Table of Contents

Python: Actually Cool Stuff

This repository contains the source code for my work-in-progress--a self-published book, titled "Python: Actually Cool Stuff."

For a quick view at what this book will have to offer, check out the Table of Contents.

Brief Overview

Are you a programmer and wondering what you can *really* do with your code? Are you tired of writing programs that manipulate numbers and basic strings? Do you want to make something *actually cool* with your skills as a programmer? Well, you're in the right place!

The goal of this book is to give Python programmers a quick, reference/study for implementing some of the tasks many beginner/intermediate programmer's find daunting, or would like to do but have no idea *how* to do them. Ultimately, you can think of this as a collection of some of my useful Python scripts and packages, and hopefully you'll find use in their existence and knowledge in learning how they function (haha, a pun).

Check out the TODO List to see current plans for the book's development.

Introduction

Prev: Table of Contents | Next: Chapter 1 - Serialization

Hello, and welcome to *"Python: Actually Cool Stuff"*. There may be a few questions on your mind at this moment, concerning this book, and I would like to first address those, as follows:

Q: *What will you learn from this book?*
A: After reading this book, you should have a good reference point for future, larger projects of your own. This book is by means intended to give a firm, complete understanding of the topics covered, but as introductory and suggestive material to help kick-start your own usage of such topics.

Q: *What should you know already, before reading this?*
A: This book is certainly not for beginners of the Python language, however I also strive to keep the complexity of the code to a minimal. It would be best if you already had the fundamentals under your belt, is all.

Q: *Who are you, Zach...King?*
A: Well, briefly, I'm a self-taught programmer and game enthusiast/developer, now studying in Computer Science to earn my B.S. For a more detailed look into my persona, feel free to check the About the Author section.

Q: *Is this book just another dry pit of jargon that will put me to sleep?*
A: I'm glad you asked--I would've too--seeing as most computer science related books are that way. However, I strive to make this book the exact *opposite* of dry (since I myself hate that kind of reading). Besides, if you pay attention, you may even find a couple of Monty Python references!

Entire Book Source

Source Code

Chapter 1: Serialization

Prev: Introduction | Next: Chapter 2 - Sending Email

Summary:

Have you ever cooked anything? I'm betting you have. I'm also betting you've had leftovers that you might have frozen or otherwise stored for later. Well, in this chapter, we're going to discuss how to do the same thing with objects in Python!

As you create projects with more and more complex classes and objects, you'll eventually find yourself wanting to save the state of the instances you create. In other words, you want to be able to keep the values of their variables and "state" and recreate them with no hassle. That's where serialization comes in to save the day. I like to think of serialization as freezing food for later—or as you'll soon see in Python, we'll call it pickling. When you're ready to save the food, or object, for later use, you just serialize it and store it away; then later when you want to use it again, you thaw it out--or in our case deserialize it!

Okay, let's jump right into it then, shall we!

Content:

```
# Example 1-1 (simple_serialize.py):
import pickle
simple_data = list(range(10))
pickle.dump(simple_data, open("simple.sav", "wb"))
```

First thing's first, we have to import our package containing needed sorcery, *pickle*. This package comes installed with Python and is awesome for easy serialization/deserialization. Then we generate some simple data that is just a list of the numbers 0-9. Finally, we want to save this object, so we just "dump" it in a file called 'simple.sav.' You typically want to use the "wb" mode on the open function, which means "write binary" so it pickle will write the data as binary. Note: you can call this file whatever you want and give it any extension or none at all; I simply like .sav for my "save" files.

Now you ask, *"Okay...but how do I get my data back?"* Here comes the deserialization part:

```
# Example 1-2 (simple_deserialize.py):
import pickle
simple_data = pickle.load(open("simple.sav", "rb"))
print(simple_data)
```

So the only difference is, now we're "loading" instead of "dumping." This time, the open function will use the "read binary" mode, since we wrote the save file as binary. Thanks to the `print()` statement at the end, we can verify that our simple data is indeed intact.

> "That's just a list though. What about my super complex class I built? How do I save all of its awesome data?"

Have no fear, Python is here. First, we'll need a "complex class" like the following:

```
# Example 1-3 (complex.py):
class ComplexThing:
    def __init__(self, name, data):
        self.name = name
        self.data = data

    def __str__(self):
        return self.name + ": " + str(self.data)
```

And then the serialization bit:

```
# Example 1-4 (complex_serialization.py):
import pickle
from complex import ComplexThing

# Create the instance
awesome_thing = ComplexThing("awesomeness", 15)
print("Before Pickling:")
print(awesome_thing)

# Pickle the instance
pickle.dump(awesome_thing, open("complex.sav", "wb"))

# Load the instance from save
loaded = pickle.load(open("complex.sav", "rb"))

# Print the loaded instance to verify intact data
print("\nAfter Loading from Pickle: ")
print(loaded)
```

You mean to pickle an instance of my custom class, I simply do the same thing as before? Yep, it's that easy. This example might seem different, but the serialization is the same; you just pass in the instance of your class to `pickle.dump()` instead of the simple data from

Example 1-1. Please note, however, that with very large or nested data, you might need to rethink your storage method (I highly recommend some research on the Python *shelve* package for this).

Wrap Up:

So now you know how to "freeze" your objects for later, and then bring them back to life when the time calls! This topic may seem boring at first, but trust me when I say you'll thank me later. An easy application of serialization is a configuration class for say...a text editor you created (totally not going to make one of those later in this book...[hint of sarcasm]). Using serialization, you can save the values of all the settings the user has set for the application, and when they hit "Save Settings" it dumps the pickled data into a save file. Then, whenever the user starts the application up, it checks for a save file and loads the settings from it if it exists.

Prev: Introduction | Next: Chapter 2 - Sending Email

Chapter 2: Sending Email

Prev: Chapter 1 - Serialization | Next: Chapter 3 - Cryptography

Summary:

As you've probably guessed after seeing the title of this chapter, we're now going to take a look at how to send emails with Python. If you've ever done any networking before, this process will seem like a walk in the park--which it is, thanks to Python.

To send emails with Python, we'll use the *smtplib* package that comes installed with Python. SMTP stands for "Simple Mail Transfer Protocol" and is how email is generally sent across the interwebs. The way we send email is using a socket connection to the SMTP port for our email service (i.e. Gmail, Yahoo, Hotmail, etc.). The default port to connect to for SMTP is the TCP (Transmission Control Protocol) port, 25. However, mail submission is done through port 587, which we'll be using. If you are more interested in networking, I would suggest looking into the *socket* package of Python.

Contents:

To dive right into things, let's take a look at this simple command-line interface (CLI) for sending an email:

```
# Example 2-1 (email_example.py):
import smtplib
from getpass import getpass
from email.mime.text import MIMEText

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
username = input("Enter your email address: ")
password = getpass(prompt="Enter your password: ")
server.login(username, password)

recipient = input("\nEnter Recipient Email Address: ")
subject = input("Enter Subject: ")
msg = input("\nEnter message: ")

mail = MIMEText(msg)
mail['Subject'] = subject
mail['From'] = username
mail['To'] = recipient

# Send mail
server.send_message(mail)
server.quit()
```

First, we import the needed tools such as the obvious, *smtplib*. We also make use of the *getpass* module of Python for retrieving the password from the user more appropriately. Lastly, the *MIMEText* class that we import will be used to construct our mail object with subject, message, etc.

After the imports are done, we can create the SMTP connection and login. The SMTP class takes two parameters here: the host and port to connect to; in this case, we are using Google's gmail in particular. After creating this, a simple call to `starttls()` which tells our connection to encrypt the sent messages for safer communication.

Next, we need to login to our Gmail account, so we get the username and password from the user and login. Using `getpass()` we can get the password without showing the characters entered by the user.

Before constructing our MIMEText object, we need to have the user enter the necessary information. So we get the recipient's email address, the subject line, and the message itself to be sent. Finally, we can build the MIMEText object and set it's headers appropriately. Note: you can actually send mail without using the MIMEText object; however, MIMEText is more verbose, and you must use other MIME objects for different media such as images.

Then, all that's left is to hit send! So we call `send_message()` and `quit()` which sends the email and logs out, closing our socket connection.

Wrap Up:

There you have it. You can now send emails to your hearts content using Python. Of course, you can delve further into this topic if you like. The best place to start is of course the Python documentation on the *smtplib* package. From there you can learn how to retrieve mail from your inbox, send images and other attachments, etc. I developed a graphical user interface (**gmailer.py**) for this program using *Tkinter* for Python (which we will discuss later), and can be found with the other source code provided with this book.

Prev: Chapter 1 - Serialization | Next: Chapter 3 - Cryptography

Chapter 3: Cryptography

Prev: Chapter 2 - Sending Email | Next: Chapter 4 - Web Scraping

Summary:

These days everyone carries gigabytes--even terabytes--of data on them at all times. Thus, we face a very important task--keeping that data locked down tight and out of the wrong hands! That's where the fun of cryptography comes in.

You may have seen some examples of encryption before, and perhaps thought it would be just as hard to write an encryption program as it is to crack the cipher behind it. But fear not, for nothing is so challenging with Python--seriously, this chapter will make you look like a genius (but of course you are one...).

So what we'll go through in this chapter is first some basic study of modern encryption (without the nitty gritty details, which we'll leave for cryptographers).
Then we'll use a third-party package for Python, known as *PyCrypto*, to write a very basic encryption/decryption program to use from the command-line.

History:

Before moving into the meat of this chapter, I'd like to give a brief history lecture regarding the origins of cryptography. If you find this boring, or already know it all, feel free to skip to the good stuff.

The true origins of cryptography date way back all the way to the Egyptians. They of course used hieroglyphs to communicate, and only the scribes knew these glyphs. Later, a more well-known instance of encryption was used by Julius Caesar for private communication; this encryption involved shifting the letters in the message by a certain number--a *key*--and is known as the *Caesar Cipher*.

It wasn't until World War II that cryptography became extremely mathematical. During the war it was used by armies to communicate without the worry of the enemy obtaining the information. A very prominent figure of this time is Alan Turing, who cracked the infamous Enigma Cipher used by the Germans. Arguably, one might say this is what led to The Allies winning the war.

Content

First, let's cover some basic concepts of cryptography. Cryptography is defined as "the art of writing or solving codes." Quite a generic definition, but that just shows how broad the topic truly is. In most instances of cryptography there are two primary components: *plaintext* and *ciphertext*. The plaintext is the message, or data, in which you wish to keep secret, yet it is simply...plain; the message is just sitting there waiting for people to read. Ciphertext is the *encrypted* version of the plaintext; this is how data is kept secure. So how does one's data go from plaintext to ciphertext?

Well, there are a variety of cryptographic methods for this task. Some involve shuffling the characters, exchanging characters, and more realistically, manipulating the data on a bit level. All of these methods make use of a *key* however. This is how the message is *decrypted* back to its original, readable format.

Nowadays, there are two types of encryption: symmetric and asymmetric.

Wrap Up:

Prev: Chapter 2 - Sending Email | Next: Chapter 4 - Web Scraping

Chapter 4: Web Scraping

Prev: Chapter 3 - Cryptography | Next: Chapter 5 - GUIs

Summary:

Big data. Big deal. In this chapter, we're going to take a gander at a subject known as "Web Scraping." What's that you ask? Is that when you crawl the Internet for days trying to escape the dark web? No, that's called poor judgement. Web scraping is when a script or program retrieves information from a website for processing, analyzing, or organizing.

So why did I mention big data? Well, since the computer is retrieving the data from the website(s), it can get that data and crunch through it pretty quickly. So, naturally, web scraping is popular among those seeking the usage and analytics of "big data." You could grab the numbers and statistics from a large collection of websites, store it in a neat data structure, and produce whatever your heart desires--graphs, charts, tables, etc. manufactured from the data.

Maybe handling a sea of data seems daunting to you though. Perhaps we could study the more common applications of web scraping? Sure! Web scraping can be used for myriad tasks, such as:

- Sports statistics
- Social media trends
- Online product comparisons
- Stock data aggregation
- Crime statistics

So, in this chapter we'll get to look at the basics of scraping the web using a pretty sweet third-party Python package called *BeautifulSoup*.

Content:

Alright, so first we need to install *BeautifulSoup* for Python. The newest version, as of the writing of this book, is v4.4.1, which supports both Python 2.6+ and Python 3. You can install the package with whatever package manager you prefer, such as *pip* or *easy_install*, like so:

```
pip install beautifulsoup4
```

```
easy_install beautifulsoup4
```

Or, if you don't like that method, you can always visit the website for the package at https://www.crummy.com/software/BeautifulSoup/bs4/download/.

To test that you installed the package successfully, you can simply start the Python interpreter, and run the following import command:

```
from bs4 import BeautifulSoup
```

If that import command completed without error, you're setup and ready to scrape!

Now then, before we actually start using the *bs4* package, let's first take a look at the general process of web scraping, as follows:

1. Study the format of target webpage
2. Retrieve content of webpage (an HTTP request)
3. Search the content for specific data based off the content format
4. Do stuff with the data found in the search

Okay, let's put this into practice. This first example is so simple, we don't need to use *bs4*.

```
# Example 4-1 (scrape0.py):
import requests
page = requests.get('http://github.com/zach-king/')
print(page.content)
```

Here, we're using the builtin *requests* package from Python (2 and 3). After importing the package, we use the `get()` function to retrieve the HTML document that the server sends back for my GitHub profile, at http://github.com/zach-king.

After storing that in a variable called `page` we can print its contents out and verify the same HTML is received this way as when we view it in a web browser. Keep in mind that you can view the source code of a webpage in your browser by right clicking and selecting either *inspect element* or *view source* (I recommend using *inspect element* since you can more accurately tell what elements you are looking at).

Now that we've seen the basic retrieval of HTML from a webpage, we can actually begin *scraping* it for our target information. The two most popular methods of doing this task in Python are using either *regular expressions* or *bs4*. However, regular expressions--which will be covered in chapter 9-- are typically not suggested in comparison with bs4. Let's take a look at an example using the `BeautifulSoup` class in bs4, and see the basics it has to offer:

```
# Example 4-2 (scrape1.py)
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://www.github.com/zach-king/CoolPython"
content = urlopen(url).read()
print(content)

soup = BeautifulSoup(content)

print(soup.prettify())
print(soup.title)
print(soup.title.string)
print(soup.p)
print(soup.a)
```

First off, note that this time we use the *urllib.request* package and its `urlopen()` function for retrieving the HTML; this is simply to showcase another way of accomplishing the same task. As you can see, we just store the HTML content in the content variable and print it out (ugly output, that is). Then, we create the mystical `BeautifulSoup` object with the HTML content passed in. After creating that, we can access many things easily, such as the `<title>` element and text, the first `<p>` (paragraph) element, and the first `<a>` (link) element. Also note that the soup object has a `prettify()` method which returns a long string of a *prettier* format than the jumbled HTML originally received.

While this is cool and all, next we need to start figuring out how to really get something useful out of the webpage. The next example demonstrates the scraping of all the links found on the repository page for this book:

```
# Example 4-3 (scrape2.py)
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://www.github.com/zach-king/CoolPython"
content = urlopen(url).read()

soup = BeautifulSoup(content)

for link in soup.find_all('a'):
    print(link.get('href'))
```

Nearly all of the code is the same, except for the last snippet. This uses one of BeautifulSoup's most powerful methods, `find_all()`. This method can be used in a plethora of ways, with different arguments passed to it. In this case, we want all of the `<a>` tags, and more specifically we want their actual link (the `href` attribute). So if we pass in 'a' to `soup.find_all()`, we will get a list of items of type `bs4.element.Tag`. This class in bs4 then contains a nifty method called `get()` used for getting the value of attributes within the tag; so to extract the href from the links we simply call this method and pass 'href' as the argument!

After scraping all the links out of a page, you might begin to wonder how you would scrape different tags *from within other tags*. This comes up when you find yourself targeting a specific `<div>` or section of a webpage, and you need to look at its *children*.

So let's take a look at yet another example to learn how to do just that!

```
# Example 4-4 (scrape3.py)
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "https://github.com/zach-king?tab=repositories"
content = urlopen(url).read()

soup = BeautifulSoup(content, "html.parser")

for repo in soup.find_all("h3", {"class":"repo-list-name"}):
    children = repo.children
    for child in children:
        print(child)
```

Once again, no surprises with the HTML and soup object. But wait! This time we used the `find_all()` method completely differently! Yes, this time we are scraping the repositories from my GitHub profile. If you first study the HTML of the repositories tab on any GitHub profile, you'll notice each repo's title is stored in an `<a>` tag *within* an `<h3>` tag that has a class attribute of 'repo-list-name.' Thus, we find these specific h3 tags using `find_all()` and pass in a dictionary of required attribute/value pairs. Then, for each matched h3 tag, we access its children with the `children` attribute, loop through them, and print them out! How easy was that?

Now, I'm sure you're thinking these examples are dull and only exist to generate more visits to my GitHub profile--boring as it is--and that's why the last 'examples' I have are actual applications of 'web scraping.'

This first application is very simple; it simply scrapes the 10 most viewed bills on congress.gov.

```
# Example 4-5 (gov-bills-scraper.py)
import requests
from bs4 import BeautifulSoup

url = "https://www.congress.gov/resources/display/content/Most-Viewed+Bills"
r = requests.get(url)
soup = BeautifulSoup(r.content, 'html.parser')
for i, bill in enumerate(soup.find_all("td", {"class":"confluenceTd"})[:30]):
    try:
        print(bill.text)
    except:
        pass
    if (i+1) % 3 == 0:
        print("\n")
```

Let's break it down. The first step in writing an application like this is to study the HTML in the web browser. I could see that the bills' information was being stored in `<td>` (table data) tags inside a table. Please note that tables are sometimes difficult to scrape information from, depending on how standardized and well-designed the table is.

The things I wanted for each bill was its Senate or House of Representatives code, its title, and its rank in the most-viewed list. These three things are found consecutively for each bill, but each bill has a span (no relation to the span tag...) of five matched td tags.

So I know that the *top 10* bills will consist of the first 30 matched td tags. Furthermore, with every third td tag, I want to print out a newline to provide some separation of the bills in the output. To do this I use the `enumerate()` function in Python which counts the iterations for me; then I use the modulus operator to print a '\n' after every three iterations. And that's it!

Now, the very last example is a bit longer than the previous ones, and for that reason, I will not go through it line by line. It is an application which takes a Python list of 'symbols' for different companies and scrapes their current value from the Yahoo! finance page. This example particularly showcases the capabilities of 'web-scraping' since it actually 'scrapes' multiple pages, and then produces a bar graph of that data--colored and all!

The foundation of this application was of course studying how the URLs of these company stock pages are formed on Yahoo! finance. I noticed that the website uses a query with a variable `s` (probably stands for symbol) which stores the value of the symbol, or company, being looked up. So the URL for Apple on Yahoo! finance is http://finance.yahoo.com/q?&s=aapl.

From there, it was all about using bs4 to extract the desired information out of the web pages. I'll leave the example here for you to study, if you so wish, and keep in mind that all source/examples can be found at the book's GitHub page. Here's the application:

```
import requests
from bs4 import BeautifulSoup
import numpy as np
import matplotlib.pyplot as plt

# Variable declarations
symbols = ["aapl", "goog", "msft", "yhoo", "amzn", "fb", "twtr", "nflx"]
numCompanies = len(symbols)
values = []
arrows = []

# Scrape the data
for s in symbols:
    url = "http://finance.yahoo.com/q?&s=" + s
    r = requests.get(url)
    soup = BeautifulSoup(r.content, "html.parser")
    price = soup.find_all("span", {"id": "yfs_l84_"+s})
    arrow = soup.find_all("span", {"id": "yfs_c63_"+s})
    arrows.append(arrow[0].next_element["class"])
    print(s.upper() + " share value: $" + price[0].text)
    values.append(price[0].text)
    symbols[symbols.index(s)] = symbols[symbols.index(s)].upper()

# convert values from strings to floats
for i in range(numCompanies):
    values[i] = float(values[i])

# Plot and figure
fig = plt.figure()
fig.canvas.set_window_title('Stock-Scraper')
ax = fig.add_subplot(111)

index = np.arange(numCompanies)
rects = ax.bar(index, values, 0.25, alpha=0.5)

# Show the number value for the share above the bar
for i in range(numCompanies):
    ax.annotate('$' + str(values[i]), xy=(i+0.12, values[i]), xytext=(i - 0.12, values
[i] + 10),
            arrowprops=dict(arrowstyle='-'),
            )

# Labels and titles on graph
plt.xlabel('Company')
plt.ylabel('Share Value')
plt.title('Current Share Value for Companies')
plt.xticks(index + 0.25, symbols)

# Color the bars based on the companies' stock status
for i, arrow in enumerate(arrows):
    if (arrow == [u'pos_arrow']):
        rects[i].set_color('g') # stock is going up
```

```
    elif (arrow == [u'neg_arrow']):
        rects[i].set_color('r') # stock is going down
    else:
        rects[i].set_color('b') # unknown data

# Show the graph
fig.tight_layout()
plt.show()
```

Wrap Up:

Well there you have it. You now have the metaphorical 'scraper' known as Python, waiting to crawl the World Wide Web. I highly recommend reading more on the documentation for bs4 if you're interested in strengthening your scraping skills.

While you do have this new power at your fingertips though, let me stress the importance of its ethical usage. To put it bluntly, the line defining the legality of 'web-scraping' is currently quite vague; there have been hearings related to the activity and some have yielded good results, while others...not so much. The concept of retrieving data from a webpage is entirely legal of course, but just be sure to keep your programs away from proprietary or sensitive information. Furthermore, I would like to remind you that I take no responsibility for your actions...

Prev: Chapter 3 - Cryptography | Next: Chapter 5 - GUIs

Chapter 5: GUIs

Prev: Chapter 4 - Web Scraping | Next: Chapter 6 - APIs

Summary:

It's time to get GUI (pronounced 'goo-ee'), and I'm not talking about cookie dough. This chapter is all about getting started with simple Graphical User Interfaces (GUIs) in Python, using the built-in *tkinter* package.

If you've programmed for more than a few hours in your life, you've probably asked yourself, "What's up with this lame terminal window? When can I start making 'normal' applications--with windows and buttons!?" Well, now is the time to start, and--like most things in Python--it's not daunting at all!

In this chapter, you'll get to study some basic concepts with GUI programming, learn how to use a handful of the basic widgets supplied by tkinter, and even learn how to make a basic, but functional text editor!

Content:

Wrap Up:

Prev: Chapter 4 - Web Scraping | Next: Chapter 6 - APIs

Chapter 6: APIs

Prev: Chapter 5 - GUIs | Next: Chapter 7 - FTP Client

Summary:

I often find myself using a service or web browser application, and I think, "Wow, wouldn't it be awesome if I could somehow use this service for my own project..." What's this, an...'API' you speak of?

That's right, this chapter is about introducing you to the brilliant topic of APIs. API stands for Application Program Interface, and is the essential messenger for how we communicate with certain services and applications. From a more general definition, you'll be glad to know that a class in Object-Oriented Programming (OOP) is technically an API. In Layman's terms, an API is where the code has already been written for you, and all you have to do is read up on how to use it, and then call the right functions.

Say you wanted to build a program that plays music from Pandora and then saves certain song information on the local machine. Or perhaps you could write an application that retrieves statistics for League of Legends tournament games. The first thing you should check with these project ideas is whether or not the general services being used already have an API or not--and usually they do. This falls back on the old saying:

> Why reinvent the wheel?

And yes, the previously mentioned examples are both very possible thanks to APIs. Thus, this chapter will cover a few simple APIs, how to authenticate in order to use them, and even showcase a full example using a Facebook API. So without further ado...

Content:

Wrap Up:

Prev: Chapter 5 - GUIs | Next: Chapter 7 - FTP Client

Chapter 7: FTP Client

Prev: Chapter 6 - APIs | Next: Chapter 8 - Distribution

Summary:

Content:

Wrap Up:

Prev: Chapter 6 - APIs | Next: Chapter 8 - Distribution

Chapter 8: Distribution

Prev: Chapter 7 - FTP Client | Next: Chapter 9 - Regular Expressions

Summary:

Content:

Wrap Up:

Prev: Chapter 7 - FTP Client | Next: Chapter 9 - Regular Expressions

Chapter 9: Regular Expressions

Prev: Chapter 8 - Distribution | Next: Chapter 10 - Multithreading

Summary:

Content:

Wrap Up:

Prev: Chapter 8 - Distribution | Next: Chapter 10 - Multithreading

Chapter 10: Multithreading

Prev: Chapter 9 - Regular Expressions

Summary:

Content:

Wrap Up:

Prev: Chapter 9 - Regular Expressions | Next: About the Author

About the Author

Prev: Chapter 10 - Multithreading