PYTHON PROGRAMMING *with* DESIGN PATTERNS

JAMES W. COOPER

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Python Programming with Design Patterns

This page intentionally left blank

Python Programming with Design Patterns

James W. Cooper

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town •
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi •
Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Python screenshots: © 2001-2021, Python Software Foundation

Cover image: spainter_vfx/Shutterstock

Screenshot of MySQL Workbench © 2021, Oracle Corporation

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the web: informit.com/aw

Library of Congress Control Number: 2021947622

Copyright © 2022 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-757993-8
ISBN-10: 0-13-757993-4

ScoutAutomatedPrintCode

Editor-in-Chief
Mark Taub

Executive Editor
Debra J. Willimans

Development Editor
Chris Zahn

Managing Editor
Sandra Schroeder

Senior Project Editor
Lori Lyons

Copy Editor
Krista Hansing Editorial Services

Production Manager
Remya Divakaran/ Codemantra

Indexer
Ken Johnson

Proofreader
Charlotte Kughen

Compositor
Codemantra

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

❖

To Vicki

❖

Contents at a Glance

I: Introduction

1 Introduction to Objects 5

2 Visual Programming in Python 17

3 Visual Programming of Tables of Data 41

4 What Are Design Patterns? 53

II: Creational Patterns

5 The Factory Pattern 61

6 The Factory Method Pattern 67

7 The Abstract Factory Pattern 75

8 The Singleton Pattern 79

9 The Builder Pattern 83

10 The Prototype Pattern 91

11 Summary of Creational Patterns 95

III: Structural Patterns

12 The Adapter Pattern 99

13 The Bridge Pattern 105

14 The Composite Pattern 111

15 The Decorator Pattern 121

16 The Façade Pattern 129

17 The Flyweight Pattern 139

18 The Proxy Pattern 145

19 Summary of Structural Patterns 151

IV: Behavioral Patterns

20 Chain of Responsibility Pattern **155**

21 The Command Pattern **167**

22 The Interpreter Pattern **177**

23 The Iterator Pattern **187**

24 The Mediator Pattern **195**

25 The Memento Pattern **203**

26 The Observer Pattern **211**

27 The State Pattern **217**

28 The Strategy Pattern **225**

29 The Template Pattern **233**

30 The Visitor Pattern **239**

V: A Brief Introduction to Python

31 Variables and Syntax in Python **249**

32 Making Decisions in Python **263**

33 Development Environments **275**

34 Python Collections and Files **279**

35 Functions **291**

A Running Python programs **295**

Index **299**

Table of Contents

I: Introduction 1

The tkinter Library 2

GitHub 2

1 Introduction to Objects 5

The Class __init__ Method 6

Variables Inside a Class 6

Collections of Classes 7

Inheritance 8

Derived Classes Created with Revised Methods 8

Multiple Inheritance 8

Drawing a Rectangle and a Square 10

Visibility of Variables 12

Properties 13

Local Variables 13

Types in Python 13

Summary 14

Programs on GitHub 15

2 Visual Programming in Python 17

Importing Fewer Names 19

Creating an Object-Oriented Version 19

Using Message Boxes 21

Using File Dialogs 22

Understanding Options for the Pack Layout Manager 23

Using the ttk Libraries 24

Responding to User Input 25

Adding Two Numbers 26

Catching the Error 26

Applying Colors in tkinter 27

Creating Radio Buttons 27

Using a Class-Level Variable 30

Communicating Between Classes 30

Using the Grid Layout 30

Creating Checkbuttons 32

Disabling Check Boxes 34

Adding Menus to Windows 35
Using the LabelFrame 39
Moving On 40
Examples on GitHub 40

3 Visual Programming of Tables of Data 41
Creating a Listbox 42
Displaying the State Data 44
Using a Combobox 46
The Treeview Widget 47
Inserting Tree Nodes 50
Moving On 51
Example Code on GitHub 51

4 What Are Design Patterns? 53
Defining Design Patterns 54
The Learning Process 55
Notes on Object-Oriented Approaches 56
Python Design Patterns 57
References 57

II: Creational Patterns 59

5 The Factory Pattern 61
How a Factory Works 61
Sample Code 62
The Two Subclasses 62
Building the Simple Factory 63
Using the Factory 63
A Simple GUI 64
Factory Patterns in Math Computation 65
Programs on GitHub 65
Thought Questions 66

6 The Factory Method Pattern 67
The Swimmer Class 68
The Event Classes 69
Straight Seeding 70
Circle Seeding 71

Our Seeding Program 72
Other Factories 74
When to Use a Factory Method 74
Programs on GitHub 74

7 The Abstract Factory Pattern 75
A GardenMaker Factory 75
How the User Interface Works 77
Consequences of the Abstract Factory Pattern 77
Thought Questions 78
Code on GitHub 78

8 The Singleton Pattern 79
Throwing the Exception 80
Creating an Instance of the Class 80
Static Classes As Singleton Patterns 81
Finding the Singletons in a Large Program 81
Other Consequences of the Singleton Pattern 82
Sample Code on GitHub 82

9 The Builder Pattern 83
An Investment Tracker 84
Calling the Builders 86
The List Box Builder 87
The Checkbox Builder 88
Displaying the Selected Securities 89
Consequences of the Builder Pattern 89
Thought Questions 89
Sample Code on GitHub 89

10 The Prototype Pattern 91
Cloning in Python 91
Using the Prototype 92
Consequences of the Prototype Pattern 94
Sample Code on GitHub 94

11 Summary of Creational Patterns 95

III: Structural Patterns 97

12 The Adapter Pattern 99
Moving Data Between Lists 99
Making an Adapter 101
The Class Adapter 103
Two-Way Adapters 103
Pluggable Adapters 103
Programs on GitHub 103

13 The Bridge Pattern 105
Creating the User Interface 107
Extending the Bridge 108
Consequences of the Bridge Pattern 109
Programs on GitHub 110

14 The Composite Pattern 111
An Implementation of a Composite 112
Salary Computation 112
The Employee Classes 112
The Boss Class 113
Building the Employee Tree 114
Printing the Employee Tree 114
Creating a Treeview of the Composite 116
Using Doubly Linked Lists 117
Consequences of the Composite Pattern 118
A Simple Composite 119
Other Implementation Issues 119
Dealing with Recursive Calls 119
Ordering Components 120
Caching Results 120
Programs on GitHub 120

15 The Decorator Pattern 121
Decorating a Button 121
Using a Decorator 122
Using Nonvisual Decorators 123
Decorated Code 124

The dataclass Decorator 125
Using dataclass with Default Values 126
Decorators, Adapters, and Composites 126
Consequences of the Decorator Pattern 126
Programs on GitHub 127

16 The Façade Pattern 129
Building the Façade Classes 131
Creating Databases and Tables 135
Using the SQLite Version 136
Consequences of the Façade 137
Programs on GitHub 137
Notes on MySQL 137
Using SQLite 138
References 138

17 The Flyweight Pattern 139
What Are Flyweights? 139
Example Code 140
Selecting a Folder 142
Copy-on-Write Objects 143
Program on GitHub 143

18 The Proxy Pattern 145
Using the Pillow Image Library 145
Displaying an Image Using PIL 146
Using Threads to Handle Image Loading 146
Logging from Threads 149
Copy-on-Write 149
Comparing Related Patterns 149
Programs on GitHub 150

19 Summary of Structural Patterns 151

IV: Behavioral Patterns 153

20 Chain of Responsibility Pattern 155
When to Use the Chain 156
Sample Code 156

The Listboxes 159
Programming a Help System 160
Receiving the Help Command 161
The First Case 162
A Chain or a Tree? 163
Kinds of Requests 164
Consequences of the Chain of Responsibility 164
Programs on GitHub 165

21 The Command Pattern 167
When to Use the Command Pattern 167
Command Objects 168
A Keyboard Example 168
Calling the Command Objects 170
Building Command Objects 171
The Command Pattern 172
Consequences of the Command Pattern 172
Providing the Undo Function 172
Creating the Red and Blue Buttons 175
Undoing the Lines 175
Summary 176
References 176
Programs on GitHub 176

22 The Interpreter Pattern 177
When to Use an Interpreter 177
Where the Pattern Can Be Helpful 177
A Simple Report Example 178
Interpreting the Language 179
How Parsing Works 180
Sorting Using attrgetter() 181
The Print Verb 182
The Console Interface 182
The User Interface 183
Consequences of the Interpreter Pattern 184
Programs on GitHub 185

23 The Iterator Pattern 187
Why We Use Iterators 187
Iterators in Python 187
A Fibonacci Iterator 188
Getting the Iterator 189
Filtered Iterators 189
The Iterator Generator 191
A Fibonacci Iterator 191
Generators in Classes 192
Consequences of the Iterator Pattern 192
Programs on GitHub 193

24 The Mediator Pattern 195
An Example System 195
Interactions Between Controls 197
Sample Code 198
Mediators and Command Objects 199
Consequences of the Mediator Pattern 200
Single Interface Mediators 200
Programs on GitHub 201

25 The Memento Pattern 203
When to Use a Memento 203
Sample Code 204
Consequences of the Memento Pattern 209
Programs on GitHub 209

26 The Observer Pattern 211
Example Program for Watching Colors Change 212
The Message to the Media 215
Consequences of the Observer Pattern 215
Programs on GitHub 215

27 The State Pattern 217
Sample Code 217
Switching Between States 221
How the Mediator Interacts with the StateManager 222
Consequences of the State Pattern 224

State Transitions 224
Programs on GitHub 224

28 The Strategy Pattern 225
Why We Use the Strategy Pattern 225
Sample Code 226
The Context 227
The Program Commands 227
The Line and Bar Graph Strategies 228
Consequences of the Strategy Pattern 230
Programs on GitHub 231

29 The Template Pattern 233
Why We Use Template Patterns 233
Kinds of Methods in a Template Class 234
Sample Code 234
Drawing a Standard Triangle 235
Drawing an Isosceles Triangle 236
The Triangle Drawing Program 237
Templates and Callbacks 238
Summary and Consequences 238
Example Code on GitHub 238

30 The Visitor Pattern 239
When to Use the Visitor Pattern 239
Working with the Visitor Pattern 241
Sample Code 241
Visiting Each Class 242
Visiting Several Classes 242
Bosses Are Employees, Too 243
Double Dispatching 245
Traversing a Series of Classes 245
Consequences of the Visitor Pattern 245
Example Code on GitHub 245

V: A Brief Introduction to Python 247

31 Variables and Syntax in Python 249

Data Types 250
Numeric Constants 250
Strings 250
Character Constants 251
Variables 252
Complex Numbers 253
Integer Division 253
Multiple Equal Signs for Initialization 254
A Simple Python Program 254
Compiling and Running This Program 255
Arithmetic Operators 255
Bitwise Operators 255
Combined Arithmetic and Assignment Statements 256
Comparison Operators 256
The input Statement 257
PEP 8 Standards 258
Variable and Function Names 258
Constants 258
Class Names 258
Indentation and Spacing 259
Comments 259
Docstrings 259
String Methods 260
Examples on GitHub 261

32 Making Decisions in Python 263

elif is “else if” 263
Combining Conditions 264
The Most Common Mistake 264
Looping Statements in Python 265
The for Loop and Lists 265
Using range in if Statements 266
Using break and continue 266
The continue Statement 267
Python Line Length 267

The print Function 267
Formatting Numbers 268
C and Java Style Formatting 269
The format string Function 269
f-string Formatting 269
Comma-Separated Numbers 270
Strings 270
Formatting Dates 271
Using the Python match Function 271
Pattern Matching 272
Reference 273
Moving On 273
Sample Code on GitHub 273

33 Development Environments 275
IDLE 275
Thonny 275
PyCharm 276
Visual Studio 276
Other Development Environments 276
LiClipse 276
Jupyter Notebook 277
Google Colaboratory 277
Anaconda 277
Wing 278
Command-Line Execution 278
CPython, IPython, and Jython 278

34 Python Collections and Files 279
Slicing 279
Slicing Strings 280
Negative Indexes 281
String Prefix and Suffix Removal 281
Changing List Contents 281
Copying a List 282
Reading Files 282
Using the with Loop 283
Handling Exceptions 284

Using Dictionaries 284
Combining Dictionaries 286
Using Tuples 286
Using Sets 287
Using the map Function 287
Writing a Complete Program 288
Impenetrable Coding 288
Using List Comprehension 289
Sample Programs on GitHub 290

35 Functions 291
Returning a Tuple 292
Where Does the Program Start? 292
Summary 293
Programs on GitHub 293

A Running Python Programs 295
If You Have Python Installed 295
Shortcuts 295
Creating an Executable Python Program 296
Command-Line Arguments 297

Index 299

Preface

When I began studying Python, I was impressed by how simple coding was and how easy it was to get started writing basic programs. I tried several development environments, and in all cases, I was able to get simple programs running in moments.

The Python syntax was simple, and there were no brackets or semicolons to remember. Other than remembering to use the Tab key (to generate those four-space indentations), coding in Python was easy.

But it was only after I played with Python for a few weeks that I began to see how sophisticated the language really is and how much you can really do with it. Python is a fully object-oriented language, making it easy to create classes that hold their own data without a lot of syntactic fussing.

In fact, I started trying to write some programs that I had written years ago in Java, and I was amazed by how much simpler they were in Python. And with the powerful IDEs, it was hard to make many mistakes.

When I realized how much I could get done quickly in Python, I also realized that it was time to write a book about powerful programs you can write in Python. This led to my writing new, clean, readable versions of the 23 classic design patterns that I had originally coded some years before.

The result is this book, which illustrates the basics of object-oriented programming, visual programming, and how to use all of the classic patterns. You can find complete working code for all these programs on GitHub at https://github.com/jwcnmr/jameswcooper/tree/main/Pythonpatterns.

This book is designed to help Python programmers broaden their knowledge of object-oriented programming (OOP) and the accompanying design patterns.

- If you are new to Python but have experience in other languages, you will be able to charge ahead by reviewing Chapter 31 through Chapter 35 and then starting back at Chapter 1.
- If you are experienced in Python but want to learn about OOP and design patterns, start at Chapter 1. If you like, you can skip Chapter 2 and Chapter 3 and go right through the rest of the book.
- If you are new to programming in general, spend some time going over Chapter 31 through 35 to try some of the programs. Then start on Chapter 1 to learn about OOP and design patterns.

You will likely find that Python is the easiest language you ever learned, as well as the most effortless language for writing the objects you use in design patterns. You'll see what they are for and how to use them in your own work.

In any case, the object-oriented programming methods presented in these pages can help you write better, more reusable program code.

Book Organization

This book is organized into five parts.

Part I, "Introduction"

Design patterns essentially describe how objects can interact effectively. This book starts by introducing objects in Chapter 1, "Introduction to Objects," and providing graphical examples that clearly illustrate how the patterns work.

Chapter 2, "Visual Programming in Python," and Chapter 3, "Visual Programming of Tables of Data," introduce the Python tkinter library, which gives you a way to create windows, buttons, lists, tables, and more with minimal complexity.

Chapter 4, "What Are Design Patterns?", begins the discussion of design patterns by exploring exactly what they are.

Part II, "Creational Patterns"

Part II starts by outlining the first group of patterns that the "Gang of Four" named Creational Patterns.

Chapter 5, "The Factory Pattern," describes the basic Factory pattern, which serves as the simple basis of the three factory patterns that follow. In this chapter, you create a Factory class that decides which of several related classes to use, based on the data itself.

Chapter 6, "The Factory Method Pattern," explores the Factory method. In this pattern, no single class makes the decision as to which subclass to instantiate. Instead, the superclass defers the decision to each subclass.

Chapter 7, "The Abstract Factory Pattern," discusses the Abstract Factory pattern. You can use this pattern when you want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several groups of classes.

Chapter 8, "The Singleton Pattern," looks at the Singleton pattern, which describes a class in which there can be no more than one instance. It provides a single global point of access to that instance. You don't use this pattern all that often, but it is helpful to know how to write it.

In Chapter 9, "The Builder Pattern," you see that the Builder pattern separates the construction of a complex object from its visual representation, so that several different representations can be created, depending on the needs of the program.

Chapter 10, "The Prototype Pattern," shows how to use the Prototype pattern when creating an instance of a class is time consuming or complex. Instead of creating more instances, you make copies of the original instance and modify them as appropriate.

Chapter 11, "Summary of Creational Patterns," just summarizes the patterns in Part II.

Part III, "Structural Patterns"

Part III begins with a short discussion of Structural Patterns.

Chapter 12, "The Adapter Pattern," examines the Adapter pattern, which is used to convert the programming interface of one class into that of another. Adapters are useful whenever you want unrelated classes to work together in a single program.

Chapter 13, "The Bridge Pattern," takes up the similar Bridge pattern, which is designed to separate a class's interface from its implementation. This enables you to vary or replace the implementation without changing the client code.

Chapter 14, "The Composite Pattern," delves into systems in which a component may be an individual object or may represent a collection of objects. The Composite pattern is designed to accommodate both cases, often in a treelike structure.

In Chapter 15, "The Decorator Pattern," we look at the Decorator pattern, which provides a way to modify the behavior of individual objects without having to create a new derived class. Although this can apply to visual objects such as buttons, the most common use in Python is to create a kind of macro that modifies the behavior of a single class instance.

In Chapter 16, "The Façade Pattern," we learn to use the Façade pattern to write a simplifying interface to code that otherwise might be unduly complex. This chapter deals with such an interface to a couple of different databases.

Chapter 17, "The Flyweight Pattern," describes the Flyweight pattern, which enables you to reduce the number of objects by moving some of the data outside the class. You can consider this approach when you have multiple instances of the same class.

Chapter 18, "The Proxy Pattern," looks at the Proxy pattern, which is used when you need to represent an object that is complex or time consuming to create, by a simpler one. If creating an object is expensive in time or computer resources, Proxy enables you to postpone creation until you need the actual object.

Chapter 19, "Summary of Structural Patterns," summarizes these Structural patterns.

Part IV, "Behavioral Patterns"

Part IV outlines the Behavioral Patterns.

Chapter 20, "Chain of Responsibility Pattern," looks at how the Chain of Responsibility pattern allows a decoupling between objects by passing a request from one object to the next in a chain until the request is recognized.

Chapter 21, "The Command Pattern," shows how the Command pattern uses simple objects to represent the execution of software commands. Additionally, this pattern enables you to support logging and undoable operations.

Chapter 22, "The Interpreter Pattern," looks at the Interpreter pattern, which provides a definition of how to create a little execution language and include it in a program.

In Chapter 23, “The Iterator Pattern,” we explore the well-known Iterator pattern, which describes the formal ways you can move through a collection of data items.

Chapter 24, “The Mediator Pattern,” takes up the important Mediator pattern. This pattern defines how communication between objects can be simplified by using a separate object to keep all objects from having to know about each other.

Chapter 25, “The Memento Pattern,” saves the internal state of an object, so you can restore it later.

In Chapter 26, “The Observer Pattern,” we look at the Observer pattern, which enables you to define the way a number of objects can be notified of a change in a program state.

Chapter 27, “The State Pattern,” describes the State pattern, which allows an object to modify its behavior when its internal state changes.

Chapter 28, “The Strategy Pattern,” describes the Strategy pattern, which, like the State pattern, switches easily between algorithms without any monolithic conditional statements. The difference between the State and Strategy patterns is that the user generally chooses which of several strategies to apply.

In Chapter 29, “The Template Pattern,” we look at the Template pattern. This pattern formalizes the idea of defining an algorithm in a class but leaves some of the details to be implemented in subclasses. In other words, if your base class is an abstract class, as often happens in these design patterns, you are using a simple form of the Template pattern.

Chapter 30, “The Visitor Pattern,” explores The Visitor pattern, which turns the tables on the object-oriented model and creates an external class to act on data in other classes. This is useful if there are a fair number of instances of a small number of classes and you want to perform some operation that involves all or most of them.

Part V, “A Brief Introduction to Python”

In this last section of the book, we provide a succinct summary of the Python language. If you are only passingly familiar with Python, this will get you up to speed. It is sufficiently thorough to instruct beginner as well.

In Chapter 31, “Variables and Syntax in Python,” we review the basic Python variables and syntax, and in Chapter 32, “Making Decisions in Python,” we illustrate the ways your programs can make decisions.

In Chapter 33, “Development Environments,” we provide a short summary of the most common development environments, and in Chapter 34, “Python Collections and Files,” we discuss arrays and files.

Finally in Chapter 35, “Functions,” we take up how to use functions on Python.

Enjoy writing design patterns and learning the ins and outs of the powerful Python language!

Register Your Book

Acknowledgments

I must start by thanking the late John Vlissides, one of the original "Gang of Four," for his clear explanations of several points about these design patterns. He worked just a few doors down from me at IBM Research and didn't mind my dropping in for a chat about patterns from time to time.

I also really appreciated early supportive comments from Arianne Dee and Ausif Mahmood, as well as Vaughn Cooper.

Of course, my editor, Debra J. Williams, has been both supportive and creative in helping me bring this project to fruition, as have the reviewers, Nick Cohron and Regina R. Monaco. And from a development point of view, Chris Zahn has been terrific.

I hope you enjoy writing patterns in Python as much as I have.

James Cooper
Wilton, CT
July 2021

About the Author

James W. Cooper holds a PhD in chemistry and worked in academia, for the scientific instrument industry, and for IBM for 25 years, primarily as a computer scientist at IBM's Thomas J. Watson Research Center. Now retired, he is the author of 20 books, including 3 on design patterns in various languages. His most recent books are *Flameout: The Rise and Fall of IBM Instruments* (2019) and *Food Myths Debunked* (2014).

James holds 11 patents and has written 60 columns for *JavaPro Magazine*. He has also written nearly 1,000 columns for the now vanished Examiner.com on foods and chemistry, and he currently writes his own blog: FoodScienceInstitute.com. Recently, he has written columns on Python for Medium.com and Substack.

He is also involved in local theater groups and is the treasurer for Troupers Light Opera, where he performs regularly.

This page intentionally left blank

4

What Are Design Patterns?

Sitting at your desk in front of your workstation, you stare into space, trying to figure out how to write a new program feature. You know intuitively what must be done, what data and what objects come into play, but you have this underlying feeling that there is a more elegant and general way to write this program.

In fact, you probably don't write any code until you can build a picture in your mind of what the code does and how the pieces of the code interact. The more you can picture this "organic whole," the more likely you are to feel comfortable that you have developed the best solution to the problem. If you don't grasp this whole right away, you might keep staring out the window for a time, even though the basic solution to the problem is quite obvious.

In one sense, you feel that the most elegant solution will be more reusable and more maintainable, but even if you are the sole likely programmer, you feel reassured when you have designed a solution that is relatively elegant and doesn't expose too many internal inelegancies.

One of the main reasons computer science researchers began to recognize design patterns is to satisfy this need for elegant but simple reusable solutions. The term *design patterns* sounds a bit formal to the uninitiated and can be somewhat off-putting when you first encounter it. But, in fact, design patterns are just convenient ways of reusing object-oriented code between projects and programmers. The idea behind design patterns is simple: to write down and catalog common interactions between objects that programmers have frequently found useful.

One frequently cited pattern from early literature on programming frameworks is the Model-View-Controller framework for Smalltalk (Krasner and Pope, 1988), which divided the user interface problem into three parts. The parts were referred to as a *data model*, containing the computational parts of the program; the *view,* which presents the user interface; and the c*ontroller*, which interacts between the user and the view (see Figure 4-1).

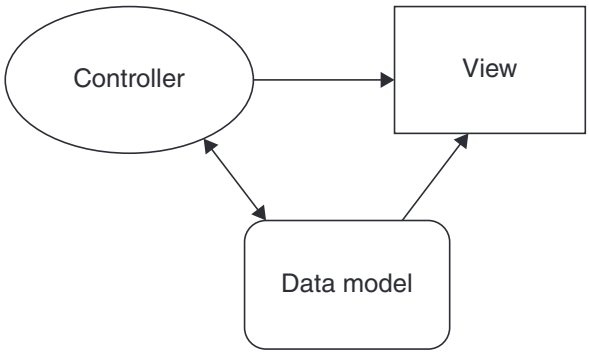

Figure 4-1 Model-View-Controller illustration

Each of these aspects of the problem is a separate object, and each has its own rules for managing its data. Communication among the user, the GUI, and the data should be carefully controlled, and this separation of functions accomplished that very nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful design pattern.

In other words, design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming. If you have been trying to keep objects minding their own business, you are probably already using some of the common design patterns.

Design patterns started to be recognized more formally in the early 1990s by Erich Gamma,[1] who described patterns incorporated in the GUI application framework ET++. The culmination of these discussions and a number of technical meetings was the book *Design Patterns: Elements of Reusable Software,* by Gamma, Helm, Johnson, and Vlissides.[2] This best-selling book, commonly referred to as the Gang of Four, or "GoF" book, has had a powerful impact on programmers seeking to understand how to use design patterns. It describes 23 commonly occurring and generally useful patterns and comments on how and when you might apply them. Throughout the following chapters, we refer to this groundbreaking book as *Design Patterns*.

Since the publication of the original *Design Patterns*, many other useful books have been published. These include our popular *Java Design Patterns: A Tutorial*[3] and an analogous book on C# design patterns.[4] Rhodes[5] maintains an interesting site describing how Python can make use of design patterns, as well.

Defining Design Patterns

We all talk about the way we do things in our everyday work, hobbies, and home life, and recognize repeating patterns all the time:

- Sticky buns are like dinner rolls, but I add brown sugar and nut filling to them.
- Her front garden is like mine, but in mine I use *astilbe*.
- This end table is constructed like that one, but in this one, the doors replace drawers.

We see the same thing in programming, when we tell a colleague how we accomplished a tricky bit of programming so that they don't have to re-create it from scratch. We simply recognize effective ways for objects to communicate while maintaining their own separate existences.

To summarize:

Design patterns are frequently used algorithms that describe convenient ways for classes to communicate.

It has become apparent that you don't just *write* a design pattern off the top of your head. In fact, most such patterns are *discovered* rather than written. The process of looking for these patterns is called *pattern mining* and is worthy of a book of its own.

The 23 design patterns selected for inclusion in the original *Design Patterns* book were patterns that had several known applications and were on a middle level of generality, where they could easily cross application areas and encompass several objects.

The authors divided these patterns into three types: creational, structural, and behavioral.

- *Creational patterns* create objects for you instead of having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- *Behavioral patterns* help you define the communication between objects in your system and control the flow in a complex program.

The Learning Process

We have found that learning design patterns is a multiple-step process:

1. Acceptance
2. Recognition
3. Internalization

First, you accept the premise that design patterns are important in your work. Then you recognize that you need to read about design patterns in order to determine when you might use them. Finally, you internalize the patterns in sufficient detail that you know which ones might help you solve a given design problem.

For some lucky people, design patterns are obvious tools, and they grasp their essential utility just by reading summaries of the patterns. For many of the rest of us, there is a slow induction period after we've read about a pattern, followed by the proverbial "Aha!" when we see how we can apply them in our work. These chapters help take you to that final stage of internalization by providing complete, working programs that you can try out for yourself.

The examples in *Design Patterns* are brief and are written in either C++ or, in some cases, Smalltalk. If you are working in another language, it is helpful to have the pattern examples in your language of choice. This part of the book attempts to fill that need for Python programmers.

Notes on Object-Oriented Approaches

The fundamental reason for using design patterns is to keep classes separated and prevent them from having to know too much about one another. Equally important, using these patterns helps you avoid reinventing the wheel and enables you to describe your programming approach succinctly in terms other programmers can easily understand.

There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance. Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all the methods of that parent class. It also has access to all its variables. However, by starting your inheritance hierarchy with a complete, working class, you might be unduly restricting yourself as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggests that you always

Program to an interface and not to an implementation.

Putting this more succinctly, you should define the top of any class hierarchy with an *abstract* class or an *interface*, which implements no methods but simply defines the methods that class will support. Then in all your derived classes, you have more freedom to implement these methods as best suits your purposes.

Python does not directly support interfaces, but it does let you write abstract classes, where the methods have no implementation. Remember the `comd` interface to the `DButton` class:

```
class DButton(Button):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        super().config(command=self.comd)

    # abstract method to be called by children
    def comd(self): pass
```

This is a good example of an abstract class. Here you fill in the code for the command method in the derived button classes. As you will see, it is also an example of the Command design pattern.

The other major concept you should recognize is *object composition*. We have already seen this approach in the `Statelist` examples. Object composition is simply the construction of objects that contain others—the encapsulation of several objects inside another one. Many beginning OO programmers tend to use inheritance to solve every problem, but as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that works best for what you want to accomplish without having all the methods of the parent classes. Thus, the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

At first this seems contrary to the customs of OO programming, but you will see any number of cases among the design patterns where we find that inclusion of one or more objects inside another is the preferred method.

Python Design Patterns

The following chapters discuss each of the 23 design patterns featured in the *Design Patterns* book, along with at least one working program example for that pattern. The programs have some sort of visual interface as well to make them more immediate to you.

Which design patterns are most useful? This depends on the individual programmer. The ones we use the most are Command, Factory, Decorator, Façade, and Mediator, but we have used nearly every one at some point.

References

1. Erich Gamma, *Object-Oriented Software Development based on ET++,* (in German) (Springer-Verlag, Berlin, 1992).
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995).
3. James Cooper, *Java Design Patterns: A Tutorial* (Boston: Addison-Wesley: 2000).
4. James Cooper, *C# Design Patterns: A Tutorial* (Boston: Addison-Wesley, 2003).
5. Brandon Rhodes, "Python Design Patterns," https://python-patterns.guide.

Index

Symbols

= (equal signs)
- assignment (=) operator, 264
- initialization, 254
- is equal to (==) operator, 264
- spacing, 259

A

Abstract Factory pattern, 75, 95
- consequences of, 77–78
- GardenMaker Factory, 75–77
- GitHub programs, 78
- purposes of, 77–78
- thought questions, 78
- user interfaces, 77

access (privileged), iterators, 192–193

accessor methods, 6

accounts (GitHub), setting up, 3

Adapter pattern, 99
- Class adapter, 103
- creating adapters, 101–102
- GitHub programs, 103
- moving data between lists, 99–101
- pluggable adapters, 103
- two-way adapters, 103

adapters
- Class adapter, 103
- creating, 101–102
- pluggable adapters, 103
- two-way adapters, 103

adding
menus, to windows, 35–38
two numbers, visual programming, 26
Anaconda, 277
AND operator, 256
arithmetic operators, 255, 259
arithmetic/assignment statements, combined, 256
arrays
lists, 265
range function, 265
assignment (=) operator, 264
attrgetter operator, sorting, 181–182

B

bar graphs, Line and Bar Graph strategies, 228–230
behavioral patterns, 55, 153
Chain of Responsibility pattern, 155–156
consequences of, 164–165
first cases, 162
GitHub programs, 165
help systems, programming, 160–161
help systems, tree structures, 163–164
listboxes, 159–160
receiving help command, 161–162
requests, 164
sample code, 156–159
using, 156
Command pattern, 167, 176
ButtonCommand objects, 175
buttons, creating, 175
buttons, Undo button, 175–176
Command objects, 168
Command objects, building, 171–172
Command objects, calling, 170–171
Command objects, containers, 172
Command objects, mediators, 199–200
consequences of, 172
GitHub programs, 176
keyboard example, 168–170
KeyModerator class, 169, 170
references, 176
Undo button, 175–176
Undo function, 172–175
Interpreter pattern, 177
attrgetter operator, sorting, 181–182
consequences of, 184–185
console interface, 182–183
GitHub programs, 185
languages, 179–180
parsing, 180–181
Print verb, 182
report generator example, 178–179
usefulness of, 177–178
user interfaces, 183–184
using, 177
Variable class, 181
Verb class, 181
Iterator pattern, 187
calling iterators, 189
composites and iterators, 193
consequences of, 192–193
creating iterators, 189
dunder methods, 188
external iterators, 193
Fibonacci iterators, 188–189, 191–192
filtered iterators, 189–191
generators in classes, 192
getting iterators, 189
GitHub programs, 193
internal iterators, 193

iterable containers, 188
iterator generator, 191
for loops as iterators, 187–188
modifying data, 192
privileged access, 192–193
using, 187
Mediator pattern, 195
Command objects, 199–200
consequences of, 200
example system, 195–197
GitHub programs, 201
interactions between controls, 197
sample code, 198–199
single interface mediators, 200–201
Memento pattern, 203
Caretaker class, 208–209
consequences of, 209
GitHub programs, 209
graphics drawing program example, 204–209
sample code, 204–209
using, 203–204
Observer pattern, 211–212
color changing program example, 212–214
consequences of, 215
GitHub programs, 215
messages to the media, 215
notifications, 215
State pattern, 217
consequences of, 224
GitHub programs, 224
graphics drawing program example, 217–220
sample code, 217–220
StateManager, mediator interactions, 222–223
StateManager, switching between states, 221
switching between states, 221
transitions, 224
Strategy pattern, 225
consequences of, 230–231
Context class, 227
GitHub programs, 231
Line and Bar Graph strategies, 228–230
PlotStategy class, 226–227
program commands, 227–228
sample code, 226–227
using, 225–226
Template pattern, 233
callbacks, 238
concrete methods, 234
consequences of, 238
empty methods, 234
GitHub programs, 238
hook methods, 234
isosceles triangles, 236
IsoscelesTriangle class, 236
Point class, 234
sample code, 234–235
standard triangles, 235
stdTriangle class, 235
summary of, 238
Template class, 234
Triangle class, 234–235, 236
Triangle Drawing program, 237
using, 233
Visitor pattern, 239, 241
Boss class, 242–243
BVacationVisitor class, 243–244
consequences of, 245
double dispatching, 245
GitHub programs, 245
sample code, 241–242
traversing a series of classes, 245

using, 239–240
visiting classes, 242
visiting several classes, 242–243
bitwise operators, 255–256
blank lines, classes/functions, 259
Blue button, creating, 175
Boolean variables, 250
Boss class
Composite pattern, 113
Visitor pattern, 242–243
break statements, 266
Bridge pattern, 105–107
consequences of, 109–110
creating user interfaces, 107
extending bridges, 107–109
GitHub programs, 110
Listbox builder, 105–106
Treeview widget, 105, 106, 107, 108–109
bridges, extending, 107–109
Builder pattern, 83–84, 95
calling builders, 86–87
Checkbox builder, 88
consequences of, 89
GitHub programs, 89
investment trackers, 84–86
Listbox builder, 87–88
selected securities, displaying, 89
thought questions, 89
ButtonCommand objects, Command pattern, 175
buttons
Blue button, creating, 175
checkbuttons, grid layouts, 32–34
creating
Command pattern, 175
object-oriented programming, 19–21
visual programming, 17–19
Decorator pattern, 121–122
Hello buttons, 17
Quit buttons, 17–18
radio buttons, 27–29
Red button, 175
Undo button, Command pattern, 175–176
BVacationVisitor class, Visitor pattern, 243–244
byte codes, 2, 278

C

C style formatting, 269
caching results, Composite pattern, 120
callbacks, Template pattern, 238
calling
builders, Builder pattern, 86–87
Command objects, 170–171
functions, 293
iterators, 189
CamelCase, 258
Caretaker class, Memento pattern, 208–209
case (upper/lower)
CamelCase, 258
classes, 258
constants, 258
functions, 258–259
variables, 249, 252
catching errors, 26–27
Chain of Responsibility pattern, 155–156
consequences of, 164–165
first cases, 162
GitHub programs, 165
help command, receiving, 161–162
help systems
programming, 160–161
tree structures, 163–164

listboxes, 159–160
requests, 164
sample code, 156–159
using, 156
character constants, 251–252
check boxes, disabling, 34
Checkbox builder, 88
checkbuttons, grid layouts, 32–34
CircleSeeding class, 71–72
Class adapter, 103
Class_init_Method, 6
classes, 5–6
blank lines, 259
Boss class
Composite pattern, 113
Visitor pattern, 242–243
BVacationVisitor class, Visitor pattern, 243–244
CamelCase, 258
Caretaker class, Memento pattern, 208–209
collections, 7
communicating between, 30
Context class, Strategy pattern, 227
creating, 7
Database class, Facade pattern, 132–133
Decorator class, 121–123
derived classes, 8
Docstrings, 259
Employee class, Composite pattern, 112–113
Facade classes, building, 131–135
Factory classes, operation of, 61–62
Factory Method pattern
CircleSeeding class, 71–72
Event classes, 69–70
Straight Seeding class, 70–71
Swimmer class, 68–69
generators, 192
indentation, 259
inheritance, 8–9
instances
creating, 6
Singleton pattern, 80–81
IsoscelesTriangle class, Template pattern, 236
KeyModerator class, Command pattern, 169, 170
naming conventions, 258
PlotStategy class, Strategy pattern, 226–227
Point class, Template pattern, 234
Query class, Facade pattern, 133–134
Results class, Facade pattern, 134
static classes, Singleton pattern, 81
stdTriangle class, Template pattern, 235
subclasses, Factory pattern, 62–63
Table class, Facade pattern, 133
Template class, methods, 234
traversing a series of classes, Visitor pattern, 245
Triangle class, Template pattern, 234–235, 236
Variable class, Interpreter pattern, 181
variables, 6–7, 30
Verb class, Interpreter pattern, 181
visiting, 242
Visitor class, 241–242
cloning, 91–92
coding, impenetrable, 288
collections
classes, 7
dictionaries
combining, 286
listing, 285
using, 284–285

GitHub programs, 290
lists
changing contents, 281–282
copying, 282
creating, 279
doubly linked lists, 117–118
moving data between, 99–101
slicing, 279
spacing, 259
sets, using, 287
tuples
returning, 292
using, 286

color
color changing program example, Observer pattern, 212–214
tkinter library, applying color with, 27

combining
arithmetic/assignment statements, 256
conditions, 264
dictionaries, 286

combo boxes, 46–47

Command pattern, 167, 176
ButtonCommand objects, 175
buttons
creating, 175
Undo button, 175–176
Command objects, 168
building, 171–172
calling, 170–171
containers, 172
mediators and, 199–200
consequences of, 172
GitHub programs, 176
keyboard example, 168–170
KeyModerator class, 169, 170
references, 176
Undo function, 172–175

command-line
arguments, 297–298
execution, 278

commands
Interp command, 180
program commands, Strategy pattern, 227–228
queuing, 168

comma-separated numbers, 270

comments
Docstrings, 259
indentation, 259
spacing, 259

common mistakes, Python decision making, 264

communicating between classes, 30

comparison operators, 256–257

compiling, simple Python program example, 255

complement operator, 250–256

complete Python programs, writing, 288

complex numbers, 253

Composite pattern, 111
Boss class, 113
caching results, 120
composite implementation, 112
consequences of, 118
doubly linked lists, 117–118
Employee class, 112–113
employee trees
building, 114
printing, 114–116
GitHub programs, 120
iterators, 193
leaf nodes, 112–113
recursive calls stet, 119
salary computation, 112
simple composites, 119
Treeviews of composites, 116–117

compound operators, spacing, 259
comprehension, lists, 289
concrete methods, 234
conditions, combining, 264
console interface, Interpreter pattern, 182–183
constants
case, upper/lower, 258
character constants, 251–252
named constants. *See* variables
naming conventions, 258
numeric constants, 250
containers
Command objects, 172
iterable containers, 188
Context class, Strategy pattern, 227
continue statements, 267
controls, interactions between, 197
copying, lists, 282
copy-on write objects
Flyweight pattern, 143
Proxy pattern, 149
CPython, 278
creational patterns, 55, 59
Abstract Factory pattern, 75, 95
consequences of, 77–78
GardenMaker Factory, 75–77
GitHub programs, 78
purposes of, 77–78
thought questions, 78
user interfaces, 77
Builder pattern, 83–84, 95
calling builders, 86–87
Checkbox builder, 88
consequences of, 89
displaying selected securities, 89
GitHub programs, 89
investment trackers, 84–86
Listbox builder, 87–88
thought questions, 89
Factory Method pattern, 67–68, 74
CircleSeeding class, 71–72
Event classes, 69–70
GitHub programs, 74
seeding program, 72–73
Straight Seeding class, 70–71
using, 74
Factory pattern, 61, 95
building, 63
Factory classes, 61–62
GitHub programs, 65
GUI, 64
math computations, 65
sample code, 62
subclasses, 62–63
thought questions, 66
using, 63–64
Prototype pattern, 91, 95
cloning, 91–92
consequences of, 94
GitHub programs, 94
using, 92–94
Singleton pattern, 79–80, 95
consequences of, 82
GitHub programs, 82
large programs, 81–82
static classes, 81
throwing exceptions, 80
summary of, 95

D

data modification, iterators, 192
data tables, 41–42
combo boxes, 46–47
listboxes, creating, 42–43
state data, displaying, 44–46

tree nodes, inserting in data tables, 50–51
Treeview widget, 47–49
data types, 250
Database class, Facade pattern, 132–133
database objects, Facade pattern, 129
databases, creating, 135–136
dataclass decorator, 125–126
dates, formatting, 271
DBObjects, Facade pattern, 134
decision making in Python
arrays, range function, 265
assignment (=) operator, 264
break statements, 266
combining conditions, 264
common mistakes, 264
continue statements, 267
elif (else if) statements, 263–264
format string function, 269
formatting
dates, 271
formatting, C style, 269
formatting, Java style, 269
f-string formatting, 269
numbers, 268, 270
strings, 270
GitHub programs, 273
if statements, 263
else clauses, 263
range function, 266
is equal to (==) operator, 264
line length, 267
lists, 265
looping statements, 265
for loops, 265
match function, 271–272
pattern matching, 272–273
print function, 267–268
declaring variables, 252
Decorator pattern, 121, 126
buttons, 121–122
consequences of, 126–127
dataclass decorator, 125–126
decorated code, 124–125
Decorator class, 121–123
GitHub programs, 127
nonvisual decorators, 123–124
def keyword, functions, 291, 293
derived classes, 8
design patterns, 57. *See also* separate entries
defined, 53, 54–55
learning process, 55–56
Model-View-Controller framework for SmallTalk, 53–54
object-oriented strategies, 56
objects over inheritance, 57
popularity of, 54
programming to interfaces, 56
resources, 54
development environments
Anaconda, 277
command-line execution, 278
CPython, 278
Google Colaboratory, 277
IDLE, 275
IPython, 278
Jupyter Notebook, 277
Jython, 278
LiClipse, 276–277
PyCharm, 276
Thonny, 275–276
Visual Studio, 276
Wing, 278
dictionaries
combining, 286
listing, 285
using, 284–285

disabling check boxes, 34
dispatching, double, 245
displaying
images with PIL, 146
selected securities, Builder pattern, 89
state data, 44–46
dividing integers, 253
Docstrings, 259
double dispatching, Visitor pattern, 245
doubly linked lists, Composite pattern, 117–118
downloading SQLite, 138
drawing
drawing program example
Memento pattern, 204–209
State pattern, 217–220
isosceles triangles, 236
rectangles/squares, 10–11
standard triangles, 235
Triangle Drawing program, 237
dunder methods, 188

E

elif (else if) statements, 263–264
else clauses, if statements, 263
Employee class, Composite pattern, 112–113
employee trees
building, 114
printing, 114–116
empty methods, 234
equal signs (=)
assignment (=) operator, 264
initialization, 254
is equal to (==) operator, 264
spacing, 259
errors
catching, 26–27
handling, 284
message boxes, creating, 21
Event classes, Factory Method pattern, 69–70
exceptions
handling, 284
throwing, Singleton pattern, 80
executable Python programs, creating, 296–297
extending bridges, 107–109
external iterators, 193

F

Facade pattern, 129–131
classes, building, 131–135
consequences of, 137
Database class, 132–133
database objects, 129
databases, creating, 135–136
DBObjects, 134
GitHub programs, 137
MySQL database, connections, 131
MySQL Workbench, 130
Query class, 133–134
Results class, 134
SQLite, 136
Table class, 133
tables
creating, 135–136
names, 134–135
Factory classes, operation of, 61–62
Factory Method pattern, 67–68, 74
CircleSeeding class, 71–72
Event classes, 69–70

GitHub programs, 74
seeding program, 72–73
Straight Seeding class, 70–71
using, 74
Factory pattern, 61, 95
building, 63
Factory classes, operation of, 61–62
GitHub programs, 65
GUI, 64
math computations, 65
sample code, 62
subclasses, 62–63
thought questions, 66
using, 63–64
Fibonacci iterators, 188–189, 191–192
file dialogs, 22–23
files
opening, 282
reading, 282–283
filtered iterators, 189–191
first cases, Chain of Responsibility pattern, 162
Flyweight pattern, 139
copy-on write objects, 143
example code, 140–142
flyweights, defined, 139
folders
as flyweights, 140–142
selecting, 142–143
GitHub programs, 143
folders
as flyweights, 140–142
selecting, Flyweight pattern, 142–143
for loops, 187–188, 265, 266
for statements, 283
format string function, 269
formatting
C style formatting, 269
f-string formatting, 269
Java style formatting, 269
numbers, 268, 270
strings, 270
frames, LabelFrame widget, 39–40
f-string formatting, 269
functions, 291
blank lines, 259
calling, 293
case, upper/lower, 258–259
def keyword, 291, 293
Docstrings, 259
format string function, 269
GitHub programs, 293
len function, strings, 251
map function, 287–288
masking function. *See* AND operator
match function, 271–272
naming conventions, 258–259
print function, 267–268
range function, arrays, 265
returning tuples, 292
starting Python programs, 292–293
Undo function, Command pattern, 172–175

G

GardenMaker Factory, 75–77
GitHub programs
Abstract Factory pattern, 78
account setup, 3
Adapter pattern, 103
Bridge pattern, 110
Builder pattern, 89
Chain of Responsibility pattern, 165
collections, 290
Command pattern, 176
Composite pattern, 120
Decorator pattern, 127
Facade pattern, 137

Factory Method pattern, 74
Factory pattern, 65
Flyweight pattern, 143
functions, 293
Interpreter pattern, 185
Iterator pattern, 193
Mediator pattern, 201
Memento pattern, 209
Observer pattern, 215
programs, 15
Prototype pattern, 94
Proxy pattern, 150
Python, decision making, 273
Python syntax, 261
Singleton pattern, 82
State pattern, 224
Strategy pattern, 231
Template pattern, 238
Visitor pattern, 245
visual programming
data tables, 51–52
examples, 40
Google Colaboratory, 277
graphics drawing program example
Memento pattern, 204–209
State pattern, 217–220
grid layouts, 30–34
GUI (Graphical User Interfaces), Factory pattern, 64

H

Hello buttons, creating, 17
help command, receiving, 161–162
help systems
programming, 160–161
tree structures, 163–164
hints, type, 13
hook methods, 234

I

icons, creating, 295–296
IDLE (Integrated Development and Learning Environment), 275
if statements, 263
else clauses, 263
range function, 266
images
displaying with PIL, 146
loading with threads, 146–148
PIL
displaying images, 146
using, 145–146
impenetrable coding, 288
importing
names to tkinter library, 19
tkinter library tools, 17
indentation
comments, 259
loops/classes, 259
statements, 263
indexes, negative, 281
inheritance, 8
multiple inheritance, 8–9
objects over inheritance, 57
initialization, equal signs (=), 254
input statements, 257
input (user), responding to, 25–26
inserting tree nodes in data tables, 50–51
installing
MySQL, 137
Python, 275
instances
classes, Singleton pattern, 80–81
creating, 6
integer division, 253
interactions between controls, Mediator pattern, 197

interfaces
programming to, 56
single interface mediators, 200–201
internal iterators, 193
Interp command, 180
Interpreter pattern, 177
attrgetter operator, sorting, 181–182
consequences of, 184–185
console interface, 182–183
GitHub programs, 185
languages, 179–180
parsing, 180–181
Print verb, 182
report generator example, 178–179
usefulness of, 177–178
user interfaces, 183–184
using, 177
Variable class, 181
Verb class, 181
investment trackers, 84–86
IPython, 278
is equal to (==) operator, 264
isosceles triangles, drawing, 236
IsoscelesTriangle class, Template pattern, 236
iterable containers, 188
Iterator pattern, 187
calling iterators, 189
composites and iterators, 193
consequences of, 192–193
creating iterators, 189
dunder methods, 188
external iterators, 193
Fibonacci iterators, 188–189, 191–192
filtered iterators, 189–191
generators in classes, 192
getting iterators, 189
GitHub programs, 193
internal iterators, 193
iterable containers, 188
iterator generator, 191
for loops as iterators, 187–188
modifying data, 192
privileged access, 192–193
using, 187

J

Java style formatting, 269
.jpg files, PIL
displaying images, 146
using, 145–146
Jupyter Notebook, 277
Jython, 278

K

keyboards, Command pattern, 168–170
KeyModerator class, Command pattern, 169, 170

L

LabelFrame widget, 39–40
languages, Interpreter pattern, 179–180
large programs, Singleton pattern, 81–82
layouts, 17
grid layouts, 30–34
pack layouts, 18–19, 23–24
leaf nodes, Composite pattern, 112–113
learning design patterns, 55–56
left/right shift operators, 256
len function, strings, 251
LiClipse, 276–277
Line and Bar Graph strategies, 228–230
line length in Python, 267
Listbox builder
Bridge pattern, 105–106
Builder pattern, 87–88

listboxes
Chain of Responsibility pattern, 159–160
creating, 42–43
lists, 265
changing contents, 281–282
comprehension, 289
copying, 282
creating, 279
dictionaries, 285
doubly linked lists, Composite pattern, 117–118
moving data between, 99–101
slicing, 279
spacing, 259
loading images with threads, 146–148
local variables, 13
logging from threads, 149
looping statements, 265
loops
break statements, 266
continue statements, 267
indentation, 259
for loops, 187–188, 265, 266
with loops, 283–284
for loops, as iterators,
lower/upper case
CamelCase, 258
classes, 258
constants, 258
functions, 258–259
variables, 249, 252

M

making decisions in Python
arrays, range function, 265
assignment (=) operator, 264
break statements, 266
combining conditions, 264
common mistakes, 264
continue statements, 267
elif (else if) statements, 263–264
format string function, 269
formatting
dates, 271
formatting, C style, 269
formatting, Java style, 269
f-string formatting, 269
numbers, 268, 270
strings, 270
GitHub programs, 273
if statements, 263
else clauses, 263
range function, 266
is equal to (==) operator, 264
line length, 267
lists, 265
looping statements, 265
for loops, 265
match function, 271–272
pattern matching, 272–273
print function, 267–268
map function, 287–288
masking function. *See* AND operator
match function, 271–272
matching patterns, 272–273
math computations, Factory pattern, 65
Mediator pattern, 195
Command objects, 199–200
consequences of, 200
example system, 195–197
GitHub programs, 201
interactions between controls, 197
sample code, 198–199
single interface mediators, 200–201
Memento pattern, 203
Caretaker class, 208–209
consequences of, 209
GitHub programs, 209

graphics drawing program example, 204–209
sample code, 204–209
using, 203–204
menus, adding to windows, 35–38
message boxes, 17
creating, 21–22
error message boxes, 21
warning message boxes, 21
messages to the media, Observer pattern, 215
methods, 5
accessor methods, 6
concrete methods, 234
dunder methods, 188
empty methods, 234
hook methods, 234
revised methods, class creation, 8
strings, 251, 260–261
Template methods, 234
mistakes (common), Python decision making, 264
Model-View-Controller framework for SmallTalk, 53–54
modifying data, iterators, 192
moving, data between lists, 99–101
multiple inheritance, 8–9
MySQL, 137
database connections, 131
installing, 137
PyCharm, 137
pymysql library, 137
MySQL Workbench, 130

N

named constants. *See* variables
naming conventions, 249, 252
classes, 258
constants, 258
functions, 258–259
variables, 258–259
negative indexes, 281
nodes (leaf), Composite pattern, 112–113
nonvisual decorators, 123–124
notifications, Observer pattern, 215
numbers
adding, visual programming, 26
comma-separated numbers, 270
complex numbers, 253
dates, formatting, 271
formatting, 268, 270, 271
numeric constants, 250

O

object-oriented programming
buttons, creating, 19–21
classes, 5–6
collections, 7
creating, 7
derived classes, 8
inheritance, 8
inheritance, multiple inheritance, 8–9
instances, 6
variables, 6–7
defined, 5
inheritance, 8
methods, 5, 8
polymorphism, 14
rectangles/squares, drawing, 10–11
types, 13
declaring, 13–14
hints, 13
variables
local variables, 13
properties, 13
visibility, 12

objects
ButtonCommand objects, Command pattern, 175
Command objects, 168
building, 171–172
calling, 170–171
containers, 172
mediators and, 199–200
copy-on write objects
Flyweight pattern, 143
Proxy pattern, 149
database objects, Facade pattern, 129
over inheritance, 57
Observer pattern, 211–212
color changing program example, 212–214
consequences of, 215
GitHub programs, 215
messages to the media, 215
notifications, 215
ODBC (Open Database Connectivity), 129
opening, files, 282
OR operator, 256
operators
AND operator, 256
arithmetic operators, 255, 259
assignment (=) operator, 264
bitwise operators, 255–256
comparison operators, 256–257
complement operator, 250–256
compound operators, spacing, 259
is equal to (==) operator, 264
left/right shift operators, 256
OR operator, 256
in strings, 251

P

pack layouts, 18–19, 23–24
parsing, Interpreter pattern, 180–181
pattern matching, 272–273
PEP 8 standards, 258
PIL (Pillow Image Library)
displaying images, 146
using, 145–146
PlotStategy class, Strategy pattern, 226–227
pluggable adapters, 103
Point class, Template pattern, 234
polymorphism, 14, 243
prefix/suffix removal, strings, 281
print function, 267–268
Print verb, Interpreter pattern, 182
printing employee trees, 114–116
privileged access, iterators, 192–193
program commands, Strategy pattern, 227–228
programming, help systems, 160–161
properties, 13
Prototype pattern, 91, 95
cloning, 91–92
consequences of, 94
GitHub programs, 94
using, 92–94
Proxy pattern, 145
comparing related patterns, 149–150
copy-on write objects, 149
GitHub programs, 150
PIL
displaying images, 146
using, 145–146
threads
loading images, 146–148
logging from, 149
PyCharm, 137, 276
pymysql library, 137
Python
arrays, lists, 265
classes

blank lines, 259
Docstrings, 259
indentation, 259
naming conventions, 258
command-line arguments, 297–298
comments
Docstrings, 259
indentation, 259
spacing, 259
complete programs, writing, 287–288
complex numbers, 253
constants
character constants, 251–252
named constants. *See* variables
naming conventions, 258
numeric constants, 250
data types, 250
decision making
assignment (=) operator, 264
break statements, 266
combining conditions, 264
common mistakes, 264
continue statements, 267
elif (else if) statements, 263–264
format string function, 269
formatting, C style, 269
formatting, dates, 271
formatting, f-string, 269
formatting, Java style, 269
formatting, numbers, 268, 270
formatting, strings, 270
GitHub programs, 273
if statements, 263
if statements, else clauses, 263
if statements, range function, 266
is equal to (==) operator, 264
line length, 267
lists, 265
looping statements, 265
for loops, 265
match function, 271–272
pattern matching, 272–273
print function, 267–268
range function, arrays, 265
development, 1–2
development environments
Anaconda, 277
command-line execution, 278
CPython, 278
Google Colaboratory, 277
IDLE, 275
IPython, 278
Jupyter Notebook, 277
Jython, 278
LiClipse, 276–277
PyCharm, 276
Thonny, 275–276
Visual Studio, 276
Wing, 278
dictionaries
listing, 285
using, 284–285
equal signs (=)
initialization with, 254
spacing, 259
executable programs, creating, 296–297
formatting, dates, 271
f-string formatting, 269
functions, 291
blank lines, 259
calling, 293
def keyword, 291, 293
Docstrings, 259
format string function, 269
GitHub programs, 293
map function, 287–288

match function, 271–272
naming conventions, 258–259
range function, arrays, 265
returning tuples, 292
starting Python programs, 292–293
GitHub programs
collections, 290
decision making, 273
syntax, 261
icons, creating, 295–296
installing, 275
integer division, 253
line length, 267
lists
changing contents, 281–282
comprehension, 289
copying, 282
slicing, 279
spacing, 259
loops
break statements, 266
continue statements, 267
for loops, 266
indentation, 259
with loops, 283–284
negative indexes, 281
operators, 250–256
AND operator, 256
arithmetic operators, 255, 259
bitwise operators, 255–256
comparison operators, 256–257
compound operators, 259
left/right shift operators, 256
OR operator, 256
pattern matching, 272–273
PEP 8 standards, 258
running programs, 295
sets, using, 287
shortcuts, creating, 295–296
simple program example, 254–255
starting programs, 292–293
statements
break statements, 266
combined arithmetic/assignment statements, 256
continue statements, 267
elif (else if) statements, 263–264
if statements, 263
if statements, else clauses, 263
if statements, range function, 266
indentation, 263
input statements, 257
looping statements, 265
for statements, 283
strings
formatting, 270
len function, 251
methods, 251, 260–261
in operators, 251
prefix/suffix removal, 281
representing, 250–251
slicing, 251, 280
tuples, using, 287
variables
Boolean variables, 250
declaring, 252
naming conventions, 249, 252, 258–259
reassigning values, 250
upper/lower case, 249, 252

Q

Query class, Facade pattern, 133–134
queuing commands, 168
Quit buttons, creating, 17–18

R

radio buttons, creating, 27–29
Radiobutton widget, 24–29
range function, arrays, 265
reading files, 282–283
reassigning variable values, 250
receiving help command, 161–162
rectangles/squares, drawing, 10–11
recursive calls stet, Composite pattern, 119
Red button, creating, 175
removing prefixes/suffixes from strings, 281
report generator example, Interpreter pattern, 178–179
requests, Chain of Responsibility pattern, 164
responding to user input, 25–26
Responsibility pattern, Chain of, 155–156
consequences of, 164–165
first cases, 162
GitHub programs, 165
help command, receiving, 161–162
help systems
programming, 160–161
tree structures, 163–164
listboxes, 159–160
requests, 164
sample code, 156–159
using, 156
Results class, Facade pattern, 134
returning tuples, 292
revised methods, class creation, 8

S

salary computation, Composite pattern, 112
securities (selected), displaying, 89
seeding program, Factory Method pattern, 72–73
selected securities, displaying, 89
selecting folders, Flyweight pattern, 142–143
series of classes, traversing, 245
sets, using, 287
shortcuts, creating, 295–296
simple composites, 119
Simple Factory pattern, 61
building, 63
Factory classes, operation of, 61–62
GitHub programs, 65
GUI, 64
math computations, 65
sample code, 62
subclasses, 62–63
thought questions, 66
using, 63–64
single interface mediators, 200–201
Singleton pattern, 79–80, 95
consequences of, 82
exceptions, throwing, 80
GitHub programs, 82
large programs, 81–82
static classes, 81
slicing
lists, 279
strings, 251, 280
SmallTalk, Model-View-Controller framework, 53–54
sorting, attrgetter operator, 181–182
spacing
arithmetic operators, 259
comments, 259
equal signs (=), 259
lists, 259
SQLite
downloading, 138
Facade pattern, 136

squares/rectangles, drawing, 10–11
standard triangles, drawing, 235
starting Python programs, 292–293
state data, displaying, 44–46
State pattern, 217
consequences of, 224
GitHub programs, 224
graphics drawing program example, 217–220
sample code, 217–220
StateManager
mediator interactions, 222–223
switching between states, 221
switching between states, 221
transitions, 224
StateManager
mediator interactions, 222–223
switching between states, 221
statements
break statements, 266
combined arithmetic/assignment statements, 256
continue statements, 267
elif (else if) statements, 263–264
for statements, 283
if statements, 263
else clauses, 263
range function, 266
indentation, 263
input statements, 257
looping statements, 265
switch statements. *See* match function
static classes, Singleton pattern, 81
stdTriangle class, Template pattern, 235
Straight Seeding class, Factory Method pattern, 70–71
Strategy pattern, 225
consequences of, 230–231
Context class, 227
GitHub programs, 231
Line and Bar Graph strategies, 228–230
PlotStategy class, 226–227
program commands, 227–228
sample code, 226–227
using, 225–226
strings
Docstrings, 259
format string function, 269
formatting, 270
f-string formatting, 269
len function, 251
methods, 251, 260–261
prefix/suffix removal, 281
representing, 250–251
slicing, 251, 280
structural patterns, 55, 59–97
Adapter pattern, 99
Class adapter, 103
creating adapters, 101–102
GitHub programs, 103
moving data between lists, 99–101
pluggable adapters, 103
two-way adapters, 103
Bridge pattern, 105–107
consequences of, 109–110
creating user interfaces, 107
extending bridges, 107–109
GitHub programs, 110
Listbox builder, 105–106
Treeview widget, 105, 106, 107, 108–109
Composite pattern, 111
Boss class, 113
building employee trees, 114
caching results, 120
composite implementation, 112
consequences of, 118

doubly linked lists, 117–118
Employee class, 112–113
GitHub programs, 120
iterators, 193
leaf nodes, 112–113
printing employee trees, 114–116
recursive calls stet, 119
salary computation, 112
simple composites, 119
Treeviews of composites, 116–117
Decorator pattern, 121, 126
buttons, 121–122
consequences of, 126–127
dataclass decorator, 125–126
decorated code, 124–125
Decorator class, 121–123
GitHub programs, 127
nonvisual decorators, 123–124
Facade pattern, 129–131
building classes, 131–135
consequences of, 137
creating databases, 135–136
creating tables, 135–136
Database class, 132–133
database objects, 129
DBObjects, 134
GitHub programs, 137
MySQL database connections, 131
MySQL Workbench, 130
Query class, 133–134
Results class, 134
SQLite, 136
Table class, 133
tables names, 134–135
Flyweight pattern, 139
copy-on write objects, 143
example code, 140–142
flyweights, defined, 139
folders as flyweights, 140–142
GitHub programs, 143
selecting folders, 142–143
Proxy pattern, 145
comparing related patterns, 149–150
copy-on write objects, 149
GitHub programs, 150
PIL, displaying images, 146
PIL, using, 145–146
threads, loading images, 146–148
threads, logging from, 149
summary of, 151

subclasses, Factory pattern, 62–63

suffix/prefix removal, 281

Swimmer class, Factory Method pattern, 68–69

switch statements. *See* match function

switching between states, 221

syntax, Python

classes
blank lines, 259
Docstrings, 259
indentation, 259
naming conventions, 258
comments
Docstrings, 259
spacing, 259
complex numbers, 253
constants
character constants, 251–252
named constants. *See* variables
naming conventions, 258
numeric constants, 250
data types, 250
equal signs (=)
initialization with, 254
spacing, 259

functions
blank lines, 259
Docstrings, 259
naming conventions, 258–259
GitHub programs, 261
integer division, 253
lists, spacing, 259
loops, indentation, 259
operators
AND operator, 256
arithmetic operators, 255, 259
bitwise operators, 255–256
comparison operators, 256–257
complement operator, 250–256
compound operators, 259
left/right shift operators, 256
OR operator, 256
PEP 8 standards, 258
Python, indentation, 259
statements
combined arithmetic/assignment statements, 256
indentation, 263
input statements, 257
strings
len function, 251
methods, 251, 260–261
in operators, 251
representing, 250–251
slicing, 251
variables
Boolean variables, 250
declaring, 252
naming conventions, 249, 252, 258–259
reassigning values, 250
upper/lower case, 249, 252

T

Table class, Facade pattern, 133
tables
creating, Facade pattern, 135–136
data tables, 41–42
combo boxes, 46–47
listboxes, creating, 42–43
state data, displaying, 44–46
tree nodes, inserting in data tables, 50–51
Treeview widget, 47–49
names, Facade pattern, 134–135
Template class, methods, 234
Template methods, 234
Template pattern, 233
callbacks, 238
concrete methods, 234
consequences of, 238
empty methods, 234
GitHub programs, 238
hook methods, 234
isosceles triangles, 236
IsoscelesTriangle class, 236
Point class, 234
sample code, 234–235
standard triangles, 235
stdTriangle class, 235
summary of, 238
Template class, 234
Triangle class, 234–235, 236
Triangle Drawing program, 237
using, 233
Thonny, 275–276
thought questions
Abstract Factory pattern, 78
Builder pattern, 89
Factory pattern, 66

threads
image loading, 146–148
logging from, 149
throwing exceptions, Singleton pattern, 80
tkinter library, 2
colors, applying, 27
importing names, 19
importing tools, 17
window setup, 17
tracking investments, 84–86
transitions, State pattern, 224
traversing a series of classes, Visitor pattern, 245
tree nodes, inserting in data tables, 50–51
Treeview widget, 47–49
Bridge pattern, 105, 106, 107, 108–109
Composite pattern, 116–117
Triangle class, Template pattern, 234–235, 236
Triangle Drawing program, 237
triangles, drawing
isosceles triangles, 236
standard triangles, 235
ttk libraries, 24
tuples
returning, 292
using, 286
two-way adapters, 103
types, 13
declaring, 13
hints, 13

U

Undo button, Command pattern, 175–176
Undo function, Command pattern, 172–175
upper/lower case
CamelCase, 258
classes, 258
constants, 258
functions, 258–259
variables, 249, 252
user input, responding to, 25–26
user interfaces
Abstract Factory pattern, 77
creating, 107
GUI, Factory pattern, 64
Interpreter pattern, 183–184

V

Variable class, Interpreter pattern, 181
variables
Boolean variables, 250
case, upper/lower, 249, 252
declaring, 252
inside classes, 6–7, 30
local variables, 13
naming conventions, 258–259
properties, 13
visibility, 12
Verb class, Interpreter pattern, 181
visibility of variables, 12
Visitor pattern, 239, 241
Boss class, 242–243
BVacationVisitor class, 243–244
consequences of, 245
double dispatching, 245
GitHub programs, 245
sample code, 241–242
traversing a series of classes, 245
using, 239–240
visiting
classes, 242
several classes, 242–243
visual programming
buttons
Hello buttons, 17
Quit buttons, 17–18
radio buttons, 27–29

check boxes, disabling, 34
classes
communicating between, 30
variables, 30
data tables, 41–42
combo boxes, 46–47
listboxes, 42–43
state data, displaying, 44–46
tree nodes, inserting, 50–51
Treeview widget, 47–49
errors, catching, 26–27
file dialogs, 22–23
Hello buttons, creating, 17
layouts, 17
grid layouts, 30–34
pack layouts, 18–19, 23–24
menus, adding to windows, 35–38
message boxes, 17
creating, 21–22
error message boxes, 21
warning message boxes, 21
numbers, adding, 26
Quit buttons, creating, 17–18
radio buttons, creating, 27–29
tkinter library
applying colors, 27
importing names, 19
importing tools, 17
window setup, 17
ttk libraries, 24
user input, responding to, 25–26
variables, inside classes, 30
widgets
grid layouts, 30–31
LabelFrame widget, 39–40
pack layout manager, options, 23–24
Radiobutton widget, 24–29
Treeview widget, 47–49
ttk libraries, 24
windows, adding menus to, 35–38
Visual Studio, 276

W - X - Y - Z

warning message boxes, creating, 21
widgets
grid layouts, 30–31
LabelFrame widget, 39–40
pack layout manager, options, 23–24
Radiobutton widget, 24–29
Treeview widget, 47–49
Bridge pattern, 105, 106, 107, 108–109
Composite pattern, 116–117
ttk libraries, 24
windows, adding menus, 35–38
Wing, 278
with loops, 283–284