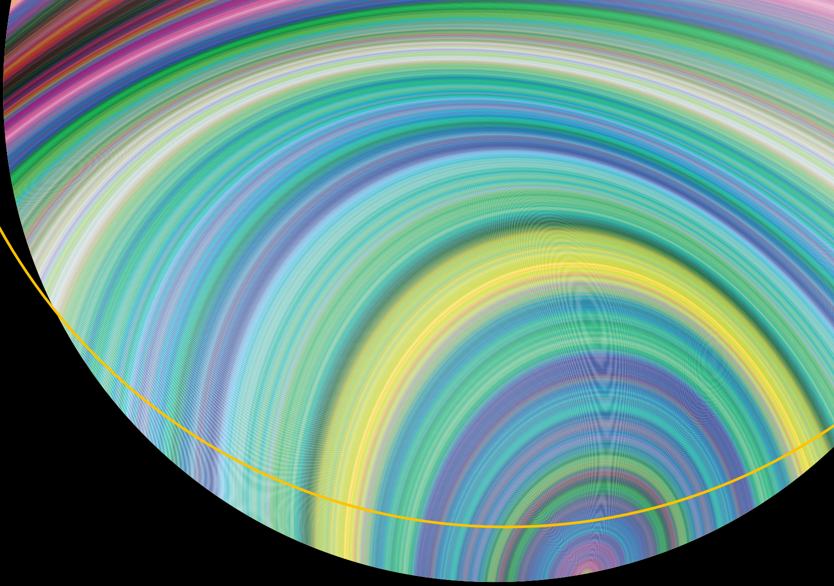


Expert Twisted

Event-Driven and Asynchronous Programming with Python

—

Mark Williams
Cory Benfield
Brian Warner
Moshe Zadka
Dustin Mitchell
Kevin Samuel
Pierre Tardy

Apress®

Expert Twisted

Event-Driven and Asynchronous Programming with Python

Mark Williams
Cory Benfield
Brian Warner
Moshe Zadka
Dustin Mitchell
Kevin Samuel
Pierre Tardy

Apress®

Expert Twisted

Mark Williams
Pasadena, CA, USA

Cory Benfield
London, UK

Brian Warner
New York, USA

Moshe Zadka
New York, USA

Dustin Mitchell
New York, USA

Kevin Samuel
Nice, France

Pierre Tardy
Toulouse, France

ISBN-13 (pbk): 978-1-4842-3741-0
ISBN-13 (electronic): 978-1-4842-3742-7
https://doi.org/10.1007/978-1-4842-3742-7

Library of Congress Control Number: 2018965166

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

*Dedicated to AZ, NZ, and TS: Twisted prevails,
and we're looking forward to the next
generation of maintainers.*

—Moshe Zadka

Table of Contents

About the Authors ... xiii

About the Technical Reviewers ... xv

Acknowledgments ... xvii

Introduction ... xix

Part 1: Foundations ... 1

Chapter 1: An Introduction to Event-Driven Programming with Twisted ... 3

- A Note About Python Versions ... 4
- What Is Event-Driven Programming? ... 4
- Multiple Events ... 5
- Application(tkinter.Tk()).mainloop() ... 6
- Multiplexing and Demultiplexing ... 7
- The select Multiplexer ... 9
  - Its History, Its Siblings, and Its Purpose ... 9
  - select and Sockets ... 9
  - The How and Why of Socket Events ... 11
  - Handling Events ... 12
  - An Event Loop with select ... 13
  - Event-Driven Clients and Servers ... 15
- Non-blocking I/O ... 18
  - Knowing When to Stop ... 18
  - Tracking State ... 19
  - State Makes Programs Complex ... 23
- Managing Complexity with Transports and Protocols ... 23
  - Reactors: Working with Transports ... 24

Transports: Working with Protocols 25
Playing Ping-Pong with Protocols and Transports 26
Clients and Servers with Protocols and Transports 31
Twisted and Reactors, Protocols, and Transports 33
The Value of Event-Driven Programming 33
Twisted and the Real World 36
Events in Time 41
Repeated Events with LoopingCall 44
Event Interfaces with zope.interface 46
Flow Control in Event-Driven Programs 49
Flow Control in Twisted with Producers and Consumers 50
Push Producers 51
Consumers 54
Pull Producers 57
Summary 57

Chapter 2: An Introduction to Asynchronous Programming with Twisted 59

Event Handlers and Composition 59
What Is Asynchronous Programming? 63
Placeholders for Future Values 63
Asynchronous Exception Handling 66
An Introduction to Twisted's Deferred 71
Callbacks 71
Errbacks and Failures 73
Composing Deferreds 76
Generators and InlineCallbacks 80
yield 80
send 81
throw 84
Asynchronous Programming with inlineCallbacks 85

Coroutines in Python ... 88
Coroutines with yield from ... 88
Coroutines async and await ... 90
Awaiting Deferreds ... 95
Coroutines to Deferreds with ensureDeferred ... 97
Multiplexing Deferreds ... 99
Testing Deferreds ... 102
Summary ... 106

Chapter 3: Applications with treq and Klein ... 109

Why Libraries? ... 109
Feed Aggregation ... 110
Introducing treq ... 111
Introducing Klein ... 115
Klein and Deferreds ... 117
Klein Templates with Plating ... 118
A First Draft of Feed Aggregation ... 121
Test-Driven Development with Klein and treq ... 128
Running Test on an Installable Project ... 128
Testing Klein with StubTreq ... 131
Testing treq with Klein ... 140
Logging with twisted.logger ... 143
Running Twisted Applications with twist ... 149
Summary ... 154

Part 2: Projects ... 155

Chapter 4: Twisted in Docker ... 157

Intro to Docker ... 157
Containers ... 157
Container Images ... 158
Runc and Containerd ... 159

Client ... 159
Registry ... 160
Build ... 160
Multi-stage Build ... 161
Python on Docker ... 163
Deployment Options ... 163
Full env ... 163
Virtualenv ... 169
Pex ... 170
Build Options ... 172
One Big Bag ... 172
Copying Wheels Between Stages ... 172
Copying Environment Between Stages ... 173
Copying the Pex Executable Between Stages ... 173
Automation with Dockerpy ... 173
Twisted on Docker ... 174
ENTRYPOINT and PID 1 ... 174
Custom Plugins ... 174
NColony ... 175
Summary ... 178

Chapter 5: Using Twisted as a WSGI Server ... 179

Introduction to WSGI ... 179
PEP ... 180
Raw Example ... 181
Reference Implementation ... 183
WebOb Example ... 185
Pyramid Example ... 186
Getting Started ... 187
WSGI Server ... 188
Finding Code ... 191

Default Path 191
PYTHONPATH 192
setup.py 192
Why Twisted 192
Production vs. Development 192
TLS 194
Server Name Indication 195
Static Files 197
Resource Model 197
Pure Static 198
Combining Static Files with WSGI 200
Built-In Scheduled Tasks 203
Control Channels 206
Strategies for Using Multiple Cores 208
Load Balancer 208
Opening Socket in Shared Mode 210
Other Options 213
Dynamic Configuration 214
A/B Testable Pyramid App 214
Custom Plugin with AMP 216
Control Program 219
Summary 221

Chapter 6: Tahoe-LAFS: The Least-Authority File System 223

How Tahoe-LAFS Works 224
System Architecture 227
How It Uses Twisted 229
Problems We've Run Into 230
Daemonization Tools 231
Internal FileNode Interfaces 232
Front-End Protocol Integration 233

The Web Front End ... 234
File Types, Content-Type, /name/ ... 237
Saving to Disk ... 238
Range Headers ... 238
Error Conversion on the Return Side ... 240
Rendering UI Elements: Nevow Templates ... 241
The FTP Front End ... 242
The SFTP Front End ... 248
Backward-Incompatible Twisted APIs ... 248
Summary ... 251
References ... 251
Chapter 7: Magic Wormhole ... 253
What It Looks Like ... 254
How It Works ... 255
Network Protocols, Transfer Latency, Client Compatibility ... 257
Network Protocols and Client Compatibility ... 258
Server Architecture ... 260
Persistent Database ... 262
Transit Client: Cancelable Deferreds ... 262
Transit Relay Server ... 265
Wormhole Client Architecture ... 267
Deferreds vs State Machines, One-Shot Observer ... 268
One-Shot Observers ... 271
Promises/Futures vs. Deferreds ... 272
Eventual-Send, Synchronous Testing ... 275
Asynchronous Testing with Deferreds ... 277
Synchronous Testing with Deferreds ... 278
Synchronous Testing and Eventual Send ... 281
Summary ... 283
References ... 283

Chapter 8: Push Data to Browsers and Micro-services with WebSocket 285
Why WebSocket? 285
WebSocket and Twisted 286
Raw WebSocket, from Python to Python 288
Raw WebSocket, Between Python and JavaScript 292
More Powerful WebSocket with WAMP 294
Summary 303

Chapter 9: Applications with asyncio and Twisted 305
Core Concepts 305
Promises 306
    Event Loops 307
Guidelines 308
Case Study: A Proxy with aiohttp and treq 312
Summary 316

Chapter 10: Buildbot and Twisted 317
History of Buildbot 317
    The Evolution of Buildbot's Async Python 318
    Migrating Synchronous APIs 321
    Async Build Steps 322
Buildbot's Code 323
    Async Utilities 323
    Debounce 323
    Async Services 324
    LRU Cache 326
    Eventual 327
    Interfacing with Synchronous Code 327
    SQLAlchemy 328
    requests 329
    Docker 332
    Concurrent Access to Shared Resources 333

Yield as a Concurrency Barrier 333
Thread-Pool Functions Should Not Mutate State 334
DeferredLocks 336
Testing 336
Fakes 338
Summary 338

Chapter 11: Twisted and HTTP/2 339

Introduction 339
Design Goals 341
Seamless Integration 341
Most-Optimized Behavior by Default 343
Separating Concerns and Reusing Code 343
Implementation Concerns 344
What Is a Connection Anyway? The Value of Standard Interfaces 345
Multiplexing and Priority 348
Backpressure 355
Backpressure in Twisted 356
Backpressure in HTTP/2 359
Current Status and Future Expansion 362
Summary 363

Chapter 12: Twisted and Django Channels 365

Introduction 365
Channels Building Blocks 367
Message Brokers and Queues 368
Distributed Multi-Layer Systems in Twisted 369
Current Status and Future Expansion 371
Summary 371

Correction to: Expert Twisted: Event-Driven and Asynchronous Programming with Python C1

Index 373

About the Authors

Mark Williams works on Twisted. At eBay and PayPal, he worked on high-performance Python web services (over a billion requests a day!), application and information security, and porting enterprise, Java-only libraries to Python.

Cory Benfield is an open source Python developer heavily involved in the Python HTTP community. He's a Requests core contributor, a urllib3 core contributor, and the lead maintainer of the Hyper Project, a collection of HTTP and HTTP/2 tools for Python. For his sins, he also helps out with the Python Cryptographic Authority on PyOpenSSL.

Brian Warner is a security engineer and software developer, having worked at Mozilla on Firefox Sync, the Add-On SDK, and Persona. He is co-founder of the Tahoe-LAFS distributed secure filesystem, and develops secure storage and communication tools.

Moshe Zadka has been part of the open source community since 1995, made his first core Python contributions in 1998, and is a founding member of the Twisted open source project. He also loves to teach Twisted and Python, having given tutorials at several conferences as well as regularly blogging.

Dustin Mitchell has contributed to Buildbot and is a member of the TaskCluster team at Mozilla, having also worked on the Release Engineering, Release Operations, and Infrastructure teams.

Kevin Samuel has been a Dev and trainer since Python 2.4 and has been putting his skills to work in East Europe, North America, Asia, and West Africa. He has been working closely with the Crossbar.io team and is an active member of the French Python community.

Pierre Tardy is a continuous integration specialist with Renault Software Labs, and he is currently the lead committer for Buildbot.

About the Technical Reviewers

Julian Berman is a New York-based software developer and open source contributor. He is the author of the jsonschema Python library, an occasional contributor to the Twisted ecosystem, and an active member of the Python community.

Shawn Shojaie lives in the clement chaparral of California's Bay Area, where he works as a back-end software engineer. He has worked at Intel, NetApp, and now SimpleLegal, where he happily builds web-based applications for legal services. He spends weekdays writing Django and tuning PostgreSQL, and his weekends contributing to open source projects like django-pylint, occasionally editing technical essays. Find out more at him at shawnshojaie.com.

Tom Most is a software engineer in the telecommunications industry. He is a Twisted committer with 10 years of experience of applying Twisted to web services, client libraries, and command-line applications. He is the maintainer of Afkak, the Twisted Kafka client. He can be found online at freecog.net and reached at twm@freecog.net.

Acknowledgments

Thanks to my wife, Jennifer Zadka, without whose support I could not have done it.

Thanks to my parents, Yaacov and Pnina Zadka, who taught me how to learn.

Thanks to my advisor, Yael Karshon, for teaching me how to write.

Thanks to Mahmoud Hashemi, for inspiration and encouragement.

Thanks to Mark Williams, for always being there for me.

Thanks to Glyph Lefkowitz, for teaching me things about Python, about programming, and about being a good person.

—Moshe Zadka

Thanks to Mahmoud Hashemi and David Karapetyan for their feedback. Thanks to Annie for putting up with me while I wrote

—Mark Williams

The original version of this book was revised. A correction to this book is available at https://doi.org/10.1007/978-1-4842-3742-7_13

Introduction

Twisted has recently celebrated its sweet sixteen birthday. It has been around for a while; and in that time, it grew to be a powerful library. In that time, some interesting applications have been built on top of it. In that time, many of us learned a lot about how to use Twisted well, how to think about networking code, and how to architect event-based programs.

After going through the introductory materials that we have on the Twisted site, a common thing to hear is "What now? How can I learn more about Twisted?" The usual way we answered that question is with a question: "What do you want to do with Twisted?" This book shows how to do interesting things with Twisted.

Each of the contributors to this book has done slightly different things with Twisted and learned different lessons. We are excited to present all of these lessons, with the goals of making them common knowledge in the community.

Enjoy!

PART 1

Foundations

CHAPTER 1

An Introduction to Event-Driven Programming with Twisted

Twisted is a powerful, well-tested, and mature concurrent networking library and framework. As we'll see in this book, many projects and individuals have used it to great effect for more than a decade.

At the same time, Twisted is large, complicated, and old. Its lexicon teems with strange names, like "reactor," "protocol," "endpoint," and "Deferred." These describe a philosophy and architecture that have baffled both newcomers and old hands with years of Python experience.

Two fundamental programming paradigms inform Twisted's pantheon of APIs: *event-driven programming* and *asynchronous programming*. The rise of JavaScript and the introduction of `asyncio` into the Python standard library have brought both further into the mainstream, but neither paradigm dominates Python programming so completely that merely knowing the language makes them familiar. They remain specialized topics reserved for intermediate or advanced programmers.

This chapter and the next introduce the motivations behind event-driven and asynchronous programming, and then show how Twisted employs these paradigms. They lay the foundation for later chapters that explore real-world Twisted programs.

We'll begin by exploring the nature of event-driven programming outside of the context of Twisted. Once we have a sense of what defines event-driven programming, we'll see how Twisted provides software abstractions that help developers write clear and effective event-driven programs. We'll also stop along the way to learn about some of the unique parts of those abstractions, like *interfaces*, and explore how they're documented on Twisted's website.

M. Williams et al., *Expert Twisted*, https://doi.org/10.1007/978-1-4842-3742-7_1

By the end of this chapter you'll know Twisted terminology: protocols, transports, reactors, consumers, and producers. These concepts form the foundation of Twisted's approach to event-driven programming, and knowing them is essential to writing useful software with Twisted.

A Note About Python Versions

Twisted itself supports Python 2 and 3, so all code examples in this chapter are written to work on both Python 2 and 3. Python 3 is the future, but part of Twisted's strength is its rich history of protocol implementations; for that reason, it's important that you're comfortable with code that runs on Python 2, even if you never write it.

What Is Event-Driven Programming?

An *event* is something that causes an event-driven program to perform an action. This broad definition allows many programs to be understood as event-driven; consider, for example, a simple program that prints either `Hello` or `World!` depending on user input:

```
import sys
line = sys.stdin.readline().strip()
if line == "h":
    print("Hello")
else:
    print("World")
```

The availability of a line of input over standard input is an event. Our program pauses on `sys.stdin.readline()`, which asks the operating system to allow the user to input a complete line. Until one is received, our program can make no progress. When the operating system receives input, and Python's internals determine it's a line, `sys.stdin.readline()` resumes our program by returning that data to it. This resumption is the event that drives our program forward. Even this simple program, then, can be understood as an *event-driven* one.

Multiple Events

A program that receives a single event and then exits doesn't benefit from an event-driven approach. Programs in which more than one thing can happen at a time, however, are more naturally organized around events. A graphical user interface implies just such a program: at any moment, a user might click a button, select an item from a menu, scroll through a text widget, and so on.

Here's a version of our previous program with a Tkinter GUI:

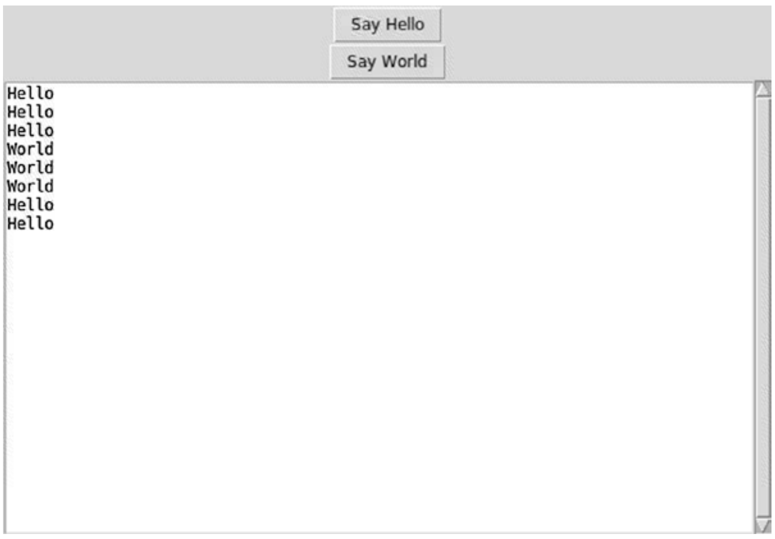
```
from six.moves import tkinter
from six.moves.tkinter import scrolledtext

class Application(tkinter.Frame):
    def __init__ (self, root):
        super(Application,self). __init__ (root)
        self.pack()
        self.helloButton = tkinter.Button(self,
                                      text="Say Hello",
                                      command=self.sayHello)
        self.worldButton = tkinter.Button(self,
                                        text="Say World",
                                        command=self.sayWorld)
         self.output = scrolledtext.ScrolledText(master=self)
         self.helloButton.pack(side="top")
        self.worldButton.pack(side="top")
         self.output.pack(side="top")
    def outputLine(self, text):
        self.output.insert(tkinter.INSERT, text+ '\n')
    def sayHello(self):
        self.outputLine("Hello")
    def sayWorld(self):
        self.outputLine("World")
```

```
Application(tkinter.Tk()).mainloop()
```

This version of our program presents the user with two buttons, either of which can generate an independent click event. This differs from our previous program, where only `sys.stdin.readline` could generate the single "line ready" event.

We cope with the possible occurrence of either button's event by associating *event handlers* with each one. Tkinter buttons accept a callable `command` to invoke when they are clicked. When the button labeled "Say Hello" generates a click event, that event drives our program to call `Application.sayHello` as shown in Figure 1-1. This, in turn, outputs a line consisting of `Hello` to a scrollable text widget. The same process applies to the button labeled "Say Hello" and `Application.sayWorld`.



```
Hello
Hello
Hello
World
World
World
Hello
Hello
```

Figure 1-1. *Our Tkinter GUI application after a series of clicks of "Say Hello" and "Say World"*

`tkinter.Frame`'s `mainloop` method, which our `Application` class inherits, waits until a button bound to it generates an event and then runs the associated event handler. After each event handler has run, `tkinter.Frame.mainloop` again begins waiting for new events. A loop that monitors event sources and dispatches their associated handlers is typical of event-driven programs, and is known as an *event loop*.

These concepts are the core of event-driven programming:

1. *Events* represent that something has occurred and to which the program should react. In both our examples, events correspond naturally to program input, but as we'll see, they can represent anything that causes our program to perform some action.
2. *Event handlers* constitute the program's reactions to events. Sometimes an event's handler just consists of a sequence of code, as in our `sys.stdin.readline` example, but more often it's encapsulated by a function or method, as in our `tkinter` example.
3. An *event loop* waits for events and invokes the event handler associated with each. Not all event-driven programs have an event loop; our `sys.stdin.readline` example did not because it only responds to a single event. However, most resemble our `tkinter` example in that they process many events before finally exiting. These kinds of programs use an event loop.

Multiplexing and Demultiplexing

The way event loops wait for events affects the way we write event-driven programs, so we must take a closer look at them. Consider our `tkinter` example and its two buttons; the event loop inside `mainloop` must wait until the user has clicked at least one button. A naive implementation might look like this:

```
def mainloop(self):
    while self.running:
        ready = [button for button in self.buttons if button.hasEvent()]
        if ready:
           self.dispatchButtonEventHandlers(ready)
```

`mainloop` continually *polls* each button for a new event, dispatching event handlers only for those that have an event ready. When no events are ready, the program makes no progress because no action has been taken that requires a response. An event-driven program must suspend its execution during these periods of inactivity.

The while loop in our `mainloop` example suspends its program until one of the buttons has been clicked and `sayHello` or `sayWorld` should run. Unless the user is supernaturally fast with a mouse, this loop spends most of its time checking buttons that haven't been clicked. This is known as a *busy wait* because the program is actively busy waiting.

A busy wait like this pauses a program's overall execution until one of its event sources reports an event, and so it suffices as a mechanism to pause an event loop.

The inner list comprehension that powers our implementation's busy wait asks a critical question: Has anything happened? The answer comes from the `ready` variable, which contains all buttons that have been clicked in a single place. The truthiness of `ready` decides the answer to the event loop's question: when `ready` is empty and thus falsey, no buttons have been clicked and so nothing has happened. When it's truthy, however, at least one has been clicked, and so something *has* happened.

The list comprehension that constructs `ready` coalesces many separate inputs into one. This is known as *multiplexing*, while the inverse process of separating different inputs out from a single coalesced input is known as *demultiplexing*. The list comprehension multiplexes our buttons into `ready` while the `dispatchButtonEventHandlers` method demultiplexes them out by invoking each event's handler.

We can now refine our understanding of event loops by precisely describing how they wait for events:

- An *event loop* waits for events by *multiplexing* their sources into a single input. When that input indicates that events have occurred, the event loop demultiplexes it into its constituent inputs and invokes the event handler associated with each.

Our `mainloop` multiplexer wastes most of its time polling buttons that haven't been clicked. Not all multiplexers are so inefficient. `tkinter.Frame.mainloop`'s actual implementation employs a similar multiplexer that polls all widgets unless the operating system provides more efficient primitives. To improve its efficiency, `mainloop`'s multiplexer exploits the insight that computers can check a GUI's widgets faster than a person can interact with them, and inserts a `sleep` call that pauses the entire program for several milliseconds. This allows the program to spend part of its busy-wait loop passively rather than actively do nothing, saving CPU time and energy at the expense of negligible latency.

While Twisted can integrate with graphical user interfaces, and in fact has special support for `tkinter`, it is at its heart a networking engine. *Sockets*, not buttons, are the fundamental object in networking, and operating systems expose efficient primitives for multiplexing socket events. Twisted's event loop uses these primitives to wait for events. To understand Twisted's approach to event-driven programming, we must understand the interaction between these sockets and these multiplexing networking primitives.

The `select` Multiplexer

Its History, Its Siblings, and Its Purpose

Almost all modern operating systems support the `select` multiplexer. `select` gets its name from its ability to take a list of sockets and "select" only those that have events ready to be handled.

`select` was born in 1983, when computers were capable of far less. Consequently, its interface prevents it from operating at maximum efficiency, especially when multiplexing a large number of sockets. Each operating system family provides its own, more efficient multiplexer, such as BSD's `kqueue` and Linux's `epoll`, but no two interoperate. Luckily their principles are similar enough to `select` that we can generalize their behavior from `select`'s. We'll use `select` to explore how these socket multiplexers behave.

`select` and Sockets

The code that follows omits error handling and will break on many edge cases that occur in practice. **It is intended only as a teaching tool. Do not use it in real applications. Use Twisted instead.** Twisted strives to correctly handle errors and edge cases; that's part of why its implementation is so complicated.

With that disclaimer out of the way, let's begin an interactive Python session and create sockets for `select` to multiplex:

```
>>> import socket
>>> listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> listener.bind(('127.0.0.1', 0))
>>> listener.listen(1)
>>> client = socket.create_connection(listener.getsockname())
>>> server, _ = listener.accept()
```

A full explanation of the socket API is beyond the scope of this book. Indeed, we expect that the parts we discuss will lead you to prefer Twisted! The preceding code, however, contains more fundamental concepts than irrelevant details:

1. `listener` - This socket can *accept* incoming connections. It is an internet (`socket.AF_INET`) and TCP (`socket.SOCK_STREAM`) socket accessible by clients on the internal, local-only network interface (which conventionally has an address of `127.0.0.1`) and on a port randomly assigned by the operating system (`0`). This listener can perform the setup necessary for one incoming connection and enqueue it until we're reading for it (`listen(1)`).

2. `client` - This socket is an outgoing connection. Python's `socket.create_connection` function accepts a `(host, port)` tuple representing the listening socket to which to connect and returns a socket connected to it. Because our listening socket is in the same process and named `listener`, we can retrieve its host and port with the `listener.getsockname()`.

3. `server` - The server's incoming connection. Once `client` has connected to our host and port, we must accept the connection from `listener`'s queue of length 1. `listener.accept` returns a `(socket, address)` tuple; we only need the socket, so we discard the address. A real program might log the address or use it to track connection metrics. The listening queue, which we set to 1 via the socket's `listen` method, holds this socket for us before we call `accept` and allows `create_connection` to return.

`client` and `server` are two ends of the same TCP connection. An established TCP connection has no concept of "client" and "server"; our `client` socket has the same privileges to read, write, or close the connection as our `server`:

```
>>> data = b"xyz"
>>> client.sendall(data)
>>> server.recv(1024) == data
True
>>> server.sendall(data)
>>> client.recv(1024) == data
True
```

The How and Why of Socket Events

Under the hood, the operating system maintains read and write buffers for each TCP socket to account for network unreliability and clients and servers that read and write at different speeds. If `server` became temporarily unable to receive data, the `b"xyz"` we passed `client.sendall` would remain in its write buffer until `server` again became active. Similarly, if we were too busy to call `client.recv` to receive the `b"xyz" server.sendall` sent, `client`'s read buffer would hold onto it until we got around to receiving it. The number that we pass `recv` represents the maximum data we're willing to remove from the read buffer. If the read buffer has less than the maximum, as it does in our example, `recv` will remove *all* the data from the buffer and return it.

Our sockets' bidirectionality implies two possible events:

1. A *readable event*, which means the socket has something available for us. A connected server socket generates this event when data has landed in the socket's receive buffer, so that calling `recv` after a readable event will immediately return that data. A disconnection is represented by `recv`ing no data. By convention, a listening socket generates this event when we can `accept` a new connection.

2. A *writable event*, which means space is available in the socket's write buffer. This is a subtle point: as long as the socket receives acknowledgment from the server for the data it's transmistted across the network faster than we add it to the send buffer, it remains writable.

`select`'s interface reflects these possible events. It accepts up to four arguments:

1. a sequence of sockets to monitor for *readable events*;

2. a sequence of sockets to monitor for *writable events*;

3. a sequence of sockets to monitor for "exceptional events." In our examples, no exceptional events will occur, so we will always pass an empty list here;

4. An optional *timeout*. This is the number of seconds `select` will wait for one of the monitor sockets to generate an event. Omitting this argument will cause `select` to wait forever.

We can ask `select` about the events our sockets have just generated:

```
>>> import select
>>> maybeReadable = [listener, client, server]
>>> maybeWritable = [client, server]
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable, [], 0)
>>> readable
[]
>>> writable == maybeWritable and writable == [client, server]
True
```

We instruct `select` not to wait for any new events by providing a timeout of 0. As explained above, our `client` and `server` sockets might be readable or writable, while our `listener`, which can only accept incoming connections, can only be readable.

If we had omitted the timeout, `select` would pause our program until one of the sockets it monitored became readable or writable. This suspension of execution is analogous to the multiplexing busy-wait that polled all buttons in our naive `mainloop` implementation above.

Invoking `select` *multiplexes* sockets more efficiently than a busy wait because the operating system will only resume our program when at least one event has been generated; inside the kernel an event loop, not unlike our `select`, waits for events from the network hardware and dispatches them to our application.

Handling Events

`select` returns a tuple with three lists, in the same order as its arguments. Iterating over each returned list *demultiplexes* `select`'s return value. None of our sockets have generated readable events, even though we've written data to both `client` and `server`; our preceding calls to `recv` emptied their read buffers, and no new connections have arrived for `listener` since we accepted `server`. Both `client` and `server` have generated a writable event, however, because there's space available in their send buffers.

Sending data from `client` to `server` causes `server` to generate a readable event, so `select` places it in the `readables` list:

```
>>> client.sendall(b'xyz')
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable, [], 0)
>>> readable == [server]
True
```

The `writable` list, interestingly, once again contains our `client` and `server` sockets:

```
>>> writable == maybeWritable and writable == [client, server]
True
```

If we called `select` again, our `server` socket would again be in `readable` and our `client` and `server` sockets again in `writable`. The reason is simple: as long as data remains in a socket's read buffer, it will continuously generate a readable event, and as long as space remains in a socket's write buffer, it will generate a writable event. We can confirm this by `recv`ing the data `client` sent to `server` and calling `select` again for new events:

```
>>> server.recv(1024) == b'xyz'
True
>>> readable, writable, _ = select.select(maybeReadable, maybeWritable,
[], 0)
>>> readable
[]
>>> writable == maybeWritable and writable == [client, server]
True
```

Emptying `server`'s read buffer has caused it to stop generating readable events, and `client` and `server` continue to generate writable events because there's still space in their write buffers.

An Event Loop with `select`

We now know how `select` multiplexes sockets:

1. Different sockets generate readable or writable events to indicate that an event-driven program should accept incoming data or connections, or write outgoing data.
2. `select` multiplexes sockets by monitoring them for readable or writable events, pausing the program until at least one is generated or the optional timeout has elapsed.
3. Sockets continue generating readable and writable events until the circumstances that led to those events changes: a socket with readable data emits readable events until its read buffer

is emptied; a listening socket emits readable events until all incoming connections have been accepted; and a writable socket emits writable events until its write buffer is filled.

With this knowledge, we can sketch an event loop around select:

```
import select

class Reactor(object):
    def __init__ (self):
        self._readers = {}
        self._writers = {}
    def addReader(self, readable, handler):
        self._readers[readable] = handler
    def addWriter(self, writable, handler):
        self._writers[writable] = handler
    def removeReader(self, readable):
        self._readers.pop(readable,None)
    def removeWriter(self, writable):
        self._writers.pop(writable,None)
    def run(self):
        while self._readers or self._writers:
            r, w, _ = select.select(list(self._readers), list
            (self._writers), [])
            for readable in r:
                self._readers[readable](self, readable)
            for writable in w:
                if writable in self._writers:
                   self._writers[writable](self, writable)
```

We call our event loop a *reactor* because it reacts to socket events. We can request our Reactor call readable event handlers on sockets with addReader and writable event handlers with addWriter. Event handlers accept two arguments: the reactor itself and the socket that generated the event.

The loop inside the run method multiplexes our sockets with select, then demultiplexes the result between sockets that have generated a read event and sockets that have generated a write event. The event handlers for each readable socket run first. Then, the event loop checks that each writable socket is still registered as a writer before running its event handler. This check is necessary because closed connections are represented as read events, so a read handler run immediately prior might remove a closed socket from the readers and writers. By the time its writable event handler runs, the closed socket would be removed from the `_writers` dictionary.

Event-Driven Clients and Servers

This simple event loop suffices for implementing a client that continually writes data to a server. We'll begin with the event handlers:

```
def accept(reactor, listener):
    server, _ = listener.accept()
    reactor.addReader(server, read)

def read(reactor, sock):
    data = sock.recv(1024)
    if data:
        print("Server received", len(data),"bytes.")
    else:
        sock.close()
        print("Server closed.")
        reactor.removeReader(sock)

DATA=[b"*",  b"*"]
def write(reactor, sock):
    sock.sendall(b"".join(DATA))
    print("Client wrote", len(DATA)," bytes.")
    DATA.extend(DATA)
```

The `accept` function handles a readable event on a listening socket by accepting the incoming connection and requesting the reactor monitor it for readable events. These are handled by the `read` function.

The `read` function handles a readable event on a socket by attempting to receive a fixed amount of data from the socket's receive buffer. The length of any received data is printed – remember, the amount passed to `recv` represents an *upper bound* on the number of bytes returned. If no data is received on a socket that has generated a readable event, then the other side of the connection has closed its socket, and the `read`