df = pd.read_excel("report.xlsx", sheet_name="Q1")
df.to_excel("output.xlsx", sheet_name="Results", index=False)
```

43.3.3 JSON

```
df = pd.read_json("data.json")
df.to_json("output.json", orient="records", indent=2)
```

43.3.4 SQL

```
import sqlite3

conn = sqlite3.connect("sales.db")
df   = pd.read_sql("SELECT * FROM orders WHERE total > 100", conn)
df.to_sql("summary", conn, if_exists="replace", index=False)
conn.close()
```

43.4 Exploring a DataFrame

```
df = pd.read_csv("sales.csv")

df.shape           # (1000, 8) --- rows, columns
df.dtypes          # data type of each column
df.columns         # column names
df.index           # row labels

df.head(5)         # first 5 rows
df.tail(5)         # last 5 rows
df.sample(5)       # 5 random rows

df.info()          # column names, non-null counts, dtypes, memory
df.describe()      # count, mean, std, min, quartiles, max for numeric cols
df.describe(include="all")    # includes non-numeric columns too

df.value_counts("category")  # frequency of each category
df.nunique()                 # number of unique values per column
df.isnull().sum()            # count missing values per column
```

43.5 Selecting Data

43.5.1 Columns

```
df["name"]                      # single column -> Series
df[["name", "score"]]           # multiple columns -> DataFrame
```

43.5.2 Rows with `.iloc` — by position

```
df.iloc[0]          # first row -> Series
df.iloc[0:5]        # first 5 rows
df.iloc[0, 2]       # row 0, column 2 (by position)
df.iloc[:, 1:4]     # all rows, columns 1-3
```

43.5.3 Rows with `.loc` — by label

```
df.loc[0]              # row with index label 0
df.loc[0:5]            # rows 0 through 5 (inclusive!)
df.loc[0, "name"]      # row 0, column "name"
df.loc[:, "name":"score"]  # all rows, columns "name" through "score"
```

The key difference: `.iloc` uses integer positions (exclusive end), `.loc` uses labels (inclusive end).

43.5.4 Boolean filtering

```
# Single condition
df[df["score"] > 80]

# Multiple conditions --- use & (and) and | (or), not Python's and/or
df[(df["score"] > 80) & (df["passed"] == True)]
df[(df["age"] < 30) | (df["score"] > 90)]

# isin --- filter on a list of values
df[df["city"].isin(["London", "Paris", "Berlin"])]

# String methods
df[df["name"].str.startswith("A")]
df[df["email"].str.contains("@gmail")]

# query() --- readable string syntax
df.query("score > 80 and age < 35")
df.query("city in ['London', 'Paris']")
```

43.6 Modifying Data

43.6.1 Adding and changing columns

```
# New column from calculation
df["tax"] = df["price"] * 0.2
df["total"] = df["price"] + df["tax"]

# Conditional column
df["grade"] = df["score"].apply(
    lambda x: "A" if x >= 90 else ("B" if x >= 80 else "C")
)

# Using np.where --- faster than apply for simple conditions
import numpy as np
df["passed"] = np.where(df["score"] >= 60, True, False)

# np.select --- multiple conditions
conditions  = [df["score"] >= 90, df["score"] >= 80, df["score"] >= 70]
choices     = ["A", "B", "C"]
df["grade"] = np.select(conditions, choices, default="D")
```

43.6.2 Renaming columns

```
df.rename(columns={"name": "full_name", "score": "exam_score"}, inplace=True)
df.columns = [c.lower().replace(" ", "_") for c in df.columns]  # clean all names
```

43.6.3 Dropping columns and rows

```
df.drop(columns=["tax", "unnecessary_col"], inplace=True)
df.drop(index=[0, 1, 2], inplace=True)          # drop rows by index
df.dropna(inplace=True)                          # drop rows with any NaN
df.dropna(subset=["score", "name"])              # only drop if these cols are NaN
```

43.6.4 The `apply` function

```
# Apply a function to each element in a column
df["name"] = df["name"].str.strip().str.title()

# Apply a function to each row (axis=1)
def classify(row):
    if row["score"] >= 90 and row["age"] < 30:
```

```
        return "Star student"
    return "Regular"

df["category"] = df.apply(classify, axis=1)
```

`apply` is flexible but slow on large DataFrames. Prefer vectorised operations (`df["col"] * 2`, `np.where`, `.str` methods) when possible.

43.7 Handling Missing Data

```
df.isnull().sum()           # count NaN per column
df.isnull().sum() / len(df) # fraction missing

# Fill missing values
df["score"].fillna(df["score"].mean(), inplace=True)    # fill with mean
df["city"].fillna("Unknown", inplace=True)               # fill with constant
df["score"].ffill(inplace=True)                          # forward fill
df["score"].bfill(inplace=True)                          # backward fill

# Drop rows with missing values
df.dropna(inplace=True)                      # any NaN -> drop row
df.dropna(subset=["score"], inplace=True)    # NaN in 'score' -> drop row
df.dropna(thresh=5, inplace=True)            # keep rows with at least 5 non-NaN
```

43.8 Sorting

```
df.sort_values("score", ascending=False)                      # descending
df.sort_values(["grade", "score"], ascending=[True, False])   # multi-column
df.sort_index()                                               # sort by row inde
```

43.9 Grouping and Aggregation

`groupby` splits the data into groups, applies a function to each group, and combines the results — the split-apply-combine pattern.

```
# Simple groupby
df.groupby("department")["salary"].mean()

# Multiple aggregations
```

```
df.groupby("department").agg(
    avg_salary   = ("salary", "mean"),
    max_salary   = ("salary", "max"),
    employee_count = ("name", "count"),
)

# Group by multiple columns
df.groupby(["department", "level"])["salary"].mean()

# Apply custom function to each group
df.groupby("category")["score"].apply(lambda s: s.max() - s.min())
```

43.10 Merging and Joining

Just like SQL JOINs:

```
customers = pd.DataFrame({
    "id":   [1, 2, 3, 4],
    "name": ["Alice", "Bob", "Carol", "Dave"],
})

orders = pd.DataFrame({
    "order_id":   [101, 102, 103, 104, 105],
    "customer_id": [1, 2, 1, 3, 5],
    "amount":      [50, 30, 80, 20, 60],
})

# Inner join --- only matching rows
pd.merge(orders, customers, left_on="customer_id", right_on="id")

# Left join --- all orders, even without matching customer
pd.merge(orders, customers, left_on="customer_id", right_on="id", how="left")

# Right join, outer join also available
pd.merge(orders, customers, left_on="customer_id", right_on="id", how="outer")

# When key columns have the same name
pd.merge(orders, customers, on="id")

# Concat --- stack DataFrames vertically
combined = pd.concat([df_q1, df_q2, df_q3], ignore_index=True)
```

43.11 Working with Dates

```
# Parse dates on load
df = pd.read_csv("sales.csv", parse_dates=["date"])

# Convert a column to datetime
df["date"] = pd.to_datetime(df["date"])

# Extract components
df["year"]    = df["date"].dt.year
df["month"]   = df["date"].dt.month
df["day"]     = df["date"].dt.day
df["weekday"] = df["date"].dt.day_name()
df["quarter"] = df["date"].dt.quarter

# Date arithmetic
df["days_since"] = (pd.Timestamp.today() - df["date"]).dt.days

# Filter by date range
start = pd.Timestamp("2024-01-01")
end   = pd.Timestamp("2024-03-31")
q1    = df[(df["date"] >= start) & (df["date"] <= end)]

# Resample time series (group by time period)
df.set_index("date").resample("ME")["revenue"].sum()    # monthly totals
df.set_index("date").resample("W")["revenue"].mean()    # weekly averages
```

43.12 Practical Project — Sales Analysis

```
"""
sales_analysis.py

End-to-end analysis of a sales dataset using pandas.
"""
import pandas as pd
import numpy as np
from io import StringIO

# ── Sample data (in a real project, use pd.read_csv("sales.csv")) ──────────

RAW = """date,product,category,units,price,region
2024-01-05,Widget A,Electronics,10,29.99,North
```

```
2024-01-12,Widget B,Electronics,5,49.99,South
2024-01-15,Gadget X,Home,8,19.99,North
2024-02-03,Widget A,Electronics,15,29.99,East
2024-02-14,Gadget Y,Home,12,14.99,West
2024-02-20,Widget B,Electronics,3,49.99,North
2024-03-01,Gadget X,Home,20,19.99,South
2024-03-10,Widget A,Electronics,7,29.99,West
2024-03-22,Gadget Y,Home,9,14.99,East
2024-03-30,Widget B,Electronics,11,49.99,South
"""

# ── Load and prepare ──────────────────────────────────────────────

df = pd.read_csv(StringIO(RAW), parse_dates=["date"])
df["revenue"] = df["units"] * df["price"]
df["month"]   = df["date"].dt.to_period("M")

print("=== Data Overview ===")
print(df.dtypes)
print(f"\nRows: {len(df)}, Columns: {len(df.columns)}")
print(f"Date range: {df['date'].min().date()} -> {df['date'].max().date()}")
print(f"Missing values:\n{df.isnull().sum()}")

# ── Summary statistics ────────────────────────────────────────────

print("\n=== Revenue Summary ===")
print(df["revenue"].describe().round(2))

# ── By category ──────────────────────────────────────────────────

print("\n=== Revenue by Category ===")
by_cat = (
    df.groupby("category")
      .agg(
          total_revenue = ("revenue", "sum"),
          total_units   = ("units",   "sum"),
          avg_price     = ("price",   "mean"),
          num_orders    = ("date",    "count"),
      )
      .round(2)
      .sort_values("total_revenue", ascending=False)
)
print(by_cat)
```

```
# ── Monthly trend ──────────────────────────────────────────────

print("\n=== Monthly Revenue ===")
monthly = df.groupby("month")["revenue"].sum().reset_index()
monthly.columns = ["month", "revenue"]
for _, row in monthly.iterrows():
    bar = "" * int(row["revenue"] / 50)
    print(f"  {row['month']}  {bar}  ${row['revenue']:,.2f}")

# ── Top products ──────────────────────────────────────────────

print("\n=== Top Products by Revenue ===")
top_products = (
    df.groupby("product")["revenue"]
      .sum()
      .sort_values(ascending=False)
      .head(5)
)
for product, rev in top_products.items():
    print(f"  {product:15s}  ${rev:,.2f}")

# ── Regional breakdown ────────────────────────────────────────

print("\n=== Revenue by Region ===")
by_region = df.pivot_table(
    values="revenue",
    index="region",
    columns="category",
    aggfunc="sum",
    fill_value=0,
).round(2)
print(by_region)

# ── Save results ──────────────────────────────────────────────

df.to_csv("sales_clean.csv", index=False)
print("\nSaved to sales_clean.csv")
```

Output:

```
=== Data Overview ===
date        datetime64[ns]
product             object
category            object
units                int64
```

```
price                float64
region                object
revenue              float64
month               period[M]

Rows: 10, Columns: 8

=== Revenue by Category ===
             total_revenue  total_units  avg_price  num_orders
category
Electronics         1664.55           51      39.99           6
Home                 847.55           49      18.32           4

=== Monthly Revenue ===
  2024-01  ████████████  $629.80
  2024-02  ████████████  $624.72
  2024-03  ████████████  $857.58

=== Top Products by Revenue ===
  Widget B         $649.87
  Widget A         $569.80
  Gadget X         $479.75
  Gadget Y         $314.73
```

43.13 What You Learned in This Chapter

- `Series` is a labelled 1D array. `DataFrame` is a labelled 2D table. Every column is a Series.
- `pd.read_csv`, `pd.read_excel`, `pd.read_json`, `pd.read_sql` load data. `.to_csv`, `.to_excel` save it.
- `df.info()`, `df.describe()`, `df.head()`, `df.isnull().sum()` explore a dataset.
- `df["col"]` selects one column. `df[["a","b"]]` selects multiple. `df.iloc[row, col]` selects by position. `df.loc[label, "col"]` selects by label.
- Boolean masks — `df[df["score"] > 80]` — filter rows without loops.
- `df["new"] = df["a"] + df["b"]` adds computed columns. `apply()` runs arbitrary functions.
- `fillna`, `dropna` handle missing values.
- `groupby("col").agg(...)` splits data, computes per-group stats, and combines results.
- `pd.merge(left, right, on=..., how=...)` joins DataFrames like SQL.

`pd.concat` stacks them vertically.

- `df["date"].dt.year`, `.dt.month`, `.dt.day_name()` extract date components. `.resample()` aggregates time series by period.

43.14 What's Next?

Chapter 44 covers **Data Visualization** — turning the DataFrames you just built into charts and graphs. You'll use `matplotlib` for control and `seaborn` for beautiful statistical plots with almost no code.

Chapter 44

Chapter 44: Data Visualization

Data in a table tells you facts. Data in a chart tells you a story.

A bar chart reveals which product dominates. A scatter plot shows whether two variables are related. A histogram shows you the shape of your data in seconds. Visualization is how you turn numbers into understanding — and how you communicate that understanding to others.

This chapter covers `matplotlib` for full control and `seaborn` for beautiful statistical plots with almost no code. Both work directly with pandas DataFrames.

44.1 Setup

```
pip install matplotlib seaborn pandas numpy
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

44.2 How matplotlib Works

matplotlib uses a figure/axes model: - **Figure** — the entire image (the canvas) - **Axes** — one plot inside the figure (you can have multiple)

```
fig, ax = plt.subplots()    # create figure and one axes
ax.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

Always use `fig, ax = plt.subplots()` — it gives you explicit control. The older `plt.plot()` shorthand works for quick scripts but gets confusing with multiple subplots.

44.3 Line Plot

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 100)

fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(x, np.sin(x), label="sin(x)",  color="steelblue",  linewidth=2)
ax.plot(x, np.cos(x), label="cos(x)",  color="coral",      linewidth=2, linesty

ax.set_title("Sine and Cosine", fontsize=14)
ax.set_xlabel("x (radians)")
ax.set_ylabel("y")
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig("line_plot.png", dpi=150)
plt.show()
```

Key options: - `figsize=(width, height)` — inches - `color=` — named color, hex, or RGB tuple - `linewidth=`, `linestyle=` (“solid”, “dashed”, “–”, “-.”, “:”) - `label=` + `ax.legend()` — add a legend - `ax.grid(True)` — gridlines - `plt.tight_layout()` — prevent clipping - `plt.savefig("file.png", dpi=150)` — save to file

---

44.4 Bar Chart

```
categories = ["Electronics", "Clothing", "Food", "Books", "Sports"]
revenue    = [45000, 32000, 28000, 15000, 19000]
```

```
fig, ax = plt.subplots(figsize=(8, 5))

bars = ax.bar(categories, revenue, color="steelblue", edgecolor="white", width=0.6

# Add value labels on top of bars
for bar in bars:
    height = bar.get_height()
    ax.text(
        bar.get_x() + bar.get_width() / 2,
        height + 500,
        f"${height:,}",
        ha="center", va="bottom", fontsize=9
    )

ax.set_title("Revenue by Category", fontsize=14)
ax.set_xlabel("Category")
ax.set_ylabel("Revenue ($)")
ax.set_ylim(0, max(revenue) * 1.15)

plt.tight_layout()
plt.show()
```

44.4.1 Horizontal bar chart

```
fig, ax = plt.subplots(figsize=(8, 5))
ax.barh(categories, revenue, color="steelblue")
ax.set_xlabel("Revenue ($)")
ax.invert_yaxis()   # largest bar at the top
plt.tight_layout()
plt.show()
```

44.5 Histogram

```
np.random.seed(42)
data = np.random.normal(loc=70, scale=12, size=500)

fig, ax = plt.subplots(figsize=(8, 5))

ax.hist(data, bins=30, color="steelblue", edgecolor="white", alpha=0.8)

# Add vertical lines for mean and median
ax.axvline(data.mean(),   color="red",    linestyle="--", label=f"Mean {data.mean(
```

```
ax.axvline(np.median(data), color="green", linestyle="--", label=f"Median {np.m

ax.set_title("Distribution of Exam Scores")
ax.set_xlabel("Score")
ax.set_ylabel("Count")
ax.legend()

plt.tight_layout()
plt.show()
```

44.6 Scatter Plot

```
np.random.seed(0)
hours_studied = np.random.uniform(1, 10, 80)
exam_score    = 50 + hours_studied * 4 + np.random.normal(0, 5, 80)

fig, ax = plt.subplots(figsize=(7, 5))

scatter = ax.scatter(
    hours_studied,
    exam_score,
    c=exam_score,        # colour by score value
    cmap="RdYlGn",       # red=low, yellow=mid, green=high
    alpha=0.7,
    edgecolors="white",
    linewidths=0.5,
    s=60,                # marker size
)

# Trend line
m, b = np.polyfit(hours_studied, exam_score, 1)
x_line = np.linspace(hours_studied.min(), hours_studied.max(), 100)
ax.plot(x_line, m * x_line + b, color="navy", linewidth=2, label="Trend")

plt.colorbar(scatter, ax=ax, label="Exam Score")
ax.set_title("Hours Studied vs Exam Score")
ax.set_xlabel("Hours Studied")
ax.set_ylabel("Exam Score")
ax.legend()

plt.tight_layout()
plt.show()
```

44.7 Pie / Donut Chart

```
labels  = ["Electronics", "Clothing", "Food", "Books", "Sports"]
sizes   = [35, 25, 20, 10, 10]
explode = [0.05, 0, 0, 0, 0]   # pull out the first slice

fig, ax = plt.subplots(figsize=(7, 6))

wedges, texts, autotexts = ax.pie(
    sizes,
    labels=labels,
    explode=explode,
    autopct="%1.1f%%",
    startangle=140,
    colors=["#4C72B0", "#DD8452", "#55A868", "#C44E52", "#8172B2"],
)

# Donut variant --- add a white circle in the centre
centre_circle = plt.Circle((0, 0), 0.65, fc="white")
ax.add_artist(centre_circle)

ax.set_title("Revenue Share by Category")
plt.tight_layout()
plt.show()
```

44.8 Box Plot

Box plots show distribution, median, quartiles, and outliers at a glance.

```
np.random.seed(7)
data = {
    "Math":    np.random.normal(72, 10, 100),
    "Science": np.random.normal(68, 15, 100),
    "English": np.random.normal(75, 8,  100),
    "History": np.random.normal(65, 12, 100),
}

fig, ax = plt.subplots(figsize=(8, 5))

ax.boxplot(
```

```
    data.values(),
    labels=data.keys(),
    patch_artist=True,                    # fill boxes with colour
    medianprops=dict(color="black", linewidth=2),
)

ax.set_title("Score Distribution by Subject")
ax.set_ylabel("Score")
ax.grid(True, axis="y", alpha=0.3)

plt.tight_layout()
plt.show()
```

44.9 Multiple Subplots

```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
fig.suptitle("Sales Dashboard", fontsize=16, fontweight="bold")

months  = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
revenue = [42000, 48000, 45000, 53000, 61000, 58000]

# Top-left: line chart
axes[0, 0].plot(months, revenue, "o-", color="steelblue", linewidth=2)
axes[0, 0].set_title("Monthly Revenue")
axes[0, 0].set_ylabel("Revenue ($)")
axes[0, 0].grid(True, alpha=0.3)

# Top-right: bar chart
axes[0, 1].bar(months, revenue, color="coral")
axes[0, 1].set_title("Revenue by Month")
axes[0, 1].set_ylabel("Revenue ($)")

# Bottom-left: histogram
np.random.seed(1)
orders = np.random.normal(250, 60, 200)
axes[1, 0].hist(orders, bins=25, color="mediumseagreen", edgecolor="white")
axes[1, 0].set_title("Order Value Distribution")
axes[1, 0].set_xlabel("Order Value ($)")
axes[1, 0].set_ylabel("Frequency")

# Bottom-right: pie chart
categories = ["Online", "In-store", "Mobile"]
```

```
sales      = [55, 30, 15]
axes[1, 1].pie(sales, labels=categories, autopct="%1.0f%%", startangle=90)
axes[1, 1].set_title("Sales by Channel")

plt.tight_layout()
plt.savefig("dashboard.png", dpi=150, bbox_inches="tight")
plt.show()
```

44.10 Seaborn — Statistical Plots Made Easy

seaborn is a high-level wrapper around matplotlib. It produces better-looking charts with less code and works natively with pandas DataFrames.

```
import seaborn as sns
import pandas as pd

sns.set_theme(style="whitegrid")    # clean background with gridlines
# Other styles: "darkgrid", "white", "dark", "ticks"
```

44.10.1 Distribution plots

```
tips = sns.load_dataset("tips")    # built-in sample dataset

# Histogram + KDE
fig, ax = plt.subplots(figsize=(7, 4))
sns.histplot(data=tips, x="total_bill", kde=True, ax=ax)
ax.set_title("Distribution of Total Bill")
plt.tight_layout()
plt.show()

# KDE only
sns.kdeplot(data=tips, x="total_bill", hue="sex", fill=True)
plt.show()
```

44.10.2 Categorical plots

```
# Box plot
sns.boxplot(data=tips, x="day", y="total_bill", hue="sex", palette="Set2")
plt.title("Total Bill by Day and Sex")
plt.show()

# Violin plot --- like boxplot but shows the full distribution
sns.violinplot(data=tips, x="day", y="total_bill", hue="sex",
```

```
                split=True, palette="Set2")
plt.show()

# Bar plot with confidence intervals
sns.barplot(data=tips, x="day", y="total_bill", hue="sex",
            estimator="mean", errorbar="ci")
plt.show()

# Strip plot --- individual points
sns.stripplot(data=tips, x="day", y="tip", hue="sex",
              dodge=True, alpha=0.5, jitter=True)
plt.show()
```

44.10.3 Scatter and relationship plots

```
# Scatter with regression line
sns.regplot(data=tips, x="total_bill", y="tip", scatter_kws={"alpha": 0.5})
plt.title("Tip vs Total Bill")
plt.show()

# Scatter with colour and size encoding
sns.scatterplot(data=tips, x="total_bill", y="tip",
                hue="day", size="size", sizes=(20, 200))
plt.show()

# Pair plot --- all variables against each other
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue="species", diag_kind="kde")
plt.show()
```

44.10.4 Heatmaps

```
# Correlation matrix
flights = sns.load_dataset("flights")
pivot   = flights.pivot_table(index="month", columns="year", values="passengers

fig, ax = plt.subplots(figsize=(10, 6))
sns.heatmap(
    pivot,
    annot=True,          # show numbers in cells
    fmt="d",             # format as integer
    cmap="YlOrRd",       # colour map: yellow -> orange -> red
    linewidths=0.5,
    ax=ax,
```

```
)
ax.set_title("Passengers per Month and Year")
plt.tight_layout()
plt.show()

# Correlation heatmap
corr = tips.select_dtypes("number").corr()
sns.heatmap(corr, annot=True, cmap="coolwarm", vmin=-1, vmax=1, center=0)
plt.title("Feature Correlations")
plt.show()
```

44.11 Plotting Directly from pandas

pandas has a `.plot()` method that wraps matplotlib:

```
import pandas as pd

df = pd.DataFrame({
    "month":    ["Jan", "Feb", "Mar", "Apr", "May"],
    "online":   [20000, 24000, 22000, 28000, 31000],
    "instore":  [15000, 16000, 14000, 17000, 18000],
})
df.set_index("month", inplace=True)

# Grouped bar chart
df.plot(kind="bar", figsize=(8, 5), color=["steelblue", "coral"])
plt.title("Sales by Channel")
plt.ylabel("Revenue ($)")
plt.xticks(rotation=0)
plt.tight_layout()
plt.show()

# Line chart
df.plot(kind="line", figsize=(8, 5), marker="o")
plt.title("Monthly Sales Trend")
plt.show()

# Stacked area chart
df.plot(kind="area", figsize=(8, 5), alpha=0.6, stacked=True)
plt.title("Stacked Revenue by Channel")
plt.show()
```

44.12 Saving High-Quality Figures

```
plt.savefig("chart.png",  dpi=300, bbox_inches="tight")   # for screen/web
plt.savefig("chart.pdf",  bbox_inches="tight")             # for print (vector)
plt.savefig("chart.svg",  bbox_inches="tight")             # for web (scalable)
```

`bbox_inches="tight"` prevents labels from being cropped. `dpi=300` is print-quality.

44.13 Project: Complete Sales Dashboard

```
"""
dashboard.py --- A 2x3 sales dashboard using matplotlib + seaborn.
"""
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import seaborn as sns
import pandas as pd
import numpy as np

sns.set_theme(style="whitegrid", palette="muted")
np.random.seed(42)

# ── Generate sample data ──────────────────────────────────────────

months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

monthly = pd.DataFrame({
    "month":    months,
    "revenue":  np.random.normal(50000, 8000, 12).clip(30000),
    "orders":   np.random.randint(200, 600, 12),
    "returns":  np.random.randint(5, 30, 12),
})

categories = pd.DataFrame({
    "category": ["Electronics", "Clothing", "Food", "Books", "Sports"],
    "revenue":  [45000, 32000, 28000, 15000, 19000],
})

order_values = np.random.lognormal(mean=5.2, sigma=0.5, size=500)

# ── Create dashboard ──────────────────────────────────────────────
```

```
fig = plt.figure(figsize=(16, 10))
fig.suptitle("Annual Sales Dashboard", fontsize=18, fontweight="bold", y=0.98)
gs  = gridspec.GridSpec(2, 3, figure=fig, hspace=0.4, wspace=0.35)

# 1 --- Revenue trend (top-left, spans 2 columns)
ax1 = fig.add_subplot(gs[0, :2])
ax1.plot(months, monthly["revenue"] / 1000, "o-",
         color="#2196F3", linewidth=2.5, markersize=6)
ax1.fill_between(range(12), monthly["revenue"] / 1000,
                 alpha=0.15, color="#2196F3")
ax1.set_title("Monthly Revenue (£000s)")
ax1.set_ylabel("Revenue (£000s)")
ax1.tick_params(axis="x", rotation=45)
ax1.grid(True, alpha=0.4)

# 2 --- Category breakdown (top-right)
ax2 = fig.add_subplot(gs[0, 2])
colors = ["#2196F3", "#FF9800", "#4CAF50", "#9C27B0", "#F44336"]
ax2.barh(categories["category"], categories["revenue"] / 1000,
         color=colors, edgecolor="white")
ax2.set_title("Revenue by Category (£000s)")
ax2.set_xlabel("Revenue (£000s)")
ax2.invert_yaxis()

# 3 --- Orders per month (bottom-left)
ax3 = fig.add_subplot(gs[1, 0])
ax3.bar(months, monthly["orders"], color="#4CAF50", edgecolor="white")
ax3.set_title("Orders per Month")
ax3.set_ylabel("Orders")
ax3.tick_params(axis="x", rotation=90)

# 4 --- Order value distribution (bottom-middle)
ax4 = fig.add_subplot(gs[1, 1])
ax4.hist(order_values, bins=35, color="#FF9800", edgecolor="white", alpha=0.85)
ax4.axvline(np.median(order_values), color="red", linestyle="--",
            label=f"Median £{np.median(order_values):.0f}")
ax4.set_title("Order Value Distribution")
ax4.set_xlabel("Order Value (£)")
ax4.set_ylabel("Frequency")
ax4.legend(fontsize=8)

# 5 --- Returns heatmap (bottom-right)
ax5 = fig.add_subplot(gs[1, 2])
```

```
quarters = np.array(monthly["returns"]).reshape(4, 3)
sns.heatmap(quarters,
            annot=True, fmt="d", cmap="Reds",
            xticklabels=["M1", "M2", "M3"],
            yticklabels=["Q1", "Q2", "Q3", "Q4"],
            ax=ax5, cbar=False)
ax5.set_title("Returns by Quarter/Month")

plt.savefig("sales_dashboard.png", dpi=150, bbox_inches="tight")
print("Dashboard saved to sales_dashboard.png")
plt.show()
```

44.14 What You Learned in This Chapter

- matplotlib uses a **Figure** (canvas) and **Axes** (individual plot) model. `fig, ax = plt.subplots()` gives you both.
- `ax.plot()` -> line chart, `ax.bar()` -> bar chart, `ax.hist()` -> histogram, `ax.scatter()` -> scatter, `ax.boxplot()` -> box plot, `ax.pie()` -> pie chart.
- Label everything: `ax.set_title()`,`ax.set_xlabel()`,`ax.set_ylabel()`, `ax.legend()`.
- `plt.subplots(rows, cols)` creates a grid of charts. Access each with `axes[row, col]`.
- `plt.savefig("file.png", dpi=300, bbox_inches="tight")` saves in PNG, PDF, or SVG.
- seaborn wraps matplotlib with better defaults. `sns.set_theme()` styles all charts.
- seaborn accepts DataFrames directly: `sns.boxplot(data=df, x="col", y="col2")`.
- `sns.histplot`, `sns.kdeplot`, `sns.boxplot`, `sns.violinplot`, `sns.barplot`, `sns.scatterplot`, `sns.regplot`, `sns.pairplot`, `sns.heatmap` cover almost every use case.
- pandas `.plot()` wraps matplotlib for quick charts directly from a DataFrame.

44.15 What's Next?

Chapter 45 covers **Machine Learning with scikit-learn** — using the DataFrames and visualisations you've built to train models that predict numbers, classify data, and find patterns automatically.

Chapter 45

Chapter 45: Machine Learning Basics with scikit-learn

Machine learning sounds intimidating. It isn't.

At its core, machine learning is this: you show a program examples, and it learns a pattern. Then you give it new data it's never seen, and it predicts the answer.

You've been doing something similar your whole life. You've seen thousands of emails. You know what spam looks like. A spam filter does the same thing — except it learned from millions of examples instead of yours.

scikit-learn is Python's standard machine learning library. It has a simple, consistent API and covers most practical use cases.

45.1 Setup

```
pip install scikit-learn pandas numpy matplotlib seaborn
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import ...   # we'll import specific parts as we go
```

45.2 The Two Types of Problems

Supervised learning — you have labeled examples. You learn to predict the label for new data. - *Regression*: predict a number (house price, temperature, salary) - *Classification*: predict a category (spam/not spam, cat/dog, disease/healthy)

Unsupervised learning — you have data with no labels. You find hidden structure. - *Clustering*: group similar data points - *Dimensionality reduction*: compress features while keeping information

This chapter focuses on supervised learning — it's where most beginners start and where most practical ML problems live.

45.3 The scikit-learn API

Every model in scikit-learn follows the same three-step API:

```
from sklearn.some_module import SomeModel

model = SomeModel()           # 1. Create the model
model.fit(X_train, y_train)   # 2. Train it on your data
predictions = model.predict(X_test)    # 3. Make predictions
score = model.score(X_test, y_test)    # 4. Evaluate it
```

Learn this pattern once and you can use any of scikit-learn's 40+ algorithms.

45.4 Step 1 — Preparing Your Data

45.4.1 Load a dataset

```
from sklearn.datasets import load_iris, load_boston, fetch_california_housing
import pandas as pd

# Built-in datasets (great for learning)
iris    = load_iris(as_frame=True)
housing = fetch_california_housing(as_frame=True)

X = housing.data      # features (DataFrame)
y = housing.target    # labels (Series)

print(X.shape)        # (20640, 8) --- 20,640 rows, 8 features
```

```
print(y.shape)          # (20640,)
print(X.head())
```

45.4.2 Train/test split

You must evaluate your model on data it has **never seen**. Always split your data first:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,      # 20% for testing, 80% for training
    random_state=42,    # reproducible split
)

print(f"Train: {X_train.shape}, Test: {X_test.shape}")
# Train: (16512, 8), Test: (4128, 8)
```

45.4.3 Feature scaling

Many algorithms (linear models, SVMs, k-nearest neighbours) are sensitive to feature scale. A feature measured in millions dominates one measured in units.

```
from sklearn.preprocessing import StandardScaler

scaler  = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)    # learn mean/std, then scale
X_test_scaled  = scaler.transform(X_test)         # use SAME mean/std as train
```

Critical rule: fit the scaler on training data only. Never fit on test data — that would be "data leakage," letting test information sneak into training.

45.5 Linear Regression — Predict a Number

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# ── Toy example ──────────────────────────────────────────────

np.random.seed(42)
X_toy   = np.random.rand(100, 1) * 10
y_toy   = 2.5 * X_toy.ravel() + np.random.randn(100) * 3
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X_toy, y_toy, test_size=0.2, random_s

model = LinearRegression()
model.fit(X_tr, y_tr)

y_pred = model.predict(X_te)

print(f"Coefficient (slope): {model.coef_[0]:.2f}")    # ~2.5
print(f"Intercept:           {model.intercept_:.2f}")
print(f"R^2 score:            {r2_score(y_te, y_pred):.3f}")    # 1.0 = perfect
print(f"RMSE:                {np.sqrt(mean_squared_error(y_te, y_pred)):.3f}")

# Plot
plt.figure(figsize=(7, 4))
plt.scatter(X_te, y_te, alpha=0.6, label="Actual")
plt.plot(X_te, y_pred, color="red", linewidth=2, label="Predicted")
plt.title("Linear Regression")
plt.legend()
plt.tight_layout()
plt.show()
```

45.5.1 On real data — California housing prices

```
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np

housing = fetch_california_housing(as_frame=True)
X, y    = housing.data, housing.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random

scaler  = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s  = scaler.transform(X_test)

model = LinearRegression()
model.fit(X_train_s, y_train)

y_pred = model.predict(X_test_s)
```

```
print(f"R^2:   {r2_score(y_test, y_pred):.3f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred)):.3f}")
# R^2:   0.606
# RMSE: 0.745
```

R^2 of 0.606 means the model explains about 60% of the variance in house prices. Not great — the relationship is non-linear. Let's try a better model.

45.6 Decision Trees and Random Forests — Flexible and Powerful

A **decision tree** splits the data on feature values, like a flowchart:

```
Is MedInc > 3.5?
  Yes -> Is HouseAge > 20?
           Yes -> predict 2.8
           No  -> predict 3.5
  No  -> predict 1.2
```

A **random forest** builds hundreds of decision trees on random subsets of the data and features, then averages their predictions. This reduces overfitting and greatly improves accuracy.

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(
    n_estimators=100,    # number of trees
    max_depth=None,      # let trees grow fully
    random_state=42,
    n_jobs=-1,           # use all CPU cores
)
rf.fit(X_train, y_train)

y_pred_rf = rf.predict(X_test)
print(f"R^2:   {r2_score(y_test, y_pred_rf):.3f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_rf)):.3f}")
# R^2:   0.805
# RMSE: 0.505

# Feature importance --- which features matter most?
importances = pd.Series(rf.feature_importances_, index=X.columns)
importances.sort_values().plot(kind="barh", figsize=(7, 4))
plt.title("Feature Importances")
```

```
plt.tight_layout()
plt.show()
```

The random forest jumps from 0.606 to 0.805 R^2 — no manual feature engineering, no scaling required.

45.7 Classification — Predict a Category

45.7.1 Logistic Regression

Despite the name, logistic regression is a **classification** algorithm. It predicts the probability that something belongs to a class.

```
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

data = load_breast_cancer(as_frame=True)
X, y = data.data, data.target  # 0=malignant, 1=benign

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

scaler   = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s  = scaler.transform(X_test)

model = LogisticRegression(max_iter=1000)
model.fit(X_train_s, y_train)

y_pred = model.predict(X_test_s)
print(classification_report(y_test, y_pred, target_names=data.target_names))
```

Output:

```
              precision    recall  f1-score   support
   malignant       0.97      0.93      0.95        43
      benign       0.96      0.98      0.97        71
    accuracy                           0.96       114
```

45.7.2 Understanding the metrics

- **Precision** — of all the times we said "malignant," how often were we right?
- **Recall** — of all actual malignant cases, how many did we catch?
- **F1-score** — harmonic mean of precision and recall. Use it when classes are imbalanced.
- **Accuracy** — overall percentage correct. Misleading when classes are very unequal.

45.7.3 Confusion matrix

```
import seaborn as sns

cm = confusion_matrix(y_test, y_pred)
sns.heatmap(
    cm, annot=True, fmt="d", cmap="Blues",
    xticklabels=data.target_names,
    yticklabels=data.target_names,
)
plt.title("Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.show()
```

45.7.4 Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
rf_clf.fit(X_train, y_train)

y_pred_rf = rf_clf.predict(X_test)
print(classification_report(y_test, y_pred_rf, target_names=data.target_names))
# accuracy: 0.96-0.97
```

45.8 Cross-Validation — A More Reliable Score

One train/test split can be lucky or unlucky. Cross-validation gives a more reliable estimate:

```
from sklearn.model_selection import cross_val_score

model = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
scores = cross_val_score(model, X, y, cv=5, scoring="accuracy")

print(f"CV scores:  {scores.round(3)}")
print(f"Mean:       {scores.mean():.3f}")
print(f"Std:        {scores.std():.3f}")
# CV scores:  [0.974 0.956 0.965 0.965 0.982]
# Mean:       0.968
# Std:        0.009
```

5-fold CV splits the data into 5 parts, trains on 4 and tests on 1, rotates 5 times. The mean score and standard deviation tell you how well and how consistently the model performs.

45.9 Pipelines — Prevent Data Leakage

Manually applying a scaler before cross-validation is dangerous — if you fit the scaler on all the data before the CV split, you've leaked test information into training.

The fix is a **Pipeline** that chains preprocessing and model together:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("model",  LogisticRegression(max_iter=1000)),
])

scores = cross_val_score(pipe, X, y, cv=5, scoring="accuracy")
print(f"Pipeline CV: {scores.mean():.3f} ± {scores.std():.3f}")
```

The pipeline scales inside each fold — scaler never sees the test fold. This is the correct way to evaluate models.

45.10 Hyperparameter Tuning with GridSearchCV

Every model has hyperparameters — settings you choose, not learned from data. Finding the best settings is called hyperparameter tuning.

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_breast_cancer

X, y = load_breast_cancer(return_X_y=True)

param_grid = {
    "n_estimators": [50, 100, 200],
    "max_depth":    [None, 5, 10],
    "min_samples_split": [2, 5],
}

grid = GridSearchCV(
    RandomForestClassifier(random_state=42, n_jobs=-1),
    param_grid,
    cv=5,
    scoring="accuracy",
    n_jobs=-1,
    verbose=1,
)
grid.fit(X, y)

print(f"Best params: {grid.best_params_}")
print(f"Best score:  {grid.best_score_:.3f}")

best_model = grid.best_estimator_
```

`GridSearchCV` tries every combination of parameters (3x3x2 = 18 combinations here) and picks the best using cross-validation.

For large grids, use `RandomizedSearchCV` instead — it samples a fixed number of random combinations, which is much faster:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dist = {
    "n_estimators":     randint(50, 500),
    "max_depth":        [None, 5, 10, 20],
    "min_samples_split": randint(2, 20),
```

```
}

random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_dist,
    n_iter=20,       # try 20 random combinations
    cv=5,
    scoring="accuracy",
    random_state=42,
)
random_search.fit(X, y)
print(random_search.best_params_)
```

45.11 Unsupervised Learning — Clustering

When you don't have labels, clustering finds natural groups:

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Generate sample data
from sklearn.datasets import make_blobs
X, true_labels = make_blobs(n_samples=300, centers=4, random_state=42)

# Scale
scaler = StandardScaler()
X_s    = scaler.fit_transform(X)

# Fit KMeans
kmeans = KMeans(n_clusters=4, random_state=42, n_init="auto")
kmeans.fit(X_s)
labels = kmeans.labels_

# Plot
plt.figure(figsize=(8, 5))
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap="tab10", alpha=0.7, s=40)
plt.scatter(
    kmeans.cluster_centers_[:, 0] * X.std(0)[0] + X.mean(0)[0],
    kmeans.cluster_centers_[:, 1] * X.std(0)[1] + X.mean(0)[1],
    marker="X", s=200, color="black", label="Centroids"
)
```

```
plt.title("KMeans Clustering")
plt.legend()
plt.tight_layout()
plt.show()
```

45.12 Full Project — House Price Predictor

```
"""
house_price_predictor.py

Predict California median house values using a
Random Forest with a Pipeline and GridSearchCV.
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error


def load_data():
    housing = fetch_california_housing(as_frame=True)
    X, y    = housing.data, housing.target
    print(f"Dataset: {X.shape[0]:,} samples, {X.shape[1]} features")
    print(f"Target range: ${y.min()*100_000:,.0f} -- ${y.max()*100_000:,.0f}")
    return X, y


def build_pipeline():
    return Pipeline([
        ("scaler", StandardScaler()),
        ("model",  RandomForestRegressor(
            n_estimators=200,
            max_depth=None,
            random_state=42,
            n_jobs=-1,
        )),
    ])
```

```python
def evaluate(model, X_test, y_test):
    y_pred = model.predict(X_test)
    r2   = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    print(f"R^2 Score:  {r2:.3f}  (1.0 = perfect)")
    print(f"RMSE:      ${rmse * 100_000:,.0f}")
    return y_pred


def plot_predictions(y_test, y_pred):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Predicted vs Actual
    axes[0].scatter(y_test, y_pred, alpha=0.3, s=10, color="steelblue")
    lims = [min(y_test.min(), y_pred.min()), max(y_test.max(), y_pred.max())]
    axes[0].plot(lims, lims, "r--", linewidth=2, label="Perfect prediction")
    axes[0].set_xlabel("Actual Price ($100k)")
    axes[0].set_ylabel("Predicted Price ($100k)")
    axes[0].set_title("Predicted vs Actual")
    axes[0].legend()

    # Residuals
    residuals = y_test - y_pred
    axes[1].hist(residuals, bins=50, color="coral", edgecolor="white", alpha=0.
    axes[1].axvline(0, color="black", linewidth=2)
    axes[1].set_title("Residuals Distribution")
    axes[1].set_xlabel("Prediction Error ($100k)")
    axes[1].set_ylabel("Count")

    plt.tight_layout()
    plt.savefig("house_predictions.png", dpi=150)
    plt.show()


def feature_importance(model, feature_names):
    rf = model.named_steps["model"]
    importances = pd.Series(rf.feature_importances_, index=feature_names)
    importances.sort_values().plot(kind="barh", figsize=(8, 4), color="steelblu
    plt.title("Feature Importances")
    plt.tight_layout()
    plt.savefig("feature_importance.png", dpi=150)
    plt.show()
```

```
if __name__ == "__main__":
    X, y = load_data()

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    pipe = build_pipeline()

    print("\nTraining model...")
    pipe.fit(X_train, y_train)

    print("\n=== Test Set Results ===")
    y_pred = evaluate(pipe, X_test, y_test)

    print("\n=== 5-Fold Cross-Validation ===")
    cv_scores = cross_val_score(pipe, X, y, cv=5, scoring="r2", n_jobs=-1)
    print(f"CV R^2 scores: {cv_scores.round(3)}")
    print(f"Mean: {cv_scores.mean():.3f}  Std: {cv_scores.std():.3f}")

    plot_predictions(y_test, y_pred)
    feature_importance(pipe, X.columns)
```

Output:

```
Dataset: 20,640 samples, 8 features
Target range: $14,999 -- $500,001

=== Test Set Results ===
R^2 Score:  0.817  (1.0 = perfect)
RMSE:       $52,831

=== 5-Fold Cross-Validation ===
CV R^2 scores: [0.808 0.812 0.821 0.806 0.810]
Mean: 0.811  Std: 0.005
```

45.13 Choosing the Right Algorithm

Problem	First try	If you need more
Predict a number	Linear Regression	Random Forest, Gradient Boosting
Classify into 2+ classes	Logistic Regression	Random Forest, SVM
Find groups	KMeans	DBSCAN, Agglomerative
Reduce dimensions	PCA	UMAP, t-SNE

Start simple. Add complexity only when the simpler model doesn't perform well enough.

45.14 What You Learned in This Chapter

- Machine learning has two main types: **supervised** (predict from labeled examples) and **unsupervised** (find structure without labels).
- The scikit-learn API is always: `model.fit(X_train, y_train)` -> `model.predict(X_test)` -> `model.score(X_test, y_test)`.
- Always split data with `train_test_split`. Never evaluate on training data.
- Scale features with `StandardScaler`. Fit it only on training data.
- **Linear Regression** predicts continuous values. Evaluate with R^2 and RMSE.
- **Random Forest** is often the best starting point — handles non-linearity, doesn't require scaling, shows feature importances.
- **Logistic Regression** classifies. Evaluate with precision, recall, F1, and a confusion matrix.
- **Cross-validation** (`cross_val_score`) gives more reliable performance estimates than a single split.
- **Pipelines** chain preprocessing and models together to prevent data leakage during cross-validation.
- **GridSearchCV** and **RandomizedSearchCV** automate hyperparameter tuning.
- **KMeans** clusters data without labels.

45.15 What's Next?

Chapter 46 covers **Automation and Scripting** — using Python to automate repetitive tasks: renaming files, sending emails, processing PDFs, working with Excel, and scheduling tasks to run daily.

Chapter 46

Chapter 46: Automation and Scripting

The best programmers are lazy — in the best way. They hate doing the same thing twice. When they spot a repetitive task, they write a script to do it for them.

Renaming 500 files? Three lines of Python. Sending a weekly report by email? Ten lines. Downloading 100 PDFs and merging them? Twenty lines. Resizing 200 images? Fifteen lines.

This chapter is your automation toolkit.

46.1 File System Automation with `pathlib` and `shutil`

You already know `pathlib` from Chapter 8. Here's how to put it to work at scale.

46.1.1 Rename files in bulk

```
from pathlib import Path

folder = Path("photos")

# Rename all .jpeg files to .jpg
for f in folder.glob("*.jpeg"):
    f.rename(f.with_suffix(".jpg"))
```

```
    print(f"Renamed: {f.name} -> {f.stem}.jpg")

# Add a date prefix to all .txt files
from datetime import date
today = date.today().strftime("%Y-%m-%d")

for f in folder.glob("*.txt"):
    new_name = f.parent / f"{today}_{f.name}"
    f.rename(new_name)
    print(f"Renamed: {f.name} -> {new_name.name}")
```

46.1.2 Organise files by extension

```
from pathlib import Path
import shutil

def organise_downloads(source: str = "Downloads") -> None:
    """Sort files in a folder into subfolders by extension."""
    source_dir = Path(source)

    extension_map = {
        "Images":    {".jpg", ".jpeg", ".png", ".gif", ".webp", ".svg"},
        "Documents": {".pdf", ".doc", ".docx", ".txt", ".md", ".xlsx"},
        "Videos":    {".mp4", ".mov", ".avi", ".mkv"},
        "Audio":     {".mp3", ".wav", ".flac", ".aac"},
        "Archives":  {".zip", ".tar", ".gz", ".rar", ".7z"},
        "Code":      {".py", ".js", ".ts", ".html", ".css", ".json"},
    }

    # Build reverse lookup: extension -> folder name
    ext_to_folder = {}
    for folder_name, extensions in extension_map.items():
        for ext in extensions:
            ext_to_folder[ext] = folder_name

    moved = 0
    for file in source_dir.iterdir():
        if not file.is_file():
            continue
        folder_name = ext_to_folder.get(file.suffix.lower(), "Other")
        dest_dir    = source_dir / folder_name
        dest_dir.mkdir(exist_ok=True)
        dest        = dest_dir / file.name
```

```
        # Avoid overwriting existing files
        if dest.exists():
            stem = file.stem
            i    = 1
            while dest.exists():
                dest = dest_dir / f"{stem}_{i}{file.suffix}"
                i   += 1

        shutil.move(str(file), str(dest))
        print(f"Moved: {file.name} -> {folder_name}/")
        moved += 1

    print(f"\nOrganised {moved} files.")


organise_downloads("Downloads")
```

46.1.3 Copy, move, and delete with `shutil`

```
import shutil
from pathlib import Path

# Copy a file
shutil.copy("report.pdf", "backup/report.pdf")

# Copy a file and preserve metadata
shutil.copy2("report.pdf", "backup/report.pdf")

# Copy an entire directory
shutil.copytree("project/", "project_backup/")

# Move (rename) a file or directory
shutil.move("old_name.txt", "new_name.txt")
shutil.move("old_folder/", "archive/old_folder/")

# Delete a directory tree
shutil.rmtree("temp_folder/")

# Get folder size
def folder_size(path: str) -> int:
    return sum(f.stat().st_size for f in Path(path).rglob("*") if f.is_file())

size_bytes = folder_size("project/")
print(f"Folder size: {size_bytes / 1_000_000:.1f} MB")
```

46.1.4 Find files matching a pattern

```
from pathlib import Path

# All Python files, recursively
for f in Path(".").rglob("*.py"):
    print(f)

# Files larger than 10 MB
for f in Path("Downloads").rglob("*"):
    if f.is_file() and f.stat().st_size > 10 * 1024 * 1024:
        print(f"{f.name}: {f.stat().st_size / 1_000_000:.1f} MB")

# Files modified in the last 7 days
import time
week_ago = time.time() - 7 * 24 * 3600
for f in Path(".").rglob("*"):
    if f.is_file() and f.stat().st_mtime > week_ago:
        print(f)
```

46.2 Scheduling Tasks with `schedule`

```
pip install schedule
```

```
import schedule
import time
from datetime import datetime

def backup_database():
    """Run every day at 2am."""
    import shutil
    shutil.copy("production.db", f"backups/backup_{datetime.now():%Y%m%d_%H%M}.
    print(f"Backup completed at {datetime.now()}")

def send_daily_report():
    print(f"Sending report at {datetime.now()}")
    # ... send email, generate PDF, etc.

def cleanup_temp():
    import shutil
    shutil.rmtree("temp/", ignore_errors=True)
    print("Temp files cleaned")
```

```
# Schedule tasks
schedule.every().day.at("02:00").do(backup_database)
schedule.every().day.at("08:00").do(send_daily_report)
schedule.every().hour.do(cleanup_temp)
schedule.every(30).minutes.do(lambda: print("Still running..."))
schedule.every().monday.at("09:00").do(send_daily_report)

# Run the scheduler
print("Scheduler running. Press Ctrl+C to stop.")
while True:
    schedule.run_pending()
    time.sleep(60)    # check every minute
```

46.3 Sending Emails with `smtplib`

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text      import MIMEText
from email.mime.base      import MIMEBase
from email                import encoders
from pathlib              import Path


def send_email(
    to:          str | list[str],
    subject:     str,
    body:        str,
    attachments: list[str] = (),
    html:        bool = False,
    smtp_host:   str = "smtp.gmail.com",
    smtp_port:   int = 587,
    username:    str = "",
    password:    str = "",
) -> None:
    """Send an email with optional attachments."""
    msg = MIMEMultipart()
    msg["From"]    = username
    msg["To"]      = to if isinstance(to, str) else ", ".join(to)
    msg["Subject"] = subject

    # Body
    mime_type = "html" if html else "plain"
```

```
    msg.attach(MIMEText(body, mime_type))

    # Attachments
    for path_str in attachments:
        path = Path(path_str)
        with open(path, "rb") as f:
            part = MIMEBase("application", "octet-stream")
            part.set_payload(f.read())
        encoders.encode_base64(part)
        part.add_header("Content-Disposition", f"attachment; filename={path.nam
        msg.attach(part)

    # Send
    with smtplib.SMTP(smtp_host, smtp_port) as server:
        server.starttls()
        server.login(username, password)
        server.send_message(msg)

    recipients = to if isinstance(to, list) else [to]
    print(f"Email sent to {len(recipients)} recipient(s): {subject}")


# Usage
import os

send_email(
    to       = "boss@example.com",
    subject  = "Daily Sales Report",
    body     = "<h1>Report</h1><p>See attached PDF.</p>",
    html     = True,
    attachments = ["report.pdf"],
    username = os.environ["EMAIL_ADDRESS"],
    password = os.environ["EMAIL_PASSWORD"],
)
```

Use environment variables for credentials — never hardcode passwords. For Gmail, create an App Password in your Google Account security settings.

46.4 Working with PDFs

```
pip install pypdf reportlab
```

46.4.1 Read and extract text

```
from pypdf import PdfReader

reader = PdfReader("annual_report.pdf")

print(f"Pages: {len(reader.pages)}")

# Extract text from all pages
full_text = ""
for page in reader.pages:
    full_text += page.extract_text() + "\n"

print(full_text[:500])
```

46.4.2 Merge PDFs

```
from pypdf import PdfReader, PdfWriter

def merge_pdfs(paths: list[str], output: str) -> None:
    writer = PdfWriter()
    for path in paths:
        reader = PdfReader(path)
        for page in reader.pages:
            writer.add_page(page)
    with open(output, "wb") as f:
        writer.write(f)
    print(f"Merged {len(paths)} PDFs into {output}")


merge_pdfs(["invoice_1.pdf", "invoice_2.pdf", "invoice_3.pdf"], "all_invoices.pdf")
```

46.4.3 Split a PDF

```
from pypdf import PdfReader, PdfWriter
from pathlib import Path

def split_pdf(path: str, pages_per_chunk: int = 10) -> None:
    reader  = PdfReader(path)
    stem    = Path(path).stem
    total   = len(reader.pages)

    for start in range(0, total, pages_per_chunk):
        writer = PdfWriter()
        end    = min(start + pages_per_chunk, total)
```

```
        for i in range(start, end):
            writer.add_page(reader.pages[i])
        output = f"{stem}_part_{start // pages_per_chunk + 1}.pdf"
        with open(output, "wb") as f:
            writer.write(f)
        print(f"Created {output} (pages {start+1}--{end})")


split_pdf("big_report.pdf", pages_per_chunk=5)
```

46.4.4 Generate a PDF from scratch with ReportLab

```
from reportlab.pdfgen  import canvas
from reportlab.lib.pagesizes import A4

def generate_invoice(filename: str, invoice_data: dict) -> None:
    c = canvas.Canvas(filename, pagesize=A4)
    width, height = A4

    # Title
    c.setFont("Helvetica-Bold", 20)
    c.drawString(50, height - 80, "INVOICE")

    # Invoice details
    c.setFont("Helvetica", 12)
    c.drawString(50, height - 120, f"Invoice #: {invoice_data['number']}")
    c.drawString(50, height - 140, f"Date: {invoice_data['date']}")
    c.drawString(50, height - 160, f"To: {invoice_data['client']}")

    # Table header
    y = height - 220
    c.setFont("Helvetica-Bold", 11)
    c.drawString(50,  y, "Description")
    c.drawString(350, y, "Qty")
    c.drawString(420, y, "Price")
    c.drawString(490, y, "Total")
    c.line(50, y - 5, 550, y - 5)

    # Line items
    c.setFont("Helvetica", 11)
    total = 0
    for item in invoice_data["items"]:
        y -= 25
        subtotal = item["qty"] * item["price"]
```

```
        total   += subtotal
        c.drawString(50,  y, item["description"])
        c.drawString(350, y, str(item["qty"]))
        c.drawString(420, y, f"${item['price']:.2f}")
        c.drawString(490, y, f"${subtotal:.2f}")

    # Total
    y -= 30
    c.line(50, y, 550, y)
    c.setFont("Helvetica-Bold", 12)
    c.drawString(420, y - 20, f"TOTAL: ${total:.2f}")

    c.save()
    print(f"Invoice saved to {filename}")


generate_invoice("invoice_001.pdf", {
    "number": "001",
    "date":   "2026-03-09",
    "client": "Acme Corp",
    "items": [
        {"description": "Python Consulting",  "qty": 10, "price": 150.00},
        {"description": "Code Review",        "qty": 5,  "price": 75.00},
        {"description": "Training Session",   "qty": 2,  "price": 500.00},
    ],
})
```

46.5 Image Processing with Pillow

```
pip install Pillow
```

```
from PIL import Image, ImageDraw, ImageFont
from pathlib import Path


# ── Basic operations ──────────────────────────────────────────────

img = Image.open("photo.jpg")
print(img.size)     # (1920, 1080)
print(img.mode)     # RGB

# Resize
```

```
img_small = img.resize((800, 450))
img_small.save("photo_800.jpg", quality=85)

# Crop
region = img.crop((100, 100, 600, 400))    # (left, top, right, bottom)
region.save("cropped.jpg")

# Rotate
rotated = img.rotate(90, expand=True)
rotated.save("rotated.jpg")

# Convert to greyscale
grey = img.convert("L")
grey.save("grey.jpg")

# Convert format
img.save("photo.png")    # saves as PNG regardless of original format


# ── Batch resize all images in a folder ──────────────────────────────

def batch_resize(
    folder:    str,
    max_width:  int = 800,
    max_height: int = 600,
    quality:    int = 85,
) -> None:
    output_dir = Path(folder) / "resized"
    output_dir.mkdir(exist_ok=True)

    image_exts = {".jpg", ".jpeg", ".png", ".webp"}
    files      = [f for f in Path(folder).iterdir()
                  if f.suffix.lower() in image_exts]

    for f in files:
        img = Image.open(f)
        img.thumbnail((max_width, max_height))    # resize in-place, keeps aspec
        dest = output_dir / f.name
        img.save(dest, quality=quality)
        print(f"Resized: {f.name} -> {img.size}")

    print(f"Done. {len(files)} images resized.")
```

```
batch_resize("product_photos")


# ── Add a watermark ──────────────────────────────────────────────

def add_watermark(image_path: str, text: str, output_path: str) -> None:
    img  = Image.open(image_path).convert("RGBA")
    overlay = Image.new("RGBA", img.size, (255, 255, 255, 0))
    draw    = ImageDraw.Draw(overlay)

    # Position: bottom-right
    font_size = max(20, img.width // 30)
    try:
        font = ImageFont.truetype("arial.ttf", font_size)
    except OSError:
        font = ImageFont.load_default()

    bbox  = draw.textbbox((0, 0), text, font=font)
    tw    = bbox[2] - bbox[0]
    th    = bbox[3] - bbox[1]
    x     = img.width  - tw - 20
    y     = img.height - th - 20

    draw.text((x, y), text, fill=(255, 255, 255, 128), font=font)

    watermarked = Image.alpha_composite(img, overlay)
    watermarked.convert("RGB").save(output_path)
    print(f"Watermark added: {output_path}")


add_watermark("product.jpg", "(c) MyBrand 2026", "product_watermarked.jpg")
```

46.6 Automating Excel with `openpyxl`

```
pip install openpyxl
```

```
import openpyxl
from openpyxl.styles import Font, PatternFill, Alignment, Border, Side
from openpyxl.chart  import BarChart, Reference


def generate_sales_report(data: list[dict], filename: str = "report.xlsx") -> None
```

```
wb = openpyxl.Workbook()
ws = wb.active
ws.title = "Sales Report"

# Header row
headers = ["Month", "Revenue", "Orders", "Avg Order"]
header_font = Font(bold=True, color="FFFFFF")
header_fill = PatternFill("solid", fgColor="2196F3")

for col, header in enumerate(headers, start=1):
    cell = ws.cell(row=1, column=col, value=header)
    cell.font      = header_font
    cell.fill      = header_fill
    cell.alignment = Alignment(horizontal="center")

# Data rows
for row_idx, row in enumerate(data, start=2):
    ws.cell(row=row_idx, column=1, value=row["month"])
    ws.cell(row=row_idx, column=2, value=row["revenue"])
    ws.cell(row=row_idx, column=3, value=row["orders"])
    ws.cell(row=row_idx, column=4, value=row["revenue"] / row["orders"])

    # Alternate row colour
    if row_idx % 2 == 0:
        fill = PatternFill("solid", fgColor="E3F2FD")
        for col in range(1, 5):
            ws.cell(row=row_idx, column=col).fill = fill

# Total row
total_row = len(data) + 2
ws.cell(row=total_row, column=1, value="TOTAL").font = Font(bold=True)
ws.cell(row=total_row, column=2,
        value=f"=SUM(B2:B{total_row-1})").font = Font(bold=True)
ws.cell(row=total_row, column=3,
        value=f"=SUM(C2:C{total_row-1})").font = Font(bold=True)

# Format numbers
for row in ws.iter_rows(min_row=2, max_row=total_row, min_col=2, max_col=4):
    for cell in row:
        cell.number_format = "#,##0.00"

# Column widths
ws.column_dimensions["A"].width = 12
ws.column_dimensions["B"].width = 15
```

```
    ws.column_dimensions["C"].width = 10
    ws.column_dimensions["D"].width = 15

    # Bar chart
    chart = BarChart()
    chart.title = "Monthly Revenue"
    chart.y_axis.title = "Revenue (£)"
    chart.x_axis.title = "Month"

    data_ref   = Reference(ws, min_col=2, min_row=1, max_row=len(data)+1)
    cats_ref   = Reference(ws, min_col=1, min_row=2, max_row=len(data)+1)
    chart.add_data(data_ref, titles_from_data=True)
    chart.set_categories(cats_ref)
    chart.shape = 4
    ws.add_chart(chart, "F2")

    wb.save(filename)
    print(f"Report saved to {filename}")


generate_sales_report([
    {"month": "Jan", "revenue": 42000, "orders": 180},
    {"month": "Feb", "revenue": 48500, "orders": 210},
    {"month": "Mar", "revenue": 45000, "orders": 195},
    {"month": "Apr", "revenue": 53000, "orders": 230},
    {"month": "May", "revenue": 61000, "orders": 265},
    {"month": "Jun", "revenue": 58000, "orders": 250},
])
```

46.7 Project: Automated Daily Report Generator

```
"""
daily_report.py

Generates a daily summary report and emails it.
Run this with: python daily_report.py
Or schedule it with cron / Task Scheduler.
"""
import os
import sqlite3
```

```
import smtplib
from datetime import date, timedelta
from email.mime.multipart import MIMEMultipart
from email.mime.text      import MIMEText
from email.mime.base      import MIMEBase
from email                import encoders
from pathlib              import Path
from reportlab.pdfgen     import canvas
from reportlab.lib.pagesizes import A4


DB_PATH      = "sales.db"
REPORT_DIR   = Path("reports")
REPORT_DIR.mkdir(exist_ok=True)


def fetch_yesterday_stats() -> dict:
    yesterday = (date.today() - timedelta(days=1)).isoformat()
    conn = sqlite3.connect(DB_PATH)
    cur  = conn.cursor()
    stats = cur.execute("""
        SELECT
            COUNT(*)           AS orders,
            SUM(total)         AS revenue,
            AVG(total)         AS avg_order,
            COUNT(DISTINCT customer_id) AS customers
        FROM orders
        WHERE DATE(created_at) = ?
    """, (yesterday,)).fetchone()
    conn.close()
    return {
        "date":      yesterday,
        "orders":    stats[0] or 0,
        "revenue":   stats[1] or 0.0,
        "avg_order": stats[2] or 0.0,
        "customers": stats[3] or 0,
    }


def generate_pdf_report(stats: dict) -> str:
    filename = REPORT_DIR / f"report_{stats['date']}.pdf"
    c = canvas.Canvas(str(filename), pagesize=A4)
    w, h = A4
```

```
    c.setFont("Helvetica-Bold", 18)
    c.drawString(50, h - 80, f"Daily Sales Report --- {stats['date']}")

    c.setFont("Helvetica", 13)
    items = [
        ("Total Orders",    f"{stats['orders']:,}"),
        ("Total Revenue",   f"£{stats['revenue']:,.2f}"),
        ("Avg Order Value", f"£{stats['avg_order']:.2f}"),
        ("Unique Customers", f"{stats['customers']:,}"),
    ]
    y = h - 140
    for label, value in items:
        c.setFont("Helvetica-Bold", 12)
        c.drawString(50, y, label + ":")
        c.setFont("Helvetica", 12)
        c.drawString(220, y, value)
        y -= 30

    c.save()
    return str(filename)


def send_report(pdf_path: str, stats: dict) -> None:
    msg = MIMEMultipart()
    msg["From"]    = os.environ["EMAIL_FROM"]
    msg["To"]      = os.environ["REPORT_RECIPIENT"]
    msg["Subject"] = f"Daily Report --- {stats['date']}"

    body = f"""
    <h2>Daily Sales Summary --- {stats['date']}</h2>
    <ul>
        <li><b>Orders:</b> {stats['orders']:,}</li>
        <li><b>Revenue:</b> £{stats['revenue']:,.2f}</li>
        <li><b>Avg Order:</b> £{stats['avg_order']:.2f}</li>
        <li><b>Customers:</b> {stats['customers']:,}</li>
    </ul>
    <p>Full report attached.</p>
    """
    msg.attach(MIMEText(body, "html"))

    with open(pdf_path, "rb") as f:
        part = MIMEBase("application", "octet-stream")
        part.set_payload(f.read())
    encoders.encode_base64(part)
```

```
    part.add_header("Content-Disposition",
                    f"attachment; filename={Path(pdf_path).name}")
    msg.attach(part)

    with smtplib.SMTP("smtp.gmail.com", 587) as server:
        server.starttls()
        server.login(os.environ["EMAIL_FROM"], os.environ["EMAIL_PASSWORD"])
        server.send_message(msg)

    print(f"Report emailed to {msg['To']}")


if __name__ == "__main__":
    stats   = fetch_yesterday_stats()
    pdf     = generate_pdf_report(stats)
    send_report(pdf, stats)
    print(f"Done. Report saved to {pdf}")
```

Schedule it on Linux/Mac with cron:

```
# Run at 7am every day
0 7 * * * /usr/bin/python3 /home/user/daily_report.py
```

Schedule it on Windows with Task Scheduler: 1. Open Task Scheduler -> Create Basic Task 2. Set trigger: Daily, 7:00 AM 3. Action: Start a program -> `python.exe` -> arguments: `C:\scripts\daily_report.py`

46.8 What You Learned in This Chapter

- `pathlib.Path.glob()` and `.rglob()` find files matching a pattern. `.rename()`, `.mkdir()`, `.stat()` manage files.
- `shutil.copy()`, `shutil.move()`, `shutil.rmtree()`, `shutil.copytree()` move and copy files and directories.
- `schedule` runs Python functions on a timer — daily, hourly, weekly, or on a custom interval.
- `smtplib` + `email.mime.*` sends emails with HTML bodies and file attachments. Always use environment variables for credentials.
- `pypdf.PdfReader` reads and extracts text from PDFs. `PdfWriter` merges and splits them. `reportlab` generates PDFs from scratch.
- `PIL.Image` (Pillow) opens, resizes, crops, rotates, converts, and saves images. `thumbnail()` resizes while preserving aspect ratio.
- `openpyxl` reads and writes Excel files with formatting, formulas, and charts.

- `schedule` + `smtplib` + `reportlab` together make a fully automated reporting pipeline.

46.9 What's Next?

Chapter 47 is the first of the **Advanced Projects** — you'll build a complete REST API with FastAPI, SQLAlchemy, JWT authentication, and a full pytest test suite from scratch.

Chapter 47

Chapter 47: Build a REST API with FastAPI and a Database

This is where everything clicks together.

You know FastAPI from Chapter 41. You know SQLAlchemy from Chapter 31. You know pytest from Chapter 28. You know JWT auth from Chapter 41. Now you're going to combine all of that into one coherent, production-ready project.

We're building a **Product Catalogue API** — a service that lets clients create, read, update, and delete products, organized into categories, with user authentication. By the end, you'll have an API you could actually ship.

47.1 What We're Building

```
GET   /products         -> list all products (with filtering, pagination)
GET    /products/{id}       -> get one product
POST   /products            -> create a product  (auth required)
PUT    /products/{id}       -> update a product  (auth required)
DELETE /products/{id}       -> delete a product  (auth required)

GET    /categories          -> list all categories
POST   /categories          -> create a category (auth required)

POST   /auth/register       -> register a new user
POST   /auth/token          -> log in, get a JWT token
```

```
GET    /health              -> health check (public)
```

47.2 Project Structure

```
catalogue/
├── src/
│   └── catalogue/
│       ├── __init__.py
│       ├── main.py           <- FastAPI app
│       ├── config.py         <- settings
│       ├── database.py       <- SQLAlchemy engine + session
│       ├── models.py         <- ORM models
│       ├── schemas.py        <- Pydantic schemas
│       ├── dependencies.py   <- shared dependencies
│       ├── routers/
│       │   ├── __init__.py
│       │   ├── auth.py
│       │   ├── products.py
│       │   └── categories.py
│       └── services/
│           ├── __init__.py
│           ├── auth.py
│           └── products.py
├── tests/
│   ├── conftest.py
│   ├── test_auth.py
│   ├── test_products.py
│   └── test_categories.py
└── pyproject.toml
```

47.3 Step 1 — Configuration (`config.py`)

```python
# src/catalogue/config.py
from functools import lru_cache
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    app_name:     str = "Product Catalogue API"
```

```
    app_version:  str = "1.0.0"
    debug:        bool = False

    database_url: str = "sqlite:///./catalogue.db"

    secret_key:   str = "change-this-secret-key-in-production"
    algorithm:    str = "HS256"
    token_expiry_hours: int = 24

    model_config = SettingsConfigDict(env_file=".env", env_file_encoding="utf-8


@lru_cache
def get_settings() -> Settings:
    return Settings()
```

47.4 Step 2 — Database (`database.py`)

```
# src/catalogue/database.py
from sqlalchemy import create_engine, event
from sqlalchemy.orm import DeclarativeBase, sessionmaker, Session
from catalogue.config import get_settings

settings = get_settings()

engine = create_engine(
    settings.database_url,
    connect_args={"check_same_thread": False},   # SQLite only
    echo=settings.debug,
)

# Enable WAL mode for SQLite --- better concurrent reads
if settings.database_url.startswith("sqlite"):
    @event.listens_for(engine, "connect")
    def set_wal_mode(dbapi_conn, _):
        dbapi_conn.execute("PRAGMA journal_mode=WAL")


SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)


class Base(DeclarativeBase):
```

```
    pass


def get_db():
    """FastAPI dependency that provides a database session."""
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

47.5 Step 3 — ORM Models (`models.py`)

```
# src/catalogue/models.py
from datetime import datetime
from sqlalchemy import (
    String, Text, Float, Integer, Boolean,
    ForeignKey, DateTime, func
)
from sqlalchemy.orm import Mapped, mapped_column, relationship
from catalogue.database import Base


class User(Base):
    __tablename__ = "users"

    id:           Mapped[int]      = mapped_column(primary_key=True)
    username:     Mapped[str]      = mapped_column(String(50), unique=True, index=
    email:        Mapped[str]      = mapped_column(String(200), unique=True, index
    hashed_password: Mapped[str]  = mapped_column(String(200))
    is_active:    Mapped[bool]     = mapped_column(Boolean, default=True)
    created_at:   Mapped[datetime] = mapped_column(DateTime, server_default=func.n

    products: Mapped[list["Product"]] = relationship(back_populates="owner")


class Category(Base):
    __tablename__ = "categories"

    id:          Mapped[int] = mapped_column(primary_key=True)
    name:        Mapped[str] = mapped_column(String(100), unique=True, index=True)
    description: Mapped[str | None] = mapped_column(Text, nullable=True)
```

```
    products: Mapped[list["Product"]] = relationship(back_populates="category")


class Product(Base):
    __tablename__ = "products"

    id:          Mapped[int]    = mapped_column(primary_key=True)
    name:        Mapped[str]    = mapped_column(String(200), index=True)
    description: Mapped[str | None] = mapped_column(Text, nullable=True)
    price:       Mapped[float]  = mapped_column(Float)
    stock:       Mapped[int]    = mapped_column(Integer, default=0)
    is_active:   Mapped[bool]   = mapped_column(Boolean, default=True)
    created_at:  Mapped[datetime] = mapped_column(DateTime, server_default=func
    updated_at:  Mapped[datetime] = mapped_column(
        DateTime, server_default=func.now(), onupdate=func.now()
    )

    category_id: Mapped[int | None] = mapped_column(
        ForeignKey("categories.id"), nullable=True
    )
    owner_id:    Mapped[int]  = mapped_column(ForeignKey("users.id"))

    category: Mapped["Category | None"] = relationship(back_populates="products
    owner:    Mapped["User"]             = relationship(back_populates="product
```

47.6 Step 4 — Pydantic Schemas (`schemas.py`)

```
# src/catalogue/schemas.py
from datetime import datetime
from typing import Optional
from pydantic import BaseModel, Field, field_validator, EmailStr


# ── User ──────────────────────────────────────────────────────────────────────

class UserCreate(BaseModel):
    username: str  = Field(..., min_length=3, max_length=50)
    email:    str  = Field(..., description="Valid email address")
    password: str  = Field(..., min_length=8)

    @field_validator("password")
```

```
    @classmethod
    def password_strength(cls, v: str) -> str:
        if not any(c.isupper() for c in v):
            raise ValueError("Password must contain at least one uppercase letter")
        if not any(c.isdigit() for c in v):
            raise ValueError("Password must contain at least one digit")
        return v

    @field_validator("email")
    @classmethod
    def email_lowercase(cls, v: str) -> str:
        if "@" not in v:
            raise ValueError("Invalid email address")
        return v.lower()


class UserOut(BaseModel):
    id:         int
    username:   str
    email:      str
    is_active:  bool
    created_at: datetime

    model_config = {"from_attributes": True}


class Token(BaseModel):
    access_token: str
    token_type:   str = "bearer"


# ── Category ──────────────────────────────────────────────────────────

class CategoryCreate(BaseModel):
    name:        str            = Field(..., min_length=1, max_length=100)
    description: Optional[str] = None


class CategoryOut(BaseModel):
    id:          int
    name:        str
    description: Optional[str]

    model_config = {"from_attributes": True}
```

```python
# ── Product ──────────────────────────────────────────────────────────

class ProductCreate(BaseModel):
    name:        str            = Field(..., min_length=1, max_length=200)
    description: Optional[str] = None
    price:       float          = Field(..., gt=0, description="Must be positiv
    stock:       int            = Field(0, ge=0)
    category_id: Optional[int] = None


class ProductUpdate(BaseModel):
    name:        Optional[str]   = Field(None, min_length=1, max_length=200)
    description: Optional[str]   = None
    price:       Optional[float] = Field(None, gt=0)
    stock:       Optional[int]   = Field(None, ge=0)
    is_active:   Optional[bool]  = None
    category_id: Optional[int]   = None


class ProductOut(BaseModel):
    id:          int
    name:        str
    description: Optional[str]
    price:       float
    stock:       int
    is_active:   bool
    created_at:  datetime
    updated_at:  datetime
    category:    Optional[CategoryOut]
    owner:       UserOut

    model_config = {"from_attributes": True}


class ProductList(BaseModel):
    items:  list[ProductOut]
    total:  int
    page:   int
    pages:  int
```

47.7 Step 5 — Services

47.7.1 Auth service (`services/auth.py`)

```
# src/catalogue/services/auth.py
from datetime import datetime, timedelta
from typing import Optional
from jose import JWTError, jwt
from passlib.context import CryptContext
from sqlalchemy.orm import Session
from catalogue.config import get_settings
from catalogue import models, schemas

settings   = get_settings()
pwd_ctx    = CryptContext(schemes=["bcrypt"], deprecated="auto")


def hash_password(plain: str) -> str:
    return pwd_ctx.hash(plain)


def verify_password(plain: str, hashed: str) -> bool:
    return pwd_ctx.verify(plain, hashed)


def create_token(data: dict) -> str:
    payload = data.copy()
    payload["exp"] = datetime.utcnow() + timedelta(hours=settings.token_expiry_hou
    return jwt.encode(payload, settings.secret_key, algorithm=settings.algorithm)


def decode_token(token: str) -> Optional[str]:
    try:
        payload = jwt.decode(
            token, settings.secret_key, algorithms=[settings.algorithm]
        )
        return payload.get("sub")
    except JWTError:
        return None


def create_user(db: Session, data: schemas.UserCreate) -> models.User:
    user = models.User(
        username=data.username,
```

```
        email=data.email,
        hashed_password=hash_password(data.password),
    )
    db.add(user)
    db.commit()
    db.refresh(user)
    return user


def authenticate_user(db: Session, username: str, password: str) -> Optional[mo
    user = db.query(models.User).filter(models.User.username == username).first
    if not user or not verify_password(password, user.hashed_password):
        return None
    return user


def get_user_by_username(db: Session, username: str) -> Optional[models.User]:
    return db.query(models.User).filter(models.User.username == username).first
```

47.7.2 Product service (`services/products.py`)

```
# src/catalogue/services/products.py
from typing import Optional
from sqlalchemy.orm import Session
from sqlalchemy import select, func
from catalogue import models, schemas


def get_products(
    db:          Session,
    page:        int = 1,
    per_page:    int = 20,
    category_id: Optional[int] = None,
    search:      Optional[str] = None,
    min_price:   Optional[float] = None,
    max_price:   Optional[float] = None,
    in_stock:    bool = False,
) -> schemas.ProductList:
    q = db.query(models.Product).filter(models.Product.is_active == True)

    if category_id:
        q = q.filter(models.Product.category_id == category_id)
    if search:
        q = q.filter(models.Product.name.ilike(f"%{search}%"))
```

```
    if min_price is not None:
        q = q.filter(models.Product.price >= min_price)
    if max_price is not None:
        q = q.filter(models.Product.price <= max_price)
    if in_stock:
        q = q.filter(models.Product.stock > 0)

    total  = q.count()
    pages  = (total + per_page - 1) // per_page
    items  = q.offset((page - 1) * per_page).limit(per_page).all()

    return schemas.ProductList(items=items, total=total, page=page, pages=pages)


def get_product(db: Session, product_id: int) -> Optional[models.Product]:
    return db.query(models.Product).filter(models.Product.id == product_id).first(


def create_product(
    db:    Session,
    data:  schemas.ProductCreate,
    owner: models.User,
) -> models.Product:
    product = models.Product(**data.model_dump(), owner_id=owner.id)
    db.add(product)
    db.commit()
    db.refresh(product)
    return product


def update_product(
    db:      Session,
    product: models.Product,
    data:    schemas.ProductUpdate,
) -> models.Product:
    update_data = data.model_dump(exclude_unset=True)
    for field, value in update_data.items():
        setattr(product, field, value)
    db.commit()
    db.refresh(product)
    return product


def delete_product(db: Session, product: models.Product) -> None:
```

```
    db.delete(product)
    db.commit()


def get_categories(db: Session) -> list[models.Category]:
    return db.query(models.Category).all()


def create_category(db: Session, data: schemas.CategoryCreate) -> models.Catego
    category = models.Category(**data.model_dump())
    db.add(category)
    db.commit()
    db.refresh(category)
    return category
```

47.8 Step 6 — Dependencies (`dependencies.py`)

```
# src/catalogue/dependencies.py
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session
from catalogue.database import get_db
from catalogue.services import auth as auth_service
from catalogue import models

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/token")


def get_current_user(
    token: str    = Depends(oauth2_scheme),
    db:    Session = Depends(get_db),
) -> models.User:
    username = auth_service.decode_token(token)
    if not username:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid or expired token",
            headers={"WWW-Authenticate": "Bearer"},
        )
    user = auth_service.get_user_by_username(db, username)
    if not user or not user.is_active:
        raise HTTPException(status_code=401, detail="User not found or inactive
```

```
    return user
```

47.9 Step 7 — Routers

47.9.1 Auth router (`routers/auth.py`)

```
# src/catalogue/routers/auth.py
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from catalogue.database import get_db
from catalogue.services import auth as auth_service
from catalogue import schemas

router = APIRouter(prefix="/auth", tags=["Authentication"])


@router.post("/register", response_model=schemas.UserOut, status_code=201)
def register(data: schemas.UserCreate, db: Session = Depends(get_db)):
    if auth_service.get_user_by_username(db, data.username):
        raise HTTPException(status_code=400, detail="Username already taken")
    return auth_service.create_user(db, data)


@router.post("/token", response_model=schemas.Token)
def login(form: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_d
    user = auth_service.authenticate_user(db, form.username, form.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
        )
    token = auth_service.create_token({"sub": user.username})
    return schemas.Token(access_token=token)
```

47.9.2 Products router (`routers/products.py`)

```
# src/catalogue/routers/products.py
from typing import Optional
from fastapi import APIRouter, Depends, HTTPException, Query
from sqlalchemy.orm import Session
from catalogue.database import get_db
```

```
from catalogue.dependencies import get_current_user
from catalogue.services import products as product_service
from catalogue import models, schemas

router = APIRouter(prefix="/products", tags=["Products"])


@router.get("", response_model=schemas.ProductList)
def list_products(
    page:        int            = Query(1, ge=1),
    per_page:    int            = Query(20, ge=1, le=100),
    category_id: Optional[int]  = None,
    search:      Optional[str]  = None,
    min_price:   Optional[float] = None,
    max_price:   Optional[float] = None,
    in_stock:    bool           = False,
    db:          Session        = Depends(get_db),
):
    return product_service.get_products(
        db, page, per_page, category_id, search, min_price, max_price, in_stock
    )


@router.get("/{product_id}", response_model=schemas.ProductOut)
def get_product(product_id: int, db: Session = Depends(get_db)):
    product = product_service.get_product(db, product_id)
    if not product:
        raise HTTPException(status_code=404, detail="Product not found")
    return product


@router.post("", response_model=schemas.ProductOut, status_code=201)
def create_product(
    data:         schemas.ProductCreate,
    db:           Session      = Depends(get_db),
    current_user: models.User  = Depends(get_current_user),
):
    return product_service.create_product(db, data, current_user)


@router.put("/{product_id}", response_model=schemas.ProductOut)
def update_product(
    product_id:   int,
    data:         schemas.ProductUpdate,
```

```
    db:           Session     = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    product = product_service.get_product(db, product_id)
    if not product:
        raise HTTPException(status_code=404, detail="Product not found")
    if product.owner_id != current_user.id:
        raise HTTPException(status_code=403, detail="Not your product")
    return product_service.update_product(db, product, data)


@router.delete("/{product_id}", status_code=204)
def delete_product(
    product_id:   int,
    db:           Session     = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    product = product_service.get_product(db, product_id)
    if not product:
        raise HTTPException(status_code=404, detail="Product not found")
    if product.owner_id != current_user.id:
        raise HTTPException(status_code=403, detail="Not your product")
    product_service.delete_product(db, product)
```

47.10 Step 8 — Main App (`main.py`)

```
# src/catalogue/main.py
from contextlib import asynccontextmanager
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
from catalogue.config import get_settings
from catalogue.database import Base, engine
from catalogue.routers import auth, products, categories

settings = get_settings()


@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup: create all tables
    Base.metadata.create_all(bind=engine)
```

```
    yield
    # Shutdown: nothing needed for SQLite


app = FastAPI(
    title=settings.app_name,
    version=settings.app_version,
    lifespan=lifespan,
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(auth.router)
app.include_router(products.router)
app.include_router(categories.router)


@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    return JSONResponse(
        status_code=500,
        content={"detail": "Internal server error"},
    )


@app.get("/health", tags=["Health"])
def health():
    return {"status": "ok", "version": settings.app_version}
```

47.11 Step 9 — Tests

47.11.1 tests/conftest.py

```
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
```

```
from catalogue.main import app
from catalogue.database import get_db, Base

TEST_DB_URL = "sqlite:///./test.db"

engine_test   = create_engine(TEST_DB_URL, connect_args={"check_same_thread": Fals
SessionTest   = sessionmaker(bind=engine_test, autocommit=False, autoflush=False)


@pytest.fixture(autouse=True)
def setup_db():
    Base.metadata.create_all(bind=engine_test)
    yield
    Base.metadata.drop_all(bind=engine_test)


@pytest.fixture
def db():
    session = SessionTest()
    try:
        yield session
    finally:
        session.close()


@pytest.fixture
def client(db):
    def override_get_db():
        try:
            yield db
        finally:
            pass
    app.dependency_overrides[get_db] = override_get_db
    with TestClient(app) as c:
        yield c
    app.dependency_overrides.clear()


@pytest.fixture
def auth_headers(client):
    client.post("/auth/register", json={
        "username": "testuser",
        "email":    "test@example.com",
        "password": "TestPass1",
```

```
    })
    response = client.post("/auth/token", data={
        "username": "testuser",
        "password": "TestPass1",
    })
    token = response.json()["access_token"]
    return {"Authorization": f"Bearer {token}"}
```

47.11.2 tests/test_auth.py

```
def test_register_success(client):
    r = client.post("/auth/register", json={
        "username": "alice",
        "email":    "alice@example.com",
        "password": "StrongPass1",
    })
    assert r.status_code == 201
    data = r.json()
    assert data["username"] == "alice"
    assert "hashed_password" not in data


def test_register_duplicate_username(client):
    payload = {"username": "bob", "email": "bob@x.com", "password": "Pass1234"}
    client.post("/auth/register", json=payload)
    r = client.post("/auth/register", json=payload)
    assert r.status_code == 400


def test_register_weak_password(client):
    r = client.post("/auth/register", json={
        "username": "carol",
        "email":    "carol@x.com",
        "password": "weak",
    })
    assert r.status_code == 422


def test_login_success(client):
    client.post("/auth/register", json={
        "username": "dave", "email": "d@x.com", "password": "Dave1234"
    })
    r = client.post("/auth/token", data={"username": "dave", "password": "Dave1
    assert r.status_code == 200
```

```
    assert "access_token" in r.json()


def test_login_wrong_password(client):
    client.post("/auth/register", json={
        "username": "eve", "email": "e@x.com", "password": "Eve12345"
    })
    r = client.post("/auth/token", data={"username": "eve", "password": "wrong"})
    assert r.status_code == 401
```

47.11.3 tests/test_products.py

```
def test_create_product(client, auth_headers):
    r = client.post("/products", json={
        "name": "Widget", "price": 9.99, "stock": 100
    }, headers=auth_headers)
    assert r.status_code == 201
    data = r.json()
    assert data["name"]  == "Widget"
    assert data["price"] == 9.99


def test_create_product_unauthenticated(client):
    r = client.post("/products", json={"name": "Widget", "price": 9.99})
    assert r.status_code == 401


def test_list_products(client, auth_headers):
    for i in range(5):
        client.post("/products", json={"name": f"Product {i}", "price": float(i+1)
                    headers=auth_headers)
    r = client.get("/products")
    assert r.status_code == 200
    data = r.json()
    assert data["total"] == 5
    assert len(data["items"]) == 5


def test_get_product_not_found(client):
    r = client.get("/products/9999")
    assert r.status_code == 404


def test_update_product(client, auth_headers):
```

```
    create_r = client.post("/products", json={"name": "Old", "price": 5.0},
                           headers=auth_headers)
    pid = create_r.json()["id"]

    r = client.put(f"/products/{pid}", json={"name": "New", "price": 10.0},
                   headers=auth_headers)
    assert r.status_code == 200
    assert r.json()["name"] == "New"


def test_delete_product(client, auth_headers):
    create_r = client.post("/products", json={"name": "ToDelete", "price": 1.0},
                           headers=auth_headers)
    pid = create_r.json()["id"]

    r = client.delete(f"/products/{pid}", headers=auth_headers)
    assert r.status_code == 204

    r = client.get(f"/products/{pid}")
    assert r.status_code == 404


def test_search_products(client, auth_headers):
    client.post("/products", json={"name": "Python Book", "price": 29.99},
                headers=auth_headers)
    client.post("/products", json={"name": "Java Book", "price": 24.99},
                headers=auth_headers)

    r = client.get("/products?search=python")
    assert r.status_code == 200
    assert r.json()["total"] == 1
    assert r.json()["items"][0]["name"] == "Python Book"


def test_filter_by_price(client, auth_headers):
    for price in [5.0, 15.0, 50.0, 100.0]:
        client.post("/products", json={"name": f"Item £{price}", "price": price},
                    headers=auth_headers)
    r = client.get("/products?min_price=10&max_price=60")
    assert r.status_code == 200
    assert r.json()["total"] == 2
```

Run all tests:

```
pip install pytest httpx
```

```
pytest tests/ -v --tb=short
```

47.12 Step 10 — Run It

```
pip install fastapi uvicorn sqlalchemy pydantic-settings python-jose passlib[bcryp

# Run the API
uvicorn catalogue.main:app --reload --app-dir src

# Visit the interactive docs
open http://127.0.0.1:8000/docs
```

The Swagger UI at `/docs` lets you: 1. Register a user (`POST /auth/register`) 2. Log in (`POST /auth/token`) — copy the token 3. Click "Authorize" -> paste the token 4. Create, list, update, and delete products — all from the browser

---

47.13 What You Built

A complete, production-ready REST API with: - **SQLAlchemy ORM** with relationships (User -> Product -> Category) - **Pydantic schemas** for validation, transformation, and documentation - **JWT authentication** — register, login, protected endpoints - **Ownership checks** — you can only edit your own products - **Filtering and pagination** — search by name, filter by price/category/stock - **Service layer** — business logic separated from route handlers - **Full test suite** — auth tests, product CRUD tests, filter tests - **Auto-generated docs** at `/docs` and `/redoc`

47.14 What's Next?

Chapter 48 covers **Building a Web Scraper and Data Pipeline** — an end-to-end project that scrapes data, cleans it with pandas, stores it in SQLite, visualises the results, and schedules the whole thing to run automatically.

Chapter 48

Chapter 48: Build a Web Scraper and Data Pipeline

Data doesn't arrive clean. It's scattered across websites, in inconsistent formats, with missing values and duplicates. A data pipeline takes raw data from the source, cleans it, transforms it, stores it, and delivers insights.

In this chapter, you build a complete, end-to-end pipeline:

```
Scrape -> Clean -> Store -> Analyse -> Visualise -> Schedule
```

We'll scrape real book data from `books.toscrape.com`, clean it with pandas, store it in SQLite, analyse it, visualise the results, and schedule the whole thing to run automatically.

48.1 The Plan

```
pipeline/
├── pipeline/
│   ├── __init__.py
│   ├── scraper.py      <- fetch + parse HTML
│   ├── cleaner.py      <- validate + transform data
│   ├── storage.py      <- SQLite with SQLAlchemy
│   ├── analyser.py     <- pandas analysis
│   ├── visualiser.py   <- matplotlib + seaborn charts
│   └── runner.py       <- orchestrate + schedule
├── output/             <- charts and reports
├── data/               <- raw and cleaned CSVs
└── books.db            <- SQLite database
```

48.2 Step 1 — The Scraper (`scraper.py`)

```
# pipeline/scraper.py
"""
Scrape books.toscrape.com --- a practice site built for scraping.
No robots.txt restrictions, no API keys needed.
"""
import time
import logging
import requests
from bs4 import BeautifulSoup
from dataclasses import dataclass, field
from datetime import datetime

log = logging.getLogger(__name__)

BASE_URL   = "https://books.toscrape.com"
RATING_MAP = {"One": 1, "Two": 2, "Three": 3, "Four": 4, "Five": 5}


@dataclass
class RawBook:
    """Raw data as scraped --- strings only, no cleaning yet."""
    title:       str
    price_str:   str        # e.g. "£12.99" --- raw, may have encoding issues
    rating_word: str        # e.g. "Three"
    in_stock:    bool
    url:         str
    scraped_at:  str = field(default_factory=lambda: datetime.utcnow().isoformat())


def fetch_html(url: str, retries: int = 3, delay: float = 1.0) -> str | None:
    """Fetch a URL with retries and exponential backoff."""
    for attempt in range(retries):
        try:
            response = requests.get(
                url,
                timeout=10,
                headers={"User-Agent": "DataPipelineBot/1.0 (educational)"},
            )
            response.raise_for_status()
```

```
            return response.text
        except requests.RequestException as e:
            log.warning(f"Attempt {attempt+1}/{retries} failed for {url}: {e}")
            if attempt < retries - 1:
                time.sleep(delay * (2 ** attempt))    # exponential backoff
    return None


def parse_catalogue_page(html: str, base_url: str) -> tuple[list[RawBook], str
    """
    Parse one catalogue page.
    Returns (list_of_books, next_page_url_or_None).
    """
    soup  = BeautifulSoup(html, "lxml")
    books = []

    for article in soup.select("article.product_pod"):
        title       = article.find("h3").find("a")["title"]
        price_str   = article.find("p", class_="price_color").text.strip()
        rating_word = article.find("p", class_="star-rating")["class"][1]
        in_stock    = "In stock" in article.find("p", class_="instock").text
        href        = article.find("h3").find("a")["href"]
        url         = f"{base_url}/catalogue/{href.replace('../', '')}"

        books.append(RawBook(
            title       = title,
            price_str   = price_str,
            rating_word = rating_word,
            in_stock    = in_stock,
            url         = url,
        ))

    # Find next page link
    next_btn = soup.select_one("li.next a")
    next_url = None
    if next_btn:
        current_path = base_url + "/catalogue"
        next_url     = f"{current_path}/{next_btn['href']}"

    return books, next_url


def scrape_catalogue(max_pages: int = 50, delay: float = 1.0) -> list[RawBook]:
    """
```

```
    Scrape the entire catalogue (50 pages = 1000 books).
    Politely waits between requests.
    """
    all_books = []
    url       = f"{BASE_URL}/catalogue/page-1.html"
    page      = 1

    while url and page <= max_pages:
        log.info(f"Scraping page {page}: {url}")
        html = fetch_html(url)

        if html is None:
            log.error(f"Failed to fetch page {page}. Stopping.")
            break

        books, next_url = parse_catalogue_page(html, BASE_URL)
        all_books.extend(books)
        log.info(f"  Page {page}: {len(books)} books (total: {len(all_books)})")

        url   = next_url
        page += 1

        if url:
            time.sleep(delay)

    log.info(f"Scraping complete. Total books: {len(all_books)}")
    return all_books
```

48.3 Step 2 — The Cleaner (`cleaner.py`)

```
# pipeline/cleaner.py
"""
Transform raw scraped data into a clean, validated DataFrame.
"""
import re
import logging
import pandas as pd
from dataclasses import asdict
from scraper import RawBook

log = logging.getLogger(__name__)
```

```
RATING_MAP = {"One": 1, "Two": 2, "Three": 3, "Four": 4, "Five": 5}


def parse_price(price_str: str) -> float | None:
    """Extract float from a price string like '£12.99' or 'Â£12.99'."""
    match = re.search(r"(\d+\.\d+)", price_str)
    if match:
        return float(match.group(1))
    return None


def clean_books(raw_books: list[RawBook]) -> pd.DataFrame:
    """
    Convert raw books to a clean DataFrame.

    Transformations:
    - Parse price string -> float
    - Map rating word -> int (1--5)
    - Normalise title (strip whitespace)
    - Drop rows that fail validation
    - Add derived columns
    """
    records = [asdict(b) for b in raw_books]
    df      = pd.DataFrame(records)

    initial_count = len(df)
    log.info(f"Cleaning {initial_count} raw records...")

    # Parse price
    df["price"] = df["price_str"].apply(parse_price)
    price_failures = df["price"].isna().sum()
    if price_failures:
        log.warning(f"  {price_failures} rows have unparseable prices --- dropp
    df = df.dropna(subset=["price"])

    # Map rating
    df["rating"] = df["rating_word"].map(RATING_MAP)
    rating_failures = df["rating"].isna().sum()
    if rating_failures:
        log.warning(f"  {rating_failures} rows have unknown rating --- dropping
    df = df.dropna(subset=["rating"])
    df["rating"] = df["rating"].astype(int)

    # Clean title
```

```
    df["title"] = df["title"].str.strip()

    # Drop duplicates by URL
    before_dedup = len(df)
    df = df.drop_duplicates(subset=["url"])
    deduped = before_dedup - len(df)
    if deduped:
        log.info(f"  Removed {deduped} duplicate URLs")

    # Derived columns
    df["price_band"] = pd.cut(
        df["price"],
        bins=[0, 10, 20, 35, 60],
        labels=["Budget", "Mid-range", "Premium", "Luxury"],
    )
    df["value_score"] = (df["rating"] / df["price"] * 10).round(2)

    # Parse scrape timestamp
    df["scraped_at"] = pd.to_datetime(df["scraped_at"])

    # Drop raw intermediate columns
    df = df.drop(columns=["price_str", "rating_word"])

    log.info(f"Cleaning done. {len(df)}/{initial_count} records kept.")
    return df


def validate_dataframe(df: pd.DataFrame) -> bool:
    """Assert data quality expectations."""
    checks = {
        "No null prices":   df["price"].isna().sum() == 0,
        "No null ratings":  df["rating"].isna().sum() == 0,
        "Price > 0":        (df["price"] > 0).all(),
        "Rating in 1-5":    df["rating"].between(1, 5).all(),
        "No empty titles":  (df["title"].str.len() > 0).all(),
    }
    all_passed = True
    for check, result in checks.items():
        status = "[x]" if result else "[-]"
        log.info(f"  [{status}] {check}")
        if not result:
            all_passed = False

    return all_passed
```

48.4 Step 3 — Storage (`storage.py`)

```
# pipeline/storage.py
"""
Persist cleaned data to SQLite via SQLAlchemy.
Also provides helpers to query the database.
"""
import logging
import pandas as pd
from sqlalchemy import create_engine, text
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from sqlalchemy import String, Float, Integer, Boolean, DateTime
from datetime import datetime

log = logging.getLogger(__name__)

DB_PATH = "books.db"
engine  = create_engine(f"sqlite:///{DB_PATH}", echo=False)


class Base(DeclarativeBase):
    pass


class Book(Base):
    __tablename__ = "books"

    id:          Mapped[int]      = mapped_column(primary_key=True)
    title:       Mapped[str]      = mapped_column(String(500))
    price:       Mapped[float]    = mapped_column(Float)
    rating:      Mapped[int]      = mapped_column(Integer)
    in_stock:    Mapped[bool]     = mapped_column(Boolean)
    url:         Mapped[str]      = mapped_column(String(500), unique=True)
    price_band:  Mapped[str]      = mapped_column(String(50))
    value_score: Mapped[float]    = mapped_column(Float)
    scraped_at:  Mapped[datetime] = mapped_column(DateTime)


def init_db() -> None:
    Base.metadata.create_all(bind=engine)
    log.info("Database initialised.")
```

```
def save_books(df: pd.DataFrame) -> int:
    """
    Upsert books into the database.
    Returns the number of new rows inserted.
    """
    with engine.connect() as conn:
        existing_urls = set(
            row[0] for row in conn.execute(text("SELECT url FROM books")).fetchall
        )

    new_books = df[~df["url"].isin(existing_urls)].copy()
    if new_books.empty:
        log.info("No new books to save.")
        return 0

    new_books.to_sql(
        "books",
        engine,
        if_exists="append",
        index=False,
        method="multi",
    )
    log.info(f"Saved {len(new_books)} new books to database.")
    return len(new_books)


def load_books() -> pd.DataFrame:
    """Load all books from the database into a DataFrame."""
    return pd.read_sql("SELECT * FROM books ORDER BY price", engine)


def get_stats() -> dict:
    """Return high-level stats from the database."""
    df = load_books()
    if df.empty:
        return {}
    return {
        "total_books":    len(df),
        "avg_price":      round(df["price"].mean(), 2),
        "avg_rating":     round(df["rating"].mean(), 2),
        "in_stock_count": int(df["in_stock"].sum()),
        "price_min":      df["price"].min(),
```

```
        "price_max":      df["price"].max(),
    }
```

48.5 Step 4 — Analysis (`analyser.py`)

```
# pipeline/analyser.py
"""
Analyse the book database and produce summary tables.
"""
import logging
import pandas as pd
from storage import load_books

log = logging.getLogger(__name__)


def analyse(df: pd.DataFrame) -> dict[str, pd.DataFrame]:
    """Return a dict of named summary DataFrames."""
    results = {}

    # Rating distribution
    results["rating_dist"] = (
        df.groupby("rating")
          .agg(count=("title", "count"), avg_price=("price", "mean"))
          .round(2)
    )

    # Price band breakdown
    results["price_bands"] = (
        df.groupby("price_band", observed=True)
          .agg(
              count      = ("title",       "count"),
              avg_rating = ("rating",      "mean"),
              avg_price  = ("price",       "mean"),
              in_stock   = ("in_stock",    "sum"),
          )
          .round(2)
    )

    # Top 10 best value books (high rating, low price)
    results["best_value"] = (
        df.sort_values("value_score", ascending=False)
```

```
            [["title", "price", "rating", "value_score"]]
            .head(10)
            .reset_index(drop=True)
        )

        # Top 10 most expensive
        results["most_expensive"] = (
            df.sort_values("price", ascending=False)
            [["title", "price", "rating"]]
            .head(10)
            .reset_index(drop=True)
        )

        # Stock availability by price band
        results["stock_by_band"] = (
            df.groupby("price_band", observed=True)["in_stock"]
            .agg(total="count", in_stock="sum")
            .assign(pct_in_stock=lambda x: (x["in_stock"] / x["total"] * 100).round(
        )

        return results


    def print_report(results: dict[str, pd.DataFrame]) -> None:
        """Print a formatted text report to the console."""
        print("\n" + "=" * 60)
        print("  BOOK CATALOGUE ANALYSIS REPORT")
        print("=" * 60)

        print("\n[chart] Books by Rating:")
        print(results["rating_dist"].to_string())

        print("\n Price Band Breakdown:")
        print(results["price_bands"].to_string())

        print("\n[trophy] Top 10 Best Value Books:")
        for _, row in results["best_value"].iterrows():
            print(f"  {row['rating']}[*]  £{row['price']:.2f}  {row['title'][:50]}")

        print("\n[pkg] Stock Availability by Price Band:")
        print(results["stock_by_band"].to_string())
```

48.6 Step 5 — Visualisation (`visualiser.py`)

```
# pipeline/visualiser.py
"""
Generate and save charts from the analysed data.
"""
import logging
from pathlib import Path
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

log = logging.getLogger(__name__)
OUTPUT_DIR = Path("output")
OUTPUT_DIR.mkdir(exist_ok=True)

sns.set_theme(style="whitegrid", palette="muted")


def _save(fig: plt.Figure, name: str) -> None:
    path = OUTPUT_DIR / name
    fig.savefig(path, dpi=150, bbox_inches="tight")
    plt.close(fig)
    log.info(f"Saved chart: {path}")


def plot_rating_distribution(df: pd.DataFrame) -> None:
    fig, ax = plt.subplots(figsize=(7, 4))
    counts = df["rating"].value_counts().sort_index()
    bars   = ax.bar(
        [f"{'[*]' * r}" for r in counts.index],
        counts.values,
        color=["#f44336", "#ff9800", "#ffc107", "#8bc34a", "#4caf50"],
        edgecolor="white",
    )
    for bar, count in zip(bars, counts.values):
        ax.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 5,
                str(count), ha="center", va="bottom", fontsize=10)
    ax.set_title("Books by Star Rating", fontsize=13)
    ax.set_xlabel("Rating")
    ax.set_ylabel("Number of Books")
    _save(fig, "rating_distribution.png")
```

```
def plot_price_distribution(df: pd.DataFrame) -> None:
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    # Histogram
    axes[0].hist(df["price"], bins=30, color="steelblue", edgecolor="white", alpha
    axes[0].axvline(df["price"].mean(), color="red", linestyle="--",
                    label=f"Mean £{df['price'].mean():.2f}")
    axes[0].axvline(df["price"].median(), color="green", linestyle="--",
                    label=f"Median £{df['price'].median():.2f}")
    axes[0].set_title("Price Distribution")
    axes[0].set_xlabel("Price (£)")
    axes[0].set_ylabel("Count")
    axes[0].legend()

    # Box plot by rating
    sns.boxplot(data=df, x="rating", y="price", palette="muted", ax=axes[1])
    axes[1].set_title("Price Distribution by Rating")
    axes[1].set_xlabel("Rating (stars)")
    axes[1].set_ylabel("Price (£)")

    _save(fig, "price_distribution.png")


def plot_price_bands(df: pd.DataFrame) -> None:
    band_counts = df["price_band"].value_counts().sort_index()

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    # Bar chart
    axes[0].bar(band_counts.index, band_counts.values,
                color=["#4caf50", "#2196f3", "#ff9800", "#f44336"],
                edgecolor="white")
    axes[0].set_title("Books by Price Band")
    axes[0].set_ylabel("Count")

    # Pie chart
    axes[1].pie(band_counts.values, labels=band_counts.index,
                autopct="%1.0f%%", startangle=90,
                colors=["#4caf50", "#2196f3", "#ff9800", "#f44336"])
    axes[1].set_title("Price Band Share")

    _save(fig, "price_bands.png")
```

```python
def plot_value_vs_price(df: pd.DataFrame) -> None:
    fig, ax = plt.subplots(figsize=(9, 5))

    scatter = ax.scatter(
        df["price"],
        df["rating"],
        c=df["value_score"],
        cmap="RdYlGn",
        alpha=0.6,
        s=30,
        edgecolors="none",
    )
    plt.colorbar(scatter, ax=ax, label="Value Score")

    ax.set_title("Price vs Rating (colour = value score)")
    ax.set_xlabel("Price (£)")
    ax.set_ylabel("Rating (stars)")
    ax.set_yticks([1, 2, 3, 4, 5])

    _save(fig, "value_vs_price.png")


def generate_all_charts(df: pd.DataFrame) -> None:
    log.info("Generating charts...")
    plot_rating_distribution(df)
    plot_price_distribution(df)
    plot_price_bands(df)
    plot_value_vs_price(df)
    log.info(f"All charts saved to {OUTPUT_DIR}/")
```

48.7 Step 6 — The Runner (`runner.py`)

```python
# pipeline/runner.py
"""
Orchestrates the full pipeline:
  scrape -> clean -> store -> analyse -> visualise

Optionally schedules it to run daily.
"""
import logging
import time
from pathlib import Path
```

```python
from datetime import datetime

import schedule

from scraper    import scrape_catalogue
from cleaner    import clean_books, validate_dataframe
from storage    import init_db, save_books, load_books, get_stats
from analyser   import analyse, print_report
from visualiser import generate_all_charts

# ── Logging setup ──────────────────────────────────────────────────────

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s  %(levelname)-8s  %(name)s --- %(message)s",
    handlers=[
        logging.StreamHandler(),
        logging.FileHandler("pipeline.log"),
    ],
)
log = logging.getLogger("runner")

# ── Pipeline ───────────────────────────────────────────────────────────

def run_pipeline(max_pages: int = 5) -> None:
    """Full pipeline: scrape -> clean -> store -> analyse -> visualise."""
    start = time.time()
    log.info("=" * 50)
    log.info(f"Pipeline started at {datetime.now():%Y-%m-%d %H:%M:%S}")

    # 1. Scrape
    log.info("STEP 1/5 --- Scraping...")
    raw_books = scrape_catalogue(max_pages=max_pages, delay=1.0)
    if not raw_books:
        log.error("No books scraped. Aborting.")
        return

    # 2. Clean
    log.info("STEP 2/5 --- Cleaning...")
    df = clean_books(raw_books)

    log.info("STEP 2b --- Validating...")
    if not validate_dataframe(df):
        log.error("Data validation failed. Aborting.")
```

```
            return

        # Save raw CSV for audit trail
        raw_path = Path("data") / f"raw_{datetime.now():%Y%m%d_%H%M}.csv"
        raw_path.parent.mkdir(exist_ok=True)
        df.to_csv(raw_path, index=False)
        log.info(f"Raw data saved to {raw_path}")

        # 3. Store
        log.info("STEP 3/5 --- Storing...")
        init_db()
        new_count = save_books(df)

        # 4. Analyse
        log.info("STEP 4/5 --- Analysing...")
        all_books = load_books()
        results   = analyse(all_books)
        print_report(results)

        stats = get_stats()
        log.info(f"Database stats: {stats}")

        # 5. Visualise
        log.info("STEP 5/5 --- Visualising...")
        generate_all_charts(all_books)

        elapsed = time.time() - start
        log.info(f"Pipeline completed in {elapsed:.1f}s. {new_count} new books adde
        log.info("=" * 50)


# ── Scheduler ──────────────────────────────────────────────────────────

def schedule_pipeline(run_at: str = "06:00") -> None:
    """Run the pipeline every day at the specified time."""
    log.info(f"Scheduler started. Pipeline will run daily at {run_at}.")
    schedule.every().day.at(run_at).do(run_pipeline)

    # Run immediately on first launch
    run_pipeline()

    while True:
        schedule.run_pending()
        time.sleep(60)
```

```
# ── Entry point ──────────────────────────────────────────────────────

if __name__ == "__main__":
    import sys
    if "--schedule" in sys.argv:
        schedule_pipeline()
    else:
        # Single run
        pages = int(sys.argv[1]) if len(sys.argv) > 1 else 5
        run_pipeline(max_pages=pages)
```

48.8 Running the Pipeline

```
# Install dependencies
pip install requests beautifulsoup4 lxml pandas sqlalchemy matplotlib seaborn sche

# Run once (scrape 5 pages = 100 books)
python pipeline/runner.py 5

# Run all 50 pages (1000 books)
python pipeline/runner.py 50

# Run continuously, daily at 6am
python pipeline/runner.py --schedule
```

After running, you'll have: - `books.db` — SQLite database with all books - `data/raw_YYYYMMDD_HHMM.csv` — raw CSV snapshot - `output/rating_distribution.png` - `output/price_distribution.png` - `output/price_bands.png` - `output/value_vs_price.png` - `pipeline.log` — full run log

---

48.9 Sample Output

```
==================================================
  BOOK CATALOGUE ANALYSIS REPORT
==================================================

[chart] Books by Rating:
        count  avg_price
```

```
rating
1           20      35.12
2           20      34.78
3           20      36.02
4           20      33.94
5           20      34.45

 Price Band Breakdown:
            count  avg_rating  avg_price  in_stock
price_band
Budget         18        2.89      10.62        18
Mid-range      29        3.10      15.43        29
Premium        39        3.21      27.81        39
Luxury         14        2.93      54.20        14

[trophy] Top 10 Best Value Books:
  1[*]  £10.00  A Light in the Attic
  5[*]  £10.00  Sharp Objects
  5[*]  £13.99  It's Only the Himalayas
  ...

[pkg] Stock Availability by Price Band:
            total  in_stock  pct_in_stock
price_band
Budget         18        18         100.0
Mid-range      29        29         100.0
Premium        39        39         100.0
Luxury         14        14         100.0
```

48.10 What You Built

A production-grade data pipeline with: - **Scraper** — polite HTTP requests, retry logic with exponential backoff, HTML parsing, pagination - **Cleaner** — regex price parsing, rating mapping, deduplication, derived columns (`value_score`, `price_band`), data quality validation with named checks - **Storage** — SQLAlchemy ORM, upsert logic (skip existing URLs), pandas -> SQL in one call - **Analyser** — rating distribution, price band breakdown, best value ranking, stock analysis - **Visualiser** — 4 charts (rating bar, price histogram + box plot, price band pie, value scatter) - **Runner** — end-to-end orchestration, structured logging to file and console, audit CSV snapshots, `schedule` for daily automation

48.11 What's Next?

Chapter 49 covers **Building a Desktop App with Tkinter** — you'll build a graphical user interface with windows, buttons, forms, and menus, turning your Python scripts into apps that anyone can use without a terminal.

Chapter 49

Chapter 49: Build a Desktop App with Tkinter

Not every program needs a browser or a terminal. Sometimes you want a real window — with buttons, text boxes, menus, and a clean interface that anyone can use without knowing Python.

Tkinter is Python's built-in GUI library. It ships with Python. No installation needed. It's not the prettiest framework, but it runs everywhere, and it's more than capable of building useful desktop apps.

In this chapter you'll build a **Personal Budget Tracker** — a desktop app with a form to add transactions, a table to view them, charts to visualise spending, and a database to persist everything.

49.1 How Tkinter Works

Tkinter is event-driven. You: 1. Create a **window** (the root) 2. Add **widgets** (buttons, labels, entry boxes, etc.) 3. Register **event handlers** (functions called when buttons are clicked, keys pressed, etc.) 4. Start the **event loop** — `root.mainloop()` — which listens for events forever

```
import tkinter as tk

root = tk.Tk()
root.title("My App")
root.geometry("400x300")    # width x height in pixels

label = tk.Label(root, text="Hello, World!", font=("Arial", 16))
```

```
label.pack(pady=20)

button = tk.Button(root, text="Click me",
                   command=lambda: label.config(text="Clicked!"))
button.pack()

root.mainloop()
```

Run this and you get a window with a label and a button. Click the button — the label changes. That's Tkinter.

49.2 Core Widgets

```
import tkinter as tk
from tkinter import ttk   # themed widgets --- always prefer ttk over tk

root = tk.Tk()
root.title("Widget Gallery")

# Label --- display text or images
tk.Label(root, text="Name:").grid(row=0, column=0, padx=5, pady=5, sticky="e")

# Entry --- single-line text input
entry = ttk.Entry(root, width=20)
entry.grid(row=0, column=1, padx=5, pady=5)

# Button
btn = ttk.Button(root, text="Submit",
                 command=lambda: print(entry.get()))
btn.grid(row=1, column=1, pady=5)

# Text --- multi-line text area
text = tk.Text(root, width=30, height=5)
text.grid(row=2, column=0, columnspan=2, padx=5, pady=5)

# Combobox (dropdown)
combo = ttk.Combobox(root, values=["Option 1", "Option 2", "Option 3"])
combo.set("Option 1")
combo.grid(row=3, column=0, columnspan=2, pady=5)

# Checkbutton
var = tk.BooleanVar()
```

```
check = ttk.Checkbutton(root, text="I agree", variable=var)
check.grid(row=4, column=0, columnspan=2)

# Radiobuttons
choice = tk.StringVar(value="A")
for i, option in enumerate(["A", "B", "C"]):
    ttk.Radiobutton(root, text=option, variable=choice, value=option).grid(
        row=5, column=i
    )

# Listbox
listbox = tk.Listbox(root, height=4)
for item in ["Apple", "Banana", "Cherry", "Date"]:
    listbox.insert(tk.END, item)
listbox.grid(row=6, column=0, columnspan=2, pady=5)

root.mainloop()
```

49.3 Layout Managers

Tkinter has three ways to position widgets:

49.3.1 `pack` — stack widgets in a direction

```
frame = tk.Frame(root)
frame.pack(fill="both", expand=True, padx=10, pady=10)

tk.Label(frame, text="Top").pack(side="top",    fill="x")
tk.Label(frame, text="Left").pack(side="left",  fill="y")
tk.Label(frame, text="Right").pack(side="right", fill="y")
tk.Label(frame, text="Bottom").pack(side="bottom", fill="x")
```

49.3.2 `grid` — position in a row/column table

```
# column=0 is left, row=0 is top
# sticky="e" aligns to the east (right) of the cell
tk.Label(root, text="Email:").grid(row=0, column=0, sticky="e", padx=5, pady=3)
ttk.Entry(root).grid(row=0, column=1, sticky="ew", padx=5, pady=3)

tk.Label(root, text="Password:").grid(row=1, column=0, sticky="e", padx=5, pady
ttk.Entry(root, show="*").grid(row=1, column=1, sticky="ew", padx=5, pady=3)
```

```
ttk.Button(root, text="Login").grid(row=2, column=1, sticky="e", padx=5, pady=5)

root.columnconfigure(1, weight=1)    # column 1 stretches when window resizes
```

49.3.3 `place` — exact pixel coordinates

```
# Use sparingly --- fragile on different screen sizes
tk.Label(root, text="Absolute").place(x=50, y=100)
```

Use `grid` for forms and complex layouts. Use `pack` for simple vertical/horizontal stacks.

49.4 Variables — Linking Widgets to Data

Tkinter variables (`StringVar`, `IntVar`, `BooleanVar`, `DoubleVar`) connect widgets to Python values:

```
name_var    = tk.StringVar()
age_var     = tk.IntVar(value=25)
agree_var   = tk.BooleanVar()
amount_var  = tk.DoubleVar()

# Entry bound to a StringVar --- changes automatically sync
entry = ttk.Entry(root, textvariable=name_var)
entry.pack()

# Read the value
print(name_var.get())    # whatever is in the entry box

# Set the value (updates the widget automatically)
name_var.set("Alice")

# React to changes
name_var.trace_add("write", lambda *_: print(f"Name changed to: {name_var.get()}")
```

49.5 Dialogs and Messageboxes

```
from tkinter import messagebox, filedialog, simpledialog

# Information dialog
messagebox.showinfo("Success", "Record saved successfully!")
```

```
# Warning
messagebox.showwarning("Warning", "This cannot be undone.")

# Error
messagebox.showerror("Error", "Failed to connect to database.")

# Yes/No question
if messagebox.askyesno("Confirm", "Delete this record?"):
    delete_record()

# Open file dialog
file_path = filedialog.askopenfilename(
    title="Open File",
    filetypes=[("CSV files", "*.csv"), ("All files", "*.*")]
)

# Save file dialog
save_path = filedialog.asksaveasfilename(
    defaultextension=".csv",
    filetypes=[("CSV files", "*.csv")]
)

# Ask for text input
value = simpledialog.askstring("Input", "Enter your name:")
```

49.6 Menus

```
def create_menu(root: tk.Tk) -> None:
    menubar = tk.Menu(root)
    root.config(menu=menubar)

    # File menu
    file_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="File", menu=file_menu)
    file_menu.add_command(label="New",     command=lambda: print("New"),  accele
    file_menu.add_command(label="Open...",  command=lambda: print("Open"), acce
    file_menu.add_command(label="Save",    command=lambda: print("Save"), accele
    file_menu.add_separator()
    file_menu.add_command(label="Exit",    command=root.quit)

    # Edit menu
```

```
    edit_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Edit", menu=edit_menu)
    edit_menu.add_command(label="Undo", accelerator="Ctrl+Z")
    edit_menu.add_command(label="Redo", accelerator="Ctrl+Y")

    # Keyboard shortcuts
    root.bind("<Control-n>", lambda e: print("New"))
    root.bind("<Control-o>", lambda e: print("Open"))
    root.bind("<Control-s>", lambda e: print("Save"))
```

49.7 Treeview — Displaying Tabular Data

`ttk.Treeview` is the right widget for displaying table data:

```
from tkinter import ttk

columns = ("id", "date", "description", "amount", "category")
tree = ttk.Treeview(root, columns=columns, show="headings", height=15)

# Configure column headers and widths
tree.heading("id",          text="ID")
tree.heading("date",        text="Date")
tree.heading("description", text="Description")
tree.heading("amount",      text="Amount")
tree.heading("category",    text="Category")

tree.column("id",          width=40,  anchor="center")
tree.column("date",        width=100, anchor="center")
tree.column("description", width=200)
tree.column("amount",      width=90,  anchor="e")
tree.column("category",    width=120)

# Scrollbar
scrollbar = ttk.Scrollbar(root, orient="vertical", command=tree.yview)
tree.configure(yscrollcommand=scrollbar.set)

tree.grid(row=0, column=0, sticky="nsew")
scrollbar.grid(row=0, column=1, sticky="ns")

# Insert a row
tree.insert("", tk.END, values=(1, "2026-03-01", "Coffee", "£3.50", "Food"))
```

```
# Delete all rows
for item in tree.get_children():
    tree.delete(item)

# Get selected row
selected = tree.selection()
if selected:
    values = tree.item(selected[0])["values"]
```

49.8 Embedding matplotlib Charts

```
import tkinter as tk
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg


def create_chart_frame(parent) -> FigureCanvasTkAgg:
    fig, ax = plt.subplots(figsize=(6, 3))
    ax.bar(["Food", "Transport", "Housing", "Fun"], [400, 120, 900, 150],
           color=["#4caf50", "#2196f3", "#ff9800", "#e91e63"])
    ax.set_title("Spending by Category")
    ax.set_ylabel("Amount (£)")
    plt.tight_layout()

    canvas = FigureCanvasTkAgg(fig, master=parent)
    canvas.draw()
    canvas.get_tk_widget().pack(fill="both", expand=True)
    return canvas
```

49.9 Project — Personal Budget Tracker

```
"""
budget_tracker.py

A full desktop budget tracking app built with Tkinter.
Features:
  - Add income and expense transactions
  - View all transactions in a sortable table
  - See spending by category in a pie chart
```

```
  - Persistent storage in SQLite
  - Export to CSV
"""
import csv
import sqlite3
import tkinter as tk
from tkinter import ttk, messagebox, filedialog
from datetime import date
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg


# ── Database ──────────────────────────────────────────────────────────

DB_PATH = "budget.db"

def init_db() -> None:
    with sqlite3.connect(DB_PATH) as conn:
        conn.execute("""
            CREATE TABLE IF NOT EXISTS transactions (
                id          INTEGER PRIMARY KEY AUTOINCREMENT,
                date        TEXT    NOT NULL,
                description TEXT    NOT NULL,
                amount      REAL    NOT NULL,
                type        TEXT    NOT NULL CHECK(type IN ('Income','Expense')),
                category    TEXT    NOT NULL
            )
        """)


def db_add(date_: str, description: str, amount: float,
           type_: str, category: str) -> int:
    with sqlite3.connect(DB_PATH) as conn:
        cur = conn.execute(
            "INSERT INTO transactions (date, description, amount, type, category)"
            " VALUES (?, ?, ?, ?, ?)",
            (date_, description, amount, type_, category)
        )
        return cur.lastrowid


def db_delete(transaction_id: int) -> None:
    with sqlite3.connect(DB_PATH) as conn:
        conn.execute("DELETE FROM transactions WHERE id = ?", (transaction_id,))
```

```
def db_fetch_all() -> list[tuple]:
    with sqlite3.connect(DB_PATH) as conn:
        return conn.execute(
            "SELECT id, date, description, amount, type, category"
            " FROM transactions ORDER BY date DESC, id DESC"
        ).fetchall()


def db_summary() -> dict:
    with sqlite3.connect(DB_PATH) as conn:
        rows = conn.execute(
            "SELECT type, SUM(amount) FROM transactions GROUP BY type"
        ).fetchall()
    totals = {"Income": 0.0, "Expense": 0.0}
    for type_, total in rows:
        totals[type_] = total or 0.0
    return totals


def db_spending_by_category() -> list[tuple]:
    with sqlite3.connect(DB_PATH) as conn:
        return conn.execute(
            "SELECT category, SUM(amount) FROM transactions"
            " WHERE type='Expense' GROUP BY category ORDER BY SUM(amount) DESC"
        ).fetchall()


# ── Main Application ──────────────────────────────────────────────────────

CATEGORIES = ["Food", "Transport", "Housing", "Utilities",
              "Entertainment", "Health", "Shopping", "Salary", "Other"]

COLORS = ["#f44336", "#e91e63", "#9c27b0", "#673ab7",
          "#3f51b5", "#2196f3", "#00bcd4", "#4caf50", "#ff9800"]


class BudgetApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Personal Budget Tracker")
        self.geometry("1000x700")
```

```
        self.resizable(True, True)
        self.configure(bg="#f5f5f5")

        init_db()
        self._build_ui()
        self.refresh()

    # ── UI Construction ──────────────────────────────────────────────

    def _build_ui(self) -> None:
        self._build_menu()

        # Top: summary bar
        self._build_summary_bar()

        # Main pane: left = form + table, right = chart
        paned = ttk.PanedWindow(self, orient="horizontal")
        paned.pack(fill="both", expand=True, padx=10, pady=5)

        left = ttk.Frame(paned)
        right = ttk.Frame(paned)
        paned.add(left,  weight=3)
        paned.add(right, weight=2)

        self._build_form(left)
        self._build_table(left)
        self._build_chart(right)

    def _build_menu(self) -> None:
        menubar = tk.Menu(self)
        self.config(menu=menubar)

        file_menu = tk.Menu(menubar, tearoff=0)
        menubar.add_cascade(label="File", menu=file_menu)
        file_menu.add_command(label="Export CSV...", command=self.export_csv)
        file_menu.add_separator()
        file_menu.add_command(label="Exit", command=self.quit)

        help_menu = tk.Menu(menubar, tearoff=0)
        menubar.add_cascade(label="Help", menu=help_menu)
        help_menu.add_command(label="About",
            command=lambda: messagebox.showinfo(
                "About", "Budget Tracker v1.0\nBuilt with Python + Tkinter"
            ))
```

```
    def _build_summary_bar(self) -> None:
        frame = ttk.Frame(self, padding=8)
        frame.pack(fill="x", padx=10, pady=(10, 0))

        self.lbl_income   = ttk.Label(frame, text="Income: £0.00",
                                      font=("Helvetica", 12, "bold"),
                                      foreground="#4caf50")
        self.lbl_expenses = ttk.Label(frame, text="Expenses: £0.00",
                                      font=("Helvetica", 12, "bold"),
                                      foreground="#f44336")
        self.lbl_balance  = ttk.Label(frame, text="Balance: £0.00",
                                      font=("Helvetica", 13, "bold"))

        self.lbl_income.pack(side="left",   padx=20)
        self.lbl_expenses.pack(side="left", padx=20)
        self.lbl_balance.pack(side="right", padx=20)

    def _build_form(self, parent) -> None:
        form = ttk.LabelFrame(parent, text="Add Transaction", padding=10)
        form.pack(fill="x", padx=5, pady=5)

        # Variables
        self.v_date        = tk.StringVar(value=date.today().isoformat())
        self.v_description = tk.StringVar()
        self.v_amount      = tk.StringVar()
        self.v_type        = tk.StringVar(value="Expense")
        self.v_category    = tk.StringVar(value="Food")

        # Form grid
        fields = [
            ("Date:",        ttk.Entry(form, textvariable=self.v_date,        w
            ("Description:", ttk.Entry(form, textvariable=self.v_description, w
            ("Amount (£):",  ttk.Entry(form, textvariable=self.v_amount,      w
            ("Type:",        ttk.Combobox(form, textvariable=self.v_type,
                                          values=["Income", "Expense"],
            ("Category:",    ttk.Combobox(form, textvariable=self.v_category,
                                          values=CATEGORIES,
        ]

        for row, (label_text, widget) in enumerate(fields):
            ttk.Label(form, text=label_text).grid(row=row, column=0,
                                                  sticky="e", padx=5, pady=3)
            widget.grid(row=row, column=1, sticky="ew", padx=5, pady=3)
```

```
        form.columnconfigure(1, weight=1)

        btn_frame = ttk.Frame(form)
        btn_frame.grid(row=len(fields), column=0, columnspan=2, pady=8)

        ttk.Button(btn_frame, text="Add Transaction",
                   command=self.add_transaction).pack(side="left", padx=5)
        ttk.Button(btn_frame, text="Clear",
                   command=self.clear_form).pack(side="left", padx=5)

    def _build_table(self, parent) -> None:
        table_frame = ttk.LabelFrame(parent, text="Transactions", padding=5)
        table_frame.pack(fill="both", expand=True, padx=5, pady=5)

        cols = ("id", "date", "description", "amount", "type", "category")
        self.tree = ttk.Treeview(table_frame, columns=cols, show="headings", heigh

        widths = {"id": 40, "date": 100, "description": 200,
                  "amount": 90, "type": 80, "category": 110}
        for col in cols:
            self.tree.heading(col, text=col.capitalize(),
                              command=lambda c=col: self._sort_by(c))
            self.tree.column(col, width=widths[col],
                             anchor="e" if col == "amount" else "center" if col in

        # Alternate row colours
        self.tree.tag_configure("income",  background="#e8f5e9")
        self.tree.tag_configure("expense", background="#ffebee")

        vsb = ttk.Scrollbar(table_frame, orient="vertical",   command=self.tree.yv
        hsb = ttk.Scrollbar(table_frame, orient="horizontal", command=self.tree.xv
        self.tree.configure(yscrollcommand=vsb.set, xscrollcommand=hsb.set)

        self.tree.grid(row=0, column=0, sticky="nsew")
        vsb.grid(row=0, column=1, sticky="ns")
        hsb.grid(row=1, column=0, sticky="ew")
        table_frame.rowconfigure(0, weight=1)
        table_frame.columnconfigure(0, weight=1)

        # Right-click menu to delete
        self.context_menu = tk.Menu(self, tearoff=0)
        self.context_menu.add_command(label="Delete Transaction",
                                      command=self.delete_selected)
```

```
        self.tree.bind("<Button-3>", self._show_context_menu)

    def _build_chart(self, parent) -> None:
        chart_frame = ttk.LabelFrame(parent, text="Spending by Category", paddi
        chart_frame.pack(fill="both", expand=True, padx=5, pady=5)

        self.fig, self.ax = plt.subplots(figsize=(4, 4))
        self.fig.patch.set_facecolor("#f5f5f5")
        self.canvas = FigureCanvasTkAgg(self.fig, master=chart_frame)
        self.canvas.get_tk_widget().pack(fill="both", expand=True)

    # ── Actions ──────────────────────────────────────────────────────────

    def add_transaction(self) -> None:
        date_   = self.v_date.get().strip()
        desc    = self.v_description.get().strip()
        amount_s = self.v_amount.get().strip()
        type_   = self.v_type.get()
        cat     = self.v_category.get()

        # Validation
        errors = []
        if not date_:
            errors.append("Date is required.")
        if not desc:
            errors.append("Description is required.")
        try:
            amount = float(amount_s)
            if amount <= 0:
                errors.append("Amount must be positive.")
        except ValueError:
            errors.append("Amount must be a number.")
            amount = 0.0

        if errors:
            messagebox.showerror("Validation Error", "\n".join(errors))
            return

        db_add(date_, desc, amount, type_, cat)
        self.clear_form()
        self.refresh()

    def delete_selected(self) -> None:
        selected = self.tree.selection()
```

```
        if not selected:
            return
        if not messagebox.askyesno("Confirm", "Delete this transaction?"):
            return
        for item in selected:
            transaction_id = self.tree.item(item)["values"][0]
            db_delete(transaction_id)
        self.refresh()

    def clear_form(self) -> None:
        self.v_date.set(date.today().isoformat())
        self.v_description.set("")
        self.v_amount.set("")
        self.v_type.set("Expense")
        self.v_category.set("Food")

    def export_csv(self) -> None:
        path = filedialog.asksaveasfilename(
            defaultextension=".csv",
            filetypes=[("CSV", "*.csv")],
            initialfile=f"budget_{date.today()}.csv",
        )
        if not path:
            return
        rows = db_fetch_all()
        with open(path, "w", newline="", encoding="utf-8") as f:
            writer = csv.writer(f)
            writer.writerow(["ID", "Date", "Description", "Amount", "Type", "Categ
            writer.writerows(rows)
        messagebox.showinfo("Export", f"Exported {len(rows)} rows to:\n{path}")

    # ── Refresh ──────────────────────────────────────────────────────────

    def refresh(self) -> None:
        self._refresh_summary()
        self._refresh_table()
        self._refresh_chart()

    def _refresh_summary(self) -> None:
        totals  = db_summary()
        income  = totals["Income"]
        expense = totals["Expense"]
        balance = income - expense
```

```
        self.lbl_income.config(  text=f"Income: £{income:,.2f}")
        self.lbl_expenses.config(text=f"Expenses: £{expense:,.2f}")
        self.lbl_balance.config( text=f"Balance: £{balance:,.2f}",
                                 foreground="#4caf50" if balance >= 0 else "#f

    def _refresh_table(self) -> None:
        for item in self.tree.get_children():
            self.tree.delete(item)
        for row in db_fetch_all():
            id_, date_, desc, amount, type_, cat = row
            tag  = "income" if type_ == "Income" else "expense"
            sign = "+" if type_ == "Income" else "-"
            self.tree.insert("", tk.END,
                             values=(id_, date_, desc, f"{sign}£{amount:.2f}",
                             tags=(tag,))

    def _refresh_chart(self) -> None:
        self.ax.clear()
        data = db_spending_by_category()
        if data:
            labels  = [row[0] for row in data]
            amounts = [row[1] for row in data]
            colors  = COLORS[:len(labels)]
            self.ax.pie(amounts, labels=labels, colors=colors,
                        autopct="%1.0f%%", startangle=90,
                        textprops={"fontsize": 8})
            self.ax.set_title("Spending by Category", fontsize=10)
        else:
            self.ax.text(0.5, 0.5, "No expenses yet",
                         ha="center", va="center", transform=self.ax.transAxes)
        self.canvas.draw()

    def _sort_by(self, column: str) -> None:
        """Sort the treeview by a column."""
        rows = [(self.tree.set(item, column), item)
                for item in self.tree.get_children("")]
        rows.sort(reverse=getattr(self, f"_sort_{column}_desc", False))
        for index, (_, item) in enumerate(rows):
            self.tree.move(item, "", index)
        setattr(self, f"_sort_{column}_desc",
                not getattr(self, f"_sort_{column}_desc", False))

    def _show_context_menu(self, event) -> None:
        item = self.tree.identify_row(event.y)
```

```
        if item:
            self.tree.selection_set(item)
            self.context_menu.tk_popup(event.x_root, event.y_root)


# ── Entry point ──────────────────────────────────────────────────────

if __name__ == "__main__":
    app = BudgetApp()
    app.mainloop()
```

Run it:

```
python budget_tracker.py
```

No installation needed — `tkinter` and `sqlite3` ship with Python. You only need `matplotlib`:

```
pip install matplotlib
```

49.10 What You Built

A fully functional desktop app with: - **Menu bar** — File -> Export CSV, Exit; Help -> About - **Summary bar** — live income, expense, and balance totals in colour - **Transaction form** — date, description, amount, type (Income/Expense), category; validation; clear button - **Treeview table** — colour-coded rows (green for income, red for expenses), sortable columns, scrollbars, right-click context menu to delete - **Live pie chart** — spending by category, embedded matplotlib, updates on every change - **SQLite persistence** — all data survives closing the app - **CSV export** — save all transactions to a file

---

49.11 What's Next?

Chapter 50 covers **Packaging and Publishing** — turning your Python scripts and apps into installable packages you can share on PyPI so anyone in the world can `pip install` your work.

Chapter 50

Chapter 50: Package Your Code and Share It

You've built something useful. Now how do you share it?

Right now your code lives on your computer. If someone else wants it, they'd have to find your files, figure out the dependencies, and set everything up manually. That's a terrible experience.

Python's packaging ecosystem solves this. You create a **package**, publish it to **PyPI** (the Python Package Index), and anyone in the world can install it with:

```
pip install your-package-name
```

This chapter walks you through every step — from structuring your project to uploading it live.

50.1 What is a Package?

A Python package is a directory containing Python modules, structured so `pip` can install and manage it.

When you run `pip install requests`, pip downloads a package from PyPI, extracts it, and puts the files where Python can find them. You can do the exact same thing with your own code.

50.2 Project Structure — The `src` Layout

There are two common layouts. The `src` layout is the modern standard because it prevents accidental imports of your development code:

```
my-package/
├── src/
│   └── mypackage/
│       ├── __init__.py
│       ├── core.py
│       └── utils.py
├── tests/
│   ├── __init__.py
│   ├── test_core.py
│   └── test_utils.py
├── docs/
├── pyproject.toml    <- the single config file (replaces setup.py)
├── README.md
├── LICENSE
└── .gitignore
```

The key rule: your actual package code lives inside `src/mypackage/`, not in the root.

50.3 `pyproject.toml` — The Modern Config File

`pyproject.toml` replaced `setup.py` and `setup.cfg`. It's the single file that configures your package, build tools, and quality tools.

Let's build a small package called `texttools` — utilities for cleaning and analysing text:

```toml
[build-system]
requires      = ["hatchling"]
build-backend = "hatchling.build"

[project]
name          = "texttools"
version       = "0.1.0"
description   = "Simple text cleaning and analysis utilities"
readme        = "README.md"
license       = { text = "MIT" }
requires-python = ">=3.11"
```

```toml
authors = [
    { name = "Alice Smith", email = "alice@example.com" }
]

keywords = ["text", "nlp", "utilities", "cleaning"]

classifiers = [
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.11",
    "Programming Language :: Python :: 3.12",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
    "Topic :: Text Processing",
]

# Runtime dependencies (installed automatically with your package)
dependencies = [
    "requests>=2.28",
]

# Optional dependency groups
[project.optional-dependencies]
dev = [
    "pytest>=7",
    "pytest-cov",
    "ruff",
    "mypy",
]
nlp = [
    "nltk>=3.8",
    "spacy>=3.6",
]

# CLI entry points --- creates a command-line script
[project.scripts]
texttools = "texttools.cli:main"

# Links shown on PyPI
[project.urls]
Homepage      = "https://github.com/alice/texttools"
Repository    = "https://github.com/alice/texttools"
Documentation = "https://texttools.readthedocs.io"
"Bug Tracker" = "https://github.com/alice/texttools/issues"
```

```
# ── Tool configuration ──────────────────────────────────────────────

[tool.hatch.build.targets.wheel]
packages = ["src/texttools"]

[tool.ruff]
line-length = 88
target-version = "py311"

[tool.ruff.lint]
select = ["E", "W", "F", "I", "B", "UP"]

[tool.mypy]
python_version = "3.11"
strict = true

[tool.pytest.ini_options]
testpaths     = ["tests"]
addopts       = "-v --tb=short"
```

50.4 Write the Package Code

src/texttools/__init__.py:

```
"""texttools --- simple text cleaning and analysis utilities."""

from .core  import clean, word_count, sentence_count, readability_score
from .utils import slugify, truncate, extract_emails, extract_urls

__version__ = "0.1.0"
__all__ = [
    "clean", "word_count", "sentence_count", "readability_score",
    "slugify", "truncate", "extract_emails", "extract_urls",
]
```

src/texttools/core.py:

```
"""Core text analysis functions."""
import re


def clean(text: str, lowercase: bool = False, remove_punctuation: bool = False) ->
    """
    Clean a text string.
```

```
    Args:
        text: The input string.
        lowercase: Convert to lowercase if True.
        remove_punctuation: Strip punctuation if True.

    Returns:
        Cleaned string.

    Examples:
        >>> clean("  Hello, World!  ")
        'Hello, World!'
        >>> clean("Hello, World!", lowercase=True)
        'hello, world!'
    """
    text = text.strip()
    if lowercase:
        text = text.lower()
    if remove_punctuation:
        text = re.sub(r"[^\w\s]", "", text)
    # Normalise whitespace
    text = re.sub(r"\s+", " ", text)
    return text


def word_count(text: str) -> int:
    """Return the number of words in text."""
    return len(text.split()) if text.strip() else 0


def sentence_count(text: str) -> int:
    """Return the approximate number of sentences."""
    sentences = re.split(r"[.!?]+", text.strip())
    return len([s for s in sentences if s.strip()])


def readability_score(text: str) -> float:
    """
    Compute a simple readability score (0--100, higher = easier to read).
    Based on average sentence length and average word length.
    """
    words     = text.split()
    sentences = sentence_count(text)
    if not words or not sentences:
```

```
        return 0.0

    avg_sentence_len = len(words) / sentences
    avg_word_len     = sum(len(w) for w in words) / len(words)

    # Simplified Flesch-like formula
    score = 206.835 - (1.015 * avg_sentence_len) - (84.6 * avg_word_len / 4.5)
    return round(max(0.0, min(100.0, score)), 1)
```

src/texttools/utils.py:

```
"""Text utility functions."""
import re


def slugify(text: str) -> str:
    """
    Convert text to a URL-friendly slug.

    Examples:
        >>> slugify("Hello, World!")
        'hello-world'
        >>> slugify("Python 3.12 is GREAT!")
        'python-312-is-great'
    """
    text = text.lower().strip()
    text = re.sub(r"[^\w\s-]", "", text)
    text = re.sub(r"[\s_]+",  "-", text)
    text = re.sub(r"-+",      "-", text)
    return text.strip("-")


def truncate(text: str, max_length: int = 100, suffix: str = "...") -> str:
    """Truncate text to max_length characters, appending suffix if truncated."""
    if len(text) <= max_length:
        return text
    return text[: max_length - len(suffix)].rstrip() + suffix


def extract_emails(text: str) -> list[str]:
    """Extract all email addresses from text."""
    pattern = r"\b[A-Za-z0-9._%+\-]+@[A-Za-z0-9.\-]+\.[A-Z|a-z]{2,}\b"
    return re.findall(pattern, text)
```

```
def extract_urls(text: str) -> list[str]:
    """Extract all URLs from text."""
    pattern = r"https?://[^\s]+"
    return re.findall(pattern, text)
```

src/texttools/cli.py:

```
"""Command-line interface for texttools."""
import sys
from texttools import clean, word_count, sentence_count, readability_score


def main() -> None:
    """Entry point for the `texttools` CLI command."""
    if len(sys.argv) < 2:
        print("Usage: texttools <text>")
        print("       texttools --file <path>")
        sys.exit(1)

    if sys.argv[1] == "--file":
        if len(sys.argv) < 3:
            print("Error: --file requires a path argument")
            sys.exit(1)
        with open(sys.argv[2], encoding="utf-8") as f:
            text = f.read()
    else:
        text = " ".join(sys.argv[1:])

    cleaned = clean(text)
    print(f"Words:       {word_count(cleaned)}")
    print(f"Sentences:   {sentence_count(cleaned)}")
    print(f"Readability: {readability_score(cleaned)}/100")
```

50.5 Write Tests

tests/test_core.py:

```
import pytest
from texttools.core import clean, word_count, sentence_count, readability_score


def test_clean_strips_whitespace():
    assert clean("  hello  ") == "hello"
```

```
def test_clean_normalises_whitespace():
    assert clean("hello   world") == "hello world"


def test_clean_lowercase():
    assert clean("Hello World", lowercase=True) == "hello world"


def test_clean_remove_punctuation():
    assert clean("Hello, World!", remove_punctuation=True) == "Hello World"


def test_word_count():
    assert word_count("hello world") == 2
    assert word_count("")            == 0
    assert word_count("  ")          == 0


def test_sentence_count():
    assert sentence_count("Hello. World.") == 2
    assert sentence_count("One sentence")  == 1


def test_readability_score_range():
    score = readability_score("The cat sat on the mat. It was a fat cat.")
    assert 0.0 <= score <= 100.0


# Parametrize tests to cover many cases at once
@pytest.mark.parametrize("text,expected", [
    ("hello",          1),
    ("hello world",    2),
    ("one two three",  3),
])
def test_word_count_parametrized(text, expected):
    assert word_count(text) == expected
```

tests/test_utils.py:

```
from texttools.utils import slugify, truncate, extract_emails, extract_urls


def test_slugify_basic():
    assert slugify("Hello World")  == "hello-world"
```

```python
def test_slugify_special_chars():
    assert slugify("Hello, World!") == "hello-world"


def test_truncate_short():
    assert truncate("hi", max_length=10) == "hi"


def test_truncate_long():
    result = truncate("Hello, World!", max_length=8)
    assert len(result) <= 8
    assert result.endswith("...")


def test_extract_emails():
    text   = "Contact alice@example.com or bob@test.org for help."
    emails = extract_emails(text)
    assert "alice@example.com" in emails
    assert "bob@test.org"      in emails


def test_extract_urls():
    text = "Visit https://python.org and https://pypi.org for more."
    urls = extract_urls(text)
    assert "https://python.org" in urls
```

50.6 Install in Editable Mode

Before publishing, install your own package locally so you can test it:

```
pip install -e ".[dev]"
```

`-e` = editable mode. Changes to `src/texttools/` take effect immediately without reinstalling.

`[dev]` installs the optional dev dependencies (`pytest`, `ruff`, `mypy`).

Now test it:

```
# Run the test suite
pytest

# Use the CLI you defined
```

```
texttools "Python is a great language for beginners and experts alike."

# Import it like any other package
python -c "import texttools; print(texttools.slugify('Hello World!'))"
```

50.7 Semantic Versioning

Version numbers follow the format `MAJOR.MINOR.PATCH`:

Change	Bump
Backwards-incompatible API change	MAJOR: `1.0.0 -> 2.0.0`
New feature, backwards-compatible	MINOR: `1.0.0 -> 1.1.0`
Bug fix, no new features	PATCH: `1.0.0 -> 1.0.1`

Start at `0.1.0` while the API is still settling. Release `1.0.0` when you're confident in the public API.

50.8 Write a Good README

Your `README.md` is the first thing people see on PyPI and GitHub. Include:

````
# texttools

Simple text cleaning and analysis utilities for Python.

## Install

```bash
pip install texttools
```

## Quick start

```python
import texttools

text = "  Hello, World!  This is a GREAT day. "
print(texttools.clean(text, lowercase=True))
# -> "hello, world! this is a great day."
````
````

```
print(texttools.word_count(text))    # -> 9
print(texttools.slugify("Hello World!"))  # -> "hello-world"
```

CLI

```bash
texttools "Hello, World! This is a test."
# Words:       6
# Sentences:   2
# Readability: 74.3/100
```

API Reference

Function	Description
`clean(text, lowercase, remove_punctuation)`	Clean and normalise text
`word_count(text)`	Count words
`sentence_count(text)`	Count sentences
`slugify(text)`	Convert to URL slug
`truncate(text, max_length)`	Truncate with ellipsis
`extract_emails(text)`	Find all email addresses
`extract_urls(text)`	Find all URLs

License

MIT

50.9 Build the Distribution Files

Install the build tool:

```
pip install build
```

Build:

```
python -m build
```

This creates:

```
dist/
├── texttools-0.1.0-py3-none-any.whl    <- wheel (fast install)
└── texttools-0.1.0.tar.gz              <- source distribution
```

The `.whl` file is a zip archive — unzip it and you'll see exactly what gets installed.

50.10 Publish to TestPyPI First

Always test on TestPyPI before publishing to the real PyPI:

```
pip install twine

# Upload to TestPyPI
twine upload --repository testpypi dist/*
```

You'll need an account at test.pypi.org. Create one, then generate an API token under Account Settings.

Install from TestPyPI to verify it works:

```
pip install --index-url https://test.pypi.org/simple/ texttools
```

50.11 Publish to PyPI

When you're happy:

```
twine upload dist/*
```

You'll need an account at pypi.org and an API token.

After uploading, anyone in the world can install your package:

```
pip install texttools
```

50.12 GitHub Actions — Automate Publishing

Create `.github/workflows/publish.yml`:

```
name: Publish to PyPI

on:
  push:
    tags:
      - "v*"    # trigger when you push a version tag like v0.1.0

jobs:
  test:
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install -e ".[dev]"
      - run: pytest --tb=short

  publish:
    needs: test
    runs-on: ubuntu-latest
    permissions:
      id-token: write   # required for trusted publishing
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install build
      - run: python -m build
      - uses: pypa/gh-action-pypi-publish@release/v1
        # Uses PyPI's Trusted Publisher --- no API token needed
```

Now publishing is:

```
git tag v0.1.0
git push origin v0.1.0
```

GitHub Actions runs your tests, builds the package, and uploads it to PyPI automatically.

50.13 CI — Test on Every Push

Create `.github/workflows/ci.yml`:

```
name: CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
```

```
      matrix:
        python-version: ["3.11", "3.12"]

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}
      - run: pip install -e ".[dev]"
      - run: ruff check src/ tests/
      - run: mypy src/
      - run: pytest --cov=texttools --cov-report=term-missing
```

Every pull request is automatically linted, type-checked, and tested on Python 3.11 and 3.12.

50.14 The Full Publishing Workflow

```
# 1. Make your changes
# 2. Run tests
pytest

# 3. Lint and type check
ruff check src/ tests/
mypy src/

# 4. Bump version in pyproject.toml
#    (or use hatch-vcs for automatic versioning from git tags)

# 5. Commit
git add .
git commit -m "feat: add extract_urls function"

# 6. Tag the release
git tag v0.2.0

# 7. Push (triggers CI + publish workflow)
git push origin main --tags
```

50.15 What You Learned in This Chapter

- Python packages live in a `src/mypackage/` directory. `pyproject.toml` is the single config file.
- `[project]` in `pyproject.toml` defines name, version, description, author, dependencies, and classifiers.
- `[project.scripts]` creates CLI commands — after installation, users can run `texttools` from their terminal.
- `[project.optional-dependencies]` defines groups like `dev`, `nlp` that users install with `pip install mypackage[dev]`.
- `pip install -e ".[dev]"` installs in editable mode — changes take effect immediately.
- Semantic versioning: `MAJOR.MINOR.PATCH`. Start at `0.1.0`.
- `python -m build` creates a `.whl` and `.tar.gz` in `dist/`.
- `twine upload --repository testpypi dist/*` publishes to TestPyPI for testing.
- `twine upload dist/*` publishes to the real PyPI.
- GitHub Actions automates testing on every push and publishing on version tags.

50.16 What's Next?

Chapter 51 covers **Git and Version Control** — the workflow every professional Python developer uses: branches, commits, pull requests, and working with GitHub from the command line.

Chapter 51

Chapter 51: Git and Version Control for Python Developers

Every professional developer uses Git. Not occasionally — every day, on every project.

Git is a tool that tracks changes to your code over time. It lets you go back to any previous version, work on multiple features simultaneously without breaking each other, collaborate with a team, and understand exactly what changed, when, and why.

This chapter teaches you Git the way Python developers actually use it.

51.1 Why Git Matters

Imagine you spend a week adding a new feature. It breaks everything. Without Git, you're stuck. With Git, you type one command and your code is back to where it was — instantly.

Or imagine working with three other developers on the same codebase. Without Git, you'd email files back and forth and constantly overwrite each other's work. With Git, everyone works on their own branch, and Git merges the changes together.

Git is non-negotiable. Let's learn it.

51.2 Setup

```
# Check if Git is installed
git --version

# Install on Windows: https://git-scm.com
# Install on Mac:    brew install git
# Install on Linux:  sudo apt install git

# Configure your identity (required before first commit)
git config --global user.name  "Your Name"
git config --global user.email "you@example.com"

# Set VS Code as your default editor
git config --global core.editor "code --wait"

# Make 'main' the default branch name (modern standard)
git config --global init.defaultBranch main
```

51.3 The Core Concepts

```
Working Directory   ->  Staging Area  ->  Repository
  (your files)          (git add)       (git commit)
```

- **Working directory** — the files on your disk
- **Staging area** — a holding area where you prepare what goes into the next commit
- **Repository** — the permanent history of all commits

Think of it like packing a box (staging area) before sealing and labelling it (committing).

51.4 Starting a Repository

```
# Create a new project
mkdir my-project
cd my-project
git init              # creates a hidden .git/ folder

# OR clone an existing repository from GitHub
```

```
git clone https://github.com/username/repo-name.git
cd repo-name
```

51.5 The Basic Workflow

```
# 1. See what's changed
git status

# 2. Stage changes (prepare for commit)
git add filename.py          # add one file
git add src/                 # add a directory
git add .                    # add everything (use carefully)
git add -p                   # interactive: stage chunks, not whole files

# 3. Commit the staged changes
git commit -m "feat: add user authentication"

# 4. See the history
git log                       # full log
git log --oneline             # compact, one line per commit
git log --oneline --graph     # visual branch graph
```

51.6 `.gitignore` — What NOT to Track

Some files should never go into Git: virtual environments, compiled files, secrets, logs.

Create a `.gitignore` in your project root:

```
# Python
__pycache__/
*.py[cod]
*.pyo
*.pyd
.Python
*.egg
*.egg-info/
dist/
build/
.eggs/
```

```
# Virtual environments
.venv/
venv/
env/
ENV/

# Testing
.pytest_cache/
.coverage
htmlcov/
.tox/

# Type checking
.mypy_cache/
.dmypy.json

# IDEs
.vscode/settings.json
.idea/
*.swp
*.swo

# Secrets and environment files
.env
.env.local
*.key
secrets.json

# macOS
.DS_Store

# Database files
*.db
*.sqlite3

# Logs
*.log
logs/

# Build artifacts
*.whl
*.tar.gz
```

GitHub provides a standard Python `.gitignore` at `github.com/github/gitignore`.

51.7 Viewing Changes

```
# What changed in the working directory (not yet staged)?
git diff

# What changed in the staging area (staged but not committed)?
git diff --staged

# What changed in a specific file?
git diff filename.py

# See a specific commit
git show abc1234

# See what a file looked like in a previous commit
git show HEAD~3:src/module.py

# Compare two commits
git diff abc1234 def5678
```

51.8 Undoing Mistakes

```
# Unstage a file (keep the changes in working dir)
git restore --staged filename.py

# Discard changes in working directory (PERMANENT --- cannot undo)
git restore filename.py

# Undo the last commit but keep the changes staged
git reset --soft HEAD~1

# Undo the last commit and unstage the changes
git reset HEAD~1

# Create a new commit that reverses a past commit (safe for shared history)
git revert abc1234

# Stash changes temporarily (save without committing)
git stash
```

```
git stash pop                  # restore the stashed changes
git stash list                 # see all stashes
git stash drop stash@{0}       # delete a stash
```

The golden rule: **never rewrite history that has been pushed to a shared repository**. Use `git revert` instead of `git reset` when working with others.

51.9 Branches — Work in Isolation

Branches let you work on a feature without touching the main codebase.

```
# See all branches
git branch             # local branches
git branch -a          # local and remote

# Create a new branch
git branch feature/user-auth

# Switch to it
git checkout feature/user-auth

# Create AND switch in one command (modern way)
git switch -c feature/user-auth

# Go back to main
git switch main

# Merge a branch into main
git switch main
git merge feature/user-auth

# Delete a branch after merging
git branch -d feature/user-auth

# Force delete (unmerged branch)
git branch -D feature/user-auth
```

51.10 Merge Conflicts — How to Resolve Them

When two branches change the same line, Git can't merge automatically. You get a conflict:

```
<<<<<<< HEAD
def greet(name):
    return f"Hello, {name}!"
=======
def greet(name: str) -> str:
    return f"Hi there, {name}!"
>>>>>>> feature/greetings
```

The section between <<< and === is your current branch. Between === and >>> is the incoming branch.

Resolve it by editing the file to the correct final version:

```
def greet(name: str) -> str:
    return f"Hello, {name}!"
```

Then:

```
git add filename.py
git commit -m "resolve: merge conflict in greet function"
```

51.11 Working with GitHub

51.11.1 Push your code

```
# Connect your local repo to a GitHub remote
git remote add origin https://github.com/username/repo-name.git

# Push and set up tracking
git push -u origin main

# After that, just:
git push
```

51.11.2 Pull changes from GitHub

```
# Fetch + merge in one command
git pull

# Fetch only (don't merge yet)
git fetch origin
```

```
git diff origin/main   # see what's different
git merge origin/main  # merge when ready
```

51.11.3 The fetch/pull difference

`git fetch` downloads changes but doesn't apply them. `git pull` downloads and applies. When in doubt, fetch first, check the diff, then merge.

51.12 The Professional Git Workflow

Real teams use a consistent branching strategy. Here's the most common one:

```
main          <- stable, production code only
  └── develop <- integration branch (optional for larger teams)
        ├── feature/add-search     <- new features
        ├── fix/login-bug          <- bug fixes
        └── hotfix/security-patch  <- urgent production fixes
```

The day-to-day workflow:

```
# 1. Always start from an up-to-date main
git switch main
git pull

# 2. Create a feature branch
git switch -c feature/add-search

# 3. Work, commit often
git add -p
git commit -m "feat: add basic search endpoint"

git add -p
git commit -m "test: add tests for search endpoint"

# 4. Push your branch
git push -u origin feature/add-search

# 5. Open a Pull Request on GitHub
# 6. Code review
# 7. Merge to main (on GitHub)
# 8. Delete your branch
git switch main
git pull
git branch -d feature/add-search
```

51.13 Conventional Commits

Use a consistent format for commit messages. This makes the history readable and enables automated changelog generation:

```
<type>(<scope>): <short description>

[optional body]

[optional footer]
```

Types:

Type	When to use
`feat`	A new feature
`fix`	A bug fix
`docs`	Documentation only
`test`	Adding or changing tests
`refactor`	Code change that is neither a feature nor a fix
`perf`	Performance improvement
`chore`	Build process, dependency updates
`ci`	CI configuration

Examples:

```
feat(auth): add JWT token refresh endpoint

fix(search): handle empty query string correctly

docs: update README with installation instructions

test(products): add tests for price filter

refactor(cleaner): extract price parsing to separate function

perf(db): add index on products.created_at column

chore: upgrade dependencies to latest versions
```

Good commit messages: - Describe **what** and **why**, not how (the code shows how) - Use imperative mood: “add” not “added”, “fix” not “fixes” - Keep the first line under 72 characters - Separate subject from body with a blank line

51.14 Useful Git Commands for Python Developers

```
# Find which commit introduced a bug (binary search)
git bisect start
git bisect bad                  # current commit is broken
git bisect good v1.0.0          # this version was good
# Git checks out middle commits --- you test and mark good/bad
git bisect good                 # or: git bisect bad
git bisect reset                # when done

# See who changed a line (blame)
git blame src/module.py
git blame -L 10,20 src/module.py   # only lines 10-20

# Search commit messages
git log --grep="fix"
git log --grep="authentication" --oneline

# Search code changes across all history
git log -S "def authenticate" --oneline  # commits that added/removed this stri

# See all files changed in a commit
git show --stat abc1234

# Cherry-pick a single commit from another branch
git cherry-pick abc1234

# Rebase (replay your commits on top of another branch)
git rebase main
```

51.15 `git add -p` — Stage Like a Pro

This is the most underused and most powerful Git command for developers. It shows you each change and lets you decide whether to stage it:

```
git add -p
```

For each change, you'll see a diff and a prompt. Options: - `y` — yes, stage this chunk - `n` — no, skip it - `s` — split into smaller chunks - `e` — edit the chunk

manually - `q` — quit

This lets you make many changes at once but commit them as separate, logical units — keeping your history clean.

51.16 Git Tags — Mark Releases

```
# Create a lightweight tag
git tag v1.0.0

# Create an annotated tag (recommended for releases)
git tag -a v1.0.0 -m "Release version 1.0.0"

# List all tags
git tag

# Push tags to GitHub
git push origin --tags

# Push a specific tag
git push origin v1.0.0

# Delete a tag locally
git tag -d v1.0.0

# Delete a remote tag
git push origin --delete v1.0.0
```

51.17 A Clean Python Project on GitHub

Here's what a well-configured Python project on GitHub looks like:

```
my-project/
├── .github/
│   ├── workflows/
│   │   ├── ci.yml           <- test on every push
│   │   └── publish.yml      <- publish on version tags
│   └── PULL_REQUEST_TEMPLATE.md
├── src/
│   └── mypackage/
├── tests/
```

```
├── docs/
├── .gitignore               <- Python gitignore
├── pyproject.toml           <- package config + tool config
├── README.md
└── LICENSE
```

`PULL_REQUEST_TEMPLATE.md` — fills the PR description automatically:

```
## What does this PR do?

## Type of change
- [ ] Bug fix
- [ ] New feature
- [ ] Refactor
- [ ] Documentation

## Testing
- [ ] I added tests for this change
- [ ] All existing tests pass

## Checklist
- [ ] My code follows the project style guide
- [ ] I've updated the documentation if needed
```

51.18 What You Learned in This Chapter

- Git tracks changes to your code. `git init` starts a repo. `git clone` copies one.
- The three areas: working directory -> staging area (`git add`) -> repository (`git commit`).
- `.gitignore` keeps generated files, secrets, and virtual environments out of the repo.
- `git diff` shows uncommitted changes. `git log --oneline` shows history compactly.
- `git restore --staged` unstages. `git restore` discards working-dir changes. `git reset --soft HEAD~1` undoes the last commit.
- Branches isolate work. `git switch -c feature/name` creates and switches. `git merge` combines. Conflicts are resolved by editing the marked sections.
- `git push -u origin main` connects your local repo to GitHub. `git pull` fetches and merges remote changes.
- Conventional commits — `feat:`, `fix:`, `docs:`, `test:` — make history readable and automatable.

- `git add -p` stages changes chunk by chunk, keeping commits logical and focused.
- `git tag -a v1.0.0` marks a release. Push tags with `git push origin --tags`.

51.19 What's Next?

Chapter 52 covers **Environment Management and Project Structure** — virtual environments in depth, choosing between `pip`, `poetry`, and `uv`, professional project layouts, and managing secrets safely.

Chapter 52

Chapter 52: Environment Management and Project Structure

Two things trip up almost every beginner Python developer:

1. "Why does my code work on my machine but not on my teammate's?"
2. "I installed something and now everything is broken."

Both problems come from the same root cause: mixing packages across projects. The solution is **virtual environments** — isolated Python installations, one per project.

This chapter covers virtual environments in depth, the modern tools that make them easier, how to handle secrets safely, and how to structure a professional Python project.

52.1 The Problem — Why You Need Virtual Environments

Python packages are installed globally by default. This causes conflicts:

```
Project A needs:  requests==2.28
Project B needs:  requests==2.32
Python global:    requests==2.28   <- Project B breaks
```

Virtual environments solve this by giving each project its own isolated copy of Python and its packages.

52.2 `venv` — Python's Built-in Solution

`venv` ships with Python. No installation needed.

```
# Create a virtual environment
python -m venv .venv

# Activate it
# Windows (PowerShell):
.\.venv\Scripts\Activate.ps1

# Windows (Command Prompt):
.\.venv\Scripts\activate.bat

# Mac / Linux:
source .venv/bin/activate

# Your prompt changes to show the active environment:
# (.venv) user@machine:~/project$

# Install packages --- goes into .venv, not global Python
pip install requests pandas fastapi

# See what's installed
pip list
pip freeze          # with version pins (for requirements.txt)

# Deactivate when done
deactivate
```

Always add `.venv` (or `venv`, `env`) to your `.gitignore`. Never commit the virtual environment folder.

52.2.1 Creating `requirements.txt`

```
pip freeze > requirements.txt
```

This captures the exact versions of every installed package:

```
fastapi==0.110.0
pydantic==2.6.4
requests==2.31.0
uvicorn==0.28.0
```

Someone else reproduces your environment with:

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

52.2.2 The problem with `pip freeze`

`pip freeze` includes indirect dependencies (packages that your packages depend on). This is fine for deployment but noisy for development.

The cleaner approach: separate your direct dependencies (what you need) from indirect ones (what they need):

```
requirements.in     <- only what YOU need
requirements.txt    <- complete frozen list (generated)
```

52.3 `pip-tools` — Clean Dependency Management

```
pip install pip-tools
```

`requirements.in` — your direct dependencies:

```
fastapi
uvicorn[standard]
pydantic-settings
sqlalchemy>=2.0
```

Generate a pinned `requirements.txt`:

```
pip-compile requirements.in
```

This produces a fully pinned `requirements.txt` with hashes:

```
# This file was autogenerated by pip-compile
fastapi==0.110.0
    # via -r requirements.in
uvicorn==0.28.0
    # via -r requirements.in
...
```

Install exactly these versions:

```
pip-sync requirements.txt
```

`pip-sync` also removes packages that aren't in `requirements.txt` — your environment stays clean.

Keep both files in Git. Regenerate `requirements.txt` when you add a dependency:

```
pip-compile requirements.in
pip-sync requirements.txt
```

52.4 uv — The Modern Fast Alternative

`uv` is a next-generation Python package manager written in Rust. It's 10-100x faster than pip and handles virtual environments, dependency resolution, and more.

```
# Install uv (replaces pip in new projects)
pip install uv

# Create a project (creates pyproject.toml + .venv automatically)
uv init my-project
cd my-project

# Add dependencies
uv add fastapi uvicorn pydantic-settings

# Add dev dependencies
uv add --dev pytest ruff mypy

# Install everything
uv sync

# Run a command inside the environment
uv run python main.py
uv run pytest

# Remove a dependency
uv remove requests

# Update all packages to latest compatible versions
uv lock --upgrade
```

`uv` stores dependencies in `pyproject.toml` and locks them in `uv.lock`. Both go in Git.

For new projects in 2026, `uv` is the recommended choice. For existing projects with `requirements.txt`, stick with `pip` or `pip-tools`.

52.5 `poetry` — Dependencies + Packaging in One

Poetry is another popular tool that combines dependency management and packaging:

```
pip install poetry

poetry new my-project       # scaffold a new project
cd my-project

poetry add fastapi          # add to [tool.poetry.dependencies]
poetry add --dev pytest     # add to dev dependencies

poetry install              # install all dependencies
poetry run python main.py   # run inside the environment
poetry shell                # activate the environment

poetry update               # update to latest compatible versions
poetry show                 # list installed packages
poetry build                # build a wheel for publishing
poetry publish              # publish to PyPI
```

Poetry is opinionated and self-contained. If you're building a library to publish on PyPI, Poetry is a solid choice.

52.6 Choosing a Tool

Need	Recommended tool
Quick personal project	`venv` + `pip`
Team project, simple deps	`venv` + `pip-tools`
New project, want speed	`uv`
Library for PyPI	`uv` or Poetry
Complex monorepo	`uv` workspaces

52.7 Professional Project Structure

This is the layout I recommend for any non-trivial Python project:

```
my-project/
├── src/
│   └── myapp/
│       ├── __init__.py
│       ├── main.py
│       ├── config.py
│       ├── models/
│       │   ├── __init__.py
│       │   └── user.py
│       ├── services/
│       │   ├── __init__.py
│       │   └── auth.py
│       └── utils/
│           ├── __init__.py
│           └── helpers.py
├── tests/
│   ├── conftest.py
│   ├── unit/
│   │   ├── test_models.py
│   │   └── test_utils.py
│   └── integration/
│       └── test_api.py
├── docs/
│   ├── index.md
│   └── api.md
├── scripts/
│   ├── seed_db.py
│   └── generate_report.py
├── .github/
│   └── workflows/
│       └── ci.yml
├── .env.example          <- template for environment variables
├── .env                  <- your actual secrets (in .gitignore)
├── .gitignore
├── pyproject.toml
├── README.md
└── LICENSE
```

Key principles: - **`src/` layout** — prevents accidental imports of your code during testing - **Separate `tests/`** — outside `src/`, with `unit/` and `integration/` subdirectories - **`scripts/`** — one-off utility scripts (seed database, generate reports) - **`docs/`** — documentation lives close to the code - **`.env.example`** — checked into Git; shows what variables are needed without exposing values

52.8 Environment Variables and Secrets

Never hardcode credentials, API keys, database URLs, or any secret in your code. Not even in comments.

52.8.1 `.env` files

```
pip install python-dotenv
```

`.env` — your actual secrets (never commit this):

```
DATABASE_URL=postgresql://user:password@localhost/mydb
SECRET_KEY=super-secret-key-here
DEBUG=false
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=myapp@gmail.com
SMTP_PASSWORD=app-password-here
STRIPE_KEY=sk_live_abc123
```

`.env.example` — the template (commit this):

```
DATABASE_URL=postgresql://user:password@localhost/mydb
SECRET_KEY=change-this-to-a-random-string
DEBUG=false
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=
SMTP_PASSWORD=
STRIPE_KEY=
```

Load them in Python:

```
from dotenv import load_dotenv
import os

load_dotenv()   # loads .env into os.environ

db_url     = os.environ["DATABASE_URL"]    # raises KeyError if missing
debug      = os.getenv("DEBUG", "false") == "true"  # with default
secret_key = os.environ.get("SECRET_KEY", "")
```

52.8.2 `pydantic-settings` — Validated Configuration

For serious projects, use `pydantic-settings`. It validates environment variables against a schema and provides autocomplete:

```python
# config.py
from functools import lru_cache
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    # App
    app_name:     str  = "My App"
    debug:        bool = False
    log_level:    str  = "INFO"

    # Database
    database_url: str

    # Auth
    secret_key:   str
    token_expiry_hours: int = 24

    # Email
    smtp_host:     str = "smtp.gmail.com"
    smtp_port:     int = 587
    smtp_user:     str = ""
    smtp_password: str = ""

    model_config = SettingsConfigDict(
        env_file=".env",
        env_file_encoding="utf-8",
        case_sensitive=False,
    )


@lru_cache
def get_settings() -> Settings:
    """Return a cached Settings instance."""
    return Settings()
```

Use it anywhere in your app:

```python
from config import get_settings

settings = get_settings()
print(settings.database_url)    # type: str --- validated and typed
print(settings.debug)           # type: bool --- "false" -> False automatically
```

If `DATABASE_URL` or `SECRET_KEY` is missing from the environment, pydantic raises a clear error at startup — not buried in a cryptic crash later.

52.8.3 Never Do This

```
# BAD --- hardcoded secret
API_KEY = "sk-abc123xyz"
db = connect("postgresql://admin:password123@prod-db.example.com/mydb")

# BAD --- secret in a comment
# password: hunter2

# BAD --- secret in a test
def test_something():
    token = "eyJhbGci..."   # a real JWT from production
```

52.8.4 Production Secrets

For production, use your hosting platform's secret management: - **Render / Railway / Heroku** -> Environment variables in the dashboard - **AWS** -> AWS Secrets Manager or Parameter Store - **GCP** -> Secret Manager - **GitHub Actions** -> Repository secrets (`Settings -> Secrets -> Actions`)

```
# .github/workflows/ci.yml --- access secrets in CI
env:
  DATABASE_URL: ${{ secrets.DATABASE_URL }}
  SECRET_KEY:   ${{ secrets.SECRET_KEY }}
```

52.9 Managing Multiple Python Versions

Different projects need different Python versions. Use `pyenv` to manage them:

```
# Install pyenv (Mac/Linux)
curl https://pyenv.run | bash

# List available versions
pyenv install --list | grep "3\."

# Install specific versions
pyenv install 3.11.9
pyenv install 3.12.3

# Set global default
pyenv global 3.12.3

# Set version for a specific project
cd my-project
```

```
pyenv local 3.11.9          # creates .python-version file
python --version            # 3.11.9

# List installed versions
pyenv versions
```

On Windows, use the Python Launcher:

```
py -3.11 -m venv .venv    # create venv with Python 3.11
py -3.12 -m venv .venv    # create venv with Python 3.12
```

52.10 A Complete Project Setup Script

Put this in a `Makefile` or `justfile` to automate setup:

```
# Makefile

.PHONY: setup test lint typecheck clean

setup:
    python -m venv .venv
    .venv/bin/pip install -e ".[dev]"
    cp -n .env.example .env || true

test:
    .venv/bin/pytest tests/ -v --cov=src --cov-report=term-missing

lint:
    .venv/bin/ruff check src/ tests/
    .venv/bin/ruff format --check src/ tests/

typecheck:
    .venv/bin/mypy src/

clean:
    rm -rf .venv __pycache__ .pytest_cache .mypy_cache dist/ build/
    find . -name "*.pyc" -delete
```

Run everything with:

```
make setup      # create venv, install deps
make test       # run tests with coverage
make lint       # check code style
make typecheck  # run mypy
```

52.11 The Checklist for Every New Python Project

```
☐ Create a virtual environment (.venv)
☐ Add .venv to .gitignore
☐ Create pyproject.toml (or requirements.in + requirements.txt)
☐ Add .env.example with all required variable names
☐ Add .env to .gitignore
☐ Set up ruff for linting (pyproject.toml [tool.ruff])
☐ Set up mypy for type checking (pyproject.toml [tool.mypy])
☐ Set up pytest (pyproject.toml [tool.pytest.ini_options])
☐ Create .github/workflows/ci.yml
☐ Create README.md with: what it does, how to install, how to run
☐ git init + first commit
```

52.12 What You Learned in This Chapter

- Virtual environments isolate packages per project. `python -m venv .venv` creates one. Always add `.venv` to `.gitignore`.
- `pip freeze > requirements.txt` pins all packages. `pip install -r requirements.txt` reproduces the environment.
- `pip-tools` separates your direct deps (`requirements.in`) from the full pinned lock (`requirements.txt`). `pip-compile` generates, `pip-sync` installs.
- `uv` is the modern, fast alternative: `uv add`, `uv sync`, `uv run`. Ideal for new projects.
- Poetry combines dependency management and packaging in one tool.
- Professional projects use `src/` layout, separate `tests/` with `unit/` and `integration/`, a `scripts/` folder, and `.env.example`.
- Secrets live in `.env` (never committed). Load them with `python-dotenv` or `pydantic-settings`.
- `pydantic-settings` validates env vars against a schema and gives you typed, autocompleted config.
- `pyenv` manages multiple Python versions. `pyenv local 3.11.9` pins a version to a project.

52.13 What's Next?

Chapter 53 covers **Logging** — replacing scattered `print()` statements with a structured logging system that tells you exactly what your application is doing in production, without wading through noise.

Chapter 53

Chapter 53: Logging

`print()` is fine for scripts. For real applications, it's a disaster.

`print()` has no timestamps, no severity levels, no way to filter messages, no way to write to a file without redirecting stdout, and no way to turn it off in production without deleting it. The moment something goes wrong at 3am, you'll wish you had proper logging.

Python's built-in `logging` module solves all of this. It takes ten minutes to set up and pays dividends every time you debug a production issue.

53.1 The Problem with `print()`

```
# This is everywhere in your codebase
print("Starting server...")
print("Connecting to database")
print("Processing 1000 records")
print("Done!")
```

Problems: - No timestamp — when did this happen? - No severity — is this normal or a warning? - Can't filter — want only errors? Too bad. - Can't write to file without shell tricks - Can't disable without editing code - No context — which module, which function?

53.2 The Logging Module — Basics

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("Detailed debugging info")
logging.info("Normal operation message")
logging.warning("Something unexpected happened")
logging.error("A serious error occurred")
logging.critical("Application cannot continue")
```

Output:

```
DEBUG:root:Detailed debugging info
INFO:root:Normal operation message
WARNING:root:Something unexpected happened
ERROR:root:A serious error occurred
CRITICAL:root:Application cannot continue
```

53.3 Log Levels

There are five standard log levels, in increasing severity:

Level	Value	When to use
`DEBUG`	10	Detailed info for debugging. Off in production.
`INFO`	20	Confirm things are working as expected.
`WARNING`	30	Something unexpected, but the app continues. Default level.
`ERROR`	40	A serious problem. The app may keep running.
`CRITICAL`	50	Fatal error. The app cannot continue.

Setting a level filters out everything below it:

```
logging.basicConfig(level=logging.WARNING)
# DEBUG and INFO messages are now silently ignored
# WARNING, ERROR, CRITICAL are shown
```

In development: `DEBUG`. In production: `INFO` or `WARNING`.

53.4 Named Loggers — The Right Way

Never use the root logger (`logging.debug(...)`) in real code. Always create a **named logger** per module:

```
# In every Python file
import logging

log = logging.getLogger(__name__)
# __name__ is the module's dotted path, e.g. "myapp.services.auth"

log.debug("Starting authentication")
log.info("User %s logged in", username)
log.warning("Failed login attempt for %s", username)
log.error("Database connection failed: %s", error)
```

Why `__name__`? It creates a hierarchy:

```
myapp
  ├── myapp.api
  ├── myapp.services
  │      └── myapp.services.auth
  └── myapp.database
```

You can configure `myapp` as a whole, or silence just `myapp.database`. The hierarchy gives you fine-grained control without touching every file.

53.5 Configuring Logging Properly

`logging.basicConfig()` is fine for scripts. For applications, configure logging explicitly:

```
import logging
import sys

def setup_logging(level: str = "INFO") -> None:
    """Configure logging for the application."""

    # Create a formatter
    formatter = logging.Formatter(
        fmt="%(asctime)s  %(levelname)-8s  %(name)s --- %(message)s",
```

```
        datefmt="%Y-%m-%d %H:%M:%S",
    )

    # Console handler --- writes to stdout
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setLevel(logging.DEBUG)
    console_handler.setFormatter(formatter)

    # Configure the root logger
    root_logger = logging.getLogger()
    root_logger.setLevel(getattr(logging, level.upper(), logging.INFO))
    root_logger.addHandler(console_handler)

    # Silence noisy third-party libraries
    logging.getLogger("urllib3").setLevel(logging.WARNING)
    logging.getLogger("httpx").setLevel(logging.WARNING)
    logging.getLogger("sqlalchemy.engine").setLevel(logging.WARNING)


# Call this once at application startup
setup_logging(level="DEBUG")
```

Output now looks like:

```
2026-03-09 14:22:01  INFO      myapp.main --- Starting application
2026-03-09 14:22:01  DEBUG    myapp.database --- Connecting to database
2026-03-09  14:22:02    INFO             myapp.services.auth  ---
User alice logged in
2026-03-09 14:22:05  WARNING  myapp.api --- Rate limit exceeded for 192.168.1.1
2026-03-09 14:22:10  ERROR    myapp.database --- Query timeout after 30s
```

53.6 Logging to Files

```
import logging
from logging.handlers import RotatingFileHandler
from pathlib import Path

def setup_logging(
    level:    str = "INFO",
    log_dir:  str = "logs",
    log_file: str = "app.log",
) -> None:
    Path(log_dir).mkdir(exist_ok=True)
```

```
    log_path = Path(log_dir) / log_file

    formatter = logging.Formatter(
        fmt="%(asctime)s  %(levelname)-8s  %(name)s --- %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    )

    # Console handler
    console = logging.StreamHandler()
    console.setFormatter(formatter)
    console.setLevel(logging.DEBUG)

    # File handler --- plain file
    file_handler = logging.FileHandler(log_path, encoding="utf-8")
    file_handler.setFormatter(formatter)
    file_handler.setLevel(logging.INFO)    # less noise in file

    # Rotating file handler --- max 10MB per file, keep 5 backups
    rotating = RotatingFileHandler(
        log_path,
        maxBytes=10 * 1024 * 1024,    # 10 MB
        backupCount=5,
        encoding="utf-8",
    )
    rotating.setFormatter(formatter)
    rotating.setLevel(logging.INFO)

    root = logging.getLogger()
    root.setLevel(getattr(logging, level.upper(), logging.INFO))
    root.addHandler(console)
    root.addHandler(rotating)
```

`RotatingFileHandler` creates `app.log`, then `app.log.1`, `app.log.2`, etc. when the file reaches 10MB. It keeps 5 backup files and deletes the oldest. Your disk never fills up.

For daily rotation use `TimedRotatingFileHandler`:

```
from logging.handlers import TimedRotatingFileHandler

daily = TimedRotatingFileHandler(
    "logs/app.log",
    when="midnight",      # rotate at midnight
    interval=1,
    backupCount=30,       # keep 30 days
    encoding="utf-8",
```

```
)
```

53.7 Logging Exceptions

Always log the full traceback when catching exceptions:

```
import logging

log = logging.getLogger(__name__)


def process_file(path: str) -> None:
    try:
        with open(path) as f:
            data = f.read()
    except FileNotFoundError:
        log.error("File not found: %s", path)
        raise
    except PermissionError:
        log.error("Permission denied reading: %s", path)
        raise
    except Exception:
        # exc_info=True includes the full traceback
        log.exception("Unexpected error reading file: %s", path)
        raise
```

`log.exception(...)` is equivalent to `log.error(..., exc_info=True)` — it logs the message AND the full traceback. Use it inside `except` blocks.

Output:

```
2026-03-09 14:30:15 ERROR    myapp.processor --- Unexpected error reading file: data.
Traceback (most recent call last):
  File "processor.py", line 8, in process_file
    data = f.read()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 4
```

53.8 Structured Logging — JSON Logs for Production

In production, logs are often parsed by tools like Datadog, Splunk, or Elasticsearch. Plain text logs are hard to query. JSON logs are easy.

```
pip install python-json-logger
```

```
import logging
import sys
from pythonjsonlogger import jsonlogger


def setup_json_logging(level: str = "INFO") -> None:
    formatter = jsonlogger.JsonFormatter(
        fmt="%(asctime)s %(levelname)s %(name)s %(message)s",
        datefmt="%Y-%m-%dT%H:%M:%S",
    )
    handler = logging.StreamHandler(sys.stdout)
    handler.setFormatter(formatter)

    root = logging.getLogger()
    root.setLevel(getattr(logging, level.upper(), logging.INFO))
    root.addHandler(handler)
```

Each log line is now a JSON object:

```
{"asctime": "2026-03-09T14:22:01", "levelname": "INFO", "name": "myapp.auth", "
{"asctime": "2026-03-09T14:22:10", "levelname": "ERROR", "name": "myapp.db", "m
```

Add extra fields to any log call:

```
log.info("User logged in", extra={"username": username, "ip": client_ip})
log.error("Query timeout", extra={"query": sql, "duration_ms": elapsed_ms})
```

53.9 Adding Context with LoggerAdapter

When processing a request, you want every log message to include the request ID without passing it to every function:

```
import logging
from typing import Any

log = logging.getLogger(__name__)


class RequestAdapter(logging.LoggerAdapter):
    """Injects request_id into every log record."""

    def process(self, msg: str, kwargs: dict) -> tuple[str, dict]:
        extra = kwargs.setdefault("extra", {})
```

```
        extra["request_id"] = self.extra.get("request_id", "-")
        return msg, kwargs


# In a FastAPI middleware or request handler:
def get_request_logger(request_id: str) -> logging.LoggerAdapter:
    return RequestAdapter(log, {"request_id": request_id})


# Usage
def handle_request(request_id: str, user_id: int) -> None:
    rlog = get_request_logger(request_id)
    rlog.info("Processing request for user %d", user_id)
    rlog.debug("Fetching from database")
    rlog.info("Request complete")
```

Output (every line has the request ID):

```
2026-03-09 14:22:01  INFO  myapp --- [req-abc123] Processing request for user 42
2026-03-09 14:22:01  DEBUG myapp --- [req-abc123] Fetching from database
2026-03-09 14:22:01  INFO  myapp --- [req-abc123] Request complete
```

53.10 Complete Production Logging Setup

```
# logging_config.py
"""
Centralized logging configuration.
Call setup_logging() once at application startup.
"""
import logging
import logging.config
import sys
from pathlib import Path


NOISY_LIBRARIES = [
    "urllib3",
    "httpx",
    "httpcore",
    "sqlalchemy.engine",
    "sqlalchemy.pool",
    "multipart",
    "passlib",
```

```
    "boto3",
    "botocore",
    "s3transfer",
]


def setup_logging(
    level:       str  = "INFO",
    log_dir:     str  = "logs",
    json_output: bool = False,
) -> None:
    """
    Configure application-wide logging.

    Args:
        level:       Minimum log level (DEBUG, INFO, WARNING, ERROR).
        log_dir:     Directory for log files. None = console only.
        json_output: If True, output JSON (for production log aggregators).
    """
    Path(log_dir).mkdir(parents=True, exist_ok=True)

    numeric_level = getattr(logging, level.upper(), logging.INFO)

    if json_output:
        try:
            from pythonjsonlogger import jsonlogger
            formatter = jsonlogger.JsonFormatter(
                fmt="%(asctime)s %(levelname)s %(name)s %(message)s",
                datefmt="%Y-%m-%dT%H:%M:%SZ",
            )
        except ImportError:
            formatter = _text_formatter()
    else:
        formatter = _text_formatter()

    handlers = []

    # Console
    console = logging.StreamHandler(sys.stdout)
    console.setFormatter(formatter)
    console.setLevel(numeric_level)
    handlers.append(console)

    # Rotating file
```

```
    if log_dir:
        from logging.handlers import RotatingFileHandler
        file_handler = RotatingFileHandler(
            Path(log_dir) / "app.log",
            maxBytes=10 * 1024 * 1024,
            backupCount=5,
            encoding="utf-8",
        )
        file_handler.setFormatter(formatter)
        file_handler.setLevel(logging.INFO)
        handlers.append(file_handler)

        # Separate error log
        error_handler = RotatingFileHandler(
            Path(log_dir) / "error.log",
            maxBytes=10 * 1024 * 1024,
            backupCount=10,
            encoding="utf-8",
        )
        error_handler.setFormatter(formatter)
        error_handler.setLevel(logging.ERROR)
        handlers.append(error_handler)

    # Configure root logger
    root = logging.getLogger()
    root.setLevel(numeric_level)
    for h in handlers:
        root.addHandler(h)

    # Silence noisy libraries
    for library in NOISY_LIBRARIES:
        logging.getLogger(library).setLevel(logging.WARNING)

    logging.getLogger(__name__).info(
        "Logging configured: level=%s, log_dir=%s, json=%s",
        level, log_dir, json_output,
    )


def _text_formatter() -> logging.Formatter:
    return logging.Formatter(
        fmt="%(asctime)s  %(levelname)-8s  %(name)s --- %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    )
```

Use it:

```
# main.py
from logging_config import setup_logging
import logging

setup_logging(level="DEBUG")    # development
# setup_logging(level="INFO", json_output=True)  # production

log = logging.getLogger(__name__)
log.info("Application started")
```

53.11 Logging in FastAPI

```
# main.py
from contextlib import asynccontextmanager
from fastapi import FastAPI, Request
import logging
import time
import uuid
from logging_config import setup_logging

setup_logging(level="INFO")
log = logging.getLogger(__name__)


@asynccontextmanager
async def lifespan(app: FastAPI):
    log.info("Application starting up")
    yield
    log.info("Application shutting down")


app = FastAPI(lifespan=lifespan)


@app.middleware("http")
async def log_requests(request: Request, call_next):
    request_id = str(uuid.uuid4())[:8]
    start      = time.time()

    log.info(
```

```
        "-> %s %s [%s]",
        request.method, request.url.path, request_id,
        extra={"request_id": request_id},
    )

    response = await call_next(request)

    duration_ms = (time.time() - start) * 1000
    log.info(
        "<- %s %s %dms [%s]",
        response.status_code, request.url.path, duration_ms, request_id,
        extra={"request_id": request_id, "duration_ms": round(duration_ms)},
    )
    return response
```

Output for every request:

```
2026-03-09 14:22:01  INFO     myapp.main --- -> GET /products [a3f2b1c4]
2026-03-09 14:22:01  INFO     myapp.main --- <- 200 /products 12ms [a3f2b1c4]
2026-03-09 14:22:02  INFO     myapp.main --- -> POST /auth/token [d7e1a8b5]
2026-03-09 14:22:02  INFO     myapp.main --- <- 200 /auth/token 45ms [d7e1a8b5]
2026-03-09 14:22:03  WARNING  myapp.main --- -> GET /products/9999 [c2f3d4e5]
2026-03-09 14:22:03  INFO     myapp.main --- <- 404 /products/9999 3ms [c2f3d4e5]
```

53.12 What You Learned in This Chapter

- `print()` has no timestamps, severity, filtering, or file output. Use `logging` for anything beyond a quick script.
- Five levels in order: `DEBUG < INFO < WARNING < ERROR < CRITICAL`. Set the level to filter below it.
- Always create named loggers with `logging.getLogger(__name__)`. Never use the root logger directly.
- Format strings: `%(asctime)s`, `%(levelname)s`, `%(name)s`, `%(message)s`. Use `%s` style in log messages (not f-strings) — lazy evaluation skips string building if the level is filtered.
- `RotatingFileHandler` caps file size and keeps backups. `TimedRotatingFileHandler` rotates daily.
- `log.exception("msg")` logs the full traceback. Use it inside `except` blocks.
- Structured JSON logging (`python-json-logger`) makes production logs queryable by log aggregators.
- `LoggerAdapter` injects context (request ID, user ID) into every log message in a scope.

- Silence noisy third-party libraries with `logging.getLogger("urllib3").setLevel(logg`
- Call your setup function once at startup. Every module just does `log = logging.getLogger(__name__)`.

53.13 What's Next?

Chapter 54 covers **Design Patterns in Python** — proven solutions to recurring problems in software design: Singleton, Factory, Observer, Strategy, Decorator, and more. You'll learn which patterns are genuinely useful in Python and which the language already handles for you.

Chapter 54

Chapter 54: Design Patterns in Python

Design patterns are battle-tested solutions to problems that come up again and again in software development.

They're not code you copy. They're blueprints — proven ways of structuring relationships between objects. The Gang of Four (GoF) book catalogued 23 patterns in 1994. Some are essential. Some are obsolete. Some are already built into Python.

This chapter teaches you the ones that actually matter in modern Python code, how they look in practice, and which patterns Python makes unnecessary.

54.1 Why Design Patterns Matter

Without patterns, codebases grow in ad-hoc ways:

- Logic duplicated in ten places
- Classes that know too much about each other
- Changes that break things across the codebase
- Code impossible to test in isolation

Patterns give you a vocabulary and a toolkit to avoid these problems.

54.2 Category 1: Creational Patterns

Creational patterns deal with object creation.

54.2.1 Singleton — One Instance Only

The Singleton pattern ensures only one instance of a class ever exists. Common uses: database connections, configuration objects, caches.

```
import threading


class Singleton:
    _instance = None
    _lock     = threading.Lock()

    def __new__(cls):
        with cls._lock:                # thread-safe
            if cls._instance is None:
                cls._instance = super().__new__(cls)
        return cls._instance


# Both variables point to the SAME object
a = Singleton()
b = Singleton()
print(a is b)    # True
```

The Pythonic way: Use a module. A Python module is a singleton by definition — it's loaded once and cached. Your `config.py` with a `@lru_cache` singleton is idiomatic Python:

```
# config.py
from functools import lru_cache

@lru_cache(maxsize=None)
def get_settings():
    return Settings()

# Anywhere in your app:
settings = get_settings()    # always the same object
```

54.2.2 Factory — Create Objects Without Specifying the Exact Class

The Factory pattern provides a function or method that creates objects. The caller doesn't need to know which class to instantiate.

```
from abc import ABC, abstractmethod


# Abstract product
class Notification(ABC):
    @abstractmethod
    def send(self, message: str, recipient: str) -> None:
        ...


# Concrete products
class EmailNotification(Notification):
    def send(self, message: str, recipient: str) -> None:
        print(f"Email to {recipient}: {message}")


class SMSNotification(Notification):
    def send(self, message: str, recipient: str) -> None:
        print(f"SMS to {recipient}: {message}")


class PushNotification(Notification):
    def send(self, message: str, recipient: str) -> None:
        print(f"Push to {recipient}: {message}")


# Factory function
def create_notification(channel: str) -> Notification:
    channels = {
        "email": EmailNotification,
        "sms":   SMSNotification,
        "push":  PushNotification,
    }
    cls = channels.get(channel.lower())
    if cls is None:
        raise ValueError(f"Unknown channel: {channel!r}. "
                         f"Valid: {list(channels)}")
    return cls()
```

```
# Usage --- caller doesn't know which class is created
notifier = create_notification("email")
notifier.send("Your order shipped!", "alice@example.com")

notifier = create_notification("sms")
notifier.send("Your code is 1234", "+44 7700 900000")
```

To add a new channel: add one line to `channels`. Nothing else changes.

54.2.3 Builder — Construct Complex Objects Step by Step

The Builder pattern constructs complex objects with many optional parameters through a fluent interface:

```
from dataclasses import dataclass, field


@dataclass
class Query:
    table:      str
    columns:    list[str]    = field(default_factory=lambda: ["*"])
    conditions: list[str]    = field(default_factory=list)
    order_by:   str | None   = None
    limit_:     int | None   = None
    offset_:    int | None   = None

    def to_sql(self) -> str:
        cols  = ", ".join(self.columns)
        sql   = f"SELECT {cols} FROM {self.table}"
        if self.conditions:
            sql += " WHERE " + " AND ".join(self.conditions)
        if self.order_by:
            sql += f" ORDER BY {self.order_by}"
        if self.limit_ is not None:
            sql += f" LIMIT {self.limit_}"
        if self.offset_ is not None:
            sql += f" OFFSET {self.offset_}"
        return sql


class QueryBuilder:
    def __init__(self, table: str):
```

```
        self._query = Query(table=table)

    def select(self, *columns: str) -> "QueryBuilder":
        self._query.columns = list(columns)
        return self

    def where(self, condition: str) -> "QueryBuilder":
        self._query.conditions.append(condition)
        return self

    def order_by(self, column: str) -> "QueryBuilder":
        self._query.order_by = column
        return self

    def limit(self, n: int) -> "QueryBuilder":
        self._query.limit_ = n
        return self

    def offset(self, n: int) -> "QueryBuilder":
        self._query.offset_ = n
        return self

    def build(self) -> Query:
        return self._query


# Fluent interface --- each method returns self
query = (
    QueryBuilder("products")
    .select("id", "name", "price")
    .where("price > 10")
    .where("in_stock = 1")
    .order_by("price")
    .limit(20)
    .offset(40)
    .build()
)

print(query.to_sql())
# SELECT id, name, price FROM products
# WHERE price > 10 AND in_stock = 1
# ORDER BY price LIMIT 20 OFFSET 40
```

54.3 Category 2: Structural Patterns

Structural patterns deal with composing classes and objects.

54.3.1 Decorator Pattern (vs Python's `@decorator`)

The design-pattern Decorator wraps an object to add behaviour. Python's `@decorator` syntax is related but different — it wraps a function.

The object-based Decorator pattern:

```
from abc import ABC, abstractmethod


class DataSource(ABC):
    @abstractmethod
    def write(self, data: str) -> None: ...
    @abstractmethod
    def read(self) -> str: ...


class FileDataSource(DataSource):
    def __init__(self, path: str):
        self._path = path

    def write(self, data: str) -> None:
        with open(self._path, "w") as f:
            f.write(data)

    def read(self) -> str:
        with open(self._path) as f:
            return f.read()


class DataSourceDecorator(DataSource):
    """Base decorator --- wraps another DataSource."""
    def __init__(self, source: DataSource):
        self._source = source

    def write(self, data: str) -> None:
        self._source.write(data)

    def read(self) -> str:
        return self._source.read()
```

```
class EncryptionDecorator(DataSourceDecorator):
    """Adds encryption on top of any DataSource."""
    def write(self, data: str) -> None:
        encrypted = data[::-1]   # toy encryption
        self._source.write(encrypted)

    def read(self) -> str:
        return self._source.read()[::-1]


class CompressionDecorator(DataSourceDecorator):
    """Adds compression on top of any DataSource."""
    def write(self, data: str) -> None:
        import zlib, base64
        compressed = base64.b64encode(zlib.compress(data.encode())).decode()
        self._source.write(compressed)

    def read(self) -> str:
        import zlib, base64
        return zlib.decompress(base64.b64decode(self._source.read())).decode()


# Stack decorators --- each adds a layer
source = FileDataSource("data.txt")
source = CompressionDecorator(source)
source = EncryptionDecorator(source)

source.write("Hello, World!")    # encrypt(compress("Hello, World!"))
print(source.read())             # -> "Hello, World!" (unwrapped transparently)
```

54.3.2 Adapter — Make Incompatible Interfaces Work Together

You have a class with one interface. You need another. The Adapter wraps the original:

```
class EuropeanSocket:
    """Provides 230V."""
    def voltage(self) -> int: return 230
    def live(self) -> int:    return 1
    def neutral(self) -> int: return -1
```

```
class USASocket:
    """Provides 110V."""
    def voltage(self) -> int: return 110
    def live(self) -> int:    return 1
    def neutral(self) -> int: return -1


class USADevice:
    """Requires a USASocket interface."""
    def run(self, socket: USASocket) -> None:
        if socket.voltage() > 150:
            raise RuntimeError("Too much voltage!")
        print(f"Running on {socket.voltage()}V")


class EuropeanToUSAAdapter(USASocket):
    """Makes a EuropeanSocket look like a USASocket."""
    def __init__(self, socket: EuropeanSocket):
        self._socket = socket

    def voltage(self) -> int:
        return 110   # step down from 230V

    def live(self) -> int:
        return self._socket.live()

    def neutral(self) -> int:
        return self._socket.neutral()


# Plug a US device into a European socket via the adapter
euro_socket = EuropeanSocket()
adapter     = EuropeanToUSAAdapter(euro_socket)
device      = USADevice()
device.run(adapter)   # "Running on 110V"
```

Real-world example: wrapping a third-party API client to match your own interface.

54.3.3 Facade — Simplify a Complex System

Facade provides a simple interface to a complex subsystem:

```
class VideoConverter:
    """
    Facade that hides the complexity of video conversion.
    Internally uses Codec, AudioMixer, BitrateReader, etc.
    """

    def convert(self, filename: str, target_format: str) -> str:
        print(f"VideoConverter: opening file {filename}")
        print(f"VideoConverter: detecting codec")
        print(f"VideoConverter: extracting audio")
        print(f"VideoConverter: re-encoding to {target_format}")
        print(f"VideoConverter: writing output file")
        output = filename.rsplit(".", 1)[0] + "." + target_format
        print(f"VideoConverter: done -> {output}")
        return output


# Client uses one simple method
converter = VideoConverter()
output    = converter.convert("movie.avi", "mp4")
```

Flask itself is a Facade — it hides the complexity of WSGI, HTTP parsing, and routing behind a simple `@app.route` decorator.

54.4 Category 3: Behavioural Patterns

Behavioural patterns deal with communication between objects.

54.4.1 Observer — Notify Multiple Objects of State Changes

When one object changes state, dependent objects are notified automatically:

```
from abc import ABC, abstractmethod
from typing import Any


class Observer(ABC):
    @abstractmethod
    def update(self, event: str, data: Any) -> None: ...


class EventBus:
```

```
    """Simple publish/subscribe event bus."""

    def __init__(self):
        self._subscribers: dict[str, list[Observer]] = {}

    def subscribe(self, event: str, observer: Observer) -> None:
        self._subscribers.setdefault(event, []).append(observer)

    def unsubscribe(self, event: str, observer: Observer) -> None:
        self._subscribers.get(event, []).remove(observer)

    def publish(self, event: str, data: Any = None) -> None:
        for observer in self._subscribers.get(event, []):
            observer.update(event, data)


# Concrete observers
class EmailService(Observer):
    def update(self, event: str, data: Any) -> None:
        if event == "user.registered":
            print(f"EmailService: sending welcome email to {data['email']}")


class AnalyticsService(Observer):
    def update(self, event: str, data: Any) -> None:
        print(f"Analytics: tracking {event} --- {data}")


class AuditLog(Observer):
    def update(self, event: str, data: Any) -> None:
        print(f"AuditLog: [{event}] {data}")


# Wire it up
bus = EventBus()
bus.subscribe("user.registered", EmailService())
bus.subscribe("user.registered", AnalyticsService())
bus.subscribe("user.registered", AuditLog())

# Publish an event --- all subscribers notified
bus.publish("user.registered", {"email": "alice@example.com", "id": 42})
```

Output:

```
EmailService: sending welcome email to alice@example.com
```

```
Analytics: tracking user.registered --- {'email': 'alice@example.com', 'id': 42}
AuditLog: [user.registered] {'email': 'alice@example.com', 'id': 42}
```

54.4.2 Strategy — Swap Algorithms at Runtime

Define a family of algorithms, encapsulate each one, make them interchangeable:

```
from abc import ABC, abstractmethod
from typing import Callable


class SortStrategy(ABC):
    @abstractmethod
    def sort(self, data: list) -> list: ...


class BubbleSort(SortStrategy):
    def sort(self, data: list) -> list:
        arr = data.copy()
        n   = len(arr)
        for i in range(n):
            for j in range(n - i - 1):
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j+1] = arr[j+1], arr[j]
        return arr


class QuickSort(SortStrategy):
    def sort(self, data: list) -> list:
        if len(data) <= 1:
            return data
        pivot = data[len(data) // 2]
        left  = [x for x in data if x < pivot]
        mid   = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return self.sort(left) + mid + self.sort(right)


class Sorter:
    def __init__(self, strategy: SortStrategy):
        self._strategy = strategy
```

```
    def set_strategy(self, strategy: SortStrategy) -> None:
        self._strategy = strategy

    def sort(self, data: list) -> list:
        return self._strategy.sort(data)


data   = [5, 2, 8, 1, 9, 3]
sorter = Sorter(BubbleSort())
print(sorter.sort(data))    # [1, 2, 3, 5, 8, 9]

sorter.set_strategy(QuickSort())
print(sorter.sort(data))    # [1, 2, 3, 5, 8, 9]
```

Pythonic version — just pass a function:

```
def sort_with(data: list, key_fn: Callable = None, reverse: bool = False) -> li
    return sorted(data, key=key_fn, reverse=reverse)

products = [{"name": "B", "price": 20}, {"name": "A", "price": 5}]
by_price = sort_with(products, key_fn=lambda p: p["price"])
by_name  = sort_with(products, key_fn=lambda p: p["name"])
```

Python's first-class functions often replace the Strategy pattern entirely.

54.4.3 Command — Encapsulate Requests as Objects

The Command pattern turns a request into a standalone object. This enables undo/redo, queuing, and logging of operations:

```
from abc import ABC, abstractmethod
from dataclasses import dataclass


class Command(ABC):
    @abstractmethod
    def execute(self) -> None: ...
    @abstractmethod
    def undo(self) -> None: ...


@dataclass
class TextEditor:
    text: str = ""
```

```
    def insert(self, pos: int, chars: str) -> None:
        self.text = self.text[:pos] + chars + self.text[pos:]

    def delete(self, pos: int, length: int) -> None:
        self.text = self.text[:pos] + self.text[pos + length:]


class InsertCommand(Command):
    def __init__(self, editor: TextEditor, pos: int, chars: str):
        self._editor = editor
        self._pos    = pos
        self._chars  = chars

    def execute(self) -> None:
        self._editor.insert(self._pos, self._chars)

    def undo(self) -> None:
        self._editor.delete(self._pos, len(self._chars))


class CommandHistory:
    def __init__(self):
        self._history: list[Command] = []

    def execute(self, cmd: Command) -> None:
        cmd.execute()
        self._history.append(cmd)

    def undo(self) -> None:
        if self._history:
            self._history.pop().undo()


editor  = TextEditor()
history = CommandHistory()

history.execute(InsertCommand(editor, 0, "Hello"))
history.execute(InsertCommand(editor, 5, ", World"))
print(editor.text)    # Hello, World

history.undo()
print(editor.text)    # Hello

history.undo()
```

```
print(editor.text)    # (empty)
```

54.4.4 Iterator — Traverse Without Exposing Internals

You've used Python iterators throughout this book — `for x in collection` is the Iterator pattern. Python makes it trivially easy with `__iter__` and `__next__` or `yield`.

```
class NumberRange:
    """Iterate over a range of numbers, step by step."""

    def __init__(self, start: int, stop: int, step: int = 1):
        self._start = start
        self._stop  = stop
        self._step  = step

    def __iter__(self):
        current = self._start
        while current < self._stop:
            yield current
            current += self._step


for n in NumberRange(0, 10, 2):
    print(n, end=" ")    # 0 2 4 6 8
```

Python's built-in `iter()` / `next()` / `StopIteration` handle the Iterator pattern natively.

54.5 Patterns Python Makes Unnecessary

Some classic GoF patterns exist to work around limitations in languages like Java or C++. Python doesn't have those limitations:

Pattern	Why Python doesn't need it
Abstract Factory	Just use a dict of classes or a factory function
Template Method	Use a base class with a method that calls `self.step()` — same thing
Prototype	`copy.deepcopy(obj)`

Pattern	Why Python doesn't need it
Memento	`dataclasses` with `copy.copy()` or pickling
Chain of Responsibility	A list of callables: `for handler in handlers: if handler(request): break`
Mediator	An event bus (shown above)

54.6 Pythonic Patterns — The Ones Python Adds

Python has its own idiomatic patterns that don't appear in GoF:

54.6.1 Context Manager (already in Chapter 24)

```
with open("file.txt") as f:
    data = f.read()
# file always closed --- even on exception
```

54.6.2 Descriptor (already in Chapter 36)

```
class Validated:
    def __set_name__(self, owner, name):
        self._name = name
    def __get__(self, obj, type=None):
        return obj.__dict__.get(self._name)
    def __set__(self, obj, value):
        if not isinstance(value, int):
            raise TypeError(f"{self._name} must be int")
        obj.__dict__[self._name] = value
```

54.6.3 Mixin

Mixins add behaviour to a class without requiring a specific base class:

```
class TimestampMixin:
    """Add created_at / updated_at to any model."""
    created_at: datetime = None
    updated_at: datetime = None

    def touch(self) -> None:
```

```
        from datetime import datetime
        now = datetime.utcnow()
        if self.created_at is None:
            self.created_at = now
        self.updated_at = now


class JSONMixin:
    """Add .to_json() to any class."""
    def to_json(self) -> str:
        import json
        return json.dumps(self.__dict__, default=str)


class User(TimestampMixin, JSONMixin):
    def __init__(self, name: str, email: str):
        self.name  = name
        self.email = email


user = User("Alice", "alice@example.com")
user.touch()
print(user.to_json())
```

54.7 When to Use Design Patterns

Use a pattern when: - You recognise the problem it solves - The pattern makes the code clearer to the next developer - You need the specific benefit (undo/redo -> Command, swap algorithms -> Strategy)

Don't use a pattern when: - It adds complexity without a clear benefit - A simpler Python feature already handles it (first-class functions, `with`, generators) - You're trying to pattern-match to a GoF name rather than solving a real problem

The best code is readable code. Patterns are tools, not goals.

54.8 What You Learned in This Chapter

- **Creational:** Singleton ensures one instance (`@lru_cache` is the Pythonic way). Factory delegates object creation to a function. Builder constructs

complex objects with a fluent interface.
- **Structural:** Decorator wraps objects to add behaviour. Adapter makes incompatible interfaces compatible. Facade simplifies a complex subsystem behind a clean interface.
- **Behavioural:** Observer notifies subscribers of events (publish/subscribe). Strategy swaps algorithms at runtime (or just pass a function). Command encapsulates operations as objects, enabling undo/redo. Iterator traverses collections (Python's `yield` and `for` loop handle this natively).
- Many GoF patterns are unnecessary in Python because the language already has first-class functions, `with` statements, generators, and powerful standard library tools.
- Pythonic patterns include Context Managers, Descriptors, and Mixins.
- Use patterns when they solve a real problem. Avoid them when a simpler Python feature already does the job.

54.9 What's Next?

Chapter 55 covers **Security Best Practices** — the common mistakes that lead to data breaches, how to validate and sanitise input, how to store passwords correctly, prevent SQL injection, manage secrets, and audit your dependencies for known vulnerabilities.

Chapter 55

Chapter 55: Security Best Practices

Security isn't a feature you add later. It's a habit you build from the start.

Most data breaches aren't caused by sophisticated hackers cracking encryption. They're caused by simple mistakes: hardcoded passwords, unsanitised inputs, weak hashing, outdated dependencies with known vulnerabilities. The good news — almost all of them are preventable with a handful of consistent practices.

This chapter covers the security mistakes Python developers make most often and exactly how to fix them.

55.1 Mistake 1: Hardcoding Secrets

This is the most common mistake. It shows up in code reviews, GitHub searches, and breach reports every day.

```
# BAD --- never do this
API_KEY    = "sk-abc123xyz987"
DB_URL     = "postgresql://admin:hunter2@prod.db.example.com/mydb"
SECRET_KEY = "mysecretkey"

# BAD --- even in comments
# Default password: admin123
```

The fix: environment variables and `.env` files (Chapter 52).

```
import os
from dotenv import load_dotenv

load_dotenv()

API_KEY    = os.environ["API_KEY"]     # raises KeyError if missing --- good
DB_URL     = os.environ["DATABASE_URL"]
SECRET_KEY = os.environ["SECRET_KEY"]
```

And add `.env` to `.gitignore` — always.

Scan your repo for accidental commits:

```
pip install detect-secrets
detect-secrets scan > .secrets.baseline
detect-secrets audit .secrets.baseline
```

55.2 Mistake 2: SQL Injection

SQL injection is the most damaging web vulnerability. It lets attackers read, modify, or delete your entire database — or run commands on your server.

```
import sqlite3

# CATASTROPHICALLY BAD --- never do this
user_input = "' OR '1'='1"   # attacker input

conn = sqlite3.connect("users.db")
query = f"SELECT * FROM users WHERE username = '{user_input}'"
# Actual query: SELECT * FROM users WHERE username = '' OR '1'='1'
# Returns ALL users
rows = conn.execute(query).fetchall()
```

The fix: always use parameterised queries:

```
# CORRECT --- parameterised queries
def get_user(username: str) -> dict | None:
    conn = sqlite3.connect("users.db")
    row  = conn.execute(
        "SELECT * FROM users WHERE username = ?",
        (username,)    # the database driver escapes this safely
    ).fetchone()
    return dict(row) if row else None

# SQLAlchemy ORM --- also safe by default
```

```
from sqlalchemy.orm import Session

def get_user_orm(db: Session, username: str):
    return db.query(User).filter(User.username == username).first()
    # SQLAlchemy uses parameterised queries automatically
```

Never concatenate user input into SQL strings. Ever. The `?` placeholder (SQLite) or `%s` (PostgreSQL/psycopg2) is always correct.

55.3 Mistake 3: Storing Passwords Incorrectly

Never store plain-text passwords. Never use MD5 or SHA-256 directly for passwords — they're too fast; attackers can try billions of passwords per second.

Use a **password hashing algorithm** specifically designed to be slow: bcrypt, Argon2, or scrypt.

```
from passlib.context import CryptContext

# Configure --- bcrypt is the safe default
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")


def hash_password(plain: str) -> str:
    """Hash a password. Store the result, never the plain text."""
    return pwd_context.hash(plain)


def verify_password(plain: str, hashed: str) -> bool:
    """Return True if plain matches the hashed password."""
    return pwd_context.verify(plain, hashed)


# Usage
hashed = hash_password("MySecretPassword1!")
print(hashed)    # $2b$12$... (bcrypt hash)

print(verify_password("MySecretPassword1!", hashed))    # True
print(verify_password("wrongpassword",      hashed))    # False
```

bcrypt is intentionally slow (it takes ~100ms to hash). That makes brute-force attacks impractical — an attacker can only try ~10 passwords per second instead of billions.

55.3.1 Password strength requirements

```
import re


def check_password_strength(password: str) -> list[str]:
    """Return a list of unmet requirements (empty = strong password)."""
    issues = []
    if len(password) < 12:
        issues.append("At least 12 characters required")
    if not re.search(r"[A-Z]", password):
        issues.append("At least one uppercase letter required")
    if not re.search(r"[a-z]", password):
        issues.append("At least one lowercase letter required")
    if not re.search(r"\d", password):
        issues.append("At least one digit required")
    if not re.search(r"[!@#$%^&*()\-_=+\[\]{}|;:,.<>?]", password):
        issues.append("At least one special character required")
    return issues


issues = check_password_strength("abc")
if issues:
    print("Password too weak:")
    for issue in issues:
        print(f"  - {issue}")
```

55.4 Mistake 4: Not Validating Input

Never trust user input. Validate everything that comes from outside your system: HTTP requests, file uploads, command-line arguments, database reads from external sources.

```
from pydantic import BaseModel, Field, field_validator
import re


class UserRegistration(BaseModel):
    username: str  = Field(..., min_length=3, max_length=30)
    email:    str  = Field(...)
    password: str  = Field(..., min_length=12)
    age:      int  = Field(..., ge=13, le=120)
```

```
    @field_validator("username")
    @classmethod
    def username_safe(cls, v: str) -> str:
        if not re.match(r"^[a-zA-Z0-9_.-]+$", v):
            raise ValueError("Username can only contain letters, numbers, _, .,
        return v.lower()

    @field_validator("email")
    @classmethod
    def email_valid(cls, v: str) -> str:
        if not re.match(r"^[^@\s]+@[^@\s]+\.[^@\s]+$", v):
            raise ValueError("Invalid email address")
        return v.lower()

    @field_validator("password")
    @classmethod
    def password_strong(cls, v: str) -> str:
        issues = check_password_strength(v)
        if issues:
            raise ValueError("; ".join(issues))
        return v
```

FastAPI uses Pydantic automatically — invalid requests return a `422` with a clear error message before your code even runs.

55.4.1 Sanitising HTML output

If you render user input in HTML, you must escape it to prevent Cross-Site Scripting (XSS):

```
import html

user_input = '<script>alert("XSS")</script>'

# BAD --- renders the script
template = f"<p>Welcome, {user_input}!</p>"

# GOOD --- escapes HTML entities
safe = html.escape(user_input)
template = f"<p>Welcome, {safe}!</p>"
# -> <p>Welcome, &lt;script&gt;alert("XSS")&lt;/script&gt;!</p>
```

In Jinja2 templates (Flask/FastAPI), `{{ variable }}` auto-escapes by default. Only `{{ variable | safe }}` disables escaping — never use `| safe` on user input.

55.5 Mistake 5: Weak or No Authentication

55.5.1 Use strong secret keys

```
import secrets

# Generate a cryptographically secure secret key
secret_key = secrets.token_hex(32)     # 64-char hex string
# b3a2f1...  <- use this as your SECRET_KEY

# Generate a secure random token (for password reset, email verification)
token = secrets.token_urlsafe(32)      # URL-safe base64

# Secure comparison (prevents timing attacks)
if secrets.compare_digest(provided_token, stored_token):
    grant_access()
```

Never use `random` for security. `random` is predictable — it's for games and simulations, not tokens or keys. Always use `secrets`.

55.5.2 Secure JWT configuration

```
from datetime import datetime, timedelta
from jose import jwt, JWTError

SECRET_KEY = "use-a-long-random-value-from-secrets.token_hex(32)"
ALGORITHM  = "HS256"


def create_token(user_id: int, expires_in: timedelta = timedelta(hours=1)) -> str:
    payload = {
        "sub": str(user_id),
        "iat": datetime.utcnow(),
        "exp": datetime.utcnow() + expires_in,
        "type": "access",
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)


def decode_token(token: str) -> dict:
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
```

```
        if payload.get("type") != "access":
            raise ValueError("Wrong token type")
        return payload
    except JWTError as e:
        raise ValueError(f"Invalid token: {e}")
```

Set short expiry times for access tokens (15 minutes to 1 hour). Use refresh tokens for longer sessions.

55.6 Mistake 6: Path Traversal

A path traversal attack uses `../../../etc/passwd` in a filename to read files outside your intended directory:

```
import os
from pathlib import Path

UPLOAD_DIR = Path("/app/uploads")


# BAD --- user controls the path
def serve_file_bad(filename: str):
    path = UPLOAD_DIR / filename
    return open(path).read()
    # filename = "../../etc/passwd" -> reads /etc/passwd


# GOOD --- resolve and check the path
def serve_file_safe(filename: str):
    # Resolve to absolute path (follows symlinks, resolves ..)
    safe_path = (UPLOAD_DIR / filename).resolve()

    # Ensure it's still inside UPLOAD_DIR
    if not str(safe_path).startswith(str(UPLOAD_DIR.resolve())):
        raise PermissionError(f"Access denied: {filename!r}")

    return safe_path.read_text()


# Even better --- use Path.is_relative_to() (Python 3.9+)
def serve_file_modern(filename: str):
    upload_dir = UPLOAD_DIR.resolve()
    requested  = (upload_dir / filename).resolve()
```

```
    if not requested.is_relative_to(upload_dir):
        raise PermissionError(f"Access denied: {filename!r}")

    return requested.read_text()
```

55.7 Mistake 7: Insecure File Uploads

```
import magic   # pip install python-magic
from pathlib import Path
import hashlib

ALLOWED_TYPES    = {"image/jpeg", "image/png", "image/webp", "image/gif"}
MAX_FILE_SIZE    = 5 * 1024 * 1024   # 5 MB
UPLOAD_DIR       = Path("uploads")


def validate_and_save_upload(filename: str, data: bytes) -> str:
    # 1. Check file size
    if len(data) > MAX_FILE_SIZE:
        raise ValueError(f"File too large: {len(data)} bytes (max {MAX_FILE_SIZE})

    # 2. Check MIME type from file content, not extension
    # (extension can be faked: malware.exe renamed to image.jpg)
    mime_type = magic.from_buffer(data, mime=True)
    if mime_type not in ALLOWED_TYPES:
        raise ValueError(f"File type not allowed: {mime_type}")

    # 3. Sanitise filename --- strip path components, keep only safe chars
    import re
    safe_name = re.sub(r"[^\w\-.]", "_", Path(filename).name)

    # 4. Use a random name to prevent guessing
    random_prefix = hashlib.sha256(data).hexdigest()[:16]
    ext      = Path(safe_name).suffix.lower()
    new_name = f"{random_prefix}{ext}"

    # 5. Save to upload directory
    UPLOAD_DIR.mkdir(exist_ok=True)
    dest = UPLOAD_DIR / new_name
    dest.write_bytes(data)
```

```
    return new_name
```

55.8 Mistake 8: Dependency Vulnerabilities

Your code is only as secure as your dependencies. Libraries have vulnerabilities.

```
# pip audit --- check installed packages against vulnerability database
pip install pip-audit
pip-audit

# Output:
# Name         Version ID                  Fix Versions
# -------      ------- ------------------  ------------
# requests     2.27.0  GHSA-j8r2-6x86-q33q 2.31.0

# safety --- another vulnerability scanner
pip install safety
safety check

# Keep dependencies up to date
pip list --outdated
pip install --upgrade requests

# Pin versions in production to prevent surprise upgrades
# But review and update regularly
```

Set up automated dependency scanning in GitHub Actions:

```
# .github/workflows/security.yml
name: Security Scan

on: [push, pull_request]

jobs:
  audit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install pip-audit
      - run: pip-audit -r requirements.txt
```

55.9 Mistake 9: Verbose Error Messages in Production

```
# BAD --- leaks stack traces, file paths, and database info to users
@app.errorhandler(500)
def internal_error(e):
    return str(e), 500   # attacker sees your database schema, file paths, etc.


# GOOD --- log the full error, show a generic message to users
import logging
log = logging.getLogger(__name__)

@app.errorhandler(500)
def internal_error(e):
    log.exception("Internal server error")   # full traceback in your logs
    return {"error": "An internal error occurred"}, 500
```

In FastAPI:

```
@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    log.exception("Unhandled exception: %s %s", request.method, request.url)
    return JSONResponse(
        status_code=500,
        content={"detail": "Internal server error"},
        # Never include str(exc) or traceback in the response
    )
```

55.10 Mistake 10: HTTPS and Headers

Always serve your application over HTTPS. In production, your hosting platform (Render, Heroku, AWS) handles this. For self-hosted:

```
# certbot generates free Let's Encrypt certificates
sudo certbot --nginx -d yourdomain.com
```

Add security headers to every response (Flask example):

```
from flask import Flask
```

```
app = Flask(__name__)


@app.after_request
def add_security_headers(response):
    response.headers["X-Content-Type-Options"]   = "nosniff"
    response.headers["X-Frame-Options"]          = "DENY"
    response.headers["X-XSS-Protection"]         = "1; mode=block"
    response.headers["Strict-Transport-Security"] = "max-age=31536000; includeS
    response.headers["Referrer-Policy"]          = "strict-origin-when-cross-o
    response.headers["Content-Security-Policy"]  = "default-src 'self'"
    return response
```

In FastAPI, use the `secure` middleware:

```
pip install secure
```

```
import secure
from fastapi import FastAPI
from fastapi.middleware.trustedhost import TrustedHostMiddleware

app = FastAPI()
secure_headers = secure.Secure()

@app.middleware("http")
async def set_secure_headers(request, call_next):
    response = await call_next(request)
    secure_headers.framework.fastapi(response)
    return response
```

55.11 A Security Checklist for Every Python Project

```
Secrets
  ☐ No credentials hardcoded in source code
  ☐ .env in .gitignore
  ☐ Scanned repo with detect-secrets

Input Validation
  ☐ All user input validated with Pydantic or explicit checks
  ☐ File uploads: size, MIME type, and filename sanitised
  ☐ Path traversal protection on file serving
```

```
Authentication
  ☐ Passwords hashed with bcrypt (never MD5/SHA256 alone)
  ☐ Tokens generated with secrets.token_hex() or token_urlsafe()
  ☐ JWT access tokens have short expiry (<=1 hour)
  ☐ No verbose error messages exposing internals

Database
  ☐ Parameterised queries everywhere --- no string concatenation
  ☐ DB user has minimum required permissions
  ☐ DB not exposed to the public internet

Dependencies
  ☐ pip-audit passes clean
  ☐ Dependencies pinned and regularly reviewed
  ☐ Automated vulnerability scanning in CI

HTTP
  ☐ HTTPS enforced in production
  ☐ Security headers set (X-Frame-Options, HSTS, CSP)
  ☐ CORS configured to allow only trusted origins
```

55.12 What You Learned in This Chapter

- Secrets belong in environment variables, never in source code. Use `detect-secrets` to scan for accidental leaks.
- SQL injection is prevented entirely by parameterised queries — `?` placeholders, never string concatenation.
- Passwords must be hashed with bcrypt or Argon2 (via `passlib`). Never MD5, SHA-256 alone, or plain text.
- Validate all input with Pydantic. Escape HTML output with `html.escape()` to prevent XSS.
- Use `secrets.token_hex()` and `secrets.token_urlsafe()` for tokens. Never `random`. Use `secrets.compare_digest()` for constant-time comparison.
- Prevent path traversal by resolving the full path and checking it stays inside the expected directory.
- Validate file uploads by checking MIME type from file content (not extension), size, and using random filenames.
- Run `pip-audit` in CI to catch known vulnerabilities in your dependencies.
- Log full errors server-side. Return only generic messages to users.
- Serve over HTTPS. Add security headers to every response.

55.13 What's Next?

Chapter 56 covers **Python Internals — How Python Really Works**: CPython's execution model, bytecode, the GIL, memory management, and the import system. Understanding the engine makes you a better driver.

Chapter 56

Chapter 56: Python Internals — How Python Really Works

You've written Python for 55 chapters. Now let's look under the hood.

Understanding how Python executes your code makes you a better developer. You'll write faster code, understand error messages more deeply, know why the GIL exists, and stop being surprised by Python's quirks. This chapter won't make you a CPython contributor — but it will make the language feel transparent instead of magical.

56.1 CPython — The Reference Implementation

Python the language is a specification. CPython is the most popular implementation of that specification — it's what you download from python.org. Other implementations exist:

Implementation	Language	Use case
CPython	C	The standard; most compatible
PyPy	Python + RPython	Faster (JIT compiler); ~5x speedup for loops

Implementation	Language	Use case
Jython	Java	Runs on JVM; integrates with Java libraries
MicroPython	C	Runs on microcontrollers (Raspberry Pi Pico)
GraalPy	Java	Oracle's high-performance Python on GraalVM

Everything in this chapter refers to CPython unless stated otherwise.

56.2 How Python Runs Your Code

When you type `python hello.py`, four things happen:

```
1. Lexing       -> source code   -> tokens
2. Parsing      -> tokens        -> Abstract Syntax Tree (AST)
3. Compilation  -> AST           -> bytecode
4. Execution    -> bytecode      -> results (CPython VM)
```

56.2.1 Step 1 — Lexing (Tokenisation)

The lexer breaks your source into tokens:

```
import tokenize, io

source = "x = 1 + 2"
tokens = list(tokenize.generate_tokens(io.StringIO(source).readline))
for tok in tokens:
    print(tok)
```

Output:

```
TokenInfo(type=1  (NAME),   string='x',   ...)
TokenInfo(type=54 (OP),     string='=',   ...)
TokenInfo(type=2  (NUMBER), string='1',   ...)
TokenInfo(type=54 (OP),     string='+',   ...)
TokenInfo(type=2  (NUMBER), string='2',   ...)
```

56.2.2 Step 2 — Parsing (AST)

The parser converts tokens into an Abstract Syntax Tree:

```
import ast

source = "x = 1 + 2"
tree   = ast.parse(source)
print(ast.dump(tree, indent=2))
```

Output:

```
Module(
  body=[
    Assign(
      targets=[Name(id='x')],
      value=BinOp(
        left=Constant(value=1),
        op=Add(),
        right=Constant(value=2)))],
  type_ignores=[])
```

The AST is a tree of node objects. Each node represents a syntactic construct: assignments, function calls, loops, conditionals.

56.2.3 Step 3 — Compilation (Bytecode)

The compiler converts the AST into **bytecode** — a sequence of simple instructions for the Python virtual machine. Use `dis` to see it:

```
import dis

def add(a, b):
    return a + b

dis.dis(add)
```

Output:

```
  2           0 RESUME           0

  3           2 LOAD_FAST        0 (a)
              4 LOAD_FAST        1 (b)
              6 BINARY_OP      0 (+)
             10 RETURN_VALUE
```

Each line is one bytecode instruction. The VM reads these instructions one by one and executes them.

Bytecode is cached in `__pycache__/` as `.pyc` files so Python doesn't recompile unchanged modules:

```
mypackage/
└── __pycache__/
    ├── module.cpython-312.pyc
    └── utils.cpython-312.pyc
```

56.2.4 Step 4 — Execution (CPython VM)

The CPython virtual machine is a **stack-based interpreter**. It maintains a call stack. Each function call pushes a new **frame** onto the stack. Each frame has:

- The bytecode being executed
- A local variables dictionary
- A reference to the global namespace
- A reference to the enclosing scope
- A data stack for intermediate values

```
import sys

def outer():
    def inner():
        frame = sys._getframe()
        print(f"Function:  {frame.f_code.co_name}")
        print(f"File:      {frame.f_code.co_filename}")
        print(f"Line:      {frame.f_lineno}")
        print(f"Locals:    {frame.f_locals}")
        print(f"Caller:    {frame.f_back.f_code.co_name}")
    inner()

outer()
```

56.3 Code Objects — The Compiled Unit

Every function, class, and module has a **code object** (`__code__`):

```
def greet(name: str, times: int = 1) -> str:
    message = f"Hello, {name}!"
    return message * times

code = greet.__code__
print(code.co_name)          # greet
```

```
print(code.co_varnames)      # ('name', 'times', 'message') --- local variable name
print(code.co_argcount)      # 2
print(code.co_consts)        # (None, 'Hello, ', '!') --- constants
print(code.co_filename)      # path to the .py file
print(code.co_firstlineno)   # 1
```

Code objects are immutable. They're created at compile time and never change at runtime.

56.4 The GIL — Global Interpreter Lock

The GIL is a mutex (lock) inside CPython that allows only one thread to execute Python bytecode at a time.

This means: - **Multithreaded Python cannot run Python code in parallel on multiple CPU cores** - **But it can do I/O in parallel** (the GIL is released during I/O operations)

56.4.1 Why does the GIL exist?

CPython's memory management uses **reference counting** — every object has a counter that tracks how many references point to it:

```
import sys

a = [1, 2, 3]
print(sys.getrefcount(a))    # 2 (a + the argument to getrefcount)

b = a
print(sys.getrefcount(a))    # 3

del b
print(sys.getrefcount(a))    # 2
```

When the reference count drops to 0, the object is immediately freed. This is fast and simple — but reference counts are not thread-safe. Without the GIL, two threads could simultaneously decrement the same counter, corrupting memory.

The GIL is a single lock that protects all Python objects instead of thousands of individual locks.

56.4.2 What the GIL means for you

```
import threading

# I/O-bound: GIL is released, threads run in parallel [x]
# (database queries, HTTP requests, file reads)
def fetch_url(url):
    import urllib.request
    return urllib.request.urlopen(url).read()

threads = [threading.Thread(target=fetch_url, args=(url,)) for url in urls]
# These run in true parallel --- the GIL is released during network I/O

# CPU-bound: GIL held, threads take turns [-]
# (number crunching, image processing, sorting)
def count_primes(n):
    return sum(1 for i in range(2, n) if all(i % j for j in range(2, i)))

# Use multiprocessing for CPU-bound work instead
```

56.4.3 Python 3.13+ — The No-GIL Build

Python 3.13 introduced an experimental "free-threaded" build that removes the GIL:

```
# Install the free-threaded build
python3.13t   # the 't' suffix indicates free-threaded

# Check if GIL is active
import sys
print(sys._is_gil_enabled())   # False in free-threaded build
```

The free-threaded build is experimental in 3.13 and will stabilise in future versions. For now, most production code should still use multiprocessing for CPU-bound parallelism.

56.5 Memory Management

56.5.1 Reference Counting

Every Python object has a reference count. When it hits 0, the memory is freed immediately:

```
import ctypes

def ref_count(obj):
    """Return the reference count of an object."""
    return sys.getrefcount(obj) - 1   # subtract the getrefcount call itself

x = [1, 2, 3]
print(ref_count(x))   # 1

y = x
print(ref_count(x))   # 2

del y
print(ref_count(x))   # 1
# When x goes out of scope, count hits 0 and memory is freed
```

56.5.2 Cyclic Garbage Collector

Reference counting can't handle cycles:

```
import gc

class Node:
    def __init__(self, value):
        self.value = value
        self.next  = None

a = Node(1)
b = Node(2)
a.next = b
b.next = a   # cycle: a -> b -> a

del a
del b
# Reference counts are now 1 for each --- never reach 0
# The cyclic GC detects and cleans this up
```

Python runs a cyclic garbage collector periodically. You can trigger it manually:

```
import gc

gc.collect()            # run all three generations
gc.collect(0)           # collect youngest generation only
print(gc.get_count())   # (gen0, gen1, gen2) object counts
```

56.5.3 Object Interning

Python reuses certain objects to save memory. Small integers (-5 to 256) and many short strings are **interned** — only one copy exists:

```
# Small integers: same object
a = 100
b = 100
print(a is b)    # True --- same object

# Large integers: different objects
a = 1000
b = 1000
print(a is b)    # False (implementation-defined, often False)

# Short strings: often interned
a = "hello"
b = "hello"
print(a is b)    # True (interned)

# Strings with spaces: NOT interned
a = "hello world"
b = "hello world"
print(a is b)    # False
```

This is why `is` should only be used for `None`, `True`, and `False`. Use `==` for all other equality checks.

56.5.4 Memory Profiling

```
import tracemalloc

tracemalloc.start()

# Your code here
data = [i for i in range(100_000)]

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics("lineno")

print("Top 3 memory consumers:")
for stat in top_stats[:3]:
    print(stat)
```

56.6 The Import System

When you write `import mymodule`, Python:

1. Checks `sys.modules` — if already imported, returns the cached module
2. Finds the module file using `sys.path`
3. Loads and compiles the `.py` file
4. Executes the module code in a new namespace
5. Caches the result in `sys.modules`

```
import sys

# See all currently imported modules
print(list(sys.modules.keys())[:10])

# See where Python looks for modules
print(sys.path)

# Reload a module (rarely needed)
import importlib
importlib.reload(mymodule)
```

56.6.1 `__all__` — Controlling What's Exported

```
# mymodule.py
__all__ = ["PublicClass", "public_function"]

class PublicClass:
    pass

class _PrivateClass:    # won't be imported with "from mymodule import *"
    pass

def public_function():
    pass

def _private_function():
    pass
```

56.6.2 `if __name__ == "__main__"`

When Python imports a module, it sets `__name__` to the module's name. When you run a file directly, `__name__` is `"__main__"`:

```
# mymodule.py
def useful_function():
```

```
    return 42

# This block only runs when the file is executed directly
# NOT when it's imported
if __name__ == "__main__":
    result = useful_function()
    print(f"Result: {result}")
```

This pattern lets a file be both a reusable module and a runnable script.

56.7 Bytecode Deep Dive

Let's trace through a slightly more complex function:

```
import dis

def fibonacci(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(n - 1):
        a, b = b, a + b
    return b

dis.dis(fibonacci)
```

Output (Python 3.12):

```
  2           RESUME           0

  3           LOAD_FAST        0 (n)
              LOAD_CONST       1 (1)
              COMPARE_OP       1 (<=)
              POP_JUMP_IF_FALSE ...

  4           LOAD_FAST        0 (n)
              RETURN_VALUE

  5           LOAD_CONST       2 (0)
              LOAD_CONST       1 (1)
              UNPACK_SEQUENCE  2
              STORE_FAST       1 (a)
              STORE_FAST       2 (b)
...
```

Key instructions: - `LOAD_FAST` — load a local variable onto the stack - `LOAD_CONST` — load a constant onto the stack - `COMPARE_OP` — compare top two stack items - `POP_JUMP_IF_FALSE` — jump if top of stack is falsy - `STORE_FAST` — pop stack and store in a local variable - `CALL` — call a function - `RETURN_VALUE` — return top of stack to caller

Understanding bytecode explains Python performance characteristics. `LOAD_FAST` (local variable) is faster than `LOAD_GLOBAL` (global variable) — that's why moving frequently accessed globals into local variables inside tight loops speeds things up.

56.8 `__slots__` — Memory-Efficient Classes

By default, Python stores instance attributes in a `__dict__` (a hash map):

```
class Regular:
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = Regular(1, 2)
print(obj.__dict__)    # {'x': 1, 'y': 2}
```

Each `__dict__` uses ~230 bytes. For classes with millions of instances, this adds up.

`__slots__` replaces the dict with fixed-size per-attribute descriptors:

```
class Slotted:
    __slots__ = ("x", "y")

    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = Slotted(1, 2)
# obj.__dict__   <- AttributeError: no __dict__
print(obj.x)     # 1
```

```
import sys

print(sys.getsizeof(Regular(1, 2)))    # ~48 bytes + ~230 for __dict__
print(sys.getsizeof(Slotted(1, 2)))    # ~56 bytes --- no __dict__ overhead
```

Use `__slots__` when you create millions of instances of a class and memory is a concern.

56.9 `__getattr__` vs `__getattribute__`

These two look similar but behave very differently:

```
class Demo:
    def __init__(self):
        self.exists = "I exist"

    def __getattr__(self, name: str):
        # Called ONLY when normal attribute lookup fails
        print(f"__getattr__: {name} not found")
        return f"default_{name}"

    def __getattribute__(self, name: str):
        # Called for EVERY attribute access --- be careful
        print(f"__getattribute__: accessing {name}")
        return super().__getattribute__(name)


d = Demo()
print(d.exists)         # __getattribute__ called -> "I exist"
print(d.missing)        # __getattribute__ called -> not found -> __getattr__ ca
```

`__getattr__` — safe to override. Use for proxy objects, lazy loading, dynamic attributes.

`__getattribute__` — called for every attribute access. Override it only if you absolutely must — it's easy to create infinite recursion. Always call `super().__getattribute__(name)`.

56.10 AST Manipulation — Code That Reads Code

Python's AST can be inspected and transformed:

```
import ast


def find_function_names(source: str) -> list[str]:
    """Extract all function names defined in source code."""
    tree  = ast.parse(source)
```

```
    names = []
    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            names.append(node.name)
    return names


source = """
def greet(name):
    return f"Hello, {name}"

def farewell(name):
    return f"Goodbye, {name}"

class MyClass:
    def method(self):
        pass
"""

print(find_function_names(source))
# ['greet', 'farewell', 'method']
```

```
def count_operations(source: str) -> dict[str, int]:
    """Count each type of binary operation in source."""
    tree  = ast.parse(source)
    ops   = {}
    for node in ast.walk(tree):
        if isinstance(node, ast.BinOp):
            op_name = type(node.op).__name__
            ops[op_name] = ops.get(op_name, 0) + 1
    return ops


source = "result = (a + b) * (c - d) / (e + f)"
print(count_operations(source))   # {'Add': 2, 'Mult': 1, 'Sub': 1, 'Div': 1}
```

AST manipulation is used by linters (`ruff`, `flake8`), formatters (`black`), type checkers (`mypy`), and code analysis tools.

56.11 What You Learned in This Chapter

- CPython is the reference implementation. PyPy is faster via JIT. MicroPython runs on microcontrollers.

- Python executes your code in four steps: **Lex** (source -> tokens) -> **Parse** (tokens -> AST) -> **Compile** (AST -> bytecode) -> **Execute** (bytecode -> results via CPython VM).
- Use `tokenize` to see tokens, `ast.parse()` + `ast.dump()` to see the AST, `dis.dis()` to see bytecode.
- Bytecode is cached in `__pycache__/*.pyc` files. The VM is stack-based.
- Each function call creates a **frame** with local variables, the code object, and a data stack. `sys._getframe()` inspects the current frame.
- Every object has a **reference count**. When it hits 0, the object is freed immediately. The cyclic GC handles reference cycles.
- The **GIL** allows only one thread to execute Python bytecode at a time. I/O releases the GIL, so threads work well for I/O-bound work. Use `multiprocessing` for CPU-bound parallelism. Python 3.13 introduced an experimental no-GIL build.
- Small integers (-5 to 256) and short strings are **interned** — shared single objects. Use `==` for equality, `is` only for `None`/`True`/`False`.
- `__slots__` replaces `__dict__` with fixed descriptors, saving 200+ bytes per instance.
- `__getattr__` fires only when normal lookup fails (safe to override). `__getattribute__` fires on every access (override with extreme care).
- The `ast` module lets you parse, inspect, and transform Python source code programmatically.

56.12 What's Next?

Chapter 57 covers **The Python Ecosystem Map** — a tour of the major libraries and frameworks in every domain: web, data science, machine learning, automation, DevOps, finance, games, and embedded systems. Think of it as a roadmap for wherever you want to go next.

Chapter 57

Chapter 57: The Python Ecosystem Map

You've learned the language. Now it's time to see the territory.

Python's ecosystem is enormous. There's a mature, well-maintained library for almost everything — web servers, machine learning, game development, financial modelling, microcontrollers, DevOps automation, computer vision, and more.

This chapter is your map. You won't master all of these. Nobody does. But knowing what exists — and roughly what each tool does — means you can pick the right one when a project demands it.

57.1 Web Development

57.1.1 Frameworks

Library	Use case	Why it's popular
FastAPI	REST APIs, async	Auto docs, Pydantic validation, modern
Flask	APIs + server-rendered web	Simple, flexible, huge ecosystem
Django	Full-stack web apps	Batteries included (ORM, admin, auth)
Starlette	ASGI foundation	Lightweight; FastAPI is built on it

Library	Use case	Why it's popular
Litestar	REST APIs	Type-first, opinionated FastAPI alternative
Tornado	Long-polling, WebSockets	Mature async framework

```
# Django --- full-featured (ORM, admin, auth built in)
# pip install django
django-admin startproject mysite
python manage.py startapp blog

# Starlette --- minimal ASGI
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route
```

57.1.2 HTTP Clients

Library	Use case
requests	Simple synchronous HTTP — the standard
httpx	Like requests but async-capable
aiohttp	Async HTTP client + server
urllib3	Low-level; used by requests internally

```
# requests --- the classic
import requests
r = requests.get("https://api.github.com/users/python")
data = r.json()

# httpx --- modern, supports async
import httpx
async with httpx.AsyncClient() as client:
    r = await client.get("https://api.github.com")
```

57.1.3 Web Scraping

Library	Use case
BeautifulSoup4	Parse HTML/XML
Scrapy	Full scraping framework (crawls, pipelines)
Playwright	Control a real browser (JS-rendered pages)
Selenium	Classic browser automation

Library	Use case
mechanize	Stateful browser — fill forms, follow links

57.2 Data Science and Analytics

57.2.1 The Core Stack

Library	Use case
NumPy	N-dimensional arrays, vectorised math
pandas	DataFrames, data cleaning, tabular analysis
matplotlib	Publication-quality plots
seaborn	Statistical visualisation, beautiful defaults
plotly	Interactive charts (web-ready)
scipy	Scientific algorithms (stats, signal, optimisation)

```
# The standard data science import block
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

57.2.2 Databases from Python

Library	Use case
SQLAlchemy	ORM + SQL toolkit (any SQL DB)
sqlite3	Built-in; zero-config SQLite
psycopg2 / psycopg3	PostgreSQL driver
pymysql	MySQL driver
motor	Async MongoDB driver
redis-py	Redis client
pymongo	MongoDB client

57.3 Machine Learning and AI

57.3.1 Classical ML

Library	Use case
scikit-learn	Classification, regression, clustering, pipelines
XGBoost	Gradient boosting — wins Kaggle competitions
LightGBM	Faster gradient boosting (Microsoft)
CatBoost	Gradient boosting for categorical data (Yandex)
statsmodels	Statistical models, hypothesis testing

```
from sklearn.ensemble import GradientBoostingClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier

# Often the fastest path to a strong model:
model = XGBClassifier(n_estimators=500, learning_rate=0.05)
model.fit(X_train, y_train)
```

57.3.2 Deep Learning

Library	Use case
PyTorch	Research + production DL (dominant in research)
TensorFlow / Keras	Production DL, mobile deployment
JAX	Accelerated NumPy with autograd (Google)
Hugging Face Transformers	Pre-trained models (BERT, GPT, Llama)
fastai	High-level PyTorch API

```
import torch
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

# Hugging Face --- use a pre-trained model in 4 lines
from transformers import pipeline
```

```
classifier = pipeline("sentiment-analysis")
result = classifier("Python is an amazing language!")
print(result)   # [{'label': 'POSITIVE', 'score': 0.9998}]
```

57.3.3 AI / LLM Tooling

Library	Use case
openai	Official OpenAI API client
anthropic	Official Anthropic (Claude) client
LangChain	LLM application framework (chains, agents)
LlamaIndex	RAG (Retrieval-Augmented Generation)
sentence-transformers	Embed text as vectors
chromadb	Local vector database

```
from openai import OpenAI

client = OpenAI()  # uses OPENAI_API_KEY env var
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "Explain Python's GIL in one sentence."}]
)
print(response.choices[0].message.content)
```

57.3.4 Computer Vision

Library	Use case
OpenCV	Image processing, video, real-time CV
Pillow	Image manipulation
scikit-image	Scientific image processing
torchvision	CV datasets and models for PyTorch
ultralytics	YOLOv8 — state-of-the-art object detection

```
import cv2

cap = cv2.VideoCapture(0)   # open webcam
while True:
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    cv2.imshow("Webcam", gray)
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break
cap.release()
```

57.4 Automation and Scripting

Library	Use case
pathlib	Built-in file system operations
shutil	Built-in file copying/moving
schedule	Run Python functions on a schedule
APScheduler	Advanced scheduler (cron, interval, date)
watchdog	Watch the file system for changes
Playwright	Browser automation
pyautogui	Mouse/keyboard automation
pynput	Monitor/control keyboard and mouse

```
# APScheduler --- more powerful than schedule
from apscheduler.schedulers.blocking import BlockingScheduler

scheduler = BlockingScheduler()

@scheduler.scheduled_job("cron", hour=8, minute=0)
def morning_report():
    print("Sending morning report...")

scheduler.start()
```

57.4.1 Office Automation

Library	Use case
openpyxl	Read/write Excel (.xlsx)
xlrd	Read legacy Excel (.xls)
python-docx	Read/write Word documents
python-pptx	Read/write PowerPoint
pypdf	Read, merge, split PDFs
reportlab	Generate PDFs from scratch
weasyprint	HTML -> PDF

57.5 Command-Line Tools

Library	Use case
argparse	Built-in CLI argument parsing
click	Decorator-based CLI (Flask author)
typer	Type-hint-based CLI (FastAPI author)
rich	Beautiful terminal output (tables, progress, colour)
textual	Full TUI (Terminal UI) apps
tqdm	Progress bars

```
# Typer --- CLI with type hints
import typer

app = typer.Typer()

@app.command()
def greet(name: str, count: int = 1, upper: bool = False):
    for _ in range(count):
        msg = f"Hello, {name}!"
        typer.echo(msg.upper() if upper else msg)

if __name__ == "__main__":
    app()
```

```
$ python hello.py Alice --count 3 --upper
HELLO, ALICE!
HELLO, ALICE!
HELLO, ALICE!
```

```
# Rich --- beautiful output
from rich.console import Console
from rich.table   import Table

console = Console()
table   = Table(title="Python Libraries")
table.add_column("Name",    style="cyan",  no_wrap=True)
table.add_column("Purpose", style="green")
table.add_row("FastAPI",   "Web APIs")
table.add_row("pandas",    "Data analysis")
table.add_row("PyTorch",   "Deep learning")
console.print(table)
```

57.6 Testing and Code Quality

Library	Use case
pytest	Test runner — the standard
hypothesis	Property-based testing (generates test cases)
factory_boy	Test fixtures / data factories
faker	Generate fake data (names, emails, etc.)
ruff	Linter + formatter (replaces flake8 + black + isort)
mypy	Static type checker
pyright	Microsoft's type checker (used in Pylance)
coverage.py	Code coverage measurement
pre-commit	Run checks before every git commit

```python
# Hypothesis --- property-based testing
from hypothesis import given, strategies as st

@given(st.lists(st.integers()))
def test_sorted_is_sorted(lst):
    result = sorted(lst)
    assert result == sorted(result)   # tautological but shows the pattern
    assert len(result) == len(lst)
    assert set(result) == set(lst)
```

57.7 DevOps and Infrastructure

Library	Use case
ansible	IT automation, server configuration
fabric	SSH automation / remote task runner
boto3	AWS SDK for Python
google-cloud	Google Cloud SDK
azure-sdk	Microsoft Azure SDK
docker	Docker SDK — manage containers from Python
kubernetes	Kubernetes Python client
paramiko	SSH client

```python
import boto3

# Upload a file to S3
s3 = boto3.client("s3")
s3.upload_file("report.pdf", "my-bucket", "reports/report.pdf")

# List EC2 instances
```

```
ec2 = boto3.resource("ec2")
for instance in ec2.instances.all():
    print(instance.id, instance.state["Name"])
```

57.8 Finance and Quantitative Analysis

Library	Use case
yfinance	Download stock data from Yahoo Finance
pandas-datareader	Financial data from multiple sources
ta	Technical analysis indicators
backtrader	Backtesting trading strategies
zipline	Algorithmic trading backtester
pyfolio	Portfolio performance analysis
quantlib	Quantitative finance library

```
import yfinance as yf
import matplotlib.pyplot as plt

# Download 1 year of Apple stock data
aapl = yf.download("AAPL", period="1y")
aapl["Close"].plot(figsize=(10, 4), title="AAPL Closing Price")
plt.tight_layout()
plt.show()
```

57.9 Game Development

Library	Use case
pygame	2D games — the classic choice
arcade	Modern 2D games, better API than pygame
Panda3D	3D game engine
pyglet	OpenGL-based windowing and multimedia
pygame-ce	Community fork of pygame, actively maintained

```
import pygame

pygame.init()
screen  = pygame.display.set_mode((800, 600))
```

```
clock   = pygame.time.Clock()
running = True

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    screen.fill((30, 30, 30))
    pygame.draw.circle(screen, (0, 120, 255), (400, 300), 50)
    pygame.display.flip()
    clock.tick(60)

pygame.quit()
```

57.10 Embedded and IoT

Library	Use case
MicroPython	Python on microcontrollers (ESP32, Pico)
CircuitPython	Adafruit's beginner-friendly MicroPython fork
RPi.GPIO	Raspberry Pi GPIO control
gpiozero	Higher-level Raspberry Pi GPIO
pyserial	Serial port communication
paho-mqtt	MQTT protocol (IoT messaging)

```
# MicroPython on Raspberry Pi Pico
from machine import Pin
import time

led = Pin(25, Pin.OUT)    # onboard LED

while True:
    led.toggle()
    time.sleep(0.5)
```

57.11 Scientific and Academic

Library	Use case
sympy	Symbolic mathematics (algebra, calculus)
scipy	Scientific algorithms
astropy	Astronomy tools
biopython	Bioinformatics
networkx	Graph theory and network analysis
nltk	Natural language processing (classic)
spacy	Industrial-strength NLP

```
import sympy as sp

x = sp.Symbol("x")

# Symbolic differentiation
expr = sp.sin(x**2) + sp.cos(x)
print(sp.diff(expr, x))      # 2*x*cos(x**2) - sin(x)

# Solve an equation
print(sp.solve(x**2 - 4, x))  # [-2, 2]

# Integrate
print(sp.integrate(x**2, x))   # x**3/3
```

57.12 Async and Concurrency

Library	Use case
asyncio	Built-in async event loop
aiohttp	Async HTTP client + server
aiofiles	Async file I/O
aiomysql / asyncpg	Async database drivers
celery	Distributed task queue
rq	Simple Redis-backed job queue
dramatiq	Message processing middleware

```
# Celery --- run tasks in the background
from celery import Celery

app = Celery("tasks", broker="redis://localhost:6379/0")

@app.task
```

```python
def send_email(to: str, subject: str, body: str):
    # This runs in a separate worker process
    ...

# In your web app:
send_email.delay("alice@example.com", "Welcome!", "Hello!")
# Returns immediately --- email sent asynchronously
```

57.13 How to Choose a Library

When you need a library for something new:

1. **Search PyPI**: `pip search` is deprecated; use pypi.org directly
2. **Check stars and activity**: GitHub stars + recent commits = active project
3. **Check downloads**: pypistats.org shows weekly download counts
4. **Read the README**: Is it well-documented? Does it have examples?
5. **Check issues**: Are bugs being fixed? Is there a responsive maintainer?
6. **Try it in a sandbox**: `pip install` in a fresh venv and experiment

Signs a library is safe to depend on: - 1000+ GitHub stars (for non-niche libraries) - Updated in the last 6-12 months - Comprehensive test suite - Semantic versioning - Published to PyPI (not just GitHub) - Used by well-known projects

---

57.14 The 20 Libraries Every Python Developer Should Know

If you master nothing else from this ecosystem map, master these:

#	Library	Why
1	**requests**	HTTP in 2 lines
2	**pandas**	Data manipulation
3	**NumPy**	Numerical computing
4	**matplotlib**	Visualisation
5	**SQLAlchemy**	Databases
6	**FastAPI**	Modern APIs
7	**Flask**	Simple web apps
8	**pytest**	Testing
9	**pydantic**	Data validation

#	Library	Why
10	**ruff**	Linting + formatting
11	**mypy**	Type checking
12	**Pillow**	Image processing
13	**rich**	Beautiful terminal output
14	**click** / **typer**	CLI tools
15	**boto3**	AWS (if you use cloud)
16	**scikit-learn**	Machine learning
17	**PyTorch**	Deep learning
18	**httpx**	Async HTTP
19	**aiohttp**	Async HTTP server/client
20	**celery**	Background jobs

57.15 What You Learned in This Chapter

- Python's ecosystem has mature, well-maintained libraries for nearly every domain.
- **Web:** FastAPI (modern APIs), Flask (simple web), Django (full-stack), requests/httpx (HTTP clients), Playwright (browser automation).
- **Data science:** NumPy + pandas + matplotlib + seaborn + scipy form the core stack.
- **ML/AI:** scikit-learn for classical ML; XGBoost/LightGBM for boosting; PyTorch/TensorFlow for deep learning; Hugging Face for pre-trained models; OpenAI/Anthropic for LLMs.
- **Automation:** pathlib + shutil (files), schedule/APScheduler (scheduling), openpyxl/python-docx/pypdf (office), pyautogui (GUI automation).
- **CLI:** argparse (built-in), click, typer (type-hint-based), rich (beautiful output), tqdm (progress bars).
- **DevOps:** boto3 (AWS), fabric (SSH), docker (containers), ansible (infrastructure).
- **Finance:** yfinance (stock data), backtrader (backtesting), quantlib (quant finance).
- **Games:** pygame/arcade (2D), Panda3D (3D).
- **Embedded:** MicroPython/CircuitPython (microcontrollers), gpiozero (Raspberry Pi).
- When evaluating a library: check stars, recent commits, downloads, documentation, and whether it's used by known projects.

57.16 What's Next?

Chapter 58 covers **Building Your Portfolio and Getting Hired** — what Python projects to build, how to present them on GitHub, how to write a Python-focused resume, and how to answer the most common Python interview questions.

Chapter 58

Chapter 58: Building Your Portfolio and Getting Hired

You've spent 57 chapters learning Python. Now you need to show the world what you can do.

The truth is this: employers don't care what courses you took. They care what you've built. A portfolio of three strong, well-documented projects will open more doors than a dozen certifications.

This chapter tells you exactly what to build, how to present it, how to write a Python-focused resume, and how to answer the most common interview questions.

58.1 What Makes a Python Portfolio Stand Out

Most beginner portfolios look the same: a to-do list, a calculator, a weather app. Hiring managers have seen them a thousand times.

A strong portfolio is different in three ways:

1. **Solves a real problem** — even a small, personal one
2. **Shows the full stack** — not just the code, but tests, docs, CI, clean structure

3. **Is easy to run** — a hiring manager should be able to clone it and run it in under two minutes

---

58.2 Five Project Ideas (With Difficulty Levels)

58.2.1 1. Personal Finance Tracker API [*][*]

Build a REST API with FastAPI that tracks income, expenses, and budgets. Store data in SQLite with SQLAlchemy. Add JWT authentication. Write a full pytest test suite.

Technologies: FastAPI, SQLAlchemy, Pydantic, JWT, pytest
What it shows: API design, database ORM, auth, testing

```
GET  /transactions          <- with filtering by date/category
POST /transactions
GET  /summary/monthly       <- grouped by category
GET  /summary/trends        <- monthly spending over time
```

58.2.2 2. Web Scraper + Data Dashboard [*][*][*]

Scrape a real website (job listings, property prices, sports stats), clean the data with pandas, store it in SQLite, and build a dashboard with Streamlit or Plotly Dash.

Technologies: requests, BeautifulSoup, pandas, SQLite, Streamlit
What it shows: data pipeline skills, visualisation, automation

```
# Streamlit dashboard in ~50 lines
import streamlit as st
import pandas as pd

st.title("Job Market Tracker")
df = pd.read_sql("SELECT * FROM jobs", conn)
st.metric("Total Jobs", len(df))
st.bar_chart(df.groupby("category").size())
st.dataframe(df.sort_values("date_posted", ascending=False).head(20))
```

58.2.3 3. CLI Tool for Developer Productivity [*][*]

Build a command-line tool that solves a real developer problem: a note-taking CLI, a git log formatter, a `.env` file manager, a snippet manager. Publish it to PyPI.

Technologies: typer, rich, pathlib, sqlite3, setuptools
What it shows: CLI design, packaging, real-world utility

This is highly impressive because it's a tool other developers might actually use. A PyPI package proves you can ship.

58.2.4 4. Machine Learning Pipeline [*][*][*]

Pick a public dataset (Kaggle, UCI, or scrape your own), build a complete ML pipeline: clean data, engineer features, train multiple models, evaluate, and serve predictions via a FastAPI endpoint.

Technologies: pandas, scikit-learn, FastAPI, Docker
What it shows: end-to-end ML, model serving, engineering skills

```
# Serve your model as an API
from fastapi import FastAPI
from pydantic import BaseModel
import joblib

app   = FastAPI()
model = joblib.load("model.pkl")

class PredictionRequest(BaseModel):
    features: list[float]

@app.post("/predict")
def predict(req: PredictionRequest):
    prediction = model.predict([req.features])
    return {"prediction": float(prediction[0])}
```

58.2.5 5. Full-Stack Web App [*][*][*][*]

Build something you would actually use: a recipe manager, a reading tracker, a workout log, a budget app with charts. FastAPI or Django backend + a simple HTML/CSS frontend or a React frontend.

Technologies: FastAPI/Django, SQLAlchemy, JWT, HTML/CSS or React
What it shows: full product thinking, frontend + backend integration, deployment

Deploy it live on Render or Railway so it has a real URL.

58.3 How to Document Your Projects on GitHub

A great GitHub project has all of these:

58.3.1 1. A compelling README

```
# Finance Tracker API

> A personal finance API with automatic categorisation,
> monthly summaries, and trend analysis.

[![Tests](https://github.com/you/finance-api/actions/workflows/ci.yml/badge.svg
[![Coverage](https://codecov.io/gh/you/finance-api/badge.svg)]()

## Live Demo

API: https://finance-api.onrender.com/docs

## Features

- [lock] JWT authentication
- [chart] Monthly spending summaries
- [tag] Automatic category detection
- [trend] Trend analysis over time
- [x] 95% test coverage

## Quick Start

```bash
git clone https://github.com/you/finance-api
cd finance-api
cp .env.example .env
pip install -e ".[dev]"
pytest          # all tests pass
uvicorn app.main:app --reload
```

58.4 Architecture

```
src/
├── api/          <- FastAPI routes
├── services/     <- business logic
├── models/       <- SQLAlchemy ORM
```
```

```
├── schemas/      <- Pydantic schemas
└── tests/        <- 50+ tests
```

58.5 Tech Stack

Python 3.12 · FastAPI · SQLAlchemy · Pydantic · pytest · Docker

```
### 2. Working CI badge

Your README should show a green CI badge. It proves the code actually works -
-- not just on your machine.

### 3. Tests with coverage

Run `pytest --cov` and include the output or a Codecov badge in the README. Even 70% cov

### 4. Live deployment URL

If it's a web app or API, deploy it. Free tiers on Render, Railway, and Fly.io are enoug

### 5. Clean git history

Use conventional commits. Hiring managers sometimes look at the commit log. Small, log

---

## Writing a Python-Focused Resume

Your resume has one job: get you an interview. Here's what works:

### Lead with what you can do
```

Python Developer | 2 years | FastAPI · SQLAlchemy · pandas · pytest

```
Not "Python enthusiast" or "aspiring developer." State what you are.

### Show numbers where you can

- "Built a REST API serving 500 requests/day"
- "Reduced data processing time by 60% with vectorised NumPy operations"
- "Maintained 94% test coverage across 1,200 lines of production code"

### List projects, not tutorials
```

Finance Tracker API github.com/you/finance-api FastAPI · SQLAlchemy · JWT · pytest · Docker render.com/finance-api - REST API with JWT auth, 50+ test cases, deployed on Render - Implemented automatic category detection using keyword matching - Achieved 95% code coverage with pytest-cov

```
### Skills section --- be specific
```

Languages: Python 3.11/3.12 Web: FastAPI, Flask, SQLAlchemy, Pydantic, JWT Data: pandas, NumPy, matplotlib, seaborn, scikit-learn Tools: pytest, ruff, mypy, Docker, Git, GitHub Actions Databases: PostgreSQL, SQLite, Redis Cloud: AWS (S3, Lambda), Render

```
Don't list "Python" by itself. List what you've done with it.

---

## Common Python Interview Questions

### Fundamentals

**Q: What is the difference between a list and a tuple?**

Lists are mutable (you can change them after creation). Tuples are immutable. Use t
-- coordinates, RGB values, function return pairs. Tuples are also slightly faster

**Q: What is a decorator?**

A decorator is a function that takes a function and returns a modified version. The

**Q: What is the GIL?**

The Global Interpreter Lock is a mutex inside CPython that allows only one thread to
safe without it). For I/O-bound work, threads still help because the GIL is release
bound work, use `multiprocessing` or PyPy.

**Q: What is the difference between `is` and `==`?**

`==` tests value equality. `is` tests identity --- whether two variables point to t

```python
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b)    # True --- same value
print(a is b)    # False --- different objects
```
```

Q: Explain list comprehensions vs generator expressions.

List comprehensions (`[x*2 for x in range(10)]`) evaluate eagerly — they create the whole list in memory at once. Generator expressions (`(x*2 for x in range(10))`) evaluate lazily — they produce one item at a time. Use generators when the sequence is large or when you only iterate once.

58.5.1 Intermediate

Q: What is a context manager?

An object that defines `__enter__` and `__exit__` methods, used with the `with` statement. `__enter__` sets up the resource, `__exit__` tears it down — even if an exception occurs. File handling (`with open(...)`) is the classic example.

Q: Explain `*args` and `**kwargs`.

`*args` collects extra positional arguments into a tuple. `**kwargs` collects extra keyword arguments into a dictionary. They allow functions to accept any number of arguments:

```
def log(message, *args, **kwargs):
    print(message.format(*args), kwargs)
```

Q: What is the difference between `@staticmethod` and `@classmethod`?

`@classmethod` receives the class (`cls`) as first argument and can create instances or access class state. `@staticmethod` receives no implicit argument — it's just a function namespaced inside the class with no access to class or instance state.

Q: How does Python's memory management work?

Python uses reference counting as the primary mechanism — each object tracks how many references point to it; when the count reaches 0, the object is freed immediately. For reference cycles (A -> B -> A), Python runs a cyclic garbage collector periodically.

58.5.2 Advanced

Q: What is the difference between `__getattr__` and `__getattribute__`?

`__getattribute__` is called on every attribute access. `__getattr__` is called only when normal attribute lookup fails. Override `__getattr__` for proxy objects or dynamic attributes. Override `__getattribute__` only when you need to intercept every access — be careful of infinite recursion.

Q: Explain Python's descriptor protocol.

A descriptor is any object that defines `__get__`, `__set__`, or `__delete__`. When you access an attribute, Python checks if the class attribute is a descriptor and

calls its `__get__` method. `property`, `classmethod`, and `staticmethod` are all descriptors.

Q: How does `async/await` work?

`async def` defines a coroutine — a function that can suspend its execution. `await` suspends the coroutine until the awaited coroutine completes, handing control back to the event loop so other coroutines can run. The event loop (`asyncio.run()`) manages all running coroutines. This is cooperative multitasking — coroutines voluntarily yield control, unlike threads which are preempted by the OS.

Q: What does `__slots__` do?

`__slots__` replaces the per-instance `__dict__` with fixed-size attribute slots. This reduces memory usage by ~200 bytes per instance and slightly speeds up attribute access. Use it for classes with millions of instances.

58.6 Coding Challenges — Practice Platforms

Platform	Best for
LeetCode	Algorithmic problems (FAANG interviews)
HackerRank	General coding + domain-specific (SQL, data)
Codewars	Fun kata-style problems, good for practice
Exercism	Mentored Python track
Advent of Code	Annual December challenge — builds problem-solving
Project Euler	Mathematical programming problems

For most Python jobs (outside FAANG), you'll solve problems like: - Parse a CSV and compute statistics - Write a function that does X with a list - Fix a bug in this code - Design a class hierarchy for this scenario

You don't need to master dynamic programming to get a Python developer job. Focus on clean code, good tests, and clear communication.

58.7 Open Source Contributions

Contributing to open source is one of the best ways to get noticed:

1. **Pick a library you use** — you already understand the domain
2. **Start small** — fix a typo in the docs, improve an error message, add a missing test
3. **Read the contributing guide** — every project has one
4. **Find good first issues** — filter by `good first issue` label on GitHub

Your first contribution doesn't have to be a new feature. A well-written bug fix with tests is impressive.

58.8 The Job Search Strategy

1. **Build 2-3 strong projects** — quality over quantity
2. **Deploy them live** — a URL you can click is worth ten screenshots
3. **Write about your projects** — a short blog post on Dev.to or your GitHub README explaining what you built and why
4. **LinkedIn** — update your headline to "Python Developer | FastAPI · pandas · pytest" and add your projects
5. **Apply early** — respond within 24 hours of postings
6. **Network** — attend Python meetups (meetup.com), PyCon, and local user groups
7. **Practice interviews** — solve 2-3 LeetCode easy/medium problems per week for 4-6 weeks before actively applying

---

58.9 What You Learned in This Chapter

- Employers care about what you've built, not what you studied. Three strong projects outperform ten certificates.
- Strong portfolio projects solve real problems, show the full engineering stack (tests, docs, CI, deployment), and are easy to run.
- Five project ideas ranked by difficulty: Finance API, Web Scraper Dashboard, CLI Tool (publish to PyPI), ML Pipeline, Full-Stack App.
- A great GitHub project has: a compelling README with badges, working CI, test coverage, a live URL, and a clean commit history.
- Resume tips: lead with skills not attitude, show numbers, list projects not tutorials, be specific in the skills section.
- Key interview questions covered: GIL, `is` vs `==`, generators vs lists, decorators, context managers, `*args`/`**kwargs`, descriptors, async/await,

`__slots__`.

- Practice on LeetCode/Codewars/Exercism. For most Python jobs, clean code and good tests matter more than algorithmic complexity.
- Open source contributions — even small ones — are highly valued by employers.

58.10 What's Next?

Chapter 59 — the final chapter — covers **Keep Learning: Resources and Community**. Where to go from here, the best books and podcasts, the communities worth joining, and a personal learning roadmap for the next 12 months.

Chapter 59

Chapter 59: Keep Learning — Resources and Community

You've reached the final chapter.

You started with `print("Hello, World!")`. You've covered variables, loops, functions, classes, files, error handling, OOP, functional programming, generators, decorators, type hints, concurrency, testing, debugging, performance, databases, web APIs, data science, machine learning, packaging, Git, logging, security, Python internals, and the entire ecosystem.

That is not a beginner's journey. That is the curriculum of a professional Python developer.

But learning never stops. The language evolves. The ecosystem grows. The best developers treat learning as a permanent habit, not a phase before the real work starts.

This chapter tells you exactly where to go from here.

59.1 How to Read the Official Python Docs

Most developers avoid the official docs. That's a mistake.

The Python documentation at docs.python.org is one of the best-written technical references in existence. Here's how to use it:

- **Tutorial** — docs.python.org/3/tutorial — A readable introduction. Worth re-reading after finishing this book — you'll understand far more now.
- **Library Reference** — docs.python.org/3/library — Your daily reference. Every standard library module is documented here. When you need `pathlib` or `datetime`, come here first.
- **Language Reference** — docs.python.org/3/reference — The formal specification of the language. Dense but authoritative.
- **What's New** — docs.python.org/3/whatsnew — Read this with every major Python release. Invaluable.
- **PEPs** — peps.python.org — Python Enhancement Proposals. PEP 8 (style), PEP 20 (Zen), PEP 484 (type hints), PEP 695 (type aliases) — reading the motivating PEPs gives you deep insight into *why* Python works the way it does.

Tip: When you Google a Python question and land on Stack Overflow, always also check the official docs link. StackOverflow answers can be outdated. The docs are always current.

59.2 Books Worth Reading Next

59.2.1 Python-Specific

Book	Best for
Fluent Python (Luciano Ramalho)	Deep mastery of Python idioms — the best intermediate Python book
Python Cookbook (Beazley & Jones)	Solutions to real problems — great reference
Effective Python (Brett Slatkin)	90 specific ways to write better Python
Architecture Patterns with Python (Percival & Gregory)	DDD, CQRS, event-driven design in Python
High Performance Python (Gorelick & Ozsvald)	Profiling and optimisation in depth
CPython Internals (Anthony Shaw)	How the interpreter works — for the deeply curious

59.2.2 Software Engineering (Language-Agnostic)

Book	Why you should read it
Clean Code (Robert Martin)	The principles of readable, maintainable code
The Pragmatic Programmer (Hunt & Thomas)	Career and craft advice that ages well
Designing Data-Intensive Applications (Kleppmann)	The best book on distributed systems and databases
A Philosophy of Software Design (Ousterhout)	How to manage complexity
Staff Engineer (Will Larson)	What senior technical leadership looks like

---

59.3 Podcasts

Podcast	Why listen
Talk Python To Me	Weekly interviews with Python ecosystem builders
Python Bytes	2x weekly — the best new Python packages and news
Real Python Podcast	Tutorials and interviews, beginner-friendly
Changelog	Broader software engineering, great guests
Software Engineering Radio	Deep dives on architecture and infrastructure
Lex Fridman	Long-form AI/CS researchers and engineers

Subscribe to at least **Talk Python To Me** and **Python Bytes**. Listen while commuting, exercising, or cooking. The cumulative effect over a year is enormous.

59.4 YouTube Channels

Channel	Content
mCoding	Python internals, CPython, advanced topics
Arjan Codes	Clean code, design patterns, OOP in Python
Tech With Tim	Beginner to intermediate Python projects
Sentdex	Python for finance, ML, and data science
3Blue1Brown	Mathematical intuition (not Python, but invaluable for ML)
Fireship	Fast, dense tech overviews — great for breadth
PyCon Talks	Conference talks from the annual PyCon conference

---

59.5 Newsletters

Newsletter	Why read it
Python Weekly	Curated Python news, articles, projects
Real Python Newsletter	Tutorials and articles from realPython.com
Pycoder's Weekly	Community newsletter, Python jobs
Data Science Weekly	ML, data, and AI articles
Import AI	Jack Clark's AI research digest

Subscribe to **Python Weekly** and **Pycoder's Weekly** at minimum. One email per week keeps you current without overwhelming you.

59.6 Communities

59.6.1 Online

Community	Where
Python Discord	discord.gg/python — 300k+ members, help channels, projects
r/learnpython	Reddit — beginner questions, welcoming community
r/Python	Reddit — news, projects, discussions
Dev.to	Write and read technical articles
Hacker News	Broader tech discussions, Python content
Python Forum	discuss.python.org — official Python community forum

59.6.2 Conferences

Conference	What
PyCon US	The flagship annual Python conference
EuroPython	Europe's largest Python conference
PyData	Data science and machine learning focused
DjangoCon	Django-specific (also web-focused Python)
local PyUGs	Python User Groups — monthly meetups worldwide

Even if you can't attend in person, all PyCon talks are recorded and free on youtube.com/c/PyConUS.

59.7 How to Read Other People's Code

This is one of the most underrated skills. Reading great code is as educational as writing it.

Start with small, well-regarded libraries: - `click` (~5k lines) — clean CLI design - `httpx` — async HTTP, well-typed - `attrs` — class definition patterns - `rich` — terminal output, beautiful code

How to read a codebase: 1. Read the README and docs first — understand the purpose 2. Find the entry point (usually `__init__.py` or `main.py`) 3. Trace one complete user flow — e.g. "what happens when I call `requests.get(url)`?" 4. Read the tests — they show intended usage 5. Look at the git log — `git log --oneline` shows how the project evolved 6. Ask "why?" for every design decision that surprises you

```
git clone https://github.com/pallets/click
cd click
# Read src/click/__init__.py to see public API
# Trace: what happens when @click.command() is called?
# Read tests/test_basic.py to see usage examples
```

59.8 A Personal Learning Roadmap for the Next 12 Months

Here is a realistic, achievable plan to go from "finished this book" to "confident professional Python developer":

59.8.1 Months 1-2: Consolidate and Ship

- Pick one project from Chapter 58 and build it end-to-end
- Deploy it live with a public URL
- Write tests (aim for 70%+ coverage)
- Set up CI/CD with GitHub Actions

59.8.2 Months 3-4: Specialise

Choose one path based on where you want to work:

Path	Focus
Backend Engineering	FastAPI + PostgreSQL + Docker + Redis + Celery
Data Engineering	pandas + SQL + dbt + Airflow + cloud storage
Machine Learning	scikit-learn + PyTorch + MLflow + model deployment
Data Analysis	pandas + seaborn + SQL + Tableau/PowerBI
DevOps/Platform	boto3 + Terraform + Docker + Kubernetes + Ansible

59.8.3 Months 5-6: Go Deeper

- Read **Fluent Python** (or the book for your specialism)
- Contribute one PR to an open source project
- Write 2-3 blog posts explaining something you learned

59.8.4 Months 7-9: Build in Public

- Start a second portfolio project, more ambitious than the first
- Share progress on LinkedIn and GitHub
- Attend one Python meetup or watch 3 PyCon talks per month

59.8.5 Months 10-12: Apply and Iterate

- Polish your resume and LinkedIn
- Apply to 2-3 positions per week
- Do mock interviews (interviewing.io, Pramp)
- Track applications in a spreadsheet — the process is a numbers game

---

59.9 The Most Important Lessons

After 59 chapters, here are the principles that matter most:

1. Build things, not notes.
Reading and highlighting teaches you little. Writing code — even broken code — teaches you everything. Every concept in this book becomes real the moment you use it in a project.

2. Read error messages completely.
Most beginners read the first line of a traceback and give up. The last line tells you what went wrong. The middle tells you where. Read all of it.

3. Break problems into small pieces.
Every complex program is just many simple programs working together. When something seems impossible, find the smallest next step and do just that.

4. Test your assumptions.
Not sure how something works? Try it in the REPL. Type it. Run it. Python rewards curiosity.

5. Write code for humans, not computers.
The computer doesn't care about variable names. Your future self and your teammates do. Clean, readable code is not a luxury — it is the baseline.

6. Simple is better than clever.
Python's Zen says it: "Simple is better than complex. Complex is better than complicated." The most impressive code is often the most obvious code.

7. Ship something.
An imperfect project deployed beats a perfect project that lives only on your machine. Get your code out into the world.

59.10 The Zen of Python — One More Time

```
import this
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one --- and preferably only one --- obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea --- let's do more of those!
```

These twenty aphorisms describe not just how to write Python, but how to think about software. Return to them every year. They mean something different each time.

59.11 A Final Word

You opened this book knowing little or nothing about Python. You're closing it as a Python developer.

You know how to write clean, tested, documented code. You know how to build web APIs, data pipelines, desktop apps, and command-line tools. You know how to manage dependencies, handle secrets, log properly, and secure your applications. You know what's happening under the hood when Python runs your code.

That knowledge is yours. Nobody can take it away.

Now go build something.

The Python community is welcoming, the ecosystem is rich, and the problems worth solving are everywhere. We'll see you at PyCon.

Python: Zero to Hero — Complete.

59 chapters. Every concept. Every pattern. Every tool.

From `print("Hello, World!")` *to production.*

Chapter 60

Appendices

60.1 Appendix A: Python Quick Reference

A single-page reference for the syntax and built-ins you use every day.

60.1.1 Built-in Types

Type	Example	Notes
`int`	`42`, `-7`, `0`	Arbitrary precision
`float`	`3.14`, `-0.5`	64-bit IEEE 754
`complex`	`3+4j`	`real` and `imag` attributes
`bool`	`True`, `False`	Subclass of `int`
`str`	`"hello"`, `'world'`	Immutable, Unicode
`bytes`	`b"hello"`	Immutable byte sequence
`list`	`[1, 2, 3]`	Mutable sequence
`tuple`	`(1, 2, 3)`	Immutable sequence
`set`	`{1, 2, 3}`	Mutable, unordered, unique
`frozenset`	`frozenset({1, 2})`	Immutable set
`dict`	`{"a": 1}`	Key-value, ordered (3.7+)
`None`	`None`	The null value

60.1.2 Numeric Operations

```
10 // 3       # 3      --- floor division
10 % 3        # 1      --- modulo
2 ** 10       # 1024   --- exponentiation
```

```
abs(-5)        # 5
round(3.567, 2)  # 3.57
divmod(10, 3)    # (3, 1)
```

60.1.3 String Methods

```
s = "  Hello, World!  "
s.strip()              # "Hello, World!"
s.lower()              # "  hello, world!  "
s.upper()              # "  HELLO, WORLD!  "
s.replace("Hello", "Hi")  # "  Hi, World!  "
s.split(", ")          # ["  Hello", "World!  "]
", ".join(["a","b","c"])  # "a, b, c"
s.startswith("  H")    # True
s.endswith("!")        # False (has trailing spaces)
s.strip().endswith("!")  # True
f"{'hello':>10}"       # "     hello"
f"{3.14159:.2f}"       # "3.14"
f"{1_000_000:,}"       # "1,000,000"
```

60.1.4 List Operations

```
lst = [3, 1, 4, 1, 5, 9]
lst.append(2)          # [3, 1, 4, 1, 5, 9, 2]
lst.extend([6, 5])     # in-place extend
lst.insert(0, 0)       # insert at index
lst.remove(1)          # remove first occurrence of 1
lst.pop()              # remove and return last
lst.pop(0)             # remove and return at index
lst.sort()             # in-place sort
lst.sort(reverse=True) # descending
lst.reverse()          # in-place reverse
lst.index(4)           # index of first 4
lst.count(1)           # count of 1s
sorted(lst)            # new sorted list
sorted(lst, key=abs)   # sort by key
```

60.1.5 Dictionary Operations

```
d = {"a": 1, "b": 2}
d["c"] = 3             # add/update
d.get("x", 0)          # 0 --- safe get with default
d.setdefault("y", [])  # set if missing, return value
del d["a"]             # delete key
```

```
"b" in d                # True
d.keys()                # dict_keys(["b", "c"])
d.values()              # dict_values([2, 3])
d.items()               # dict_items([("b", 2), ("c", 3)])
d.pop("b")              # remove and return
d.update({"e": 5})      # merge in-place
{**d, "f": 6}           # merge into new dict (3.9+: d | {"f":6})
```

60.1.6 Set Operations

```
a = {1, 2, 3}
b = {2, 3, 4}
a | b           # {1, 2, 3, 4} --- union
a & b           # {2, 3}       --- intersection
a - b           # {1}          --- difference
a ^ b           # {1, 4}       --- symmetric difference
a.issubset(b)   # False
a.issuperset(b)# False
a.isdisjoint({5, 6})   # True
```

60.1.7 Control Flow

```
# Ternary
x = "yes" if condition else "no"

# Walrus operator (3.8+)
if (n := len(data)) > 10:
    print(f"Too many: {n}")

# Match statement (3.10+)
match command:
    case "quit":
        quit()
    case "help" | "?":
        show_help()
    case str(msg) if msg.startswith("/"):
        handle_command(msg)
    case _:
        print("Unknown")
```

60.1.8 Comprehensions

```
# List
squares = [x**2 for x in range(10)]
```

```
evens   = [x for x in range(20) if x % 2 == 0]

# Dict
word_len = {w: len(w) for w in words}

# Set
unique_lengths = {len(w) for w in words}

# Generator (lazy)
gen = (x**2 for x in range(10**9))
next(gen)  # 0

# Nested
matrix = [[i*j for j in range(3)] for i in range(3)]
flat   = [x for row in matrix for x in row]
```

60.1.9 Functions

```
def greet(name: str, greeting: str = "Hello") -> str:
    """Return a greeting string."""
    return f"{greeting}, {name}!"

# Unpack
def add(*args: int) -> int:
    return sum(args)

def configure(**kwargs: str) -> None:
    for key, val in kwargs.items():
        print(f"{key} = {val}")

# Lambda
double = lambda x: x * 2

# Call with unpack
args = [1, 2, 3]
add(*args)

d = {"name": "Ada"}
greet(**d)
```

60.1.10 Built-in Functions

```
len(obj)            # length
range(n)            # 0 to n-1
```

```
range(a, b, step)    # a to b-1, stepping
enumerate(iterable) # (index, value) pairs
zip(a, b)           # pair up iterables
map(fn, iterable)    # apply fn lazily
filter(fn, iterable)# filter by fn lazily
sorted(iterable)     # new sorted list
reversed(sequence)   # reversed iterator
sum(iterable)        # numeric sum
min(iterable)        # minimum
max(iterable)        # maximum
any(iterable)        # True if any truthy
all(iterable)        # True if all truthy
isinstance(obj, T)   # type check
issubclass(A, B)     # class hierarchy check
hasattr(obj, name)   # attribute check
getattr(obj, name)   # attribute access
setattr(obj, name, val)   # attribute set
dir(obj)             # list attributes
vars(obj)            # __dict__
id(obj)              # memory address
type(obj)            # type
repr(obj)            # developer string
hash(obj)            # hash value
callable(obj)        # is it callable?
open(path, mode)     # file handle
print(*args, end="\n", sep=" ", file=sys.stdout)
input(prompt)        # read user input
int(), float(), str(), bool(), list(), dict(), set(), tuple()
```

60.2 Appendix B: Standard Library Cheat Sheet

The most useful modules from the Python standard library:

60.2.1 `os` — Operating System Interface

```
import os

os.getcwd()                  # current directory
os.chdir("/path")            # change directory
os.listdir(".")              # list files
os.makedirs("a/b", exist_ok=True)  # create dirs
```

```
os.remove("file.txt")         # delete file
os.rename("old", "new")       # rename
os.environ.get("HOME")        # env var
os.path.join("a", "b.txt")   # path join (prefer pathlib)
os.path.exists("file.txt")   # existence check
os.cpu_count()                # CPU count
```

60.2.2 `pathlib` — File Paths (Modern)

```
from pathlib import Path

p = Path("/home/user/data.csv")
p.name          # "data.csv"
p.stem          # "data"
p.suffix        # ".csv"
p.parent        # Path("/home/user")
p.exists()      # True/False
p.is_file()     # True
p.is_dir()      # False
p.read_text()  # file contents as string
p.write_text("hello")
p.read_bytes() # binary content
p.mkdir(parents=True, exist_ok=True)
list(p.glob("*.csv"))       # matching files
list(p.rglob("*.py"))       # recursive
p.with_suffix(".json")      # change extension
p / "subdir" / "file.txt"  # path joining
p.stat().st_size            # file size bytes
p.stat().st_mtime           # modification time
```

60.2.3 `sys` — System Information

```
import sys

sys.argv           # command-line arguments
sys.exit(0)        # exit with code
sys.path           # module search path
sys.version        # Python version string
sys.platform       # "linux", "win32", "darwin"
sys.stdin          # standard input
sys.stdout         # standard output
sys.stderr         # standard error
sys.getsizeof(obj) # size in bytes
sys.getrecursionlimit()
```

```
sys.setrecursionlimit(2000)
```

60.2.4 `datetime` — Dates and Times

```
from datetime import datetime, date, timedelta, timezone

now = datetime.now()
today = date.today()
utc_now = datetime.now(tz=timezone.utc)

# Create
dt = datetime(2024, 12, 25, 9, 30, 0)
d  = date(2024, 12, 25)

# Format
dt.strftime("%Y-%m-%d %H:%M:%S")   # "2024-12-25 09:30:00"
dt.isoformat()                      # "2024-12-25T09:30:00"

# Parse
datetime.strptime("2024-12-25", "%Y-%m-%d")
datetime.fromisoformat("2024-12-25T09:30:00")

# Arithmetic
tomorrow   = today + timedelta(days=1)
last_week  = today - timedelta(weeks=1)
diff       = dt2 - dt1   # timedelta

# Attributes
dt.year, dt.month, dt.day
dt.hour, dt.minute, dt.second
dt.weekday()    # 0=Monday, 6=Sunday
```

60.2.5 `collections` — Specialised Containers

```
from collections import defaultdict, Counter, deque, namedtuple, OrderedDict

# defaultdict
word_count = defaultdict(int)
word_count["hello"] += 1

# Counter
c = Counter(["a", "b", "a", "c", "a", "b"])
c.most_common(2)  # [("a", 3), ("b", 2)]
```

```
# deque --- fast append/pop from both ends
q = deque(maxlen=5)
q.append(1)
q.appendleft(0)
q.pop()
q.popleft()

# namedtuple
Point = namedtuple("Point", ["x", "y"])
p = Point(1, 2)
p.x, p.y

# OrderedDict (mostly redundant since Python 3.7 dicts are ordered)
od = OrderedDict()
```

60.2.6 `itertools` — Iterator Building Blocks

```
import itertools

itertools.chain([1,2], [3,4])          # 1 2 3 4
itertools.chain.from_iterable([[1,2],[3]])  # flatten
itertools.count(10, 2)                 # 10 12 14 ...
itertools.cycle([1,2,3])               # 1 2 3 1 2 3 ...
itertools.repeat(7, 3)                 # 7 7 7
itertools.islice(gen, 5)               # first 5
itertools.takewhile(pred, it)          # while pred is true
itertools.dropwhile(pred, it)          # drop while pred true
itertools.filterfalse(pred, it)        # where pred is false
itertools.compress([1,2,3], [1,0,1])  # 1 3
itertools.starmap(fn, [(a,b)])         # fn(*args)
itertools.accumulate([1,2,3,4])        # 1 3 6 10
itertools.groupby(sorted_data, key_fn)
itertools.combinations([1,2,3], 2)     # (1,2) (1,3) (2,3)
itertools.permutations([1,2,3], 2)     # all 2-perms
itertools.product([1,2], [3,4])        # cartesian product
itertools.batched(it, 3)               # groups of 3 (3.12+)
```

60.2.7 `functools` — Higher-Order Functions

```
from functools import reduce, partial, lru_cache, wraps, cached_property

reduce(lambda acc, x: acc + x, [1,2,3,4])  # 10
double = partial(pow, exp=2)                # pow with exp=2
```

```
@lru_cache(maxsize=128)
def fib(n):
    return n if n < 2 else fib(n-1) + fib(n-2)

@wraps(original_fn)  # preserve metadata on wrapped functions
def wrapper(*args, **kwargs): ...

class Circle:
    @cached_property        # computed once, then cached
    def area(self): return 3.14 * self.radius ** 2
```

60.2.8 `json` — JSON Encoding/Decoding

```
import json

# Encode
json.dumps({"name": "Ada", "age": 36})     # string
json.dumps(data, indent=2, sort_keys=True)  # pretty
json.dump(data, file_handle)               # to file

# Decode
json.loads('{"name": "Ada"}')              # dict from string
json.load(file_handle)                     # from file

# Custom encoder
class DateEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, date):
            return obj.isoformat()
        return super().default(obj)

json.dumps(data, cls=DateEncoder)
```

60.2.9 `re` — Regular Expressions

```
import re

re.match(r"^\d+", text)         # match at start
re.search(r"\d+", text)         # first match anywhere
re.findall(r"\d+", text)        # all matches
re.finditer(r"\d+", text)       # iterator of matches
re.sub(r"\s+", " ", text)       # replace
re.split(r"[,;]\s*", text)      # split
```

```python
# Compile for reuse
pattern = re.compile(r"(\d{4})-(\d{2})-(\d{2})")
m = pattern.search("Date: 2024-12-25")
m.group(0)     # "2024-12-25"
m.group(1)     # "2024"
m.groups()     # ("2024", "12", "25")

# Flags
re.search(r"hello", text, re.IGNORECASE)
re.DOTALL      # . matches newline
re.MULTILINE   # ^ and $ match line starts/ends
```

60.2.10 `subprocess` — Run System Commands

```python
import subprocess

# Simple run
result = subprocess.run(
    ["git", "log", "--oneline", "-5"],
    capture_output=True,
    text=True,
    check=True,           # raises if returncode != 0
)
print(result.stdout)
print(result.returncode)  # 0

# Shell command (avoid when possible --- injection risk)
subprocess.run("ls -la | grep .py", shell=True)
```

60.2.11 `typing` — Type Hints

```python
from typing import Optional, Union, Any, Callable, TypeVar
from typing import Sequence, Mapping, Iterable
from collections.abc import Generator, Iterator, AsyncGenerator

# Python 3.10+ --- use | instead of Union
def foo(x: int | None) -> str | int: ...

T = TypeVar("T")

def first(items: list[T]) -> T:
    return items[0]

# Callable
```

```python
Handler = Callable[[str, int], bool]

# TypedDict
from typing import TypedDict
class Movie(TypedDict):
    title: str
    year: int
```

60.3 Appendix C: Virtual Environment and Project Setup Cheat Sheet

60.3.1 Quick Start (venv)

```
# Create and activate
python -m venv .venv

# Windows
.venv\Scripts\activate

# Mac/Linux
source .venv/bin/activate

# Install and freeze
pip install requests pandas
pip freeze > requirements.txt
pip install -r requirements.txt

# Deactivate
deactivate
```

60.3.2 Quick Start (uv — recommended)

```
# Install uv
pip install uv

# New project
uv init myproject
cd myproject

# Add dependencies
uv add requests pandas pytest
```

```
# Run
uv run python script.py
uv run pytest

# Sync from lockfile
uv sync
```

60.3.3 Project Structure Template

```
my-project/
├── src/
│   └── my_project/
│       ├── __init__.py
│       ├── core.py
│       └── utils.py
├── tests/
│   ├── conftest.py
│   ├── test_core.py
│   └── test_utils.py
├── docs/
├── scripts/
├── .env.example
├── .gitignore
├── pyproject.toml
├── README.md
└── Makefile
```

60.3.4 Python `.gitignore` (essential lines)

```
# Virtual environments
.venv/
venv/
env/

# Python cache
__pycache__/
*.py[cod]
*.pyo

# Distribution
dist/
build/
*.egg-info/
```

```
# Pytest
.pytest_cache/
.coverage
htmlcov/

# Environment variables
.env

# IDE
.idea/
.vscode/
*.swp

# OS
.DS_Store
Thumbs.db
```

60.4 Appendix D: Git Quick Reference

```
# Setup
git config --global user.name  "Your Name"
git config --global user.email "you@example.com"

# Start
git init
git clone https://github.com/user/repo.git

# Core workflow
git status
git add file.py
git add .
git add -p                # interactive staging
git commit -m "feat: add user login"
git log --oneline --graph

# Branches
git switch -c feature/login
git switch main
git merge feature/login
git branch -d feature/login

# Remote
```

```
git remote add origin https://github.com/user/repo.git
git push -u origin main
git pull
git fetch origin

# Undo
git restore file.py       # discard changes
git restore --staged file # unstage
git revert HEAD           # undo last commit safely
git stash
git stash pop

# Useful
git diff
git show HEAD
git log --author="You" --since="1 week ago"
git tag v1.0.0
git bisect start
```

60.4.1 Conventional Commits

```
feat:     new feature
fix:      bug fix
docs:     documentation change
test:     add or update tests
refactor: code change with no feature/fix
perf:     performance improvement
chore:    build, config, or tool change
ci:       CI/CD configuration
```

60.5 Appendix E: pytest Quick Reference

```
# Basic test
def test_add():
    assert add(1, 2) == 3

# Exception testing
def test_divide_zero():
    with pytest.raises(ZeroDivisionError):
        divide(1, 0)

# Parametrize
```

```python
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),
    (0, 0, 0),
    (-1, 1, 0),
])
def test_add(a, b, expected):
    assert add(a, b) == expected

# Fixture
@pytest.fixture
def db():
    conn = sqlite3.connect(":memory:")
    init_db(conn)
    yield conn
    conn.close()

def test_query(db):
    result = query(db, "SELECT 1")
    assert result == [(1,)]

# Mock
from unittest.mock import patch, Mock

def test_api_call():
    with patch("mymodule.requests.get") as mock_get:
        mock_get.return_value.json.return_value = {"status": "ok"}
        result = fetch_status()
    assert result == "ok"
```

```
pytest                    # run all tests
pytest -v                 # verbose
pytest tests/test_core.py # specific file
pytest -k "login"         # filter by name
pytest --cov=src          # coverage
pytest --cov=src --cov-report=html  # HTML report
pytest -x                 # stop on first failure
pytest --tb=short         # shorter tracebacks
```

60.6 Appendix F: f-String Formatting Reference

```
name  = "Ada"
value = 3.14159
count = 1_000_000

# Basic
f"Hello, {name}!"           # "Hello, Ada!"
f"Pi = {value}"             # "Pi = 3.14159"

# Debug (3.8+)
f"{name=}"                  # "name='Ada'"
f"{value=:.2f}"             # "value=3.14"

# Number formatting
f"{value:.2f}"              # "3.14"
f"{value:.4e}"              # "3.1416e+00"
f"{count:,}"                # "1,000,000"
f"{count:_}"                # "1_000_000"
f"{0.1234:.0%}"             # "12%"
f"{255:#010b}"              # "0b11111111"
f"{255:#x}"                 # "0xff"

# Alignment
f"{'left':<10}"             # "left      "
f"{'right':>10}"            # "     right"
f"{'center':^10}"           # "  center  "
f"{'padded':*^10}"          # "**padded**"

# Width from variable
width = 10
f"{'hello':>{width}}"       # "     hello"

# Nested expressions
f"{2 + 2}"                  # "4"
f"{'yes' if True else 'no'}"  # "yes"
f"{', '.join(['a','b','c'])}"  # "a, b, c"
```

Chapter 61

Glossary

Abstract Base Class (ABC) A class that cannot be instantiated directly and defines an interface that subclasses must implement. Created using the `abc` module. Subclasses that do not implement all abstract methods also cannot be instantiated.

Argument A value passed to a function when it is called. Contrast with *parameter*, which is the variable in the function definition. `greet("Ada")` — `"Ada"` is the argument.

Assertion A statement that a condition must be true. In Python, `assert condition` raises `AssertionError` if `condition` is false. Used in tests (`pytest`) and for sanity checks in code.

Asynchronous Programming A style of concurrent programming where tasks can pause and resume without blocking other tasks. Python implements this with `async`/`await` and the `asyncio` event loop.

Attribute A variable or method belonging to an object or class. Accessed with dot notation: `obj.attribute`.

Augmented Assignment A shorthand operator that combines an operation with assignment. `x += 1` is equivalent to `x = x + 1`. Available for most arithmetic operators: `+=`, `-=`, `*=`, `/=`, `//=`, `**=`, `%=`, `|=`, `&=`.

Boolean A data type with two values: `True` and `False`. A subclass of `int` in Python (`True == 1`, `False == 0`).

Bytecode An intermediate representation of Python code, compiled from source. Stored in `.pyc` files inside `__pycache__`. The CPython virtual machine executes bytecode, not raw source code.

Callable Any object that can be called with parentheses. Functions, methods, classes, and objects with a `__call__` method are all callable.

Class A blueprint for creating objects. Defines attributes (data) and methods (behaviour) that instances of the class will have.

Class Method A method decorated with `@classmethod` that receives the class (`cls`) as its first argument instead of an instance. Often used as alternative constructors.

Closure A function that retains access to variables from the enclosing scope, even after the enclosing function has finished executing. Used to create factory functions and to implement the Decorator pattern.

Comprehension A concise syntax for creating collections. List comprehensions (`[x*2 for x in range(10)]`), dictionary comprehensions (`{k: v for k, v in pairs}`), and set comprehensions (`{len(w) for w in words}`) are all available.

Concurrency The ability to handle multiple tasks that overlap in time. Python supports concurrency through threading, multiprocessing, and async I/O.

Context Manager An object that defines `__enter__` and `__exit__` methods for use with the `with` statement. Ensures setup and teardown happen correctly, even if exceptions occur. Files and database connections are classic examples.

Coroutine A function defined with `async def`. Can be paused with `await` and resumed later by the event loop. The building block of `asyncio` concurrency.

CPython The reference implementation of Python, written in C. When people say "Python," they almost always mean CPython. Alternative implementations include PyPy, Jython, and MicroPython.

Data Class A class created with the `@dataclass` decorator from the `dataclasses` module. Automatically generates `__init__`, `__repr__`, and `__eq__` based on class-level type-annotated attributes.

Decorator A function that takes a function and returns a modified or wrapped version of it. Applied with the `@` syntax above a function definition. Common examples: `@property`, `@staticmethod`, `@classmethod`, `@lru_cache`.

Default Parameter A function parameter with a default value that is used when the caller does not provide that argument. Example: `def greet(name, greeting="Hello")`.

Descriptor An object that defines `__get__`, `__set__`, or `__delete__` to customise attribute access. `property`, `classmethod`, and `staticmethod` are all descriptors.

Dictionary A mutable, ordered (Python 3.7+) collection of key-value pairs. Keys must be hashable and unique. Created with `{}` or `dict()`.

Docstring A string literal placed as the first statement of a module, class, or function. Used to document the object. Accessible via `obj.__doc__` or `help(obj)`.

Duck Typing Python's type system philosophy: if an object has the right methods and attributes, it can be used in a given context, regardless of its class. "If it walks like a duck and quacks like a duck, it is a duck."

Dunder (Magic) Method A method with double underscores on both sides of the name, like `__init__`, `__str__`, `__len__`. Also called "magic methods." They define how objects behave with Python's built-in operations.

Event Loop The central mechanism in `asyncio` that schedules and runs coroutines. `asyncio.run()` creates an event loop, runs the given coroutine, and closes the loop.

Exception An error condition that interrupts normal program flow. Python raises exceptions as objects (instances of `BaseException` subclasses). Handled with `try`/`except` blocks.

Expression Any piece of code that evaluates to a value. `2 + 2`, `"hello".upper()`, `x if x > 0 else -x` are all expressions.

f-String A string literal prefixed with `f` that can contain expressions inside `{}` braces that are evaluated at runtime. Example: `f"Hello, {name}!"`. Available since Python 3.6.

First-Class Function Functions that can be stored in variables, passed as arguments, and returned from other functions. Python functions are first-class.

Fixture (testing) In `pytest`, a function decorated with `@pytest.fixture` that provides a pre-configured resource (database connection, test data, etc.) to test functions via dependency injection.

Float A floating-point number — a real number with a decimal point. Stored as a 64-bit IEEE 754 double. Subject to floating-point rounding: `0.1 + 0.2 != 0.3`.

Function A named, reusable block of code defined with `def`. Takes zero or more parameters and optionally returns a value.

Generator A function that uses `yield` to produce a sequence of values lazily, one at a time. Memory-efficient for large sequences. Returns a generator object.

Generator Expression A lazy version of a list comprehension: `(x*2 for x in range(10))`. Produces values one at a time instead of creating a full list in memory.

GIL (Global Interpreter Lock) A mutex in CPython that ensures only one thread executes Python bytecode at a time. Simplifies memory management

but limits CPU-bound parallelism in threads. Use `multiprocessing` for CPU-bound parallel work.

Global Scope The top-level scope of a module. Variables defined at module level are in global scope and accessible from anywhere within that module.

hashable An object is hashable if it has a `__hash__` method that returns an integer and an `__eq__` method, and its hash never changes. Hashable objects can be dictionary keys or set members. Strings, numbers, and tuples of hashables are hashable. Lists and dicts are not.

Higher-Order Function A function that takes one or more functions as arguments or returns a function as its result. `map()`, `filter()`, and `sorted(key=...)` are higher-order functions.

Immutable An object whose value cannot be changed after it is created. Numbers, strings, tuples, and frozensets are immutable. Attempting to modify them creates a new object instead.

Import The mechanism for making code from another module available in the current namespace. `import os`, `from pathlib import Path`, `from . import utils`.

Index An integer used to access a specific element of a sequence. Python uses zero-based indexing: the first element is at index `0`. Negative indices count from the end: `-1` is the last element.

Inheritance A mechanism where a class (subclass) derives attributes and methods from another class (superclass). Allows code reuse and creates an "is-a" relationship.

Instance A specific object created from a class. Each instance has its own attribute values. `person = Person("Ada")` creates one instance of the `Person` class.

Integer A whole number with no decimal point. Python integers have arbitrary precision — they can be as large as available memory allows.

Iterable Any object that can be iterated over — that returns an iterator when `iter()` is called on it. Lists, strings, dicts, files, generators, and ranges are all iterables.

Iterator An object with a `__next__()` method that returns the next item in a sequence or raises `StopIteration` when exhausted. All generators are iterators.

Keyword Argument An argument passed to a function using the parameter name: `greet(name="Ada", greeting="Hi")`. Allows arguments in any order.

Lambda An anonymous function defined with the `lambda` keyword. Limited to a single expression. Example: `lambda x: x * 2`. Useful as a short function

passed to `sorted()` or `map()`.

List A mutable, ordered sequence that can hold items of any type. Created with `[]`. Items can be added, removed, and changed.

Local Scope The scope inside a function. Variables defined inside a function are local to that function and not accessible outside it.

Method A function defined inside a class. When called on an instance, the instance is automatically passed as the first argument (`self`).

Module A Python file (`.py`) containing definitions and statements. Imported with `import`. Every `.py` file is a module.

Mutable An object whose value can be changed after it is created. Lists, dicts, and sets are mutable.

Namespace A container that maps names to objects. Python has local, enclosing, global, and built-in namespaces, searched in that order (LEGB rule).

None Python's null value. A singleton — there is only one `None` object. Check with `is None`, not `== None`.

Object An instance of a class. In Python, everything is an object: integers, strings, functions, classes themselves.

Package A directory containing an `__init__.py` file and one or more modules. Organises related modules into a hierarchy.

Parameter A variable in a function definition that receives an argument when the function is called. Contrast with *argument*.

Polymorphism The ability of different objects to respond to the same method call in different ways. In Python, implemented via duck typing and method overriding.

Property An attribute managed by getter/setter/deleter methods, defined with `@property`. Allows attribute access syntax (`obj.name`) while running code behind the scenes.

Protocol A structural typing construct (from `typing.Protocol`) that defines an interface by the presence of methods and attributes, without requiring inheritance. Python's formal version of duck typing.

Raise The `raise` keyword is used to trigger an exception manually. `raise ValueError("Invalid input")`.

Regular Expression (Regex) A pattern syntax for matching, searching, and replacing text. Python's `re` module provides regex support.

Return The `return` statement exits a function and optionally provides a value to the caller. A function without `return` (or with bare `return`) returns `None`.

Scope The region of code where a variable name is accessible. Python scopes: local, enclosing, global, built-in (LEGB).

Sequence An ordered collection that supports indexing, slicing, and `len()`. Lists, tuples, strings, and ranges are sequences.

Set An unordered collection of unique, hashable objects. Useful for membership testing and set operations (union, intersection, difference). Created with `{}` (not `{}` for empty — use `set()`).

Slice A way to extract a subset of a sequence using `[start:stop:step]` notation. `lst[1:4]` returns elements at indices 1, 2, 3.

Stack Trace (Traceback) The output Python shows when an exception is raised. It shows the chain of function calls that led to the error. Always read the last line first.

Statement A complete unit of code that performs an action. `x = 5`, `print("hi")`, `if x > 0:` are all statements. Contrast with expressions, which evaluate to values.

Static Method A method decorated with `@staticmethod` that receives no implicit first argument. It's a plain function namespaced inside a class.

String An immutable sequence of Unicode characters. Created with single `'`, double `"`, or triple `"""` quotes.

Subclass A class that inherits from another class (the superclass or parent class). Uses `super()` to access the parent's methods.

Tuple An immutable, ordered sequence. Created with `()`. Faster than lists and can be used as dictionary keys. Often used for fixed-length data.

Type Annotation A syntax for declaring the expected type of a variable, parameter, or return value. `def greet(name: str) -> str:`. Not enforced at runtime but checked by tools like `mypy`.

Type Hint Synonymous with *type annotation*. The `typing` module provides tools for complex type hints: `Optional`, `Union`, `List`, `Dict`, `Callable`, `TypeVar`, etc.

`*args` A function parameter that collects any additional positional arguments into a tuple. Example: `def add(*numbers): return sum(numbers)`.

`**kwargs` A function parameter that collects any additional keyword arguments into a dictionary. Example: `def configure(**options): ...`.

Variable A name that refers to an object in memory. In Python, variables are references — multiple variables can point to the same object.

Virtual Environment An isolated Python environment with its own Python interpreter and packages. Created with `python -m venv .venv`. Prevents

package version conflicts between projects.

`with` Statement Syntax that uses a context manager to ensure proper setup and teardown. `with open("file.txt") as f:` automatically closes the file.

`yield` A keyword used inside a generator function to produce a value and pause execution until the next value is requested.

Chapter 62

Index

62.1 A

`abc` module, Ch. 21
Abstract Base Classes (ABCs), Ch. 21
`abstractmethod`, Ch. 21
`aiohttp`, Ch. 28
Alembic (migrations), Ch. 37
`all()`, Ch. 23
`any()`, Ch. 23
`argparse`, Ch. 57
`args` (`*args`), Ch. 8
`assert`, Ch. 30
Async I/O, Ch. 28
`async def`, Ch. 28
`asyncio`, Ch. 28
`asyncio.gather()`, Ch. 28
`asyncio.run()`, Ch. 28
Augmented assignment (`+=`, `-=`...), Ch. 2

62.2 B

`BeautifulSoup`, Ch. 39
Big O notation, Ch. 33
Boolean values (`True`/`False`), Ch. 4
`boto3`, Ch. 57

`break` statement, Ch. 5
Breakpoints (`breakpoint()`), Ch. 31
Broadcasting (NumPy), Ch. 42
`build` (packaging), Ch. 50
Builder pattern, Ch. 54
Bytecode, Ch. 56

62.3 C

`cached_property`, Ch. 18
Callable, Ch. 22
`cProfile`, Ch. 33
Class attributes, Ch. 17
Class methods (`@classmethod`), Ch. 17
Classes (basics), Ch. 17
Closures, Ch. 22
`collections.Counter`, App. B
`collections.defaultdict`, App. B
`collections.deque`, App. B
`collections.namedtuple`, App. B
Command pattern, Ch. 54
Comparison operators, Ch. 4
Comprehensions (list, dict, set), Ch. 23
`concurrent.futures`, Ch. 28
`conftest.py`, Ch. 30
Context managers, Ch. 25
`contextlib.contextmanager`, Ch. 25
`contextlib.suppress`, Ch. 25
`continue` statement, Ch. 5
Conventional commits, Ch. 51
Coroutines, Ch. 28
`cProfile`, Ch. 33
CPython internals, Ch. 56
`csv` module, Ch. 39

62.4 D

`dataclasses`, Ch. 17
Data Science (overview), Ch. 42–45
`datetime` module, App. B
Debugging, Ch. 31
Decorator pattern, Ch. 54
Decorators (function), Ch. 22

Default parameters, Ch. 8
`defaultdict`, App. B
Dependency injection (`Depends`), Ch. 41
Descriptors, Ch. 21
Dictionary comprehensions, Ch. 23
Dictionaries, Ch. 3
`dis` module, Ch. 56
Docker, Ch. 41, Ch. 50
Docstrings, Ch. 8
`dotenv` (`python-dotenv`), Ch. 52
Duck typing, Ch. 19
Dunder methods, Ch. 17

62.5 E

`email` (module), Ch. 46
`enumerate()`, Ch. 23
Environment variables, Ch. 52, Ch. 55
Error handling (`try`/`except`), Ch. 11
Event loop (`asyncio`), Ch. 28
Exception hierarchy, Ch. 11
`ExitStack`, Ch. 25
`f-strings`, Ch. 3, App. F

62.6 F

Facade pattern, Ch. 54
Factory pattern, Ch. 54
FastAPI, Ch. 41, Ch. 47
`ffill()` / `bfill()`, Ch. 43
File I/O, Ch. 10
`filter()`, Ch. 23
Flask, Ch. 40
`for` loops, Ch. 5
`frozenset`, App. A
`functools`, Ch. 22, App. B
`functools.lru_cache`, Ch. 22, Ch. 33
`functools.partial`, Ch. 22
`functools.reduce`, Ch. 22
`functools.wraps`, Ch. 22

62.7 G

Garbage collection, Ch. 56
Generator expressions, Ch. 24
Generators, Ch. 24
`get_db()` (SQLAlchemy), Ch. 37
`getattr()`, Ch. 56
`gh` (GitHub CLI), Ch. 51
GIL (Global Interpreter Lock), Ch. 28, Ch. 56
Git, Ch. 51
`git bisect`, Ch. 51
`git blame`, Ch. 51
`git stash`, Ch. 51
GitHub Actions, Ch. 50, Ch. 51
`global` keyword, Ch. 8
Glossary, App. (Glossary)
`GridSearchCV`, Ch. 45
`groupby()` (pandas), Ch. 43
Gunicorn, Ch. 40

62.8 H

Hashability, Glossary
Higher-order functions, Ch. 22
`html.escape()`, Ch. 55

62.9 I

`if`/`elif`/`else`, Ch. 4
`import` system, Ch. 12, Ch. 56
Inheritance, Ch. 20
`__init__`, Ch. 17
`__init__.py`, Ch. 50
`inspect` module, Ch. 31
Instance attributes, Ch. 17
Integer (`int`), Ch. 2
`isinstance()`, Ch. 20
`issubclass()`, Ch. 20
`itertools`, Ch. 24, App. B

62.10 J

Jinja2 templates, Ch. 40
JSON (`json` module), Ch. 13, App. B
JWT (JSON Web Tokens), Ch. 41, Ch. 55

62.11 K

`kwargs` (`**kwargs`), Ch. 8

62.12 L

Lambda functions, Ch. 22
LEGB rule (scope), Ch. 8
`line_profiler`, Ch. 33
Lists, Ch. 6
`loc` / `iloc` (pandas), Ch. 43
`logging` module, Ch. 53
Logical operators (`and`, `or`, `not`), Ch. 4
`lru_cache`, Ch. 22, Ch. 33

62.13 M

`map()`, Ch. 23
Matplotlib, Ch. 44
Memory management, Ch. 56
`memory_profiler`, Ch. 33
Method Resolution Order (MRO), Ch. 21
`merge()` (pandas), Ch. 43
`mock` / `unittest.mock`, Ch. 30
Modules, Ch. 12
Multiprocessing, Ch. 28
Multiple inheritance, Ch. 21
`mypy`, Ch. 26

62.14 N

Nested loops, Ch. 5
Nested lists, Ch. 6
`nonlocal` keyword, Ch. 22
`None`, Ch. 8
NumPy, Ch. 42

62.15 O

Object-Oriented Programming (OOP), Ch. 17–21
Observer pattern, Ch. 54
`openpyxl`, Ch. 46
`os` module, App. B
`Optional` (typing), Ch. 26

62.16 P

`packaging` (pyproject.toml), Ch. 50
Packages, Ch. 12
`pandas`, Ch. 43
`parameterized` queries, Ch. 37
`passlib`, Ch. 55
`pathlib`, Ch. 10, App. B
`pdb` debugger, Ch. 31
Performance profiling, Ch. 33
`Pillow` (PIL), Ch. 46
`pip`, Ch. 12
`pip-audit`, Ch. 55
`pip-tools`, Ch. 52
`playwright`, Ch. 39
Poetry, Ch. 52
Polymorphism, Ch. 20
PostgreSQL (`psycopg2`), Ch. 37
`print()` debugging, Ch. 31
`property` decorator, Ch. 17
Protocol (typing), Ch. 26
`psycopg2`, Ch. 37
PyPI / TestPyPI, Ch. 50
`pyproject.toml`, Ch. 50, Ch. 52
`pytest`, Ch. 30, App. E
`pytest-cov`, Ch. 30

62.17 R

`raise`, Ch. 11
`random` module, Ch. 13
`range()`, Ch. 5
Reference counting, Ch. 56
Regular expressions (`re`), App. B
`reportlab`, Ch. 46

`requests` library, Ch. 39
`requests.Session`, Ch. 39
`return`, Ch. 8
`robots.txt`, Ch. 39
`ruff` (linter), Ch. 50

62.18 S

`schedule` module, Ch. 46
scikit-learn, Ch. 45
Scope (local, global, enclosing), Ch. 8
Seaborn, Ch. 44
Security, Ch. 55
`selenium`, Ch. 39
`self`, Ch. 17
Semantic versioning, Ch. 50
Sets, Ch. 3
`shutil`, Ch. 46
Singleton pattern, Ch. 54
Slicing, Ch. 6
`smtplib`, Ch. 46
SQLAlchemy ORM, Ch. 37
SQLite (`sqlite3`), Ch. 37
`src` layout, Ch. 50, Ch. 52
Static methods (`@staticmethod`), Ch. 17
Strategy pattern, Ch. 54
`str` methods, App. A
`subprocess`, App. B
`super()`, Ch. 20
`sys` module, App. B

62.19 T

Test-driven development (TDD), Ch. 30
Threading, Ch. 28
`timeit`, Ch. 33
Tkinter, Ch. 49
`traceback`, Ch. 11
`tracemalloc`, Ch. 56
Tracebacks, Ch. 31
`try`/`except`/`finally`/`else`, Ch. 11
Tuples, Ch. 3
`twine`, Ch. 50

Type aliases, Ch. 26
Type hints, Ch. 26
TypedDict, Ch. 26
TypeVar, Ch. 26

62.20 U

`unittest.mock`, Ch. 30
`uv` (package manager), Ch. 52
`uvicorn`, Ch. 41

62.21 V

Variables, Ch. 2
`venv`, Ch. 12, Ch. 52
Virtual environments, Ch. 12, Ch. 52

62.22 W

Walrus operator (`:=`), Ch. 4
Web scraping, Ch. 39
`while` loops, Ch. 5
`with` statement, Ch. 25

62.23 Y

`yield`, Ch. 24
`yield from`, Ch. 24

62.24 Z

`zip()`, Ch. 23