THE EXPERT'S VOICE® IN OPEN SOURCE

THIRD EDITION

Foundations of Python Network Programming

THE COMPREHENSIVE GUIDE TO NETWORK PROGRAMMING AND MANAGEMENT WITH PYTHON 3

Brandon Rhodes and John Goerzen

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

It is an exciting moment for the Python community. After two decades of careful innovation that saw the language gain features such as context managers, generators, and comprehensions in a careful balance with its focus on remaining simple in both its syntax and its concepts, Python is finally taking off.

Instead of being seen as a boutique language that can be risked only by top-notch programming shops such as Google and NASA, Python is now experiencing rapid adoption, both in traditional programming roles, such as web application design, and in the vast world of "reluctant programmers," such as scientists, data specialists, and engineers—people who learn to program not for its own sake but because they must write programs if they are to make progress in their field. The benefits that a simple programming language offers for the occasional or nonexpert programmer cannot, I think, be overstated.

Python 3

After its debut in 2008, Python 3 went through a couple of years of reworking and streamlining before it was ready to step into the role of its predecessor. But as it now enters its second half-decade, it has emerged as the preferred platform for innovation in the Python community.

Whether one looks at fundamental improvements, like the fact that true Unicode text is now the default string type in Python 3, or at individual improvements, like correct support for SSL, a built-in `asyncio` framework for asynchronous programming, and tweaks to Standard Library modules large and small, the platform that Python 3 offers the network programmer is in nearly every way improved. This is a significant achievement. Python 2 was already one of the best languages for making programmers quickly and effectively productive on the modern Internet.

This book is not a comprehensive guide to switching from Python 2 to Python 3. It will not tell you how to add parentheses to your old `print` statements, rename Standard Library module imports to their new names, or debug deeply flawed network code that relied on Python 2's dangerous automatic conversion between byte strings and Unicode strings—conversions that were always based on rough guesswork. There are already excellent resources to help you with that transition or even to help you write libraries carefully enough so that their code will work under both Python 2 and Python 3, in case you need to support both audiences.

Instead, this book focuses on network programming, using Python 3 for every example script and snippet of code at the Python prompt. These examples are intended to build a comprehensive picture of how network clients, network servers, and network tools can best be constructed from the tools provided by the language. Readers can study the transition from Python 2 to Python 3 by comparing the scripts used in each chapter of the second edition of this book with the listings here in the third edition—both of which are available at `https://github.com/brandon-rhodes/fopnp/tree/m/` thanks to the excellent Apress policy of making source code available online. The goal in each of the following chapters is simply to show you how Python 3 can best be used to solve modern network programming problems.

By focusing squarely on how to accomplish things the right way with Python 3, this book hopes to prepare both the programmer who is getting ready to write a new application from the ground up and the programmer preparing to transition an old code base to the new conventions. Both programmers should come away knowing what correct networking code looks like in Python 3 and therefore knowing the look and flavor of the kind of code that ought to be their goal.

Improvements in This Edition

There are several improvements by which this book attempts to update the previous edition, beyond the move to Python 3 as its target language and the many updates to both Standard Library and third-party Python modules that have occurred in the past half-decade.

- Every Python program listing is now written as a module. That is, each one performs its imports and defines its functions or classes but then carefully guards any import-time actions inside an `if` statement that fires only if the module `__name__` has the special string value `'__main__'` indicating that the module is being run as the main program. This is a Python best practice that was almost entirely neglected in the previous edition of this book and whose absence made it more difficult for the sample listings to be pulled into real codebases and used to solve reader problems. By putting their executable logic at the left margin instead of inside an `if` statement, the older program listings may have saved a line or two of code, but they gave novice Python programmers far less practice in how to lay out real code.
- Instead of making ad hoc use of the raw `sys.argv` list of strings in a bid to interpret the command line, most of the scripts in this book now use the Standard Library `argparse` module to interpret options and arguments. This not only clarifies and documents the semantics that each script expects during invocation but also lets the user of each script use the `-h` or `--help` query option to receive interactive assistance when launching the script from the Windows or Unix command line.
- Program listings now make an effort to perform proper resource control by opening files within a controlling `with` statement that will close the files automatically when it completes. In the previous edition, most listings relied instead on the fact that the C Python runtime from the main Python web site usually assures that files are closed immediately thanks to its aggressive reference counting.
- The listings, for the most part, have transitioned to the modern `format()` method for performing string interpolation and away from the old modulo operator hack `string % tuple` that made sense in the 1990s, when most programmers knew the C language, but that is less readable today for new programmers entering the field—and less powerful since individual Python classes cannot override percent formatting like they can with the new kind.
- The three chapters on HTTP and the World Wide Web (Chapters 9 through 11) have been rewritten from the ground up with an emphasis on better explaining the protocol and on introducing the most modern tools that Python offers the programmer writing for the Web. Explanations of the HTTP protocol now use the Requests library as their go-to API for performing client operations, and Chapter 11 has examples in both Flask and Django.
- The material on SSL/TLS (Chapter 6) has been completely rewritten to match the vast improvement in support that Python 3 delivers for secure applications. While the `ssl` module in Python 2 is a weak half-measure that does not even verify that the server's certificate matches the hostname to which Python is connecting, the same module in Python 3 presents a much more carefully designed and extensive API that provides generous control over its features.

This edition of the book is therefore a better resource for the learning programmer simply in terms of how the listings and examples are constructed, even apart from the improvements that Python 3 has made over previous versions of the language.

The Network Playground

The source code to the program listings in this book is available online so that both current owners of this book and potential readers can study them. There is a directory for each chapter of this edition of the book. You can find the chapter directories here:

https://github.com/brandon-rhodes/fopnp/tree/m/py3

But program listings can go only so far toward supporting the curious student of network programming. There are many features of network programming that are difficult to explore from a single host machine. Thus, the source code repository for the book provides a sample network of 12 machines, each implemented as a Docker container. A setup script is provided that builds the images, launches them, and networks them. You can find the script and the images in the source code repository here:

https://github.com/brandon-rhodes/fopnp/tree/m/playground

You can see the 12 machines and their interconnections in Figure 1. The network is designed to resemble a tiny version of the Internet.

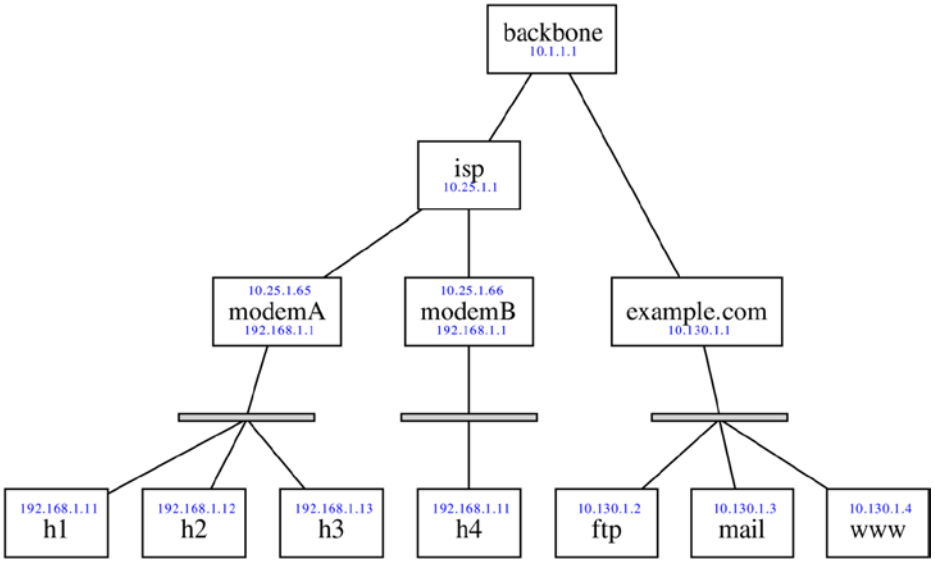

Figure 1. *The network playground's topology*

- Representing the typical situation of a client in a home or coffee shop are the client machines behind `modemA` and `modemB` that not only offer no services to the Internet but that are in fact not visible on the wider Internet at all. They possess merely local IP addresses, which are meaningful only on the subnet that they share with any other hosts in the same home or coffee shop. When they make connections to the outside world, those connections will appear to originate from the IP addresses of the modems themselves.
- Direct connections allow the modems to connect to an `isp` gateway out on the wider Internet, which is represented by a single `backbone` router that forwards packets between the networks to which it is connected.

- `example.com` and its associated machines represent the configuration of a simple service-oriented machine room. Here, no network translation or masquerading is taking place. The three servers behind `example.com` have service ports that are fully exposed to client traffic from the Internet.
- Each of the service machines `ftp`, `mail`, and `www` has correctly configured daemons up and running so that Python scripts from this book can be run on the other machines in the playground to connect successfully to representative examples of each service.
- All of the service machines have correctly installed TLS certificates (see Chapter 6), and the client machines all have the `example.com` signing certificate installed as a trusted certificate. This means Python scripts demanding true TLS authentication will be able to achieve it.

The network playground will continue to be maintained as both Python and Docker continue to evolve. Instructions will be maintained in the repository for how to download and run the network locally on your own machine, and they will be tweaked based on user reports to make sure that a virtual machine, which offers the playground, can be run by readers on Linux, Mac OS X, and Windows machines.

With the ability to connect and run commands within any of the playground machines, you will be able to set up packet tracing at whichever point on the network you want to see traffic passing between clients and servers. The example code demonstrated in its documentation, combined with the examples and instruction in this book, should help you reach a solid and vivid understanding of how networks help clients and servers communicate.

CHAPTER 1

Introduction to Client-Server Networking

This book explores network programming in the Python language. It covers the basic concepts, modules, and third-party libraries that you are likely to use when communicating with remote machines using the most popular Internet communication protocols.

The book lacks the space to teach you how to program in Python if you have never seen the language before or if you have never even written a computer program at all; it presumes that you have already learned something about Python programming from the many excellent tutorials and books on the subject. I hope that the Python examples in the book give you ideas about how to structure and write your own code. But I will be using all sorts of advanced Python features without explanation or apology—though, occasionally, I might point out how I am using a particular technique or construction when I think it is particularly interesting or clever.

On the other hand, this book does *not* start by assuming you know any networking! As long as you have ever used a web browser or sent an e-mail, you should know enough to start reading this book at the beginning and learn about computer networking along the way. I will approach networking from the point of view of an application programmer who is either implementing a network-connected service—such as a web site, an e-mail server, or a networked computer game—or writing a client program that is designed to use such a service.

Note that you will not, however, learn how to set up or configure networks from this book. The disciplines of network design, server room management, and automated provisioning are full topics all on their own, which tend not to overlap with the discipline of computer programming as covered in this particular book. While Python is indeed becoming a big part of the provisioning landscape thanks to projects such as OpenStack, SaltStack, and Ansible, you will want to search for books and documentation that are specifically about provisioning and its many technologies if you want to learn more about them.

The Building Blocks: Stacks and Libraries

As you begin to explore Python network programming, there are two concepts that will appear over and over again.

- The idea of a *protocol stack*, in which simpler network services are used as the foundation on which to build more sophisticated services.
- The fact that you will often be using Python *libraries* of previously written code—whether modules from the built-in standard library that ships with Python or packages from third-party distributions that you download and install—that already know how to speak the network protocol that you want to use.

In many cases, network programming simply involves selecting and using a library that already supports the network operations that you need to perform. The major purposes of this book are to introduce you to several key networking libraries available for Python while also teaching you about the lower-level network services on which those libraries are built. Knowing the lower-level material is useful, both so that you understand how the libraries work and so that you will understand what is happening when something at a lower level goes wrong.

Let's begin with a simple example. Here is a mailing address:

```
207 N. Defiance St
Archbold, OH
```

I am interested in knowing the latitude and longitude of this physical address. It just so happens that Google provides a Geocoding API that can perform such a conversion. What would you have to do to take advantage of this network service from Python?

When looking at a new network service that you want to use, it is always worthwhile to start by finding out whether someone has already implemented the protocol—in this case, the Google Geocoding protocol—which your program will need to speak. Start by scrolling through the Python Standard Library documentation, looking for anything having to do with geocoding.

http://docs.python.org/3/library/

Do you see anything about geocoding? No, neither do I. But it is important for a Python programmer to look through the Standard Library's table of contents pretty frequently, even if you usually do not find what you are looking for, because each read-through will make you more familiar with the services that are included with Python. Doug Hellmann's "Python Module of the Week" blog is another great reference from which you can learn about the capabilities that come with Python thanks to its Standard Library.

Since in this case the Standard Library does not have a package to help, you can turn to the Python Package Index, an excellent resource for finding all sorts of general-purpose Python packages contributed by other programmers and organizations from across the world. You can also, of course, check the web site of the vendor whose service you will be using to see whether it provides a Python library to access it. Or, you can do a general Google search for *Python* plus the name of whatever web service you want to use and see whether any of the first few results link to a package that you might want to try.

In this case, I searched the Python Package Index, which lives at this URL:

https://pypi.python.org/

There I entered *geocoding*, and I immediately found a package that is named `pygeocoder`, which provides a clean interface to Google's geocoding features (though, you will note from its description, it is *not* vendor-provided but was instead written by someone besides Google).

http://pypi.python.org/pypi/pygeocoder/

This is such a common situation—finding a Python package that sounds like it might already do exactly what you want and that you want to try it on your system—that I should pause for a moment and introduce you to the best Python technology for quickly trying a new library: `virtualenv`!

In the old days, installing a Python package was a gruesome and irreversible act that required administrative privileges on your machine and that left your system Python install permanently altered. After several months of heavy Python development, your system Python install could become a wasteland of dozens of packages, all installed by hand, and you could even find that new packages you tried to install would break because they were incompatible with the old packages sitting on your hard drive from a project that ended months ago.

Careful Python programmers do not suffer from this situation any longer. Many of us install only one Python package systemwide—ever—and that is `virtualenv`! Once `virtualenv` is installed, you have the power to create any number of small, self-contained "virtual Python environments" where packages can be installed and un-installed and with which you can experiment, all without contaminating your systemwide Python. When a particular project or experiment is over, you simply remove its virtual environment directory, and your system is clean.

In this case, you want to create a virtual environment in which to test the `pygeocoder` package. If you have never installed `virtualenv` on your system before, visit this URL to download and install it:

http://pypi.python.org/pypi/virtualenv

Once you have `virtualenv` installed, you can create a new environment using the following commands. (On Windows, the directory containing the Python binary in the virtual environment will be named `Scripts` instead of `bin`.)

```
$ virtualenv -p python3 geo_env
$ cd geo_env
$ ls
bin/  include/  lib/
$ . bin/activate
$ python -c 'import pygeocoder'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'pygeocoder'
```

As you can see, the `pygeocoder` package is not yet available. To install it, use the `pip` command that is inside your virtual environment that is now on your path thanks to your having run the `activate` command.

```
$ pip install pygeocoder
Downloading/unpacking pygeocoder
  Downloading pygeocoder-1.2.1.1.tar.gz
  Running setup.py egg_info for package pygeocoder

Downloading/unpacking requests>=1.0 (from pygeocoder)
  Downloading requests-2.0.1.tar.gz (412kB): 412kB downloaded
  Running setup.py egg_info for package requests

Installing collected packages: pygeocoder, requests
  Running setup.py install for pygeocoder

  Running setup.py install for requests

Successfully installed pygeocoder requests
Cleaning up...
```

The `python` binary inside the `virtualenv` will now have the `pygeocoder` package available.

```
$ python -c 'import pygeocoder'
```

Now that you have the `pygeocoder` package installed, you should be able to run the simple program named `search1.py`, as shown in Listing 1-1.

Listing 1-1. Fetching a Longitude and Latitude

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter01/search1.py

from pygeocoder import Geocoder

if __name__ == '__main__':
    address = '207 N. Defiance St, Archbold, OH'
    print(Geocoder.geocode(address)[0].coordinates)
```

By running it at the command line, you should see a result like this:

```
$ python3 search1.py
(41.521954, -84.306691)
```

And there, right on your computer screen is the answer to our question about the address's latitude and longitude! The answer has been pulled directly from Google's web service. The first example program is a rousing success.

Are you annoyed to have opened a book on Python network programming only to have found yourself immediately directed to download and install a third-party package that turned what might have been an interesting networking problem into a boring three-line Python script? Be at peace! Ninety percent of the time, you will find that this is exactly how programming challenges are solved—by finding other programmers in the Python community who have already tackled the problem you are facing and then building intelligently and briefly upon their solutions.

You are not yet done exploring this example, however. You have seen that a complex network service can often be accessed quite trivially. But what is behind the pretty `pygeocoder` interface? How does the service actually work? You will now explore, in detail, how this sophisticated service is actually just the top layer of a network stack that involves at least a half-dozen different levels.

Application Layers

The first program listing used a third-party Python library, downloaded from the Python Package Index, to solve a problem. It knew all about the Google Geocoding API and the rules for using it. But what if that library had not already existed? What if you had to build a client for Google's Maps API on your own?

For the answer, take a look at `search2.py`, as shown in Listing 1-2. Instead of using a geocoding-aware third-party library, it drops down one level and uses the popular `requests` library that lies behind `pygeocoding` and that, as you can see from the `pip install` command earlier, has also been installed in your virtual environment.

Listing 1-2. Fetching a JSON Document from the Google Geocoding API

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter01/search2.py

import requests

def geocode(address):
    parameters = {'address': address, 'sensor': 'false'}
    base = 'http://maps.googleapis.com/maps/api/geocode/json'
    response = requests.get(base, params=parameters)
```

```
    answer = response.json()
    print(answer['results'][0]['geometry']['location'])

if __name__ == '__main__':
    geocode('207 N. Defiance St, Archbold, OH')
```

Running this Python program returns an answer quite similar to that of the first script.

```
$ python3 search2.py
{'lat': 41.521954, 'lng': -84.306691}
```

The output is not *exactly* the same—you can see, for example, that the JSON data encoded the result as an "object" that `requests` has handed to you as a Python dictionary. But it is clear that this script has accomplished much the same thing as the first one.

The first thing that you will notice about this code is that the semantics offered by the higher-level `pygeocoder` module are absent. Unless you look closely at this code, you might not even see that it's asking about a mailing address at all! Whereas `search1.py` asked directly for an address to be turned into a latitude and longitude, the second listing painstakingly builds both a base URL and a set of query parameters whose purpose might not even be clear to you unless you have already read the Google documentation. If you want to read the documentation, by the way, you can find the API described here:

http://code.google.com/apis/maps/documentation/geocoding/

If you look closely at the dictionary of query parameters in `search2.py`, you will see that the `address` parameter provides the particular mailing address about which you are asking. The other parameter informs Google that you are not issuing this location query because of data pulled live from a mobile device location sensor.

When you receive a document back as a result of looking up this URL, you manually call the `response.json()` method to interpret it as JSON and then dive into the multilayered resulting data structure to find the correct element inside that holds the latitude and longitude.

The `search2.py` script then does the same thing as `search1.py`—but instead of doing so in the language of addresses and latitudes, it talks about the gritty details of constructing a URL, fetching a response, and parsing it as JSON. This is a common difference when you step down a level from one layer of a network stack to the layer beneath it: whereas the high-level code talked about what a request *meant*, the lower-level code can see only the details of how the request is *constructed*.

Speaking a Protocol

So, the second example script creates a URL and fetches the document that corresponds to it. That operation sounds quite simple, and, of course, your web browser works hard to make it look quite elementary. But the real reason that a URL can be used to fetch a document, of course, is that the URL is a kind of recipe that describes where to find—and how to fetch—a given document on the Web. The URL consists of the name of a protocol, followed by the name of the machine where the document lives, and finishes with the path that names a particular document on that machine. The reason then that the `search2.py` Python program is able to resolve the URL and fetch the document at all is that the URL provides instructions that tell a lower-level protocol how to find the document.

The lower-level protocol that the URL uses, in fact, is the famous Hypertext Transfer Protocol (HTTP), which is the basis of nearly all modern web communications. You will learn more about it in Chapters 9, 10, and 11 of this book. It is HTTP that provides the mechanism by which the Requests library is able to fetch the result from Google. What do you think it would look like if you were to strip that layer of magic off—what if you wanted to use HTTP to fetch the result directly? The result is `search3.py`, as shown in Listing 1-3.

Listing 1-3. Making a Raw HTTP Connection to Google Maps

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter01/search3.py

import http.client
import json
from urllib.parse import quote_plus

base = '/maps/api/geocode/json'

def geocode(address):
    path = '{}?address={}&sensor=false'.format(base, quote_plus(address))
    connection = http.client.HTTPConnection('maps.google.com')
    connection.request('GET', path)
    rawreply = connection.getresponse().read()
    reply = json.loads(rawreply.decode('utf-8'))
    print(reply['results'][0]['geometry']['location'])

if __name__ == '__main__':
    geocode('207 N. Defiance St, Archbold, OH')
```

In this listing, you are directly manipulating the HTTP protocol: asking it to connect to a specific machine, to issue a `GET` request with a path that you have constructed by hand, and finally to read the reply directly from the HTTP connection. Instead of being able conveniently to provide your query parameters as separate keys and values in a dictionary, you are having to embed them directly, by hand, in the path that you are requesting by first writing a question mark (`?`) followed by the parameters in the format `name=value` separated by `&` characters.

The result of running the program, however, is much the same as for the programs shown previously.

```
$ python3 search3.py
{'lat': 41.521954, 'lng': -84.306691}
```

As you will see throughout this book, HTTP is just one of many protocols for which the Python Standard Library provides a built-in implementation. In `search3.py`, instead of having to worry about all of the details of how HTTP works, your code can simply ask for a request to be sent and then take a look at the resulting response. The protocol details that the script has to deal with are, of course, more primitive than those of `search2.py`, because you have stepped down another level in the protocol stack, but at least you are still able to rely on the Standard Library to handle the actual network data and make sure that you get it right.

A Raw Network Conversation

HTTP cannot simply send data between two machines using thin air, of course. Instead, the HTTP protocol must operate by using some even simpler abstraction. In fact, it uses the capacity of modern operating systems to support a plain-text network conversation between two different programs across an IP network by using the TCP protocol. The HTTP protocol, in other words, operates by dictating *exactly* what the text of the messages will look like that pass back and forth between two hosts that can speak TCP.

When you move beneath HTTP to look at what happens below it, you are dropping down to the lowest level of the network stack that you can still access easily from Python. Take a careful look at `search4.py`, as shown in Listing 1-4. It makes exactly the same networking request to Google Maps as the previous three programs, but it does so by sending a raw text message across the Internet and receiving a bundle of text in return.

Listing 1-4. Talking to Google Maps Through a Bare Socket

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter01/search4.py

import socket
from urllib.parse import quote_plus

request_text = """\
GET /maps/api/geocode/json?address={}&sensor=false HTTP/1.1\r\n\
Host: maps.google.com:80\r\n\
User-Agent: search4.py (Foundations of Python Network Programming)\r\n\
Connection: close\r\n\
\r\n\
"""

def geocode(address):
    sock = socket.socket()
    sock.connect(('maps.google.com', 80))
    request = request_text.format(quote_plus(address))
    sock.sendall(request.encode('ascii'))
    raw_reply = b''
    while True:
        more = sock.recv(4096)
        if not more:
            break
        raw_reply += more
    print(raw_reply.decode('utf-8'))

if __name__ == '__main__':
    geocode('207 N. Defiance St, Archbold, OH')
```

In moving from `search3.py` to `search4.py`, you have passed an important threshold. In every previous program listing, you were using a Python library—written in Python itself—that knew how to speak a complicated network protocol on your behalf. But here you have reached the bottom: you are calling the raw `socket()` function that is provided by the host operating system to support basic network communications on an IP network. You are, in other words, using the same mechanisms that a low-level system programmer would use in the C language when writing this same network operation.

You will learn more about sockets over the next few chapters. For now, you can notice in `search4.py` that raw network communication is a matter of sending and receiving *byte strings*. The request that you send is one byte string, and the reply—that, in this case, you simply print to the screen so that you can experience it in all of its low-level glory—is another large byte string. (See the section "Encoding and Decoding," later in this chapter for the details of why you decode the string before printing it.) The HTTP request, whose text you can see inside the `sendall()` function, consists of the word *GET*—the name of the operation you want performed—followed by the path of the document you want fetched and the version of HTTP you support.

```
GET /maps/api/geocode/json?address=207+N.+Defiance+St%2C+Archbold%2C+OH&sensor=false HTTP/1.1
```

Then there are a series of headers that each consist of a name, a colon, and a value, and finally a carriage-return/newline pair that ends the request.

The reply, which will print as the script's output if you run `search4.py`, is shown as Listing 1-5. I chose simply to print the reply to the screen in this example, rather than write the complex text-manipulation code that would be able to interpret the response. I did so because I thought that simply reading the HTTP reply on your screen would give you a much better idea of what it looks like than if you had to decipher code designed to interpret it.

Listing 1-5. The Output of Running `search4.py`

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Sat, 23 Nov 2013 18:34:30 GMT
Expires: Sun, 24 Nov 2013 18:34:30 GMT
Cache-Control: public, max-age=86400
Vary: Accept-Language
Access-Control-Allow-Origin: *
Server: mafe
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic
Connection: close

{
   "results" : [
      {
         ...
         "formatted_address" : "207 North Defiance Street, Archbold, OH 43502, USA",
         "geometry" : {
            "location" : {
               "lat" : 41.521954,
               "lng" : -84.306691
            },
            ...
         },
         "types" : [ "street_address" ]
      }
   ],
   "status" : "OK"
}
```

You can see that the HTTP reply is quite similar in structure to the HTTP request. It begins with a status line, which is followed by a number of headers. After a blank line, the response content itself is shown: a JavaScript data structure, in a simple format known as JSON, that answers your query by describing the geographic location that the Google Geocoding API search has returned.

All of these status lines and headers, of course, are exactly the sort of low-level details that Python's `httplib` was taking care of in the earlier listings. Here, you see what the communication looks like if that layer of software is stripped away.

Turtles All the Way Down

I hope you have enjoyed these initial examples of what Python network programming can look like. Stepping back, I can use this series of examples to make several points about network programming in Python.

First, you can perhaps now see more clearly what is meant by the term *protocol stack*: it means building a high-level, semantically sophisticated conversation ("I want the geographic location of this mailing address") on top of simpler, and more rudimentary, conversations that ultimately are just text strings sent back and forth between two computers using their network hardware.

The particular protocol stack that you have just explored is four protocols high.

- On top is the Google Geocoding API, which tells you how to express your geographic queries as URLs that fetch JSON data containing coordinates.
- URLs name documents that can be retrieved using HTTP.
- HTTP supports document-oriented commands such as GET using raw TCP/IP sockets.
- TCP/IP sockets know how only to send and receive byte strings.

Each layer of the stack, you see, uses the tools provided by the layer beneath it and in turn offers capabilities to the next higher layer.

A second point made clear through these examples is how very complete the Python support is for every one of the network levels at which you have just operated. Only when using a vendor-specific protocol, and needing to format requests so that Google would understand them, was it necessary to resort to using a third-party library; I chose `requests` for the second listing not because the Standard Library lacks the `urllib.request` module but because its API is overly clunky. Every single one of the other protocol levels you encountered already had strong support inside the Python Standard Library. Whether you wanted to fetch the document at a particular URL or send and receive strings on a raw network socket, Python was ready with functions and classes that you could use to get the job done.

Third, note that my programs decreased considerably in quality as I forced myself to use increasingly lower-level protocols. The `search2.py` and `search3.py` listings, for example, started to hard-code things such as the form structure and hostnames in a way that is inflexible and that might be hard to maintain later. The code in `search4.py` is even worse: it includes a handwritten, unparameterized HTTP request whose structure is completely opaque to Python. And, of course, it contains none of the actual logic that would be necessary to parse and interpret the HTTP response and understand any network error conditions that might occur.

This illustrates a lesson that you should remember throughout every subsequent chapter of this book: that implementing network protocols correctly is difficult and that you should use the Standard Library or third-party libraries whenever possible. Especially when you are writing a network client, you will always be tempted to oversimplify your code; you will tend to ignore many error conditions that might arise, to prepare for only the most likely responses, to avoid properly escaping parameters because you fondly believe that your query strings will only ever include simple alphabetic characters, and, in general, to write very brittle code that knows as little about the service it is talking to as is technically possible. By instead using a third-party library that has developed a thorough implementation of a protocol, which has had to support many different Python developers who are using the library for a variety of tasks, you will benefit from all of the edge cases and awkward corners that the library implementer has already discovered and learned how to handle properly.

Fourth, it needs to be emphasized that higher-level network protocols—such as the Google Geocoding API for resolving a street address—generally work by *hiding* the network layers beneath them. If you only ever used the `pygeocoder` library, you might not even be aware that URLs and HTTP are the lower-level mechanisms that are being used to construct and answer your queries!

An interesting question, whose answer varies depending on how carefully a Python library has been written, is whether the library correctly hides errors at those lower levels. Could a network error that makes Google temporarily unreachable from your location raise a raw, low-level networking exception in the middle of code that's just trying to find the coordinates of a street address? Or will all errors be changed into a higher-level exception specific to geocoding? Pay careful attention to the topic of catching network errors as you go forward throughout this book, especially in the chapters of this first part with their emphasis on low-level networking.

Finally, we have reached the topic that will occupy you for the rest of this first part of the book: the `socket()` interface used in `search4.py` is *not*, in fact, the lowest protocol level in play when you make this request to Google! Just as the example has network protocols operating above the level above raw sockets, so also there are protocols down *beneath* the sockets abstraction that Python cannot see because your operating system manages them instead.

The layers operating below the `socket()` API are the following:

- The Transmission Control Protocol (TCP) supports two-way conversations made of streams of bytes by sending (or perhaps re-sending), receiving, and re-ordering small network messages called *packets*.
- The Internet Protocol (IP) knows how to send packets between different computers.
- The "link layer," at the very bottom, consists of network hardware devices such as Ethernet ports and wireless cards, which can send physical messages between directly linked computers.

Throughout the rest of this chapter, and in the two chapters that follow, you will explore these lowest protocol levels. You will start in this chapter by examining the IP level and then proceed in the following chapters to see how two quite different protocols—UDP and TCP—support the two basic kinds of conversation that are possible between applications on a pair of Internet-connected hosts.

But first, a few words about bytes and characters.

Encoding and Decoding

The Python 3 language makes a strong distinction between strings of characters and low-level sequences of bytes. *Bytes* are the actual binary numbers that computers transmit back and forth during network communication, each consisting of eight binary digits and ranging from the binary value 00000000 to 11111111 and thus from the decimal integer 0 to 255. Strings of *characters* in Python can contain Unicode symbols like *a* ("Latin small letter A," the Unicode standard calls it) or } ("right curly bracket") or ∅ (empty set). While each Unicode character does indeed each have a numeric identifier associated with it, called its *code point*, you can treat this as an internal implementation detail—Python 3 is careful to make characters always behave like characters, and only when you ask will Python convert the characters to and from actual externally visible bytes.

These two operations have formal names.

Decoding is what happens when bytes are on their way *into* your application and you need to figure out what they mean. Think of your application, as it receives bytes from a file or across the network, as a classic Cold War spy whose task is to decipher the transmission of raw bytes arriving from across a communications channel.

Encoding is the process of taking character strings that you are ready to present to the outside world and turning them into bytes using one of the many encodings that digital computers use when they need to transmit or store symbols using the bytes that are their only real currency. Think of your spy as having to turn their message back into numbers for transmission, as turning the symbols into a code that can be sent across the network.

These two operations are exposed quite simply and obviously in Python 3 as a `decode()` method that you can apply to byte strings after reading them in and as an `encode()` method that you can call on character strings when you are ready to write them back out. The techniques are illustrated in Listing 1-6.

Listing 1-6. Decoding Input Bytes and Encoding Characters for Output

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter01/stringcodes.py

if __name__ == '__main__':
    # Translating from the outside world of bytes to Unicode characters.
    input_bytes = b'\xff\xfe4\x001\x003\x00 \x00i\x00s\x00 \x00i\x00n\x00.\x00'
```

```
input_characters = input_bytes.decode('utf-16')
print(repr(input_characters))

# Translating characters back into bytes before sending them.
output_characters = 'We copy you down, Eagle.\n'
output_bytes = output_characters.encode('utf-8')
with open('eagle.txt', 'wb') as f:
    f.write(output_bytes)
```

The examples in this book attempt to differentiate carefully between bytes and characters. Note that the two have different appearances when you display their `repr()`: byte strings start with the letter *b* and look like `b'Hello'`, while real full-fledged character strings take no initial character and simply look like `'world'`. To try to discourage confusion between byte strings and character strings, Python 3 offers most string methods only on the character string type.

The Internet Protocol

Both *networking*, which occurs when you connect several computers with a physical link so that they can communicate, and *internetworking*, which links adjacent physical networks to form a much larger system like the Internet, are essentially just elaborate schemes to allow resource sharing.

All sorts of things in a computer, of course, need to be shared: disk drives, memory, and the CPU are all carefully guarded by the operating system so that the individual programs running on your computer can access those resources without stepping on each other's toes. The network is yet another resource that the operating system needs to protect so that programs can communicate with one another without interfering with other conversations that happen to be occurring on the same network.

The physical networking devices that your computer uses to communicate—like Ethernet cards, wireless transmitters, and USB ports—are themselves each designed with an elaborate ability to share a single physical medium among many different devices that want to communicate. A dozen Ethernet cards might be plugged into the same hub; 30 wireless cards might be sharing the same radio channel; and a DSL modem uses frequency-domain multiplexing, a fundamental concept in electrical engineering, to keep its own digital signals from interfering with the analog signals sent down the line when you talk on the telephone.

The fundamental unit of sharing among network devices—the currency, if you will, in which they trade—is the packet. A *packet* is a byte string whose length might range from a few bytes to a few thousand bytes, which is transmitted as a single unit between network devices. Although specialized networks do exist, especially in realms such as telecommunications, where each individual byte coming down a transmission line might be separately routed to a different destination, the more general-purpose technologies used to build digital networks for modern computers are all based on the larger unit of the packet.

A packet often has only two properties at the physical level: the byte-string data it carries and an address to which it is to be delivered. The address of a physical packet is usually a unique identifier that names one of the other network cards attached to the same Ethernet segment or wireless channel as the computer transmitting the packet. The job of a network card is to send and receive such packets without making the computer's operating system care about the details of how the network uses wires, voltages, and signals to operate.

What, then, is the Internet Protocol?

The *Internet Protocol* is a scheme for imposing a uniform system of addresses on all of the Internet-connected computers in the entire world and to make it possible for packets to travel from one end of the Internet to the other. Ideally, an application like your web browser should be able to connect to a host anywhere without ever knowing which maze of network devices each packet is traversing on its journey.

It is rare for a Python program to operate at such a low level that it sees the Internet Protocol itself in action, but it is helpful, at least, to know how it works.

IP Addresses

The original version of the Internet Protocol assigns a 4-byte address to every computer connected to the worldwide network. Such addresses are usually written as four decimal numbers, separated by periods, which each represent a single byte of the address. Each number can therefore range from 0 to 255. So, a traditional four-byte IP address looks like this:

```
130.207.244.244
```

Because purely numeric addresses can be difficult for humans to remember, the people using the Internet are generally shown *hostnames* rather than IP addresses. The user can simply type `google.com` and forget that behind the scene this resolves to an address like `74.125.67.103`, to which their computer can actually address packets for transmission over the Internet.

In the `getname.py` script, shown in Listing 1-7, you can see a simple Python program that asks the operating system—Linux, Mac OS, Windows, or on whatever system the program is running—to resolve the hostname `www.python.org`. The particular network service, called the Domain Name System, which springs into action to answer hostname queries is fairly complex, and I will discuss it in greater detail in Chapter 4.

Listing 1-7. Turning a Hostname into an IP Address

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter01/getname.py

import socket

if __name__ == '__main__':
    hostname = 'www.python.org'
    addr = socket.gethostbyname(hostname)
    print('The IP address of {} is {}'.format(hostname, addr))
```

For now, you just need to remember two things.

- First, however fancy an Internet application might look, the actual Internet Protocol always uses numeric IP addresses to direct packets toward their destination.
- Second, the complicated details of how hostnames are resolved to IP addresses are usually handled by the operating system.

Like most details of the operation of the Internet Protocol, your operating system prefers to take care of them itself, hiding the details both from you and from your Python code.

Actually, the addressing situation can be a bit more complex these days than the simple 4-byte scheme just described. Because the world is beginning to run out of 4-byte IP addresses, an extended address scheme, called IPv6, is being deployed that allows absolutely gargantuan 16-byte addresses that should serve humanity's needs for a long time to come. They are written differently from 4-byte IP addresses and look like this:

```
fe80::fcfd:4aff:fecf:ea4e
```

But as long as your code accepts IP addresses or hostnames from the user and passes them directly to a networking library for processing, you will probably never need to worry about the distinction between IPv4 and IPv6. The operating system on which your Python code is running will know which IP version it is using and should interpret addresses accordingly.

Generally, traditional IP addresses can be read from left to right: the first one or two bytes specify an organization, and then the next byte often specifies the particular subnet on which the target machine resides. The last byte narrows down the address to that specific machine or service. There are also a few special ranges of IP address that have a special meaning.

- `127.*.*.*`: IP addresses that begin with the byte `127` are in a special, reserved range that is local to the machine on which an application is running. When your web browser or FTP client or Python program connects to an address in this range, it is asking to speak to some other service or program that is running on the same machine. Most machines make use of only one address in this entire range: the IP address `127.0.0.1` is used universally to mean "this machine itself that this program is running on" and can often be accessed through the hostname `localhost`.
- `10.*.*.*, 172.16–31.*.*, 192.168.*.*`: These IP ranges are reserved for what are called *private subnets*. The authorities who run the Internet have made an absolute promise: they will never hand out IP addresses in any of these three ranges to real companies setting up servers or services. Out on the Internet at large, therefore, these addresses are guaranteed to have no meaning; they name no host to which you could want to connect. Therefore, these addresses are free for you to use on any of your organization's internal networks where you want to be free to assign IP addresses internally, without choosing to make those hosts accessible from other places on the Internet.

You are even likely to see some of these private addresses in your own home: your wireless router or DSL modem will often assign IP addresses from one of these private ranges to your home computers and laptops and hide all of your Internet traffic behind the single "real" IP address that your Internet service provider has allocated for your use.

Routing

Once an application has asked the operating system to send data to a particular IP address, the operating system has to decide how to transmit that data using one of the physical networks to which the machine is connected. This decision (that is, the choice of where to send each Internet Protocol packet based on the IP address that it names as its destination) is called *routing*.

Most, or perhaps all, of the Python code you write during your career will be running on hosts out at the edge of the Internet, with a single network interface that connects them to the rest of the world. For such machines, routing becomes a quite simple decision.

- If the IP address looks like `127.*.*.*`, then the operating system knows that the packet is destined for another application running on the same machine. It will not even be submitted to a physical network device for transmission but handed directly to another application via an internal data copy by the operating system.
- If the IP address is in the same subnet as the machine itself, then the destination host can be found by simply checking the local Ethernet segment, wireless channel, or whatever the local network happens to be, and sending the packet to a locally connected machine.
- Otherwise, your machine forwards the packet to a *gateway machine* that connects your local subnet to the rest of the Internet. It will then be up to the gateway machine to decide where to send the packet after that.

Of course, routing is only this simple at the edge of the Internet, where the only decisions are whether to keep the packet on the local network or to send it winging its way across the rest of the Internet. You can imagine that routing decisions are much more complex for the dedicated network devices that form the Internet's backbone! There, on the switches that connect entire continents, elaborate routing tables have to be constructed, consulted, and constantly updated in order to know that packets destined for Google go in one direction, packets directed to an Amazon IP address go in another, and packets directed to your machine go in yet another. But it is rare for Python applications to run on Internet backbone routers, so the simpler routing situation just outlined is nearly always the one you will see in action.

I have been a bit vague in the previous paragraphs about how your computer decides whether an IP address belongs to a local subnet or whether it should instead be forwarded through a gateway to the rest of the Internet. To illustrate the idea of a subnet, all of whose hosts share the same IP address prefix, I have been writing the prefix followed by asterisks for the parts of the address that could vary. Of course, the binary logic that runs your operating system's network stack does not actually insert little ASCII asterisks into its routing table! Instead, subnets are specified by combining an IP address with a mask that indicates how many of its most significant bits have to match to make a host belong to that subnet. If you keep in mind that every byte in an IP address represents eight bits of binary data, then you will be able to read subnet numbers easily. They look like this:

- *127.0.0.0/8*: This pattern, which describes the IP address range discussed previously and is reserved for the local host, specifies that the first 8 bits (1 byte) must match the number 127 and that the remaining 24 bits (3 bytes) can have any value they want.
- *192.168.0.0/16*: This pattern will match any IP address that belongs in the private 192.168 range because the first 16 bits must match perfectly. The last 16 bits of the 32-bit address are allowed to have whatever value they want.
- *192.168.5.0/24*: Here you have a specification for one particular individual subnet. This is probably the most common subnet mask on the entire Internet. The first three bytes of the address are completely specified, and they have to match for an IP address to fall into this range. Only the last byte (the last eight bits) is allowed to vary between machines in this range. This leaves 256 unique addresses. Typically, the `.0` address is used as the name of the subnet, and the `.255` address is used as the destination for a "broadcast packet" that addresses all of the hosts on the subnet (as you will see in the next chapter), which leaves 254 addresses free to be assigned to computers. The address `.1` is often used for the gateway that connects the subnet to the rest of the Internet, but some companies and schools choose to use another number for their gateways instead.

In nearly all cases, your Python code will simply rely on its host operating system to make packet routing choices correctly—just as it relies upon the operating system to resolve hostnames to IP addresses in the first place.

Packet Fragmentation

One last Internet Protocol concept that deserves mention is packet fragmentation. While it is supposed to be an obscure detail that is successfully hidden from your program by the cleverness of your operating system's network stack, it has caused enough problems over the Internet's history that it deserves at least a brief mention here.

Fragmentation is necessary because the Internet Protocol supports very large packets—they can be up to 64KB in length—but the actual network devices from which IP networks are built usually support much smaller packet sizes. Ethernet networks, for example, support only 1,500-byte packets. Internet packets therefore include a "don't fragment" (DF) flag with which the sender can choose what they want to happen if the packet proves too big to fit across one of the physical networks that lies between the source computer and the destination:

- If the DF flag is unset, then fragmentation is permitted, and when the packet reaches the threshold of the network onto which it cannot fit, the gateway can split it into smaller packets and mark them to be reassembled at the other end.
- If the DF flag is set, then fragmentation is prohibited, and if the packet cannot fit, then it will be discarded and an error message will be sent back—in a special signaling packet called an Internet Control Message Protocol (ICMP) packet—to the machine that sent the packet so that it can try splitting the message into smaller pieces and re-sending it.

Your Python programs will usually have no control over the DF flag; instead, it is set by the operating system. Roughly, the logic that the system will usually use is this: If you are having a UDP conversation (see Chapter 2) that consists of individual datagrams winging their way across the Internet, then the operating system will leave DF unset so that each datagram reaches the destination in however many pieces are needed; but if you are having a TCP conversation (see Chapter 3) whose long stream of data might be hundreds or thousands of packets long, then the operating system will set the DF flag so that it can choose exactly the right packet size to let the conversation flow smoothly, without its packets constantly being fragmented *en route*, which would make the conversation slightly less efficient.

The biggest packet that an Internet subnet can accept is called its *maximum transmission unit (MTU)*, and there used to be a big problem with MTU processing that caused problems for lots of Internet users. In the 1990s, Internet service providers (most notably phone companies offering DSL links) started using PPPoE, a protocol that puts IP packets inside a capsule that leaves them room for only 1,492 bytes instead of the full 1,500 bytes usually permitted across Ethernet. Many Internet sites were unprepared for this because they used 1,500-byte packets by default and had blocked all ICMP packets as a misguided security measure. As a consequence, their servers could never receive the ICMP errors telling them that their large, 1,500-byte "don't fragment" packets were reaching customers' DSL links and were unable to fit across them.

The maddening symptom of this situation was that small files or web pages could be viewed without a problem, and interactive protocols such as Telnet and SSH would work since both of these activities tend to send small packets that are less than 1,492 bytes long anyway. But once the customer tried downloading a large file or once a Telnet or SSH command disgorged several screens full of output at once, the connection would freeze and become unresponsive.

Today this problem is rarely encountered, but it illustrates how a low-level IP feature can generate user-visible symptoms and, therefore, why it is good to keep all of the features of IP in mind when writing and debugging network programs.

Learning More About IP

In the next chapters, you will step up to the protocol layers above IP and see how your Python programs can have different kinds of network conversations by using the different services built on top of the Internet Protocol. But what if you have been intrigued by the preceding outline of how IP works and want to learn more?

The official resources that describe the Internet Protocol are the requests for comment (RFCs) published by the IETF that describe exactly how the protocol works. They are carefully written and, when combined with a strong cup of coffee and a few hours of free reading time, will let you in on every single detail of how the Internet Protocols operate. Here, for example, is the RFC that defines the Internet Protocol itself:

```
http://tools.ietf.org/html/rfc791
```

You can also find RFCs referenced on general resources such as Wikipedia, and RFCs will often cite other RFCs that describe further details of a protocol or addressing scheme.

If you want to learn everything about the Internet Protocol and the other protocols that run on top of it, you might be interested in acquiring the venerable text, *TCP/IP Illustrated, Volume 1: The Protocols (2nd Edition)* by Kevin R. Fall and W. Richard Stevens (Addison-Wesley Professional, 2011). It covers, in fine detail, all of the protocol operations at which this book will only have the space to gesture. There are also other good books on networking in general, and that might help with network configuration in particular if setting up IP networks and routing is something you do either at work or even just at home to get your computers on the Internet.

Summary

All network services except the most rudimentary ones are implemented atop some other, more basic network function.

You explored such a "stack" in the opening sections of this chapter. The TCP/IP protocol (to be covered in Chapter 3) supports the mere transmission of byte strings between a client and server. The HTTP protocol (see Chapter 9) describes how such a connection can be used for a client to request a particular document and for the server to respond by providing it. The World Wide Web (Chapter 11) encodes the instructions for retrieving an HTTP-hosted document into a special address called a URL, and the standard JSON data format is popular for when the document returned by the server needs to present structured data to the client. And atop this entire edifice, Google offers a geocoding service that lets programmers build a URL to which Google replies with a JSON document describing a geographic location.

Whenever textual information is to be transmitted on the network—or, for that matter, saved to persistent byte-oriented storage such as a disk—the characters need to be encoded as bytes. There are several widely used schemes for representing characters as bytes. The most common on the modern Internet are the simple and limited ASCII encoding and the powerful and general Unicode system, especially its particular encoding known as UTF-8. Python byte strings can be converted to real characters using their `decode()` method, and normal character strings can be changed back through their `encode()` method. Python 3 tries never to convert bytes to strings automatically—an operation that would require it simply to guess at the encoding you intend—and so Python 3 code will often feature more calls to `decode()` and `encode()` than might have been your practice under Python 2.

For the IP network to transmit packets on an application's behalf, it is necessary that network administrators, appliance vendors, and operating system programmers have conspired together to assign IP addresses to individual machines, establish routing tables at both the machine and the router level, and configure the Domain Name System (Chapter 4) to associate IP addresses with user-visible names. Python programmers should know that each IP packet winds its own way across the network toward the destination and that a packet might be fragmented if it is too large to fit across one of the "hops" between routers along its path.

There are two basic ways to use IP from most applications. They are either to use each packet as a stand-alone message or to ask for a stream of data that gets split into packets automatically. These protocols are named UDP and TCP, and they are the subjects to which this book turns in Chapter 2 and Chapter 3.