



Mark Summerfield

Foreword by Doug Hellmann,  
Senior Developer, DreamHost

# Python in Practice

Create Better Programs Using  
Concurrency, Libraries, and Patterns

**Developer's Library**



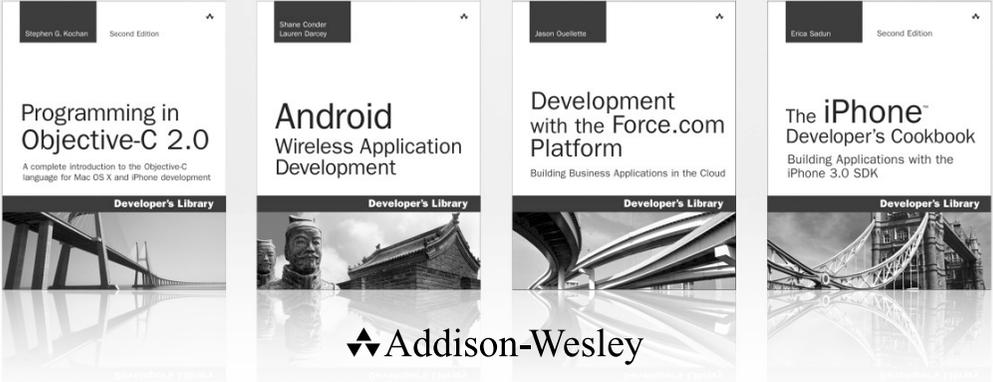
FREE SAMPLE CHAPTER

SHARE WITH OTHERS



# Python in Practice

# Developer's Library Series



Visit [developers-library.com](http://developers-library.com) for a complete list of available products

**T**he **Developer's Library Series** from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that's useful for other programmers.

Developer's Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

PEARSON

↕ Addison-Wesley

Cisco Press

EXAM/CRAM

IBM Press

que

PRENTICE HALL

SAMS

Safari  
BOOK ONLINE

# Python in Practice

Create Better Programs Using  
Concurrency, Libraries, and Patterns

Mark Summerfield

◆ Addison-Wesley

Upper Saddle River, NJ · Boston · Indianapolis · San Francisco  
New York · Toronto · Montreal · London · Munich · Paris · Madrid  
Capetown · Sydney · Tokyo · Singapore · Mexico City

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.

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.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales  
(800) 382-3419  
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales  
international@pearsoned.com

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: 2013942956

Copyright © 2014 Qtrac Ltd.

All rights reserved. Printed in the United States of America. 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-90563-5

ISBN-10: 0-321-90563-6

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.  
First printing, August 2013

*This book is dedicated to  
free and open-source software contributors  
everywhere—your generosity benefits us all.*

*This page intentionally left blank*

# Contents at a Glance

<b>Contents</b> .....	ix
<b>Foreword</b> .....	xiii
<b>Introduction</b> .....	1
<b>Chapter 1. Creational Design Patterns in Python</b> .....	5
<b>Chapter 2. Structural Design Patterns in Python</b> .....	29
<b>Chapter 3. Behavioral Design Patterns in Python</b> .....	73
<b>Chapter 4. High-Level Concurrency in Python</b> .....	141
<b>Chapter 5. Extending Python</b> .....	179
<b>Chapter 6. High-Level Networking in Python</b> .....	203
<b>Chapter 7. Graphical User Interfaces with Python and Tkinter</b>	231
<b>Chapter 8. OpenGL 3D Graphics in Python</b> .....	263
<b>Appendix A. Epilogue</b> .....	283
<b>Appendix B. Selected Bibliography</b> .....	285
<b>Index</b> .....	289

[www.qtrac.eu/pipbook.html](http://www.qtrac.eu/pipbook.html)

*This page intentionally left blank*

# Contents

<b>Foreword</b> .....	xiii
<b>Introduction</b> .....	1
Acknowledgments .....	3
<b>Chapter 1. Creational Design Patterns in Python</b> .....	5
1.1. Abstract Factory Pattern .....	5
1.1.1. A Classic Abstract Factory .....	6
1.1.2. A More Pythonic Abstract Factory .....	9
1.2. Builder Pattern .....	11
1.3. Factory Method Pattern .....	17
1.4. Prototype Pattern .....	24
1.5. Singleton Pattern .....	26
<b>Chapter 2. Structural Design Patterns in Python</b> .....	29
2.1. Adapter Pattern .....	29
2.2. Bridge Pattern .....	34
2.3. Composite Pattern .....	40
2.3.1. A Classic Composite/Noncomposite Hierarchy .....	41
2.3.2. A Single Class for (Non)composites .....	45
2.4. Decorator Pattern .....	48
2.4.1. Function and Method Decorators .....	48
2.4.2. Class Decorators .....	54
2.4.2.1. Using a Class Decorator to Add Properties .....	57
2.4.2.2. Using a Class Decorator Instead of Subclassing .....	58
2.5. Façade Pattern .....	59
2.6. Flyweight Pattern .....	64
2.7. Proxy Pattern .....	67
<b>Chapter 3. Behavioral Design Patterns in Python</b> .....	73
3.1. Chain of Responsibility Pattern .....	74
3.1.1. A Conventional Chain .....	74
3.1.2. A Coroutine-Based Chain .....	76
3.2. Command Pattern .....	79

3.3. Interpreter Pattern .....	83
3.3.1. Expression Evaluation with eval() .....	84
3.3.2. Code Evaluation with exec() .....	88
3.3.3. Code Evaluation Using a Subprocess .....	91
3.4. Iterator Pattern .....	95
3.4.1. Sequence Protocol Iterators .....	95
3.4.2. Two-Argument iter() Function Iterators .....	96
3.4.3. Iterator Protocol Iterators .....	97
3.5. Mediator Pattern .....	100
3.5.1. A Conventional Mediator .....	101
3.5.2. A Coroutine-Based Mediator .....	104
3.6. Memento Pattern .....	106
3.7. Observer Pattern .....	107
3.8. State Pattern .....	111
3.8.1. Using State-Sensitive Methods .....	114
3.8.2. Using State-Specific Methods .....	115
3.9. Strategy Pattern .....	116
3.10. Template Method Pattern .....	119
3.11. Visitor Pattern .....	123
3.12. Case Study: An Image Package .....	124
3.12.1. The Generic Image Module .....	125
3.12.2. An Overview of the Xpm Module .....	135
3.12.3. The PNG Wrapper Module .....	137
<b>Chapter 4. High-Level Concurrency in Python .....</b>	<b>141</b>
4.1. CPU-Bound Concurrency .....	144
4.1.1. Using Queues and Multiprocessing .....	147
4.1.2. Using Futures and Multiprocessing .....	152
4.2. I/O-Bound Concurrency .....	155
4.2.1. Using Queues and Threading .....	156
4.2.2. Using Futures and Threading .....	161
4.3. Case Study: A Concurrent GUI Application .....	164
4.3.1. Creating the GUI .....	165
4.3.2. The ImageScale Worker Module .....	173
4.3.3. How the GUI Handles Progress .....	175
4.3.4. How the GUI Handles Termination .....	177

<b>Chapter 5. Extending Python</b> .....	179
5.1. Accessing C Libraries with ctypes .....	180
5.2. Using Cython .....	187
5.2.1. Accessing C Libraries with Cython .....	188
5.2.2. Writing Cython Modules for Greater Speed .....	193
5.3. Case Study: An Accelerated Image Package .....	198
<b>Chapter 6. High-Level Networking in Python</b> .....	203
6.1. Writing XML-RPC Applications .....	204
6.1.1. A Data Wrapper .....	205
6.1.2. Writing XML-RPC Servers .....	208
6.1.3. Writing XML-RPC Clients .....	210
6.1.3.1. A Console XML-RPC Client .....	210
6.1.3.2. A GUI XML-RPC Client .....	214
6.2. Writing RPyC Applications .....	219
6.2.1. A Thread-Safe Data Wrapper .....	220
6.2.1.1. A Simple Thread-Safe Dictionary .....	221
6.2.1.2. The Meter Dictionary Subclass .....	224
6.2.2. Writing RPyC Servers .....	225
6.2.3. Writing RPyC Clients .....	227
6.2.3.1. A Console RPyC Client .....	227
6.2.3.2. A GUI RPyC Client .....	228
<b>Chapter 7. Graphical User Interfaces with Python and Tkinter</b> .....	231
7.1. Introduction to Tkinter .....	233
7.2. Creating Dialogs with Tkinter .....	235
7.2.1. Creating a Dialog-Style Application .....	237
7.2.1.1. The Currency Application's main() Function .....	238
7.2.1.2. The Currency Application's Main.Window Class .....	239
7.2.2. Creating Application Dialogs .....	244
7.2.2.1. Creating Modal Dialogs .....	245
7.2.2.2. Creating Modeless Dialogs .....	250
7.3. Creating Main-Window Applications with Tkinter .....	253
7.3.1. Creating a Main Window .....	255
7.3.2. Creating Menus .....	257
7.3.2.1. Creating a File Menu .....	258
7.3.2.2. Creating a Help Menu .....	259
7.3.3. Creating a Status Bar with Indicators .....	260

<b>Chapter 8. OpenGL 3D Graphics in Python</b> .....	263
8.1. A Perspective Scene .....	264
8.1.1. Creating a Cylinder with PyOpenGL .....	265
8.1.2. Creating a Cylinder with pyglet .....	270
8.2. An Orthographic Game .....	272
8.2.1. Drawing the Board Scene .....	275
8.2.2. Handling Scene Object Selection .....	277
8.2.3. Handling User Interaction .....	280
<b>Appendix A. Epilogue</b> .....	283
<b>Appendix B. Selected Bibliography</b> .....	285
<b>Index</b> .....	289

# Foreword to Python in Practice

I have been building software with Python for 15 years in various application areas. Over that time I have seen our community mature and grow considerably. We are long past the days of having to “sell” Python to our managers in order to be able to use it in work-related projects. Today’s job market for Python programmers is strong. Attendance at Python-related conferences is at an all time high, for regional conferences as well as the big national and international events. Projects like OpenStack are pushing the language into new arenas and attracting new talent to the community at the same time. As a result of the robust and expanding community, we have more and better options for books about Python than ever before.

Mark Summerfield is well known in the Python community for his technical writing about Qt and Python. Another of Mark’s books, *Programming in Python 3*, is at the top of my short list of recommendations for learning Python, a question I am asked frequently as the organizer of the user group in Atlanta, Georgia. This new book will also go on my list, but for a somewhat different audience.

Most programming books fall at either end of a spectrum that ranges from basic introductions to a language (or programming in general) to more advanced books on very focused topics like web development, GUI applications, or bioinformatics. As I was writing *The Python Standard Library by Example*, I wanted to appeal to readers who fall into the gap between those extremes—established programmers and generalists, both familiar with the language but who want to enhance their skills by going beyond the basics without being restricted to a specific application area. When my editor asked me to review the proposal for Mark’s book, I was pleased to see that *Python in Practice* is designed for the same types of readers.

It has been a long time since I have encountered an idea in a book that was immediately applicable to one of my own projects, without it being tied to a specific framework or library. For the past year I have been working on a system for metering OpenStack cloud services. Along the way, the team realized that the data we are collecting for billing could be useful for other purposes, like reporting and monitoring, so we designed the system to send it to multiple consumers by passing the samples through a pipeline of reusable transformations and publishers. At about the same time that the code for the pipeline was being finalized, I was also involved in the technical review for this book. After reading the first few sections of the draft for Chapter 3, it became clear that our pipeline implementation was much more complicated than necessary. The coroutine chaining technique Mark demonstrates is so much more elegant and easy to understand that

I immediately added a task to our roadmap to change the design during the next release cycle.

*Python in Practice* is full of similarly useful advice and examples to help you improve your craft. Generalists like me will find introductions to several interesting tools that may not have been encountered before. And whether you are already an experienced programmer or are making the transition out of the beginner phase of your career, this book will help you think about problems from different perspectives and give you techniques to create more effective solutions.

Doug Hellmann  
Senior Developer, DreamHost  
May 2013

# Introduction to Python in Practice

This book is aimed at Python programmers who want to broaden and deepen their Python knowledge so that they can improve the quality, reliability, speed, maintainability, and usability of their Python programs. The book presents numerous practical examples and ideas for improved Python programming.

The book has four key themes: design patterns for coding elegance, improved processing speeds using concurrency and compiled Python (*Cython*), high-level networking, and graphics.

The book *Design Patterns: Elements of Reusable Object-Oriented Software* (see the Selected Bibliography for details; ► 285) was published way back in 1995, yet still exerts a powerful influence over object-oriented programming practices. *Python in Practice* looks at all of the design patterns in the context of Python, providing Python examples of those that are useful, as well as explaining why some are irrelevant to Python programmers. These patterns are covered in Chapter 1, Chapter 2, and Chapter 3.

Python’s GIL (Global Interpreter Lock) prevents Python code from executing on more than one processor core at a time.\* This has led to the myth that Python can’t do threading or take advantage of multi-core hardware. For CPU-bound processing, concurrency can be done using the `multiprocessing` module, which is not limited by the GIL and can take full advantage of all the available cores. This can easily achieve the speedups we would expect (i.e., roughly proportional to the number of cores). For I/O-bound processing we can also use the `multiprocessing` module—or we can use the `threading` module or the `concurrent.futures` module. If we use threading for I/O-bound concurrency, the GIL’s overhead is usually dominated by network latency and so may not be an issue in practice.

Unfortunately, low- and medium-level approaches to concurrency are very error-prone (in any language). We can avoid such problems by avoiding the use of explicit locks, and by making use of Python’s high-level queue and `multiprocessing` modules’ queues, or the `concurrent.futures` module. We will see how to achieve significant performance improvements using high-level concurrency in Chapter 4.

Sometimes programmers use C, C++, or some other compiled language because of another myth—that Python is slow. While Python is in general slower than compiled languages, on modern hardware Python is often more than fast

---

\*This limitation applies to CPython—the reference implementation that most Python programmers use. Some Python implementations don’t have this constraint, most notably, Jython (Python implemented in Java).

enough for most applications. And in those cases where Python really isn't fast enough, we can still enjoy the benefits of programming in Python—and at the same time have our code run faster.

To speed up long-running programs we can use the PyPy Python interpreter ([pypy.org](http://pypy.org)). PyPy has a just-in-time compiler that can deliver significant speedups. Another way to increase performance is to use code that runs as fast as compiled C; for CPU-bound processing this can comfortably give us 100× speedups. The easiest way to achieve C-like speed is to use Python modules that are already written in C under the hood: for example, use the standard library's array module or the third-party numpy module for incredibly fast and memory-efficient array processing (including multi-dimensional arrays with numpy). Another way is to profile using the standard library's cProfile module to discover where the bottlenecks are, and then write any speed-critical code in Cython—this essentially provides an enhanced Python syntax that compiles into pure C for maximum runtime speed.

Of course, sometimes the functionality we need is already available in a C or C++ library, or a library in another language that uses the C calling convention. In most such cases there will be a third-party Python module that provides access to the library we require—these can be found on the Python Package Index (PyPI; [pypi.python.org](http://pypi.python.org)). But in the uncommon case that such a module isn't available, the standard library's ctypes module can be used to access C library functionality—as can the third-party Cython package. Using preexisting C libraries can significantly reduce development times, as well as usually providing very fast processing. Both ctypes and Cython are covered in Chapter 5.

The Python standard library provides a variety of modules for networking, from the low-level socket module, to the mid-level socketserver module, up to the high-level xmlrpc module. Although low- and mid-level networking makes sense when porting code from another language, if we are starting out in Python we can often avoid the low-level detail and just focus on what we want our networking applications to do by using high-level modules. In Chapter 6 we will see how to do this using the standard library's xmlrpc module and the powerful and easy-to-use third-party RPyC module.

Almost every program must provide some kind of user interface so that the program can determine what work it must do. Python programs can be written to support command-line user interfaces, using the argparse module, and full-terminal user interfaces (e.g., on Unix using the third-party urwid package; [excess.org/urwid](http://excess.org/urwid)). There are also a great many web frameworks—from the lightweight bottle ([bottlepy.org](http://bottlepy.org)) to heavyweights like Django ([www.djangoproject.com](http://www.djangoproject.com)) and Pyramid ([www.pyramidproject.org](http://www.pyramidproject.org))—all of which can be used to provide applications with a web interface. And, of course, Python can be used to create GUI (graphical user interface) applications.

The death of GUI applications in favor of web applications is often reported—and still hasn't happened. In fact, people seem to prefer GUI applications to web applications. For example, when smartphones became very popular early in the twenty-first century, users invariably preferred to use a purpose-built “app” rather than a web browser and web page for things they did regularly. There are many ways to do GUI programming with Python using third-party packages. However, in Chapter 7 we will see how to create modern-looking GUI applications using Tkinter, which is supplied as part of Python's standard library.

Most modern computers—including laptops and even smartphones—come equipped with powerful graphics facilities, often in the form of a separate GPU (Graphics Processing Unit) that's capable of impressive 2D and 3D graphics. Most GPUs support the OpenGL API, and Python programmers can get access to this API through third-party packages. In Chapter 8, we will see how to make use of OpenGL to do 3D graphics.

The purpose of this book is to illustrate how to write better Python applications that have good performance and maintainable code, and are easy to use. This book assumes prior knowledge of Python programming and is intended to be the kind of book people turn to once they've learned Python, whether from Python's documentation or from other books—such as *Programming in Python 3, Second Edition* (see the Selected Bibliography for details; ► 287). The book is designed to provide ideas, inspiration, and practical techniques to help readers take their Python programming to the next level.

All the book's examples have been tested with Python 3.3 (and where possible Python 3.2 and Python 3.1) on Linux, OS X (in most cases), and Windows (in most cases). The examples are available from the book's web site, [www.qtrac.eu/pipbook.html](http://www.qtrac.eu/pipbook.html), and should work with all future Python 3.x versions.

## Acknowledgments

As with all my other technical books, this book has greatly benefited from the advice, help, and encouragement of others: I am very grateful to them all.

Nick Coghlan, a Python core developer since 2005, provided plenty of constructive criticism, and backed this up with lots of ideas and code snippets to show alternative and better ways to do things. Nick's help was invaluable throughout the book, and particularly improved the early chapters.

Doug Hellmann, an experienced Python developer and author, sent me lots of useful comments, both on the initial proposal, and on every chapter of the book itself. Doug gave me many ideas and was kind enough to write the foreword.

Two friends—Jasmin Blanchette and Trenton Schulz—are both experienced programmers, and with their widely differing Python knowledge, they are ideal representatives of many of the book's intended readership. Jasmin and

Trenton's feedback has lead to many improvements and clarifications in the text and in the examples.

I am glad to thank my commissioning editor, Debra Williams Cauley, who once more provided support and practical help as the work progressed.

Thanks also to Elizabeth Ryan who managed the production process so well, and to the proofreader, Anna V. Popick, who did such excellent work.

As always, I thank my wife, Andrea, for her love and support.

# 1

# Creational Design Patterns in Python

---

---

§1.1. Abstract Factory Pattern ▶ 5

§1.1.1. A Classic Abstract Factory ▶ 6

§1.1.2. A More Pythonic Abstract Factory ▶ 9

§1.2. Builder Pattern ▶ 11

§1.3. Factory Method Pattern ▶ 17

§1.4. Prototype Pattern ▶ 24

§1.5. Singleton Pattern ▶ 26

---

---

Creational design patterns are concerned with how objects are created. Normally we create objects by calling their constructor (i.e., calling their class object with arguments), but sometimes we need more flexibility in how objects are created—which is why the creational design patterns are useful.

For Python programmers, some of these patterns are fairly similar to each other—and some of them, as we will note, aren’t really needed at all. This is because the original design patterns were primarily created for the C++ language and needed to work around some of that language’s limitations. Python doesn’t have those limitations.

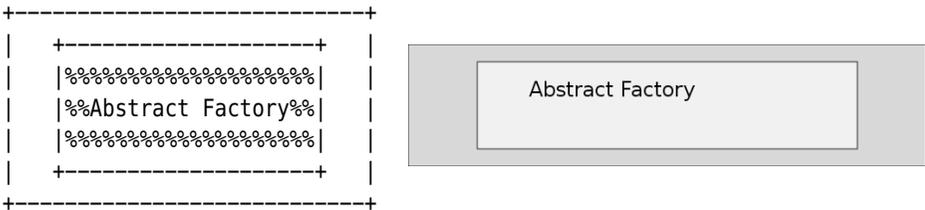
## 1.1. Abstract Factory Pattern

The Abstract Factory Pattern is designed for situations where we want to create complex objects that are composed of other objects and where the composed objects are all of one particular “family”.

For example, in a GUI system we might have an abstract widget factory that has three concrete subclass factories: `MacWidgetFactory`, `XfceWidgetFactory`, and `WindowsWidgetFactory`, all of which provide methods for creating the same objects (`make_button()`, `make_spinbox()`, etc.), but that do so using the platform-appropriate styling. This allows us to create a generic `create_dialog()` function that takes a factory instance as argument and produces a dialog with the OS X, Xfce, or Windows look and feel, depending on which factory we pass it.

### 1.1.1. A Classic Abstract Factory

To illustrate the Abstract Factory Pattern we will review a program that produces a simple diagram. Two factories will be used: one to produce plain text output, and the other to produce SVG (Scalable Vector Graphics) output. Both outputs are shown in Figure 1.1. The first version of the program we will look at, `diagram1.py`, shows the pattern in its pure form. The second version, `diagram2.py`, takes advantage of some Python-specific features to make the code slightly shorter and cleaner. Both versions produce identical output.\*



**Figure 1.1** *The plain text and SVG diagrams*

We will begin by looking at the code common to both versions, starting with the `main()` function.

```
def main():
    ...
    txtDiagram = create_diagram(DiagramFactory()) ❶
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory()) ❷
    svgDiagram.save(svgFilename)
```

First we create a couple of filenames (not shown). Next, we create a diagram using the plain text (default) factory (❶), which we then save. Then, we create and save the same diagram, only this time using an SVG factory (❷).

```
def create_diagram(factory):
    diagram = factory.make_diagram(30, 7)
    rectangle = factory.make_rectangle(4, 1, 22, 5, "yellow")
    text = factory.make_text(7, 3, "Abstract Factory")
    diagram.add(rectangle)
    diagram.add(text)
    return diagram
```

\* All the book's examples are available for download from [www.qttrac.eu/pipbook.html](http://www.qttrac.eu/pipbook.html).

This function takes a diagram factory as its sole argument and uses it to create the required diagram. The function doesn't know or care what kind of factory it receives so long as it supports our diagram factory interface. We will look at the `make_...()` methods shortly.

Now that we have seen how the factories are used, we can turn to the factories themselves. Here is the plain text diagram factory (which is also the factory base class):

```
class DiagramFactory:
    def make_diagram(self, width, height):
        return Diagram(width, height)

    def make_rectangle(self, x, y, width, height, fill="white",
                       stroke="black"):
        return Rectangle(x, y, width, height, fill, stroke)

    def make_text(self, x, y, text, fontsize=12):
        return Text(x, y, text, fontsize)
```

Despite the word “abstract” in the pattern's name, it is usual for one class to serve both as a base class that provides the interface (i.e., the abstraction), and also as a concrete class in its own right. We have followed that approach here with the `DiagramFactory` class.

Here are the first few lines of the SVG diagram factory:

```
class SvgDiagramFactory(DiagramFactory):
    def make_diagram(self, width, height):
        return SvgDiagram(width, height)
    ...
```

The only difference between the two `make_diagram()` methods is that the `DiagramFactory.make_diagram()` method returns a `Diagram` object and the `SvgDiagramFactory.make_diagram()` method returns an `SvgDiagram` object. This pattern applies to the two other methods in the `SvgDiagramFactory` (which are not shown).

We will see in a moment that the implementations of the plain text `Diagram`, `Rectangle`, and `Text` classes are radically different from those of the `SvgDiagram`, `SvgRectangle`, and `SvgText` classes—although every class provides the same interface (i.e., both `Diagram` and `SvgDiagram` have the same methods). This means that we can't mix classes from different families (e.g., `Rectangle` and `SvgText`)—and this is a constraint automatically applied by the factory classes.

Plain text `Diagram` objects hold their data as a list of lists of single character strings where the character is a space or `+`, `|`, `-`, and so on. The plain text `Rect-`

angle and Text and a list of lists of single character strings that are to replace those in the overall diagram at their position (and working right and down as necessary).

```
class Text:
    def __init__(self, x, y, text, fontsize):
        self.x = x
        self.y = y
        self.rows = [list(text)]
```

This is the complete Text class. For plain text we simply discard the fontsize.

```
class Diagram:
    ...
    def add(self, component):
        for y, row in enumerate(component.rows):
            for x, char in enumerate(row):
                self.diagram[y + component.y][x + component.x] = char
```

Here is the Diagram.add() method. When we call it with a Rectangle or Text object (the component), this method iterates over all the characters in the component's list of lists of single character strings (component.rows) and replaces corresponding characters in the diagram. The Diagram.\_\_init\_\_() method (not shown) has already ensured that its self.diagram is a list of lists of space characters (of the given width and height) when Diagram(width, height) is called.

```
SVG_TEXT = """<text x="{x}" y="{y}" text-anchor="left" \
font-family="sans-serif" font-size="{fontsize}">{text}</text>"""
SVG_SCALE = 20
class SvgText:
    def __init__(self, x, y, text, fontsize):
        x *= SVG_SCALE
        y *= SVG_SCALE
        fontsize *= SVG_SCALE // 10
        self.svg = SVG_TEXT.format(**locals())
```

This is the complete SvgText class and the two constants it depends on.\* Incidentally, using `**locals()` saves us from having to write `SVG_TEXT.format(x=x, y=y, text=text, fontsize=fontsize)`. From Python 3.2 we could write `SVG_TEXT.for-`

---

\* Our SVG output is rather crudely done—but it is sufficient to show this design pattern. Third-party SVG modules are available from the Python Package Index (PyPI) at [pypi.python.org](http://pypi.python.org).

`mat_map(locals())` instead, since the `str.format_map()` method does the mapping unpacking for us. (See the “Sequence and Mapping Unpacking” sidebar, ► 13.)

```
class SvgDiagram:
    ...
    def add(self, component):
        self.diagram.append(component.svg)
```

For the `SvgDiagram` class, each instance holds a list of strings in `self.diagram`, each one of which is a piece of SVG text. This makes adding new components (e.g., of type `SvgRectangle` or `SvgText`) really easy.

### 1.1.2. A More Pythonic Abstract Factory

The `DiagramFactory` and its `SvgDiagramFactory` subclass, and the classes they make use of (`Diagram`, `SvgDiagram`, etc.), work perfectly well and exemplify the design pattern.

Nonetheless, our implementation has some deficiencies. First, neither of the factories needs any state of its own, so we don’t really need to create factory instances. Second, the code for `SvgDiagramFactory` is almost identical to that of `DiagramFactory`—the only difference being that it returns `SvgText` rather than `Text` instances, and so on—which seems like needless duplication. Third, our top-level namespace contains all of the classes: `DiagramFactory`, `Diagram`, `Rectangle`, `Text`, and all the SVG equivalents. Yet we only really need to access the two factories. Furthermore, we have been forced to prefix the SVG class names (e.g., using `SvgRectangle` rather than `Rectangle`) to avoid name clashes, which is untidy. (One solution for avoiding name conflicts would be to put each class in its own module. However, this approach would not solve the problem of code duplication.)

In this subsection we will address all these deficiencies. (The code is in `diagram2.py`.)

The first change we will make is to nest the `Diagram`, `Rectangle`, and `Text` classes inside the `DiagramFactory` class. This means that these classes must now be accessed as `DiagramFactory.Diagram` and so on. We can also nest the equivalent classes inside the `SvgDiagramFactory` class, only now we can give them the same names as the plain text classes since a name conflict is no longer possible—for example, `SvgDiagramFactory.Diagram`. We have also nested the constants the classes depend on, so our only top-level names are now `main()`, `create_diagram()`, `DiagramFactory`, and `SvgDiagramFactory`.

```
class DiagramFactory:
    @classmethod
    def make_diagram(Class, width, height):
```

```

    return Class.Diagram(width, height)

    @classmethod
    def make_rectangle(Class, x, y, width, height, fill="white",
                      stroke="black"):
        return Class.Rectangle(x, y, width, height, fill, stroke)

    @classmethod
    def make_text(Class, x, y, text, fontsize=12):
        return Class.Text(x, y, text, fontsize)

    ...

```

Here is the start of our new `DiagramFactory` class. The `make_...()` methods are now all class methods. This means that when they are called the class is passed as their first argument (rather like `self` is passed for normal methods). So, in this case a call to `DiagramFactory.make_text()` will mean that `DiagramFactory` is passed as the `Class`, and a `DiagramFactory.Text` object will be created and returned.

This change also means that the `SvgDiagramFactory` subclass that inherits from `DiagramFactory` does not need any of the `make_...()` methods at all. If we call, say, `SvgDiagramFactory.make_rectangle()`, since `SvgDiagramFactory` doesn't have that method the base class `DiagramFactory.make_rectangle()` method will be called instead—but the `Class` passed will be `SvgDiagramFactory`. This will result in an `SvgDiagramFactory.Rectangle` object being created and returned.

```

def main():
    ...
    txtDiagram = create_diagram(DiagramFactory)
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory)
    svgDiagram.save(svgFilename)

```

These changes also mean that we can simplify our `main()` function since we no longer need to create factory instances.

The rest of the code is almost identical to before, the key difference being that since the constants and non-factory classes are now nested inside the factories, we must access them using the factory name.

```

class SvgDiagramFactory(DiagramFactory):
    ...
    class Text:
        def __init__(self, x, y, text, fontsize):
            x *= SvgDiagramFactory.SVG_SCALE
            y *= SvgDiagramFactory.SVG_SCALE

```

```

fontsize *= SvgDiagramFactory.SVG_SCALE // 10
self.svg = SvgDiagramFactory.SVG_TEXT.format(**locals())

```

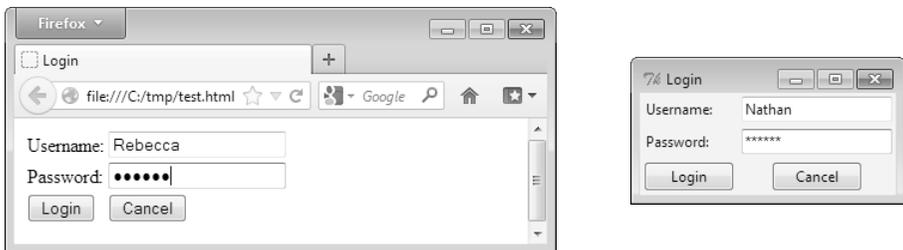
Here is the `SvgDiagramFactory`'s nested `Text` class (equivalent to `diagram1.py`'s `SvgText` class), which shows how the nested constants must be accessed.

## 1.2. Builder Pattern

The Builder Pattern is similar to the Abstract Factory Pattern in that both patterns are designed for creating complex objects that are composed of other objects. What makes the Builder Pattern distinct is that the builder not only provides the methods for building a complex object, it also holds the representation of the entire complex object itself.

This pattern allows the same kind of compositionality as the Abstract Factory Pattern (i.e., complex objects are built out of one or more simpler objects), but is particularly suited to cases where the representation of the complex object needs to be kept separate from the composition algorithms.

We will show an example of the Builder Pattern in a program that can produce forms—either web forms using HTML, or GUI forms using Python and Tkinter. Both forms work visually and support text entry; however, their buttons are non-functional.\* The forms are shown in Figure 1.2; the source code is in `formbuilder.py`.



**Figure 1.2** *The HTML and Tkinter forms on Windows*

Let's begin by looking at the code needed to build each form, starting with the top-level calls.

```

htmlForm = create_login_form(HtmlFormBuilder())
with open(htmlFilename, "w", encoding="utf-8") as file:
    file.write(htmlForm)

tkForm = create_login_form(TkFormBuilder())

```

\*All the examples must strike a balance between realism and suitability for learning, and as a result a few—as in this case—have only basic functionality.

```
with open(tkFilename, "w", encoding="utf-8") as file:
    file.write(tkForm)
```

Here, we have created each form and written it out to an appropriate file. In both cases we use the same form creation function (`create_login_form()`), parameterized by an appropriate builder object.

```
def create_login_form(builder):
    builder.add_title("Login")
    builder.add_label("Username", 0, 0, target="username")
    builder.add_entry("username", 0, 1)
    builder.add_label("Password", 1, 0, target="password")
    builder.add_entry("password", 1, 1, kind="password")
    builder.add_button("Login", 2, 0)
    builder.add_button("Cancel", 2, 1)
    return builder.form()
```

This function can create any arbitrary HTML or Tkinter form—or any other kind of form for which we have a suitable builder. The `builder.add_title()` method is used to give the form a title. All the other methods are used to add a widget to the form at a given row and column position.

Both `HtmlFormBuilder` and `TkFormBuilder` inherit from an abstract base class, `AbstractFormBuilder`.

```
class AbstractFormBuilder(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def add_title(self, title):
        self.title = title

    @abc.abstractmethod
    def form(self):
        pass

    @abc.abstractmethod
    def add_label(self, text, row, column, **kwargs):
        pass

    ...
```

Any class that inherits this class must implement all the abstract methods. We have elided the `add_entry()` and `add_button()` abstract methods because, apart from their names, they are identical to the `add_label()` method. Incidentally, we are required to make the `AbstractFormBuilder` have a metaclass of `abc.ABCMeta` to allow it to use the `abc` module's `@abstractmethod` decorator. (See §2.4, ► 48 for more on decorators.)

## Sequence and Mapping Unpacking



Unpacking means extracting all the items in a sequence or map individually. One simple use case for sequence unpacking is to extract the first or first few items, and then the rest. For example:

```
first, second, *rest = sequence
```

Here we are assuming that `sequence` has at least three items: `first == sequence[0]`, `second == sequence[1]`, and `rest == sequence[2:]`.

Perhaps the most common uses of unpacking are related to function calls. If we have a function that expects a certain number of positional arguments, or particular keyword arguments, we can use unpacking to provide them. For example:

```
args = (600, 900)
kwargs = dict(copies=2, collate=False)
print_setup(*args, **kwargs)
```

The `print_setup()` function requires two positional arguments (width and height) and accepts up to two optional keyword arguments (`copies` and `collate`). Rather than passing the values directly, we have created an `args` tuple and a `kwargs` dict, and used sequence unpacking (`*args`) and mapping unpacking (`**kwargs`) to pass in the arguments. The effect is exactly the same as if we had written `print_setup(600, 900, copies=2, collate=False)`.

The other use related to function calls is to create functions that can accept any number of positional arguments, or any number of keyword arguments, or any number of either. For example:

```
def print_args(*args, **kwargs):
    print(args.__class__.__name__, args,
          kwargs.__class__.__name__, kwargs)

print_args() # prints: tuple () dict {}
print_args(1, 2, 3, a="A") # prints: tuple (1, 2, 3) dict {'a': 'A'}
```

The `print_args()` function accepts any number of positional or keyword arguments. Inside the function, `args` is of type `tuple`, and `kwargs` is of type `dict`. If we wanted to pass these on to a function called inside the `print_args()` function, we could, of course, use unpacking in the call (e.g., `function(*args, **kwargs)`). Another common use of mapping unpacking is when calling the `str.format()` method—for example, `s.format(**locals())`—rather than typing all the `key=value` arguments manually (e.g., see `SvgText.__init__()`; 8 ◀).

Giving a class a metaclass of `abc.ABCMeta` means that the class cannot be instantiated, and so must be used as an abstract base class. This makes particular sense for code being ported from, say, C++ or Java, but does incur a tiny runtime overhead. However, many Python programmers use a more laid back approach: they don't use a metaclass at all, and simply document that the class should be used as an abstract base class.

```
class HtmlFormBuilder(ABCFormBuilder):
    def __init__(self):
        self.title = "HtmlFormBuilder"
        self.items = {}

    def add_title(self, title):
        super().add_title(escape(title))

    def add_label(self, text, row, column, **kwargs):
        self.items[(row, column)] = ('<td><label for="{}">{}</label></td>'
                                     .format(kwargs["target"], escape(text)))

    def add_entry(self, variable, row, column, **kwargs):
        html = "<td><input name="{}" type="{}" /></td>".format(
            variable, kwargs.get("kind", "text"))
        self.items[(row, column)] = html
    ...
```

Here is the start of the `HtmlFormBuilder` class. We provide a default title in case the form is built without one. All the form's widgets are stored in an `items` dictionary that uses *row*, *column* 2-tuple keys, and the widgets' HTML as values.

We must reimplement the `add_title()` method since it is abstract, but since the abstract version has an implementation we can simply call that implementation. In this case we must preprocess the title using the `html.escape()` function (or the `xml.sax.saxutil.escape()` function in Python 3.2 or earlier).

The `add_button()` method (not shown) is structurally similar to the other `add_...()` methods.

```
def form(self):
    html = ["<!doctype html>\n<html><head><title>{}</title></head>"
           "<body>".format(self.title), '<form><table border="0">']
    thisRow = None
    for key, value in sorted(self.items.items()):
        row, column = key
        if thisRow is None:
            html.append(" <tr>")
        elif thisRow != row:
```

```

        html.append(" </tr>\n <tr>")
        thisRow = row
        html.append(" " + value)
        html.append(" </tr>\n</table></form></body></html>")
    return "\n".join(html)

```

The `HtmlFormBuilder.form()` method creates an HTML page consisting of a `<form>`, inside of which is a `<table>`, inside of which are rows and columns of widgets. Once all the pieces have been added to the `html` list, the list is returned as a single string (with newline separators to make it more human-readable).

```

class TkFormBuilder(AbstractFormBuilder):

    def __init__(self):
        self.title = "TkFormBuilder"
        self.statements = []

    def add_title(self, title):
        super().add_title(title)

    def add_label(self, text, row, column, **kwargs):
        name = self._canonicalize(text)
        create = ""self.{}Label = ttk.Label(self, text="{}:")"".format(
            name, text)
        layout = ""self.{}Label.grid(row={}, column={}, sticky=tk.W, \
padx="0.75m", pady="0.75m")"".format(name, row, column)
        self.statements.extend((create, layout))

    ...
    def form(self):
        return TkFormBuilder.TEMPLATE.format(title=self.title,
            name=self._canonicalize(self.title, False),
            statements="\n " + "\n ".join(self.statements))

```

This is an extract from the `TkFormBuilder` class. We store the form's widgets as a list of statements (i.e., as strings of Python code), two statements per widget.

The `add_label()` method's structure is also used by the `add_entry()` and `add_button()` methods (neither of which is shown). These methods begin by getting a canonicalized name for the widget and then make two strings: `create`, which has the code to create the widget and `layout`, which has the code to lay out the widget in the form. Finally, the methods add the two strings to the list of statements.

The `form()` method is very simple: it just returns a `TEMPLATE` string parameterized by the title and the statements.

```

TEMPLATE = """#!/usr/bin/env python3
import tkinter as tk
import tkinter.ttk as ttk

class {name}Form(tk.Toplevel): ❶
    def __init__(self, master):
        super().__init__(master)
        self.withdraw()      # hide until ready to show
        self.title("{title}") ❷
        {statements} ❸
        self.bind("<Escape>", lambda *args: self.destroy())
        self.deiconify()    # show when widgets are created and laid out
        if self.winfo_viewable():
            self.transient(master)
        self.wait_visibility()
        self.grab_set()
        self.wait_window(self)

if __name__ == "__main__":
    application = tk.Tk()
    window = {name}Form(application) ❹
    application.protocol("WM_DELETE_WINDOW", application.quit)
    application.mainloop()
"""

```

The form is given a unique class name based on the title (e.g., `LoginForm`, ❶; ❹). The window title is set early on (e.g., “Login”, ❷), and this is followed by all the statements to create and lay out the form’s widgets (❸).

The Python code produced by using the template can be run stand-alone thanks to the `if __name__ ...` block at the end.

```

def _canonicalize(self, text, startLower=True):
    text = re.sub(r"\W+", "", text)
    if text[0].isdigit():
        return "_" + text
    return text if not startLower else text[0].lower() + text[1:]

```

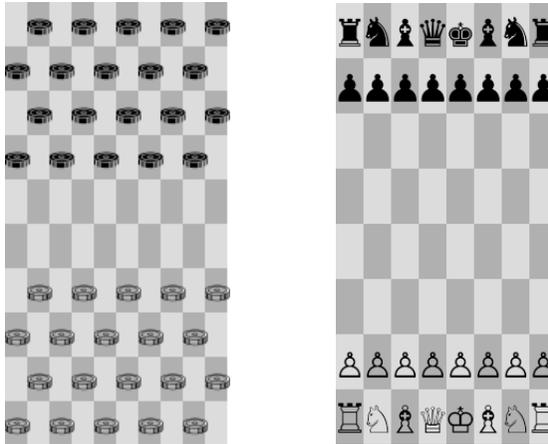
The code for the `_canonicalize()` method is included for completeness. Incidentally, although it looks as if we create a fresh regex every time the function is called, in practice Python maintains a fairly large internal cache of compiled regexes, so for the second and subsequent calls, Python just looks up the regex rather than recompiling it.\*

## 1.3. Factory Method Pattern

The Factory Method Pattern is intended to be used when we want subclasses to choose which classes they should instantiate when an object is requested. This is useful in its own right, but can be taken further and used in cases where we cannot know the class in advance (e.g., the class to use is based on what we read from a file or depends on user input).

In this section we will review a program that can be used to create game boards (e.g., a checkers or chess board). The program's output is shown in Figure 1.3, and the four variants of the source code are in the files `gameboard1.py..gameboard4.py`.<sup>◦</sup>

We want to have an abstract board class that can be subclassed to create game-specific boards. Each board subclass will populate itself with its initial layout of pieces. And we want every unique kind of piece to belong to its own class (e.g., `BlackDraught`, `WhiteDraught`, `BlackChessBishop`, `WhiteChessKnight`, etc.). Incidentally, for individual pieces, we have used class names like `WhiteDraught` rather than, say, `WhiteChecker`, to match the names used in Unicode for the corresponding characters.



**Figure 1.3** *The checkers and chess game boards on a Linux console*

---

\* This book assumes a basic knowledge of regexes and Python's `re` module. Readers needing to learn this can download a free PDF of "Chapter 13. Regular Expressions" from this author's book *Programming in Python 3, Second Edition*; see [www.qtrac.eu/py3book.html](http://www.qtrac.eu/py3book.html).

◦ Unfortunately, Windows consoles' UTF-8 support is rather poor, with many characters unavailable, even if code page 65001 is used. So, for Windows, the programs write their output to a temporary file and print the filename they used. None of the standard Windows monospaced fonts seems to have the checkers or chess piece characters, although most of the variable-width fonts have the chess pieces. The free and open-source `DejaVu Sans` font has them all ([dejavu-fonts.org](http://dejavu-fonts.org)).

We will begin by reviewing the top-level code that instantiates and prints the boards. Next, we will look at the board classes and some of the piece classes—starting with hard-coded classes. Then we will review some variations that allow us to avoid hard-coding classes and at the same time use fewer lines of code.

```
def main():
    checkers = CheckersBoard()
    print(checkers)

    chess = ChessBoard()
    print(chess)
```

This function is common to all versions of the program. It simply creates each type of board and prints it to the console, relying on the `AbstractBoard`'s `__str__()` method to convert the board's internal representation into a string.

```
BLACK, WHITE = ("BLACK", "WHITE")

class AbstractBoard:

    def __init__(self, rows, columns):
        self.board = [[None for _ in range(columns)] for _ in range(rows)]
        self.populate_board()

    def populate_board(self):
        raise NotImplementedError()

    def __str__(self):
        squares = []
        for y, row in enumerate(self.board):
            for x, piece in enumerate(row):
                square = console(piece, BLACK if (y + x) % 2 else WHITE)
                squares.append(square)
            squares.append("\n")
        return "".join(squares)
```

The `BLACK` and `WHITE` constants are used here to indicate each square's background color. In later variants they are also used to indicate each piece's color. This class is quoted from `gameboard1.py`, but it is the same in all versions.

It would have been more conventional to specify the constants by writing: `BLACK, WHITE = range(2)`. However, using strings is much more helpful when it comes to debugging error messages, and should be just as fast as using integers thanks to Python's smart interning and identity checks.

The board is represented by a list of rows of single-character strings—or `None` for unoccupied squares. The `console()` function (not shown, but in the source code),

returns a string representing the given piece on the given background color. (On Unix-like systems this string includes escape codes to color the background.)

We could have made the `AbstractBoard` a formally abstract class by giving it a metaclass of `abc.ABCMeta` (as we did for the `AbstractFormBuilder` class; 12 ◀). However, here we have chosen to use a different approach, and simply raise a `NotImplementedError` exception for any methods we want subclasses to reimplement.

```
class CheckersBoard(AbstractBoard):
    def __init__(self):
        super().__init__(10, 10)

    def populate_board(self):
        for x in range(0, 9, 2):
            for row in range(4):
                column = x + ((row + 1) % 2)
                self.board[row][column] = BlackDraught()
                self.board[row + 6][column] = WhiteDraught()
```

This subclass is used to create a representation of a  $10 \times 10$  international checkers board. This class's `populate_board()` method is *not* a factory method, since it uses hard-coded classes; it is shown in this form as a step on the way to making it into a factory method.

```
class ChessBoard(AbstractBoard):
    def __init__(self):
        super().__init__(8, 8)

    def populate_board(self):
        self.board[0][0] = BlackChessRook()
        self.board[0][1] = BlackChessKnight()
        ...
        self.board[7][7] = WhiteChessRook()
        for column in range(8):
            self.board[1][column] = BlackChessPawn()
            self.board[6][column] = WhiteChessPawn()
```

This version of the `ChessBoard`'s `populate_board()` method—just like the `CheckersBoard`'s one—is *not* a factory method, but it does illustrate how the chess board is populated.

```
class Piece(str):
    __slots__ = ()
```

This class serves as a base class for pieces. We could have simply used `str`, but that would not have allowed us to determine if an object is a piece (e.g., using `isinstance(x, Piece)`). Using `__slots__ = ()` ensures that instances have no data, a topic we'll discuss later on (§2.6, ► 65).

```
class BlackDraught(Piece):
    __slots__ = ()
    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):
    __slots__ = ()
    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```

These two classes are models for the pattern used for all the piece classes. Every one is an immutable `Piece` subclass (itself a `str` subclass) that is initialized with a one-character string holding the Unicode character that represents the relevant piece. There are fourteen of these tiny subclasses in all, each one differing only by its class name and the string it holds: clearly, it would be nice to eliminate all this near-duplication.

```
def populate_board(self):
    for x in range(0, 9, 2):
        for y in range(4):
            column = x + ((y + 1) % 2)
            for row, color in ((y, "black"), (y + 6, "white")):
                self.board[row][column] = create_piece("draught",
                                                       color)
```

This new version of the `CheckersBoard.populate_board()` method (quoted from `gameboard2.py`) is a factory method, since it depends on a new `create_piece()` factory function rather than on hard-coded classes. The `create_piece()` function returns an object of the appropriate type (e.g., a `BlackDraught` or a `WhiteDraught`), depending on its arguments. This version of the program has a similar `ChessBoard.populate_board()` method (not shown), which also uses string color and piece names and the same `create_piece()` function.

```
def create_piece(kind, color):
    if kind == "draught":
        return eval("{}{}{}".format(color.title(), kind.title()))
    return eval("{}Chess{}".format(color.title(), kind.title()))
```

This factory function uses the built-in `eval()` function to create class instances. For example, if the arguments are "knight" and "black", the string to be `eval()`'d will be `"BlackChessKnight()"`. Although this works perfectly well, it is potentially risky since pretty well anything could be `eval()`'d into existence—we will see a solution, using the built-in `type()` function, shortly.

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    exec("""\
class {}(Piece):
    __slots__ = ()
    def __new__(Class):
        return super().__new__(Class, "{}")""".format(name, char))
```

Instead of writing the code for fourteen very similar classes, here we create all the classes we need with a single block of code.

The `itertools.chain()` function takes one or more iterables and returns a single iterable that iterates over the first iterable it was passed, then the second, and so on. Here, we have given it two iterables, the first a 2-tuple of the Unicode code points for black and white checkers pieces, and the second a range-object (in effect, a generator) for the black and white chess pieces.

For each code point we create a single character string (e.g., "♘") and then create a class name based on the character's Unicode name (e.g., "black chess knight" becomes `BlackChessKnight`). Once we have the character and the name we use `exec()` to create the class we need. This code block is a mere dozen lines—compared with around a hundred lines for creating all the classes individually.

Unfortunately, though, using `exec()` is potentially even more risky than using `eval()`, so we must find a better way.

```
DRAUGHT, PAWN, ROOK, KNIGHT, BISHOP, KING, QUEEN = ("DRAUGHT", "PAWN",
    "ROOK", "KNIGHT", "BISHOP", "KING", "QUEEN")

class CheckersBoard(AbstractBoard):
    ...
    def populate_board(self):
        for x in range(0, 9, 2):
            for y in range(4):
                column = x + ((y + 1) % 2)
```

```

for row, color in ((y, BLACK), (y + 6, WHITE)):
    self.board[row][column] = self.create_piece(DRAUGHT,
        color)

```

This `CheckersBoard.populate_board()` method is from `gameboard3.py`. It differs from the previous version in that the piece and color are both specified using constants rather than easy to mistype string literals. Also, it uses a new `create_piece()` factory to create each piece.

An alternative `CheckersBoard.populate_board()` implementation is provided in `gameboard4.py` (not shown)—this version uses a subtle combination of a list comprehension and a couple of `itertools` functions.

```

class AbstractBoard:
    __classForPiece = {(DRAUGHT, BLACK): BlackDraught,
        (PAWN, BLACK): BlackChessPawn,
        ...
        (QUEEN, WHITE): WhiteChessQueen}
    ...
    def create_piece(self, kind, color):
        return AbstractBoard.__classForPiece[kind, color]()

```

This version of the `create_piece()` factory (also from `gameboard3.py`, of course) is a method of the `AbstractBoard` that the `CheckersBoard` and `ChessBoard` classes inherit. It takes two constants and looks them up in a static (i.e., class-level) dictionary whose keys are (*piece kind, color*) 2-tuples, and whose values are class objects. The looked-up value—a class—is immediately called (using the `()` call operator), and the resulting piece instance is returned.

The classes in the dictionary could have been individually coded (as they were in `gameboard1.py`) or created dynamically but riskily (as they were in `gameboard2.py`). But for `gameboard3.py`, we have created them dynamically and safely, without using `eval()` or `exec()`.

```

for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!

```

This code has the same overall structure as the code shown earlier for creating the fourteen piece subclasses that the program needs (21 ◀). Only this time instead of using `eval()` and `exec()` we take a somewhat safer approach.

Once we have the character and name we create a new function (called `new()`) by calling a custom `make_new_method()` function. We then create a new class using the built-in `type()` function. To create a class this way we must pass in the type's name, a tuple of its base classes (in this case, there's just one, `Piece`), and a dictionary of the class's attributes. Here, we have set the `__slots__` attribute to an empty tuple (to stop the class's instances having a private `__dict__` that isn't needed), and set the `__new__` method attribute to the `new()` function we have just created.

Finally, we use the built-in `setattr()` function to add to the current module (`sys.modules[__name__]`) the newly created class (`Class`) as an attribute called `name` (e.g., "WhiteChessPawn"). In `gameboard4.py`, we have written the last line of this code snippet in a nicer way:

```
globals()[name] = Class
```

Here, we have retrieved a reference to the dict of `globals` and added a new item whose key is the name held in `name`, and whose value is our newly created `Class`. This does exactly the same thing as the `setattr()` line used in `gameboard3.py`.

```
def make_new_method(char): # Needed to create a fresh method each time
    def new(Class): # Can't use super() or super(Piece, Class)
        return Piece.__new__(Class, char)
    return new
```

This function is used to create a `new()` function (that will become a class's `__new__()` method). We cannot use a `super()` call since at the time the `new()` function is created there is no class context for the `super()` function to access. Note that the `Piece` class (19 ◀) doesn't have a `__new__()` method—but its base class (`str`) does, so that is the method that will actually be called.

Incidentally, the earlier code block's `new = make_new_method(char)` line and the `make_new_method()` function just shown could both be deleted, so long as the line that called the `make_new_method()` function was replaced with these:

```
new = (lambda char: lambda Class: Piece.__new__(Class, char))(char)
new.__name__ = "__new__"
```

Here, we create a function that creates a function and immediately calls the outer function parameterized by `char` to return a `new()` function. (This code is used in `gameboard4.py`.)

All lambda functions are called "lambda", which isn't very helpful for debugging. So, here, we explicitly give the function the name it should have, once it is created.

```
def populate_board(self):
    for row, color in ((0, BLACK), (7, WHITE)):
        for columns, kind in (((0, 7), ROOK), ((1, 6), KNIGHT),
                              ((2, 5), BISHOP), ((3,), QUEEN), ((4,), KING)):
            for column in columns:
                self.board[row][column] = self.create_piece(kind,
                                                             color)
    for column in range(8):
        for row, color in ((1, BLACK), (6, WHITE)):
            self.board[row][column] = self.create_piece(PAWN, color)
```

For completeness, here is the ChessBoard.populate\_board() method from gameboard3.py (and gameboard4.py). It depends on color and piece constants (which could be provided by a file or come from menu options, rather than being hard-coded). In the gameboard3.py version, this uses the create\_piece() factory function shown earlier (22 ◀). But for gameboard4.py, we have used our final create\_piece() variant.

```
def create_piece(kind, color):
    color = "White" if color == WHITE else "Black"
    name = {DRAUGHT: "Draught", PAWN: "ChessPawn", ROOK: "ChessRook",
            KNIGHT: "ChessKnight", BISHOP: "ChessBishop",
            KING: "ChessKing", QUEEN: "ChessQueen"}[kind]
    return globals()[color + name]()
```

This is the gameboard4.py version's create\_piece() factory function. It uses the same constants as gameboard3.py, but rather than keeping a dictionary of class objects it dynamically finds the relevant class in the dictionary returned by the built-in globals() function. The looked-up class object is immediately called and the resulting piece instance is returned.

## 1.4. Prototype Pattern

The Prototype Pattern is used to create new objects by cloning an original object, and then modifying the clone.

As we have already seen, especially in the previous section, Python supports a wide variety of ways of creating new objects, even when their types are only known at runtime—and even if we have only their types' names.

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Given this classic Point class, here are seven ways to create new points:

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14) # Could have used any of point1 to point6
```

Point `point1` is created conventionally (and statically) using the `Point` class object as a constructor.\* All the other points are created dynamically, with `point2`, `point3`, and `point4` parameterized by the class name. As the creation of `point3` (and `point4`) makes clear, there is no need to use a risky `eval()` to create instances (as we did for `point2`). The creation of `point4` works exactly the same way as for `point3`, but using nicer syntax by relying on Python's built-in `globals()` function. Point `point5` is created using a generic `make_object()` function that accepts a class object and the relevant arguments. Point `point6` is created using the classic prototype approach: first, we clone an existing object, then we initialize or configure it. Point `point7` is created by using point `point1`'s class object, plus new arguments.

Point `point6` shows that Python has built-in support for prototyping using the `copy.deepcopy()` function. However, `point7` shows that Python can do better than prototyping: instead of needing to clone an existing object and modify the clone, Python gives us access to any object's class object, so that we can create a new object directly and much more efficiently than by cloning.

---

\* Strictly speaking, an `__init__()` method is an initializer, and a `__new__()` method is a constructor. However, since we almost always use `__init__()` and rarely use `__new__()`, we will refer to them both as "constructors" throughout the book.

## 1.5. Singleton Pattern

The Singleton Pattern is used when we need a class that has only a single instance that is the one and only instance accessed throughout the program.

For some object-oriented languages, creating a singleton can be surprisingly tricky, but this isn't the case for Python. The Python Cookbook ([code.activestate.com/recipes/langs/python/](http://code.activestate.com/recipes/langs/python/)) provides an easy-to-use Singleton class that any class can inherit to become a singleton—and a Borg class that achieves the same end in a rather different way.

However, the easiest way to achieve singleton functionality in Python is to create a module with the global state that's needed kept in private variables and access provided by public functions. For example, in Chapter 7's currency example (► 237), we need a function that will return a dictionary of currency rates (name keys, conversion rate values). We may want to call the function several times, but in most cases we want the rates fetched only once. We can achieve this by using the Singleton Pattern.

```
_URL = "http://www.bankofcanada.ca/stats/assets/csv/fx-seven-day.csv"

def get(refresh=False):
    if refresh:
        get.rates = {}
    if get.rates:
        return get.rates
    with urllib.request.urlopen(_URL) as file:
        for line in file:
            line = line.rstrip().decode("utf-8")
            if not line or line.startswith("#", "Date"):
                continue
            name, currency, *rest = re.split(r"\s*,\s*", line)
            key = "{} {}".format(name, currency)
            try:
                get.rates[key] = float(rest[-1])
            except ValueError as err:
                print("error {}: {}".format(err, line))
    return get.rates
get.rates = {}
```

This is the code for the `currency/Rates.py` module (as usual, excluding the imports). Here, we create a rates dictionary as an attribute of the `Rates.get()` function—this is our private value. When the public `get()` function is called for the first time (or if it is called with `refresh=True`), we download the rates afresh; otherwise, we simply return the rates we most recently downloaded. There is

no need for a class, yet we have still got a singleton data value—the rates—and we could easily add more singleton values.

---

All of the creational design patterns are straightforward to implement in Python. The Singleton Pattern can be implemented directly by using a module, and the Prototype Pattern is redundant (although still doable using the copy module), since Python provides dynamic access to class objects. The most useful creational design patterns for Python are the Factory and Builder Patterns; these can be implemented in a number of ways. Once we have created basic objects, we often need to create more complex objects by composing or adapting other objects. We'll look at how this is done in the next chapter.

*This page intentionally left blank*

# Index

## Symbols

♠ black chess knight character, 21  
!= not equal operator, 48  
& bitwise and operator, 133  
() call, generator, and tuple operator, 22; *see also* `__call__()`  
\* multiplication and sequence unpacking operator, 13, 26, 30–31, 43, 49, 70, 109  
\*\* mapping unpacking operator, 13, 222, 241–242, 248  
< less than operator, 48; *see also* `__lt__()`  
<= less than or equal operator, 48  
== equal operator, 48; *see also* `__eq__()`  
> greater than operator, 48  
>= greater than or equal operator, 48  
>> bitwise right shift operator, 133  
@ at symbol, 48, 52; *see also* decorator

## A

abc (module), 30–32, 35  
    ABCMeta (type); *see* top-level entry  
    abstractmethod(), 12–14, 42, 120  
    abstractproperty(), 42  
ABCMeta (type; abc module), 12–14, 30–32, 35, 42, 120  
abs() (built-in), 130  
abspath() (os.path module), 146  
abstractmethod() (abc module), 12–14, 42, 120  
abstractproperty() (abc module), 42  
adding properties, 57–58  
after() (tkinter module), 214, 239, 261  
all() (built-in), 31, 36  
antialiasing, 266  
append() (list type), 43, 102, 274  
application  
    design, 216  
    dialog-style, 237–244  
    main-window-style, 253–261  
    modal, 214, 236–237  
argparse (module), 145–146, 157, 209  
arguments  
    keyword and positional, 13, 50, 51–52  
    maximum, 134  
array (module), 124; *see also* numpy module  
Array (type; multiprocessing module), 144, 154  
as\_completed() (concurrent.futures module), 153–154  
assert (statement), 30, 37, 55, 69, 82, 127, 131  
AssertionError (exception), 30  
ast (module)  
    literal\_eval(), 88  
asynchronous I/O, 142  
atexit (module)  
    register(), 67, 187, 193  
Atom format, 159  
atomic operations, 143  
AttributeError (exception), 56, 103, 113

## B

- Bag1.py (example), 98–99
  - Bag2.py (example), 100
  - Bag3.py (example), 100
  - barchart1.py (example), 35–40
  - barchart2.py (example), 38, 124
  - barchart3.py (example), 36
  - basename() (os.path module), 136, 176
  - benchmark\_Scale.py (example), 194
  - bind() (tkinter.ttk.Widget), 169, 242, 249, 252, 257; *see also* keyboard handling
  - binding, late, 58
  - bool (type; built-in), 46, 55, 102, 113
  - bound method, 63, 83, 102
  - Bresenham's line algorithm, 130–131
  - built-in, *see also* statements
    - abs(), 130
    - all(), 31, 36
    - bool (type); *see* top-level entry
    - callable(), 32, 82
    - chr(), 21, 87, 95, 136
    - @classmethod, 9–10, 30–31, 36, 45–46, 127
    - dict (type); *see* top-level entry
    - dir(), 86–87
    - divmod(), 118, 278
    - enumerate(), 8, 51, 136, 150
    - eval(), 20–21, 25, 84–88; *see also* ast.literal\_eval()
    - exec(), 21, 88–91
    - \_\_file\_\_, 156, 162, 181, 256
    - getattr(), 25, 56, 58, 87, 113, 129, 248
    - globals(), 23, 24, 25, 86, 88
    - id(), 66
    - input(), 85–86, 213
    - isinstance(), 20, 30, 31, 51, 55, 58, 133, 207, 278
    - iter(), 44, 47, 95–97, 99
    - len(), 32, 63
    - list (type); *see* top-level entry
    - locals(), 8, 9, 13, 86
    - map(), 123
    - max(), 35, 151
    - min(), 151, 273
    - \_\_name\_\_, 16, 50, 146
    - next(), 76, 77
    - NotImplemented, 31, 36
    - open(), 11, 64, 120, 136, 160
    - ord(), 87, 95
    - @property, 42, 44, 48, 54, 56, 58, 60, 104, 110, 115, 278
    - range(), 18, 21
    - reversed(), 83
    - round(), 111, 131, 132, 195
    - setattr(), 22–23, 56, 58, 59, 113
    - sorted(), 14, 87, 208, 244
    - @staticmethod, 120, 121, 129, 133
    - str (type); *see* top-level entry
    - sum(), 99, 163
    - super(), 14, 19, 20, 23, 44, 76, 110, 168, 234
    - type(), 22–23, 30, 86
    - zip(), 51, 138
  - Button (type; tkinter.ttk module), 234; *inherits* Widget
  - bytes (type), 66, 67, 266
    - decode(), 26, 93, 185, 190
    - find(), 190
- ## C
- C foreign function interface for Python (CFFI), 179, 183
  - C/C++, 179, 182, 187, 189
  - c\_char\_p (ctypes module), 183–184, 185
  - c\_int (ctypes module), 184, 185
  - c\_void\_p (ctypes module), 183–184
  - calculator.py (example), 84–88
  - \_\_call\_\_() (special method), 82, 96–97, 113; *see also* () operator
  - call() (tkinter.Tk), 239
  - callable() (built-in), 32, 82

- callback; *see* function reference
- Canvas (type; tkinter module), 257
- cast, 192, 197
- CDLL() (ctypes module), 183
- CFFI (C Foreign Function Interface for Python), 179, 183
- cget() (tkinter.ttk.Widget), 171, 241, 244
- chain() (itertools module), 21, 46, 109
- ChainMap (type; collections module), 30–31, 36
- checking interfaces, 30–32, 35–36
- choice() (random module), 206, 220, 274
- chr() (built-in), 21, 87, 95, 136
- cimport (Cython), 190, 195
- class
  - attributes, *see also* special methods
    - `__class__`, 25, 222
    - `__doc__`, 50
    - `__mro__`, 30–31, 36
    - `__slots__`, 19–20, 22–23, 65
  - decorator, 36, 48, 54–59
  - dynamic creation, 20–24
  - methods, 10; *see also* `__new__()`
  - nested, 121–122
  - object, 25, 120
  - `__class__` (class attribute), 25, 222
- classes and windows, 234
- @classmethod (built-in), 9–10, 30–31, 36, 45–46, 127
- cloning, object, 24–25
- close button, 166, 177
- closure, 52, 62–63, 77, 81
- code generation, 91–94
- collections (module)
  - ChainMap (type), 30–31, 36
  - Counter (type), 100
  - defaultdict (type), 102, 114, 134
  - namedtuple (type), 88, 147, 160, 196, 205
  - OrderedDict (type), 85–86, 87
- colors; *see* Image example and cyImage example
- Combobox (type; tkinter.ttk module), 234, 240; *inherits* Widget
- command-line parsing; *see* argparse module
- compile() (re module), 120
- comprehensions, list, 18, 38, 49, 80, 123
- concatenation, tuple, 109
- concurrent.futures (module), 152–154
  - `as_completed()`, 153–154, 163
  - Executor (type), 152, 174
  - Future (type), 152, 153, 173–174, 175
  - ProcessPoolExecutor (type), 152, 153, 173
  - ThreadPoolExecutor (type), 152, 154, 162
  - `wait()`, 173–174
- conditional expression, 18, 47, 115, 130, 135, 137, 206, 244
- config() (tkinter.ttk.Widget), 170, 176, 243, 244, 247, 257; *alias of* `configure()`
- connect() (rpyc module), 227, 229
- connect\_by\_service() (rpyc module), 228
- ConnectionError (exception), 212, 213, 215, 216, 218, 228, 229
- constants, 18
- constructor; *see* `__init__()` and `__new__()`
- `__contains__()` (special method), 99, 223
- context manager; *see* with statement
- convert, 192, 197
- copy (module)
  - `deepcopy()`, 25, 144, 154
- coroutine, 76–79, 104–106, 116
- @coroutine, 77, 78, 105

Counter (type; collections module), 100  
 cProfile (module), 193–194, 201  
 cpu\_count() (multiprocessing module), 145–146, 157, 173  
 cpython.pycapsule (module); *see* pycapsule module  
 create\_command() (tkinter.Tk), 258–259  
 create\_string\_buffer() (ctypes module), 182, 185, 186  
 ctypes (module), 180–187  
   c\_char\_p, 183–184, 185  
   c\_int, 184, 185  
   c\_void\_p, 183–184  
   CDLL(), 183  
   create\_string\_buffer(), 182, 185, 186  
   POINTER(), 184  
   util (module)  
     find\_library(), 183  
 currency (example), 26, 237–244  
 cyImage (example), 151, 198–201; *see also* Image example  
 cylinder painting, 268  
 cylinder1.pyw (example), 264–270  
 cylinder2.pyw (example), 265, 270–272  
 Cython, 52, 179, 187–201; *see also* pycapsule module  
 cimport, 190, 195

## D

dæmon, 149, 150, 157, 158, 173  
 data wrapper, thread-safe, 219–224  
 datatype; *see* class  
 date() (datetime.datetime type), 84  
 datetime (type; datetime module)  
   date(), 84  
   fromtimestamp(), 108–109  
   isoformat(), 208–209, 226  
   now(), 208–209, 213, 218, 226  
   strptime(), 84, 207, 227

DateTime (type; xmlrpc.client module), 207  
 DBM database; *see* shelve module  
 decode() (bytes type), 26, 93, 185, 190  
 decorator  
   class, 36, 48, 54–59  
   @coroutine, 77, 78  
   function and method, 48–53  
 deepcopy() (copy module), 25, 144, 154  
 defaultdict (type; collections module), 102, 114, 134  
 deiconify() (tkinter.Tk), 166, 250–252, 255  
 del (statement), 98, 115, 126  
 \_\_delitem\_\_() (special method), 98, 223  
 descriptor, 67  
 diagram1.py (example), 6–9  
 diagram2.py (example), 9–11  
 dialog, 244–253  
   dumb, 236  
   modal, 245–250  
   modeless, 236–237, 244, 250–253  
   smart, 236, 245  
   -style applications, 237–244  
   vs. main windows, 235  
 dict (type), 26, 88, 98, 221, 223, 241–242, 248  
   get(), 14, 88, 98, 103, 115, 186, 191, 222  
   items(), 14, 58, 87, 208  
   keys(), 223  
   values(), 99, 187, 224  
 dictionary, thread-safe, 221–224  
 dir() (built-in), 86–87  
 dirname() (os.path module), 126, 156, 162, 181, 216, 256  
 DISABLED (tkinter module), 170  
 distutils (module), 188  
 divmod() (built-in), 118, 278  
 \_\_doc\_\_ (class attribute), 50

dock window, 261–262  
 documentation, tkinter module, 234  
 domain specific language, 84  
 draw() (pygame.graphics module), 271  
 drawing; *see* painting  
 DSL (Domain Specific Language),  
   84  
 dumb dialog, 236  
 dynamic class creation, 20–24  
 dynamic code generation, 91–94  
 dynamic instance creation, 24–25

## E

encode() (str type), 185, 186, 190,  
   191, 266  
 endswith() (str type), 62, 120, 121  
 \_\_enter\_\_() (special method), 61,  
   279  
 enumerate() (built-in), 8, 51, 136, 150  
 EOFError (exception), 85–86  
 \_\_eq\_\_() (special method), 48; *see*  
   *also* ==  
 escape() (html module), 14, 33  
 eval() (built-in), 20–21, 25, 84–88;  
   *see also* ast.literal\_eval()  
 evaluation, lazy, 60  
 events, 266; *see also* keyboard handling  
   and mouse handling  
   loop, 166, 167; *see also* glutMain-  
   Loop() and mainloop()  
   real and virtual, 242–243  
 examples  
   Bag1.py, 98–99  
   Bag2.py, 100  
   Bag3.py, 100  
   barchart1.py, 35–40  
   barchart2.py, 38, 124  
   barchart3.py, 36  
   benchmark\_Scale.py, 194  
   calculator.py, 84–88  
   currency, 26, 237–244  
   cyImage, 151, 198–201; *see also* Im-  
   age example  
   cylinder1.pyw, 264–270  
   cylinder2.pyw, 265, 270–272  
   diagram1.py, 6–9  
   diagram2.py, 9–11  
   formbuilder.py, 11–16  
   gameboard1.py, 17–20  
   gameboard2.py, 17–18, 20–21  
   gameboard3.py, 17–18, 22–23, 24  
   gameboard4.py, 17–18, 23–24  
   genome1.py, 88–91  
   genome2.py, 91–94  
   genome3.py, 91–94  
   gravitate, 245–261  
   gravitate2, 261  
   gravitate3d.pyw, 272–282  
   hello.pyw, 233–235  
   Hyphenate1.py, 181–187  
   Hyphenate2, 188–193  
   Image, 124–139, 151, 193, 199; *see*  
   *also* cyImage example  
   imageproxy1.py, 68–71  
   imageproxy2.py, 69, 70  
   ImageScale, 164–177  
   imagescale.py, 199  
   imagescale-c.py, 145  
   imagescale-cy.py, 199  
   imagescale-m.py, 145, 152–154,  
   164, 199  
   imagescale-q-m.py, 145, 147–152  
   imagescale-s.py, 145, 199  
   imagescale-t.py, 145  
   mediator1.py, 59, 101–104  
   mediator1d.py, 59  
   mediator2.py, 104–106  
   mediator2d.py, 106  
   Meter.py, 205–208  
   meter-rpc.pyw, 214–228  
   meter-rpyc.pyw, 228–229  
   meterclient-rpc.py, 210–219  
   meterclient-rpyc.py, 227–228  
   MeterLogin.py, 214–215

examples (*continued*)

MeterMT.py, 219–224  
 meterserver-rpc.py, 208–210  
 meterserver-rpyc.py, 225–227  
 multiplexer1.py, 112–115  
 multiplexer2.py, 115–116  
 multiplexer3.py, 116  
 observer.py, 107–111  
 pointstore1.py, 65–67  
 pointstore2.py, 65–67  
 render1.py, 29–34  
 render2.py, 32  
 stationery1.py, 40–45  
 stationery2.py, 45–47  
 tabulator1.py, 117  
 tabulator2.py, 117  
 tabulator3.py, 116–119  
 tabulator4.py, 118  
 texteditor, 261–262  
 texteditor2, 261–262  
 Unpack.py, 60–64  
 whatsnew-q.py, 155, 156–161  
 whatsnew-t.py, 155, 161–164  
 wordcount1.py, 119–122  
 wordcount2.py, 120

Exception (exception), 87, 128, 167, 183

exceptions, 207
 

- AssertionError, 30
- AttributeError, 56, 103, 113
- ConnectionError, 212, 213, 215, 216, 218, 228, 229
- EOFError, 85–86
- Exception, 87, 128, 167, 183
- ImportError, 125, 126, 137, 199
- IndexError, 95, 130
- KeyboardInterrupt, 148, 154, 160, 163, 209
- KeyError, 98, 206
- NotImplementedError, 18, 19, 120
- StopIteration, 96, 97
- TypeError, 30, 49, 89
- ValueError, 26, 51, 113

exec() (built-in), 21, 88–91  
 executable module, 238  
 executable (sys module), 92, 216  
 Executor (type; concurrent.futures module), 152, 174  
 exists() (os.path module), 146, 172, 181, 217  
 \_\_exit\_\_() (special method), 61, 279  
 exit() (sys module), 91  
 expression, conditional, 18, 47, 115, 130, 135, 137, 206, 244  
 expression, generator; *see* generator  
 extend() (List type), 15, 46

## F

feedparser (module), 159  
 \_\_file\_\_ (built-in), 156, 162, 181, 256  
 filedialog (tkinter module), 166  
 fill() (textwrap module), 32  
 find() (bytes type), 190  
 find\_library() (ctypes.util module), 183  
 findall() (re module), 90  
 flicker, 166  
 focus() (tkinter.ttk.Widget), 168, 169, 217, 239  
 format() (str type), 13, 21, 32, 37, 92, 109  
 format\_map() (str type), 8–9  
 formbuilder.py (example), 11–16  
 Frame (type; tkinter.ttk module), 166, 168, 234, 247, 255, 260; *inherits* Widget  
 fromiter() (numpy module), 128  
 fromtimestamp() (datetime.datetime type), 108–109  
 function, 83
 

- arguments, 13, 50
- decorators, 48–53
- nested, 58
- object, 63

reference, 60–62, 70, 87, 112  
 functools (module)  
   total\_ordering(), 48  
   wraps(), 50, 51, 77  
 functor; *see* `__call__()`  
 Future (type; concurrent.futures  
   module), 152, 153, 173–174, 175

## G

gameboard1.py (example), 17–20  
 gameboard2.py (example), 17–18,  
   20–21  
 gameboard3.py (example), 17–18,  
   22–23, 24  
 gameboard4.py (example), 17–18,  
   23–24  
 generator, 76–79, 97, 100, 139, 153,  
   160  
   expression, 44, 100  
   send(), 77, 78, 105–106, 112  
   throw(), 77  
 genome1.py (example), 88–91  
 genome2.py (example), 91–94  
 genome3.py (example), 91–94  
 geometry() (tkinter.Tk), 253  
 get()  
   dict (type), 14, 88, 98, 103, 115,  
     186, 191, 222  
   multiprocessing.Queue (type), 149,  
     150  
   queue.Queue (type), 158  
   tkinter.StringVar (type), 170,  
     172, 218, 243, 249  
 getattr() (built-in), 25, 56, 58, 87,  
   113, 129, 248  
 \_\_getattr\_\_() (special method), 66  
 \_\_getitem\_\_() (special method),  
   95–96, 222  
 getpass() (getpass module), 212  
 getpass (module)  
   getpass(), 212  
   getuser(), 212  
 gettempdir() (tempfile module), 39,  
   65, 161, 216  
 getuser() (getpass module), 212  
 GIF (image format), 38, 124, 239  
 GIL (Global Interpreter Lock), 142,  
   221  
 GL (OpenGL module), 265  
   glBegin(), 268, 271  
   glClear(), 267, 275  
   glClearColor(), 267  
   glColor3f(), 268, 271  
   glColor3ub(), 268, 277  
   glColorMaterial(), 267  
   glDisable(), 279  
   glEnable(), 267, 279  
   glEnd(), 268, 271  
   GLfloat, 267  
   glHint(), 267  
   glLightfv(), 267  
   glLoadIdentity(), 269, 275  
   glMaterialf(), 267  
   glMaterialfv(), 267  
   glMatrixMode(), 267, 269, 275  
   glOrtho(), 275  
   glPopMatrix(), 268, 276, 277  
   glPushMatrix(), 267, 268, 276,  
     277  
   glReadPixels(), 280  
   glRotatef(), 267, 276  
   glShadeModel(), 279  
   glTranslatef(), 267, 268, 277  
   GLubyte, 280  
   glVertex3f(), 268, 271  
   glViewport(), 269, 275  
 global grab, 236  
 global interpreter lock (GIL), 142,  
   221  
 global modal, 235–237  
 globals() (built-in), 23, 24, 25, 86, 88  
 GLU (OpenGL module), 265  
   gluCylinder(), 268  
   gluDeleteQuadric(), 268, 276  
   gluNewQuadric(), 268, 276

GLU (OpenGL module) (*continued*)  
 gluPerspective(), 269  
 gluQuadricNormals(), 268, 276  
 gluSphere(), 277  
 GLUT (OpenGL module), 265  
 glutDestroyWindow(), 269  
 glutDisplayFunc(), 266, 276  
 glutDisplayString(), 266  
 glutInit(), 265  
 glutInitWindowSize(), 265  
 glutKeyboardFunc(), 266, 269  
 glutMainLoop(), 266  
 glutPostRedisplay(), 269  
 glutReshapeFunc(), 266, 275  
 glutSpecialFunc(), 266  
 grab, 236  
 gravitate (example), 245–261  
 gravitate2 (example), 261  
 gravitate3d.pyw (example), 272–282  
 grid() (tkinter.ttk.Widget), 235,  
 241–242, 248, 260  
 gzip (module), 64

## H

hashlib (module), 206  
 hello.pyw (example), 233–235  
 hierarchy, object, 40–48  
 hierarchy, ownership, 234  
 html (module)  
 escape(), 14, 33  
 HTMLParser (html.parser module),  
 121–122  
 Hyphenate1.py (example), 181–187  
 Hyphenate2 (example), 188–193  
 hypot() (math module), 85

## I

id() (built-in), 66  
 Image (example), 124–139, 151, 193,  
 199; *see also* cyImage example  
 image formats, 38; *see also* PhotoImage

GIF, 38, 124, 239  
 PGM, 124, 239  
 PNG, 38, 39, 124, 125, 137–139,  
 239  
 PPM, 124, 239  
 SVG, 6  
 XBM, 39, 125, 136  
 XPM, 39, 125, 135–137  
 image references, 256  
 imageproxy1.py (example), 68–71  
 imageproxy2.py (example), 69, 70  
 ImageScale (example), 164–177  
 imagescale.py (example), 199  
 imagescale-c.py (example), 145  
 imagescale-cy.py (example), 199  
 imagescale-m.py (example), 145,  
 152–154, 164, 199  
 imagescale-q-m.py (example), 145,  
 147–152  
 imagescale-s.py (example), 145, 199  
 imagescale-t.py (example), 145  
 import (statement), 125, 137, 166,  
 181, 189, 190, 195, 199, 200, 265  
 import\_module() (importlib module),  
 126  
 ImportError (exception), 125, 126,  
 137, 199  
 IndexError (exception), 95, 130  
 indirection; *see* pointer  
 \_\_init\_\_() (special method), 25, 43,  
 44, 45, 54, 76, 110, 113, 168, 234  
 initialize OpenGL, 267  
 initializer; *see* \_\_init\_\_()  
 input() (built-in), 85–86, 213  
 instance; *see* object  
 instate() (tkinter.ttk.Widget), 171,  
 218, 244  
 inter-process communication (IPC),  
 141  
 interaction, handling, 280–282  
 interface checking, 30–32, 35–36  
 I/O, asynchronous, 142

IPC (Inter-Process Communication), 141  
 is\_alive() (threading.Thread type), 177  
 isdigit() (str type), 16, 136  
 isidentifier() (str type), 113  
 isinstance() (built-in), 20, 30, 31, 51, 55, 58, 133, 207, 278  
 isoformat() (datetime.datetime type), 208–209, 226  
 items() (dict type), 14, 58, 87, 208  
 iter() (built-in), 44, 47, 95–97, 99  
 \_\_iter\_\_() (special method), 42, 44, 97, 99–100  
 iterator protocol, 97–100  
 itertools (module)  
   chain(), 21, 46, 109  
   product(), 276, 280

## J

JIT (Just In Time compiler), 179  
 join()  
   multiprocessing.JoinableQueue (type), 148, 150  
   os.path (module), 39, 65, 150, 256  
   queue.Queue (type), 160–161  
   str (type), 15, 18, 87, 92, 111, 122, 136  
   threading.Thread (type), 177, 226  
 JoinableQueue (type; multiprocessing module), 144, 148, 154, 158; *see also* Queue type  
 json (module), 92–93, 106–107  
 JSON-RPC, 204; *see also* xmlrpc module  
 just in time compiler (JIT), 179

## K

keyboard handling, 234, 242, 249, 250, 252, 257, 258, 269, 272, 281; *see also* bind()  
 KeyboardInterrupt (exception), 148, 154, 160, 163, 209

KeyError (exception), 98, 206  
 keys() (dict type), 223  
 keyword arguments, 13, 51–52  
 kill() (os module), 219  
 Kivy, 231

## L

Label (type)  
   pyglet.text (module), 274  
   tkinter.ttk (module), 234, 240, 247–248, 252, 260; *inherits* Widget  
 lambda (statement), 23–24, 64, 93, 116, 134, 181, 242, 249  
 late binding, 58  
 layouts; *see* grid(), pack(), and place()  
 lazy evaluation, 60  
 len() (built-in), 32, 63  
 \_\_len\_\_() (special method), 99, 223  
 library, shared, 180, 183, 188  
 line algorithm, Bresenham's, 130–131  
 lines, painting, 130–131, 271  
 list comprehensions, 18, 38, 49, 80, 123  
 list (type; built-in), 8, 70, 102, 138, 274  
   append(), 43, 102, 274  
   extend(), 15, 44, 46  
   remove(), 43  
   slicing, 138  
 listdir() (os module), 126, 150  
 literal\_eval() (ast module), 88  
 load() (pyglet.image module), 270  
 local grab, 236  
 locals() (built-in), 8, 9, 13, 86  
 Lock (type; threading module), 175–176, 220, 221  
 locking, 142, 143–144, 154, 169, 170, 175–176, 223–224  
 loose coupling, 104

lower() (str type), 16, 120  
 lstrip() (str type), 93  
 \_\_lt\_\_() (special method), 48; *see also* <

## M

magic number, 135  
 main-window applications, 253–261  
 main windows vs. dialogs, 235  
 mainloop() (tkinter.Tk), 166, 234, 238, 255  
 makedirs() (os module), 146, 172  
 Manager (type; multiprocessing module), 144, 168, 169  
 map() (built-in), 123  
 mapping; *see* dict type and collections.OrderedDict  
 mapping unpacking, 13, 222, 241–242, 248  
 MappingProxyType (type; types module), 223  
 math (module), 87  
   hypot(), 85  
 max() (built-in), 35, 151  
 mediator1.py (example), 59, 101–104  
 mediator1d.py (example), 59  
 mediator2.py (example), 104–106  
 mediator2d.py (example), 106  
 memory, shared, 141  
 Menu (type; tkinter module), 257, 258–259  
 menus, 257–260  
 messagebox (tkinter module), 166, 217, 218, 243  
 metaclasses, 12–14, 30–31, 35, 42, 59, 120  
 Meter.py (example), 205–208  
 meter-rpc.pyw (example), 214–228  
 meter-rpyc.pyw (example), 228–229  
 meterclient-rpc.py (example), 210–219  
 meterclient-rpyc.py (example), 227–228

MeterLogin.py (example), 214–215  
 meterserver-rpc.py (example), 208–210  
 meterserver-rpyc.py (example), 225–227  
 method  
   bound and unbound, 63, 69, 70, 83, 102  
   class, 10  
   decorators; *see* function decorators  
   special; *see* special methods  
   state-sensitive, 114–115  
   state-specific, 115–116  
 min() (built-in), 151, 273  
 minsize() (tkinter.Tk), 242  
 mkdir() (os module), 146  
 mock objects, 67  
 modality, 214, 235–237, 244, 245–250  
 model/view/controller (MVC), 107  
 modeless, 236–237, 244, 250–253  
 module, executable, 238  
 modules dictionary (sys module), 22–23, 25  
 mouse handling, 234, 280  
 \_\_mro\_\_ (class attribute), 30–31, 36  
 multiplexer1.py (example), 112–115  
 multiplexer2.py (example), 115–116  
 multiplexer3.py (example), 116  
 multiplexing, 100–106, 107–114  
 multiprocessing (module), 142–144, 146–154, 164, 173  
   Array (type), 144, 154  
   cpu\_count(), 145–146, 157, 173  
   JoinableQueue (type), 144, 148, 154, 158  
   join(), 148, 150  
   task\_done(), 148, 150  
   *for other methods; see* Queue type  
   Manager (type), 144, 168, 169  
   Process (type), 149, 150

Queue (type), 144, 148, 154, 158  
     get(), 149, 150  
     put(), 149, 150  
 Value (type), 144, 154, 168, 169  
 multithreading; *see* threading module  
 MVC (Model/View/Controller), 107

## N

\_\_name\_\_ (built-in), 16, 50, 146  
 name() (unicodedata module), 21  
 namedtuple (type; collections module), 88, 147, 160, 196, 205  
 nested class, 121–122  
 nested function, 58  
 \_\_new\_\_() (special method), 20, 22–23, 25  
 next() (built-in), 76, 77  
 \_\_next\_\_() (special method), 97  
 Notebook (type; tkinter.ttk module), 234, 261; *inherits* Widget  
 NotImplemented (built-in), 31, 36  
 NotImplementedError (exception), 18, 19, 120  
 now() (datetime.datetime type), 208–209, 213, 218, 226  
 Numba, 179  
 number, magic, 135  
 Number (numbers module), 55–56  
 numbers (module)  
     Number, 55–56  
 numpy (module), 124, 196; *see also* array module  
     fromiter(), 128  
     zeros(), 128

## O

-O optimize flag, 30  
 object  
     class, 25, 120  
     cloning, 24–25  
     dynamic creation, 24–25

function, 63  
 hierarchy, 40–48  
 mock, 67  
 reference, 64–67  
 selection in scene, 277–279  
 observer.py (example), 107–111  
 open()  
     built-in, 11, 64, 120, 136, 160  
     webbrowser (module), 161, 162  
 OpenGL (PyOpenGL), 264–270  
 GL (module), 265  
     glBegin(), 268, 271  
     glClear(), 267, 275  
     glClearColor(), 267  
     glColor3f(), 268, 271  
     glColor3ub(), 268, 277  
     glColorMaterial(), 267  
     glDisable(), 279  
     glEnable(), 267, 279  
     glEnd(), 268, 271  
     GLfloat, 267  
     glHint(), 267  
     glLightfv(), 267  
     glLoadIdentity(), 269, 275  
     glMaterialf(), 267  
     glMaterialfv(), 267  
     glMatrixMode(), 267, 269, 275  
     glOrtho(), 275  
     glPopMatrix(), 268, 276, 277  
     glPushMatrix(), 267, 268, 276, 277  
     glReadPixels(), 280  
     glRotatef(), 267, 276  
     glShadeModel(), 279  
     glTranslatef(), 267, 268, 277  
     GLubyte, 280  
     glVertex3f(), 268, 271  
     glViewport(), 269, 275  
 GLU (module), 265  
     gluCylinder(), 268  
     gluDeleteQuadric(), 268, 276  
     gluNewQuadric(), 268, 276  
     gluPerspective(), 269

## OpenGL (PyOpenGL) (continued)

## GLU (module) (continued)

gluQuadricNormals(), 268, 276  
gluSphere(), 277

## GLUT (module), 265

glutDestroyWindow(), 269  
glutDisplayFunc(), 266, 276  
glutDisplayString(), 266  
glutInit(), 265  
glutInitWindowSize(), 265  
glutKeyboardFunc(), 266, 269  
glutMainLoop(), 266  
glutPostRedisplay(), 269  
glutReshapeFunc(), 266, 275  
glutSpecialFunc(), 266

initialize, 267

operations, atomic, 143

## operators

!= not equal, 48  
& bitwise and, 133  
() call, generator, and tuple, 22;  
  *see also* `__call__()`  
\* multiplication and sequence un-  
  packing, 13, 26, 30–31, 43,  
  49, 70, 109  
\*\* mapping unpacking, 13, 222,  
  241–242, 248  
< less than, 48; *see also* `__lt__()`  
<= less than or equal, 48  
== equal, 48; *see also* `__eq__()`  
> greater than, 48  
>= greater than or equal, 48  
>> bitwise right shift, 133

optimization; *see* Cython

option\_add() (tkinter.Tk), 255

ord() (built-in), 87, 95

OrderedDict (type; collections mod-  
  ule), 85–86, 87

orthographic projection, 264, 275

## os (module)

kill(), 219  
listdir(), 126, 150  
makedirs(), 146, 172

mkdir(), 146

remove(), 217

## os.path (module)

abspath(), 146  
basename(), 136, 176  
dirname(), 126, 156, 162, 181, 216,  
  256  
exists(), 146, 172, 181, 217  
join(), 39, 65, 150, 256  
realpath(), 216, 256  
splittext(), 62–63, 126, 135

ownership, hierarchy, 234

## P

pack() (tkinter.ttk.Widget), 235, 252,  
  257, 260

packaging tools, 188

## painting

cylinders, 268  
lines, 130–131, 271  
spheres, 277  
windows, 267, 275

Panedwindow (type; tkinter.ttk mod-  
  ule), 261; *inherits* Widget

parsing, 84

parsing, command-line; *see* argparse  
  module

path list (sys module), 126

perspective projection, 264, 269

PGM (image format), 124, 239

PhotoImage (type; tkinter module),  
  124, 239, 256

pickle (module), 66, 94, 106–107,  
  152, 173

pipe; *see* subprocess module

pipeline; *see* coroutine

## pkgutil (module)

walk\_packages(), 126

place() (tkinter.ttk.Widget), 235

platform (sys module), 85–86

PNG (image format), 38, 124, 125,  
  137–139, 239

- pointer, 182, 190
  - POINTER() (ctypes module), 184
  - pointstore1.py (example), 65–67
  - pointstore2.py (example), 65–67
  - positional arguments, 13, 51–52
  - PPM (image format), 124, 239
  - Process (type; multiprocessing module), 149, 150
  - ProcessPoolExecutor (type; concurrent.futures module), 152, 153, 173
  - product() (itertools module), 276, 280
  - profiling; *see* cProfile module
  - properties, adding, 57–58
  - @property (built-in), 42, 44, 48, 54, 56, 58, 60, 104, 110, 115, 278
  - protocol, iterator, 97–100
  - protocol, sequence, 95–96
  - protocol() (tkinter.Tk), 166, 252, 255
  - put()
    - multiprocessing.Queue (type), 149, 150
    - queue.Queue (type), 158
  - .pxd suffix; *see* Cython
  - pycapsule (cpython module), 190
    - PyCapsule\_GetPointer(), 191–192
    - PyCapsule\_IsValid(), 191–192
    - PyCapsule\_New(), 191–192
  - pyglet, 263–264, 270–282
    - app (module)
      - run(), 270
    - clock (module)
      - schedule\_once(), 281
    - graphics (module)
      - draw(), 271
      - vertex\_list, 271
    - image (module)
      - load(), 270
    - text (module)
      - Label (type), 274
      - window (module)
        - Window (type), 270, 274
  - PyGObject, 232
  - PyGtk, 232
  - PyOpenGL, 263–270; *see also* OpenGL
  - PyPng (module), 137; *see also* PNG
  - PyPy, 179, 183
  - PyQt4, 232
  - PySide, 232
  - .pyw suffix, 237
  - .pyx suffix; *see* Cython
- ## Q
- qsize() (queue.Queue type), 160
  - queue (module), 144
    - Queue (type), 156, 158
      - get(), 158
      - join(), 160–161
      - put(), 158
      - qsize(), 160
      - task\_done(), 158
  - Queue (type)
    - multiprocessing (module), 144, 148, 154, 158; *see also* JoinableQueue type
    - queue (module); *see* top-level entry
  - quit() (tkinter.ttk.Widget), 177, 219, 242, 243
- ## R
- raise (statement), 30, 51, 55, 175, 183
  - randint() (random module), 206, 216, 220
  - random (module)
    - choice(), 206, 220, 274
    - randint(), 206, 216, 220
    - shuffle(), 274
  - range() (built-in), 18, 21
  - re (module), 17
    - compile(), 120
    - findall(), 90

- re (module) (*continued*)
    - search(), 63
    - split(), 26
    - sub(), 16, 39, 90–91, 136–137
    - subn(), 90–91
  - real events, 242–243
  - realpath() (os.path module), 216, 256
  - recursion, 44
  - references
    - function, 60–62, 70, 87
    - image, 256
    - object, 64–67
  - regex; *see* re module
  - register() (atexit module), 67, 187, 193
  - regular expression; *see* re module
  - remote procedure call (RPC); *see* rpyc module and xmlrpc module
  - remove()
    - list (type), 43
    - os (module), 217
  - render1.py (example), 29–34
  - render2.py (example), 32
  - replace() (str type), 21, 93, 185
  - resizable() (tkinter.Tk), 251–252, 256
  - resizing, window, 269, 275
  - reversed() (built-in), 83
  - Rich Site Summary (RSS) format, 155, 158, 159
  - round() (built-in), 111, 131, 132, 195
  - RPC (Remote Procedure Call); *see* rpyc module and xmlrpc module
  - rpyc (module), 203, 219–229
    - connect(), 227, 229
    - connect\_by\_service(), 228
    - Service (type), 226–227
    - utils (module)
      - server (module), 225–227
  - RSS (Rich Site Summary) format, 155, 158, 159
  - rstrip() (str type), 26, 122
  - run() (pyglet.app module), 270
- ## S
- Scalable Vector Graphics; *see* SVG image format
  - scene object selection, 277–279
  - schedule\_once() (pyglet.clock module), 281
  - search() (re module), 63
  - selection, of scene objects, 277–279
  - send() (generator method), 77, 78, 105–106, 112
  - sentinel, 96
  - sequence protocol, 95–96
  - sequence unpacking, 13, 26, 30–31, 70, 109
  - serialized access, 141–142, 143–144, 154
  - server; *see* xmlrpc module and rpyc module
  - ServerProxy (type; xmlrpc.client module), 211, 215
  - Service (type; rpyc module), 226–227
  - set() (tkinter.StringVar), 172, 176, 214, 217, 218, 261
  - set (type), 103, 109, 152–153, 162
  - setattr() (built-in), 22–23, 56, 58, 59, 113
  - \_\_setattr\_\_() (special method), 67
  - \_\_setitem\_\_() (special method), 222
  - setuptools (module), 188
  - SHA-256, 206
  - shared data, 141, 143–144, 154
  - shared library, 180, 183, 188
  - shared memory, 141
  - shelve (module), 65–67
  - shuffle() (random module), 274
  - signal (module), 219
  - SimpleNamespace (type; types module), 85–86

- SimpleXMLRPCRequestHandler (type; xmlrpc.server module), 210
- SimpleXMLRPCServer (type; xmlrpc.server module), 209, 210
- sleep() (time module), 217
- slicing, 138; *see also* list type
- \_\_slots\_\_ (class attribute), 19–20, 22–23, 65
- smart dialog, 236, 245
- sorted() (built-in), 14, 87, 208, 244
- special methods, *see also* class attributes
  - \_\_call\_\_(), 82, 96–97, 113; *see also* () operator
  - \_\_contains\_\_(), 99, 223
  - \_\_delitem\_\_(), 98, 223
  - \_\_enter\_\_(), 61, 279
  - \_\_eq\_\_(), 48; *see also* ==
  - \_\_exit\_\_(), 61, 279
  - \_\_getattr\_\_(), 66
  - \_\_getitem\_\_(), 95–96, 222
  - \_\_init\_\_(), 25, 43, 44, 45, 54, 76, 110, 113, 168, 234
  - \_\_iter\_\_(), 42, 44, 97, 99–100
  - \_\_len\_\_(), 99, 223
  - \_\_lt\_\_(), 48; *see also* <
  - \_\_new\_\_(), 20, 22–23, 25
  - \_\_next\_\_(), 97
  - \_\_setattr\_\_(), 67
  - \_\_setitem\_\_(), 222
  - \_\_str\_\_(), 18
  - \_\_subclasshook\_\_, 36
- sphere, painting, 277
- Spinbox (type; tkinter.ttk module), 240; *inherits* Widget
- split() (re module), 26
- splitext() (os.path module), 62–63, 126, 135
- startswith() (str type), 26, 63
- state-sensitive methods, 114–115
- state-specific methods, 115–116
- state() (tkinter.ttk.Widget), 170, 244
- statements, *see also* built-in
  - assert, 30, 37, 55, 69, 82, 127, 131
  - del, 98, 115, 126
  - import, 125, 137, 166, 181, 189, 190, 195, 199, 200, 265
  - lambda, 23–24, 64, 93, 116, 134, 181, 242, 249
  - raise, 30, 51, 55, 175, 183
  - with, 26, 61, 64, 92, 153, 159, 162, 176, 220, 222, 276, 279
  - yield, 44, 76–79, 100, 105, 139, 153, 160
- @staticmethod (built-in), 120, 121, 129, 133
- static type checking, 52
- stationery1.py (example), 40–45
- stationery2.py (example), 45–47
- status bar, 260–261
- StopIteration (exception), 96, 97
- \_\_str\_\_() (special method), 18
- str (type; built-in), 19–20, 66
  - encode(), 185, 186, 190, 191, 266
  - endswith(), 62, 120, 121
  - format(), 13, 21, 32, 37, 92, 109
  - format\_map(), 8–9
  - isdigit(), 16, 136
  - isidentifier(), 113
  - join(), 15, 18, 87, 92, 111, 122, 136
  - lower(), 16, 120
  - lstrip(), 93
  - replace(), 21, 93, 185
  - rstrip(), 26, 122
  - startswith(), 26, 63
  - strip(), 136
  - title(), 20–21
- string (module), 63
- StringVar (type; tkinter module), 168, 240, 247, 256
  - get(), 170, 172, 218, 243, 249
  - set(), 172, 176, 214, 217, 218, 261
- strip() (str type), 136

`strptime()` (`datetime.datetime` type),  
 84, 207, 227  
`sub()` (re module), 16, 39, 90–91,  
 136–137  
`__subclasshook__` (special method),  
 36  
 subclassing, alternative to, 58–59  
`subn()` (re module), 90–91  
`subprocess` (module), 92, 216  
`sum()` (built-in), 99, 163  
`super()` (built-in), 14, 19, 20, 23, 44,  
 76, 110, 168, 234  
 SVG (image format), 6  
`sys` (module)  
   `executable`, 92, 216  
   `exit()`, 91  
   modules (dictionary), 22–23, 25  
   `path` (list), 126  
   platform, 85–86

## T

`tabulator1.py` (example), 117  
`tabulator2.py` (example), 117  
`tabulator3.py` (example), 116–119  
`tabulator4.py` (example), 118  
`tarfile` (module), 62–63  
`task_done()`  
   `multiprocessing.JoinableQueue`  
   (type), 148, 150  
   `queue.Queue` (type), 158  
 Tcl/Tk; *see* tkinter module  
`tempfile` (module)  
   `gettempdir()`, 39, 65, 161, 216  
 testing, unit, 67  
 Text (type; tkinter module), 261  
`texteditor` (example), 261–262  
`texteditor2` (example), 261–262  
`textwrap` (module)  
   `fill()`, 32  
 Thread (type; threading module), 150,  
 157, 172, 226  
   `is_alive()`, 177

`join()`, 177, 226  
 thread-safe data wrapper, 219–224  
 thread-safe dictionary, 221–224  
 threading (module), 142–144, 164  
   Lock (type), 175–176, 220, 221  
   Thread (type); *see* top-level entry  
`ThreadPoolExecutor` (type; `concurrent.futures` module), 152, 154,  
 162  
`throw()` (generator method), 77  
`time` (module)  
   `sleep()`, 217  
   `time()`, 109, 110  
`time()` (time module), 109, 110  
 timer; *see* `after()` and `schedule_once()`  
 timings, 145, 152, 156, 199  
`title()`  
   str (type), 20–21  
   Tk (tkinter module), 215, 234,  
   238, 255  
 Tk (tkinter module), 166, 238, 255; *see*  
   *also* `Widget`  
   `after()`, 214, 239, 261  
   `call()`, 239  
   `create_command()`, 258–259  
   `deiconify()`, 166, 250–252, 255  
   `geometry()`, 253  
   `mainloop()`, 166, 234, 238, 255  
   `minsize()`, 242  
   `option_add()`, 255  
   `protocol()`, 166, 252, 255  
   `resizable()`, 251–252, 256  
   `title()`, 215, 234, 238, 255  
   `wait_visibility()`, 251–252  
   `withdraw()`, 166, 251–252, 253,  
   255  
 tkinter (module), 166  
   Canvas (type), 257  
   DISABLED, 170  
   documentation, 234  
   `filedialog`, 166  
   Menu (type), 257, 258–259

- messagebox, 166, 217, 218, 243
  - PhotoImage (type), 124, 239, 256
  - StringVar (type), 168
  - Text (type), 261
  - Tk; *see* top-level entry
  - Toplevel (type), 251–252
  - ttk (module); *see* top-level entry
  - tkinter.ttk (module), 166
    - Button (type), 234; *inherits* Widget
    - Combobox (type), 234, 240; *inherits* Widget
    - Frame (type), 166, 168, 234, 247, 255, 260; *inherits* Widget
    - Label (type), 234, 240, 247–248, 252, 260; *inherits* Widget
    - Notebook (type), 234, 261; *inherits* Widget
    - Panedwindow (type), 261; *inherits* Widget
    - Spinbox (type), 240; *inherits* Widget
    - Treeview (type), 234, 261; *inherits* Widget
    - Widget (type)
      - bind(), 169, 242, 249, 252, 257; *see also* keyboard handling
      - cget(), 171, 241, 244
      - config(), 170, 176, 243, 244, 247, 257; *alias of* configure()
      - focus(), 168, 169, 217, 239
      - grid(), 235, 241–242, 248, 260
      - instate(), 171, 218, 244
      - pack(), 235, 252, 257, 260
      - place(), 235
      - quit(), 177, 219, 242, 243
      - state(), 170, 244
      - update(), 170, 176
  - toolbar, 261–262
  - Toplevel (type; tkinter module), 251–252
  - total\_ordering() (functools module), 48
  - Treeview (type; tkinter.ttk module), 234, 261; *inherits* Widget
  - ttk; *see* tkinter.ttk module
  - tuple concatenation, 109
  - two-part application design, 216
  - type; *see* class
  - type() (built-in), 22–23, 30, 86
  - type checking, static, 52
  - TypeError (exception), 30, 49, 89
  - types (module)
    - MappingProxyType (type), 223
    - SimpleNamespace (type), 85–86
- ## U
- unbound method, 63, 69, 70
  - Unicode, 17, 20, 21, 131, 215, 243; *see also* UTF-8
  - unicodedata (module)
    - name(), 21
  - unit testing, 67
  - Unpack.py (example), 60–64
  - unpacking, mapping and sequence, 13, 26, 30–31, 70, 109, 222, 241–242, 248
  - update() (tkinter.ttk.Widget), 170, 176
  - urllib.request (module)
    - urlopen(), 26, 159
  - urlopen() (urllib.request module), 26, 159
  - user interaction, handling, 280–282
  - UTF-8, 17, 66, 121, 160, 182, 185, 243
- ## V
- Value (type; multiprocessing module), 144, 154, 168, 169
  - ValueError (exception), 26, 51, 113
  - values() (dict type), 99, 187, 224

vertex\_list (pyglet.graphics module), 271

virtual events, 242–243

## W

wait() (concurrent.futures module), 173–174

wait\_visibility() (tkinter.Tk), 251–252

walk\_packages() (pkgutil module), 126

warn() (warnings module), 126

webbrowser (module)

open(), 161, 162

whatsnew-q.py (example), 155, 156–161

whatsnew-t.py (example), 155, 161–164

Widget (type; tkinter.ttk module)

bind(), 169, 242, 249, 252, 257; *see also* keyboard handling

cget(), 171, 241, 244

config(), 170, 176, 243, 244, 247, 257; *alias of* configure()

focus(), 168, 169, 217, 239

grid(), 235, 241–242, 248, 260

instate(), 171, 218, 244

pack(), 235, 252, 257, 260

place(), 235

quit(), 177, 219, 242, 243

state(), 170, 244

update(), 170, 176

Window (type; pyglet.window module), 270, 274

windows

and classes, 234

dock, 261–262

main, 255–261

main vs. dialogs, 235

modal, 236–237

painting, 267, 275

resizing, 269, 275

with (statement), 26, 61, 64, 92, 153, 159, 162, 176, 220, 222, 276, 279

withdraw() (tkinter.Tk), 166, 251–252, 253, 255

wordcount1.py (example), 119–122

wordcount2.py (example), 120

wrappers; *see* decorator, class and decorator, and function

wraps() (functools module), 50, 51, 77

wxPython, 232

## X

XBM (image format), 39, 125, 136

XML (eXtensible Markup Language), 155, 204

xmlrpc (module), 203, 204–219

client (module), 210–219

DateTime (type), 207

ServerProxy (type), 211, 215

server (module), 207, 208–210

SimpleXMLRPCRequestHandler (type), 210

SimpleXMLRPCServer (type), 209, 210

XPM (image format), 39, 125, 135–137

## Y

yield (statement), 44, 76–79, 100, 105, 139, 153, 160

## Z

zeros() (numpy module), 128

zip() (built-in), 51, 138

zipfile (module), 62–63

zombie, 149, 177

## Mark Summerfield

Mark is a computer science graduate with many years experience working in the software industry, primarily as a programmer and documenter. Mark owns Qtrac Ltd. ([www.qtrac.eu](http://www.qtrac.eu)), where he works as an independent author, editor, consultant, and trainer, specializing in the C++, Go, and Python languages, and the Qt, PyQt, and PySide libraries.

Other books by Mark Summerfield include

- *Programming in Go* (2012, ISBN-13: 978-0-321-77463-7)
- *Advanced Qt Programming* (2011, ISBN-13: 978-0-321-63590-7)
- *Programming in Python 3* (First Edition, 2009, ISBN-13: 978-0-13-712929-4; Second Edition, 2010, ISBN-13: 978-0-321-68056-3)
- *Rapid GUI Programming with Python and Qt* (2008, ISBN-13: 978-0-13-235418-9)

Other books by Jasmin Blanchette and Mark Summerfield include

- *C++ GUI Programming with Qt 4* (First Edition, 2006, ISBN-13: 978-0-13-187249-3; Second Edition, 2008, ISBN-13: 978-0-13-235416-5)
- *C++ GUI Programming with Qt 3* (2004, ISBN-13: 978-0-13-124072-8)

---

## Production

The text was written using the *gvim* editor. The typesetting—including all the diagrams—was done using the *lout* typesetting language. All of the code snippets were automatically extracted directly from the example programs and from test programs using custom tools. The index was compiled by the author. The text and source code was version-controlled using *Mercurial*. The monospaced code font was derived from a condensed version of *DejaVu Mono* and modified using *FontForge*. The book was previewed using *evince* and *gv*, and converted to PDF by *Ghostscript*. The cover was provided by the publisher. Note that only English print editions are definitive; *ebook* versions and translations are not under the author's control and may introduce errors.

All the editing and processing was done on Debian Linux systems. All the book's examples have been tested with Python 3.3 (and, where possible, Python 3.2 and Python 3.1) on Linux, OS X (in most cases), and Windows (in most cases). The examples are available from the book's web site, [www.qtrac.eu/pipbook.html](http://www.qtrac.eu/pipbook.html), and should work with all future Python 3.x versions.

*This page intentionally left blank*

# REGISTER



## THIS PRODUCT

[informit.com/register](http://informit.com/register)

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **[informit.com/register](http://informit.com/register)** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

**informIT.com**

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

## LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit [informit.com/newsletters](http://informit.com/newsletters).
- Access FREE podcasts from experts at [informit.com/podcasts](http://informit.com/podcasts).
- Read the latest author articles and sample chapters at [informit.com/articles](http://informit.com/articles).
- Access thousands of books and videos in the Safari Books Online digital library at [safari.informit.com](http://safari.informit.com).
- Get tips from expert blogs at [informit.com/blogs](http://informit.com/blogs).

Visit [informit.com/learn](http://informit.com/learn) to discover all the ways you can access the hottest technology content.

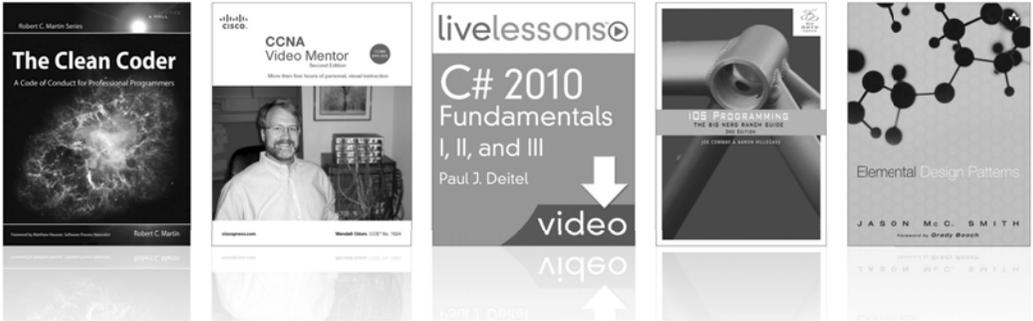
### Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit [informit.com/socialconnect](http://informit.com/socialconnect).



# Try Safari Books Online FREE for 15 days

## Get online access to Thousands of Books and Videos



**Safari**<sup>®</sup>  
Books Online

**FREE 15-DAY TRIAL + 15% OFF\***  
[informit.com/safaritrial](http://informit.com/safaritrial)

### ➤ Feed your brain

Gain unlimited access to thousands of books and videos about technology, digital media and professional development from O'Reilly Media, Addison-Wesley, Microsoft Press, Cisco Press, McGraw Hill, Wiley, WROX, Prentice Hall, Que, Sams, Apress, Adobe Press and other top publishers.

### ➤ See it, believe it

Watch hundreds of expert-led instructional videos on today's hottest topics.

## WAIT, THERE'S MORE!

### ➤ Gain a competitive edge

Be first to learn about the newest technologies and subjects with Rough Cuts pre-published manuscripts and new technology overviews in Short Cuts.

### ➤ Accelerate your project

Copy and paste code, create smart searches that let you know when new books about your favorite topics are available, and customize your library with favorites, highlights, tags, notes, mash-ups and more.

\* Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.



Adobe Press

Cisco Press



IBM Press

Microsoft Press



O'REILLY



PEARSON  
IT Certification



QUE

SAMS

vmware PRESS

