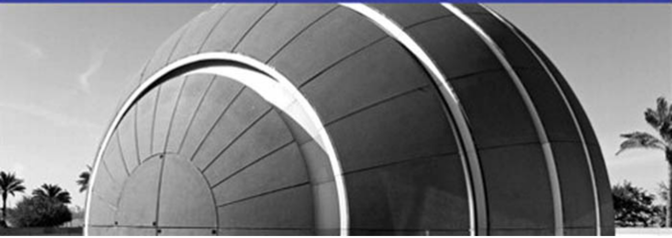
Mark Summerfield

Foreword by Doug Hellmann,
Senior Developer, DreamHost

Python in Practice

Create Better Programs Using Concurrency, Libraries, and Patterns

Developer's Library



FREE SAMPLE CHAPTER

Python in Practice

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

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

Library of Congress Control Number: 2013942956

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

`www.qtrac.eu/pipbook.html`

This page intentionally left blank

Contents

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`). 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`). 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 `xmlrpclib` 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 `xmlrpclib` 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`). There are also a great many web frameworks—from the lightweight `bottle` (`bottlepy.org`) to heavyweights like Django (`www.djangoproject.com`) and Pyramid (`www.pylonsproject.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`, 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

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.*

```
+----------------------------+
|   +--------------------+   |
|   |%%%%%%%%%%%%%%%%%%%%|   |
|   |%%Abstract Factory%%|   |
|   |%%%%%%%%%%%%%%%%%%%%|   |
|   +--------------------+   |
+----------------------------+
```

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.qtrac.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`.

`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.

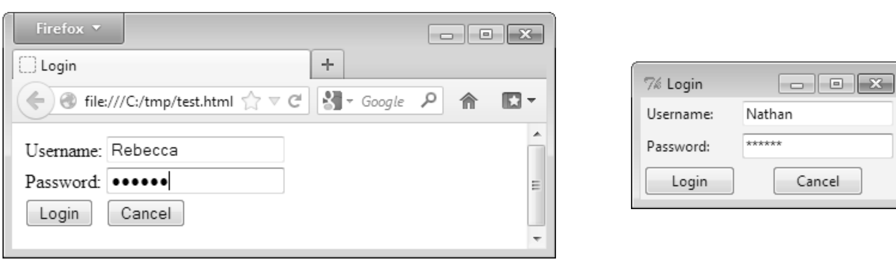


Figure 1.2 *The HTML and Tkinter forms on Windows*

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(AbstractFormBuilder):

    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        ".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`.[✪]

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.

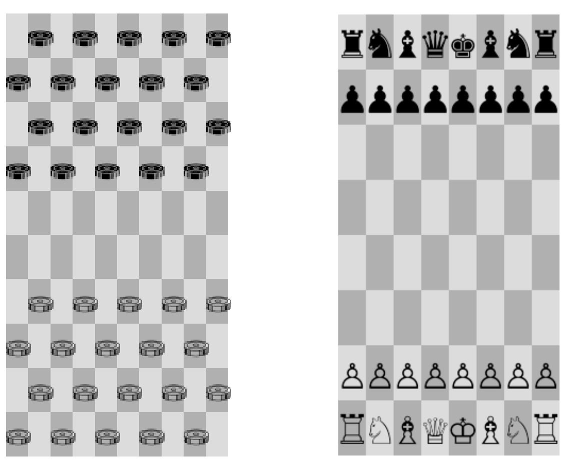


Figure 1.3 *The checkers and chess game boards on a Linux console*

[★] 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`.

[✪] 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`).

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 × 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/`) 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

B

C

E

F

G

P

Q

R

U

V

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`), 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`, and should work with all future Python 3.*x* versions.

This page intentionally left blank