Foundations of Python Network Programming

Introduction to Client-Server Networking

Google offers a geocoding service that lets programmers build a URL to which Google replies with a JSON document describing a geographic location.

The best Python technology for quickly trying a new library: virtualenv!

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'
```

To install pygeocoder, 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

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.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UDP

When you craft programs that accept port numbers from user input such as the command line or configuration files, it is friendly to allow not just numeric port numbers but human-readable names for well-known ports. These names are standard, and they are available through the getservbyname() function inside Python's standard socket module. If you want to ask the port for the Domain Name Service, you can find out this way:

```
>>> import socket
>>> socket.getservbyname('domain')
53
```

Python's Standard Library simply provides an object-based interface to all of the normal, gritty, low-level operating system calls that are normally used to accomplish networking tasks on POSIX-compliant operating systems.

Many socket options are specific to particular operating systems, and they may be finicky about how their options are presented. Here are some of the more common options:

- SO_BROADCAST: This allows broadcast UDP packets to be sent and received.
- SO_DONTROUTE: Only be willing to send packets that are addressed to hosts on subnets to which this computer is connected directly.
- SO_TYPE: When passed to getsockopt(), this returns to you whether a socket is of type SOCK_DGRAM and can be used for UDP or whether it is of type SOCK_STREAM and instead supports the semantics of TCP.

There are a few good reasons to use UDP.

- Because you are implementing a protocol that already exists and it uses UDP.
- Because you are designing a time-critical media stream whose redundancy allows for occasional packet loss and you never want this second's data getting hung up waiting for old data from several seconds ago that has not yet been delivered (as happens with TCP).
- Because unreliable LAN subnet multicast is a great pattern for your application and UDP supports it perfectly.

The POSIX network stack gives you access to UDP through the idea of a "socket," which is a communications endpoint that can sit at an IP address and UDP port number—these two things together are called the socket's *name* or *address*—and send and receive datagrams. Python offers these primitive network operations through the built-in socket module.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

TCP

TCP basic tenets:

- Every TCP packet is given a sequence number so that the system on the receiving end can put them back together in the right order and can also notice missing packets in the sequence and ask that they be retransmitted.

- Instead of using sequential integers (1, 2, 3...) to sequence packets, TCP uses a counter that counts the number of bytes transmitted. A 1,024-byte packet with a sequence number of 7,200, for example, would be followed by a packet with a sequence number of 8,224. This means that a busy network stack does not have to remember how it broke up a data stream into packets. If asked for a retransmission, it can break up the stream into new packets some other way (which might let it fit more data into a packet if more bytes are now waiting for transmission), and the receiver can still put the packets back together.

- The initial sequence number, in good TCP implementations, is chosen randomly so that villains cannot assume that every connection starts at byte zero. Predictable sequence numbers unfortunately make it easier to craft forged packets that might interrupt a conversation by looking like they are a legitimate part of its data.

- Rather than running very slowly in lock step by needing every packet to be acknowledged before it sends the next one, TCP sends whole bursts of packets at a time before expecting a response. The amount of data that a sender is willing to have on the wire at any given moment is called the size of the TCP window.

- The TCP implementation on the receiving end can regulate the window size of the transmitting end and thus slow or pause the connection. This is called flow control. This lets a receiver forbid the transmission of additional packets in cases where its input buffer is full, and it would have to discard more data anyway even if it were to arrive.

- Finally, if TCP believes that packets are being dropped, it assumes that the network is becoming congested and reduces how much data it sends every second. This can be something of a disaster on wireless networks and other media where packets are lost simply because of noise. It can also ruin connections that are running fine until a router reboots andthe endpoints cannot talk for, say, 20 seconds. By the time the network comes back up, the two TCP peers will have decided that the network is extraordinarily overloaded with traffic, and upon reestablishing contact, they will at first refuse to send each other data at anything other than a trickle.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Socket Names and DNS

In Python, you can test directly for whether the underlying platform supports IPv6 by checking the has_ipv6 Boolean attribute inside the socket module.

```
>>> import socket
>>> socket.has_ipv6
True
```

To make code simple, powerful, and immune from the complexities of the transition from IPv4 to IPv6, should turn attention to one of the most powerful tools in the Python socket user's arsenal: getaddrinfo().

The getaddrinfo() function sits in the socket module along with most other operations that involve addresses.

Its approach is simple. Rather than making attack the addressing problem piecemeal, which is necessary when using the older routines in the socket module, it lets specify everything you know about the connection that you need to make in a single call. In response, it returns all of the coordinates, which are necessary for you to create and connect a socket to the named destination.

```
>>> from pprint import pprint
>>> infolist = socket.getaddrinfo('gatech.edu', 'www')
>>> pprint(infolist)
[(2, 1, 6, '', ('130.207.244.244', 80)),
(2, 2, 17, '', ('130.207.244.244', 80))]
>>> info = infolist[0]
>>> info[0:3]
(2, 1, 6)
>>> s = socket.socket(*info[0:3])
>>> info[4]
('130.207.244.244', 80)
>>> s.connect(info[4])
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Network Data and Network Errors

Python makes it easy to see the difference between the two endians. Simply use the struct module, which provides a variety of operations for converting data to and from popular binary formats. Here is the number 4253 represented first in a little-endian format and then in a big-endian order:

```
>>> import struct
>>> struct.pack('<i', 4253)
b'\x9d\x10\x00\x00'
>>> struct.pack('>i', 4253)
b'\x00\x00\x10\x9d'
```

Advice for preparing binary data for transmission across a network socket:

- Use the struct module to produce binary data for transmission on the network and to unpack it upon arrival.
- Select network byte order with the '!' prefix if you control the data format.
- If someone else has designed the protocol and specified little-endian, then you will have to use '<' instead.

TLS/SSL

Several open source implementations of TLS are available. The Python Standard Library opts to wrap the most popular, the OpenSSL library.

Through its introduction of the ssl.create_default_context() function, Python 3.4 makes it dramatically easier for Python applications to use TLS safely than did earlier versions of Python.

Here are the TLS-aware protocols that come with the Python Standard Library: (сделать слайд по первым словам)

- http.client: When you build an HTTPSConnection object, you can use the constructor's context keyword to pass in an SSLContext with your own settings. Unfortunately, neither urllib.request nor the third-party Requests library currently accept an SSLContext argument as part of their APIs.
- smtplib: When you build an SMTP_SSL object, you can use the constructor's context keyword to pass in an SSLContext with your own settings. If instead you create a plain SMTP object and only later call its starttls() method, then you provide the context parameter to that method call.
- poplib: When you build a POP3_SSL object, you can use the constructor's context keyword to pass in an SSLContext with your own settings. If instead you create a plain POP3 object and only later call its stls() method, then you would provide the context parameter to that method call.
- imaplib: When you build an IMAP4_SSL object, you can use the constructor's ssl_context keyword to pass in an SSLContext with your own settings. If instead you create a plain IMAP4 object and only later call its starttls() method, then you would provide the ssl_context parameter to that method call.
- ftplib: When you build an FTP_TLS object, you can use the constructor's context keyword to pass in an SSLContext with your own settings. Note that the first line or two of the FTP conversation will always pass in the clear (such as the "220" welcome message that often includes the server hostname) before you have the chance to turn on encryption. An FTP_TLS object will automatically turn on encryption before the login() method sends a username and password. If you are not logging in to the remote server but want encryption turned on anyway, you will have to call the auth() method manually as the first action you take after connecting.
- nntplib: Although the NNTP network news (Usenet) protocol is not covered in this book, I should note that it too can be secured. If you build an NNTP_SSL, you can use the constructor's ssl_context keyword to pass in an SSLContext with your own settings. If instead you create a plain NNTP object and only later call its starttls() method, then you would provide the context parameter to that method call.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Server Architecture

Client Program for Example Zen-of-Python Protocol

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter07/client.py
# Simple Zen-of-Python client that asks three questions then disconnects.

import argparse, random, socket, zen_utils

def client(address, cause_error=False):
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(address)
aphorisms = list(zen_utils.aphorisms)
if cause_error:
        sock.sendall(aphorisms[0][:-1])
        return
for aphorism in random.sample(aphorisms, 3):
        sock.sendall(aphorism)
        print(aphorism, zen_utils.recv_until(sock, b'.'))
sock.close()

if __name__ == '__main__':
parser = argparse.ArgumentParser(description='Example client')
parser.add_argument('host', help='IP or hostname')
parser.add_argument('-e', action='store_true', help='cause an error')
parser.add_argument('-p', metavar='port', type=int, default=1060,
                                help='TCP port (default 1060)')
args = parser.parse_args()
address = (args.host, args.p)
client(address, args.e)
```

The Simplest Possible Server Is Single-Threaded

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter07/srv_single.py
# Single-threaded server that serves one client at a time; others must wait.

import zen_utils

if __name__ == '__main__':
        address = zen_utils.parse_command_line('simple single-threaded server')
        listener = zen_utils.create_srv_socket(address)
        zen_utils.accept_connections_forever(listener)
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Caches and Message Queues

Using Memcached to Accelerate an Expensive Operation

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter08/squares.py
# Using memcached to cache expensive results.

import memcache, random, time, timeit

def compute_square(mc, n):
    value = mc.get('sq:%d' % n)
    if value is None:
        time.sleep(0.001) # pretend that computing a square is expensive
        value = n * n
        mc.set('sq:%d' % n, value)
return value

def main():
    mc = memcache.Client(['127.0.0.1:11211'])

    def make_request():
        compute_square(mc, random.randint(0, 5000))

    print('Ten successive runs:')
    for i in range(1, 11):
        print(' %.2fs' % timeit.timeit(make_request, number=2000), end='')
    print()

if __name__ == '__main__':
    main()
```

The Memcached daemon needs to be running on your machine at port 11211 for this example to succeed.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

HTTP Clients

There is an important symmetry built into HTTP: the request and response use the same rules to establish formatting and framing. Here is an example request and response to which you can refer as you read the description of the protocol that follows:

```
GET /ip HTTP/1.1
User-Agent: curl/7.35.0
Host: localhost:8000
Accept: */*
HTTP/1.1 200 OK
Server: gunicorn/19.1.1
Date: Sat, 20 Sep 2014 00:18:00 GMT
Connection: close
Content-Type: application/json
Content-Length: 27
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
{
"origin": "127.0.0.1"
}
```

The request is the block of text that begins with GET. The response begins with the version HTTP/1.1, and it continues through the blank line below the headers to include the three lines of JSON text. Both the request and the response are called an HTTP *message* in the standard, and each message is composed of three parts.

- A first line that names a method and document in the request and names a return code and description in the response. The line ends with a carriage return and linefeed (CR-LF, ASCII codes 13 and 10).

- Zero or more headers that consist of a name, a colon, and a value. Header names are case-insensitive, so they can be capitalized however a client or server desires. Each header ends with a CR-LF. A blank line then terminates the entire list of headers—the four bytes CR-LF-CR-LF that form a pair of end-of-line sequences with nothing in between them.

This blank line is mandatory whether any headers appear above it or not.

- An optional body that immediately follows the blank line that end the headers. There are several options for framing the entity, as you will learn shortly.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

HTTP Servers

Normal, synchronous HTTP in Python is usually mediated by the WSGI standard. Servers parse the incoming request to produce a dictionary full of information, and applications examine the dictionary before returning HTTP headers and an optional response body. This lets you use any web server you want with any standard Python web framework.

A Simple HTTP Service Written as a WSGI Client

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter10/wsgi_env.py
# A simple HTTP service built directly against the low-level WSGI spec.

from pprint import pformat
from wsgiref.simple_server import make_server

def app(environ, start_response):
    headers = {'Content-Type': 'text/plain; charset=utf-8'}
    start_response('200 OK', list(headers.items()))
    yield 'Here is the WSGI environment:\r\n\r\n'.encode('utf-8')
    yield pformat(environ).encode('utf-8')

if __name__ == '__main__':
    httpd = make_server('', 8000, app)
    host, port = httpd.socket.getsockname()
    print('Serving on', host, 'port', port)
    httpd.serve_forever()
```

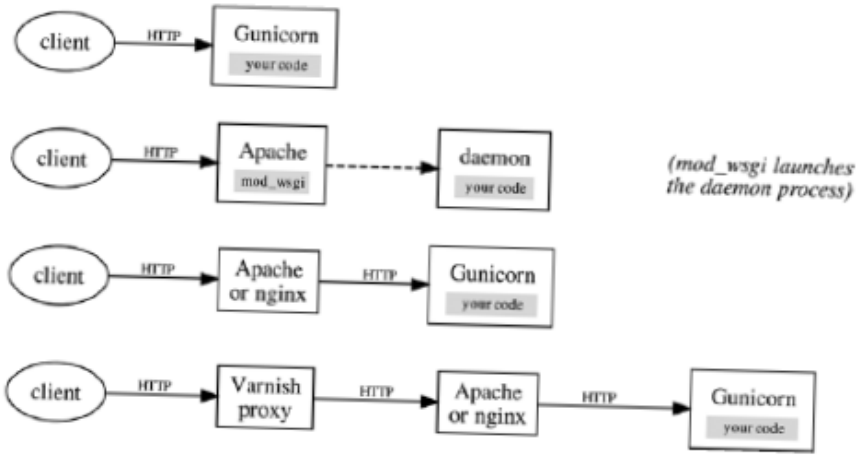

Figure *Four common techniques for deploying Python code stand-alone or behind reverse HTTP proxies*

Four architectures are popular for serving HTTP from Python. A stand-alone server can be run using Gunicorn or other pure-Python server implementations such as CherryPy. Other architects opt to run their Python under the control of Apache through mod_wsgi. However, now that the concept of a reverse proxy is a go-to pattern for web services of all kinds, many architects find it simpler to put Gunicorn or another pure-Python server directly behind nginx or Apache as a separate HTTP service to which they can forward requests for paths whose resources are generated dynamically.

- Run a server that is itself written in Python and that can call your WSGI endpoint directly from its own code. The Green Unicorn ("gunicorn") server is the most popular at the moment, but other production-ready, pure-Python servers are available. The old battle-tested CherryPy server, for example, is still used in projects today, and Flup still attracts users. (It is best to avoid prototype servers such as wsgiref, unless your service will be under light load and internal to an organization.) If you use an async server engine, then both the server and the framework will necessarily live in the same process.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- Run Apache with mod_wsgi configured to run your Python code inside of a separate WSGIDaemonProcess, producing a hybrid approach: two different languages are at work but within a single server. Static resources can be served directly from Apache's C-language engine, while dynamic paths are submitted to mod_wsgi so that it can call the Python interpreter to run your application code. (This option is not available for async web frameworks because WSGI provides no mechanism by which an application could yield control temporarily and then finish its work later.)

- Run a Python HTTP server like Gunicorn (or whatever server is dictated by your choice of async framework) behind a web server that can serve static files directly but also act a reverse proxy for the dynamic resources that you have written in Python. Both Apache and nginx are popular front-end servers for this task. They can also load-balance requests between several back-end servers if your Python application outgrows a single box.

- Run a Python HTTP server behind Apache or nginx that itself sits behind a pure reverse proxy like Varnish, creating a third tier that faces the real world. These reverse proxies can be geographically distributed so that cached resources are served from locations close to client machines instead of all from the same continent. *Content delivery networks* such as Fastly operate by deploying armies of Varnish servers to machine rooms on each continent and then using them to offer you a turnkey service that both terminates your externally facing TLS certificates and forwards requests to your central servers.

The World Wide Web

You can investigate how your favorite web sites package up information through two features of a modern browser like Google Chrome or Firefox. They will show you the HTML code—syntax highlighted, no less—for the page you are looking at if you press Ctrl+U. You can right-click any element and select Inspect Element to bring up debugging tools that let you investigate how each document element relates to the content that you are seeing on the page, as shown in Figure.

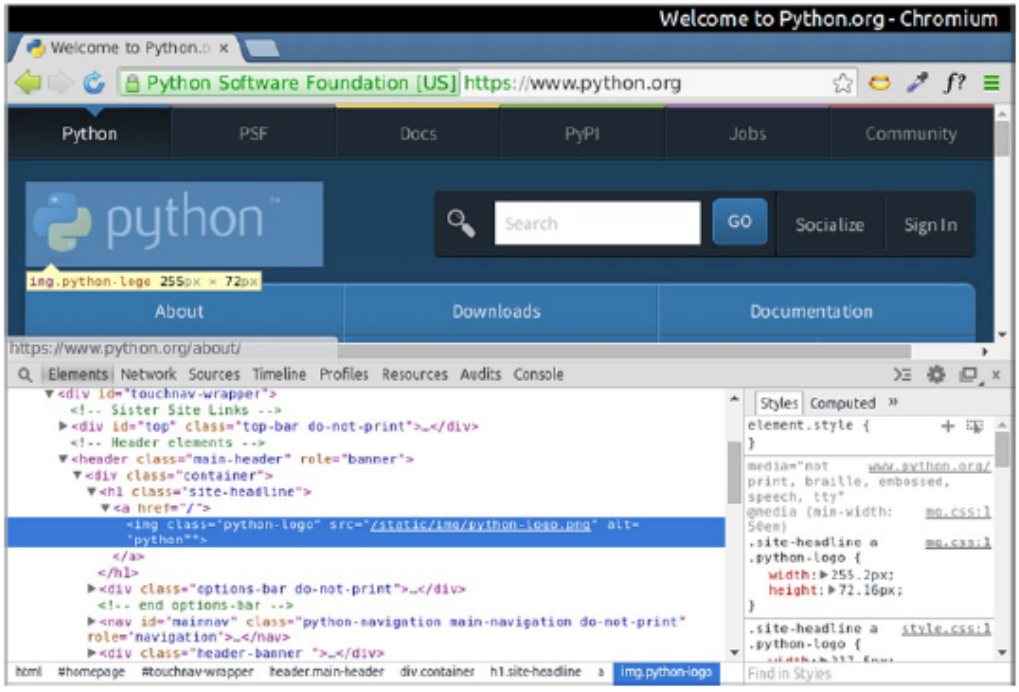

Figure *The Inspect tab in Google Chrome*

And while in the inspector, you can switch to a Network tab that will show you all of the other resources that were downloaded and displayed as the result of visiting the page.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Building and Parsing E-Mail

Generating a Simple Text E-Mail Message

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
#https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter12/build_basic_email.py

import email.message, email.policy, email.utils, sys

text = """Hello,
This is a basic message from Chapter 12.
- Anonymous"""

def main():
    message = email.message.EmailMessage(email.policy.SMTP)
    message['To'] = 'recipient@example.com'
    message['From'] = 'Test Sender <sender@example.com>'
    message['Subject'] = 'Test Message, Chapter 12'
    message['Date'] = email.utils.formatdate(localtime=True)
    message['Message-ID'] = email.utils.make_msgid()
    message.set_content(text)
    sys.stdout.buffer.write(message.as_bytes())

if __name__ == '__main__':
main()
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

SMTP

SMTP is used to transmit e-mail messages to e-mail servers. Python provides the smtplib module for SMTP clients to use. By calling the sendmail() method of SMTP objects, you can transmit messages.

ESMTP is an extension to SMTP. It allows you to discover the maximum message size supported by a remote SMTP server prior to transmitting a message. ESMTP also permits TLS, which is a way to encrypt your conversation with a remote server.

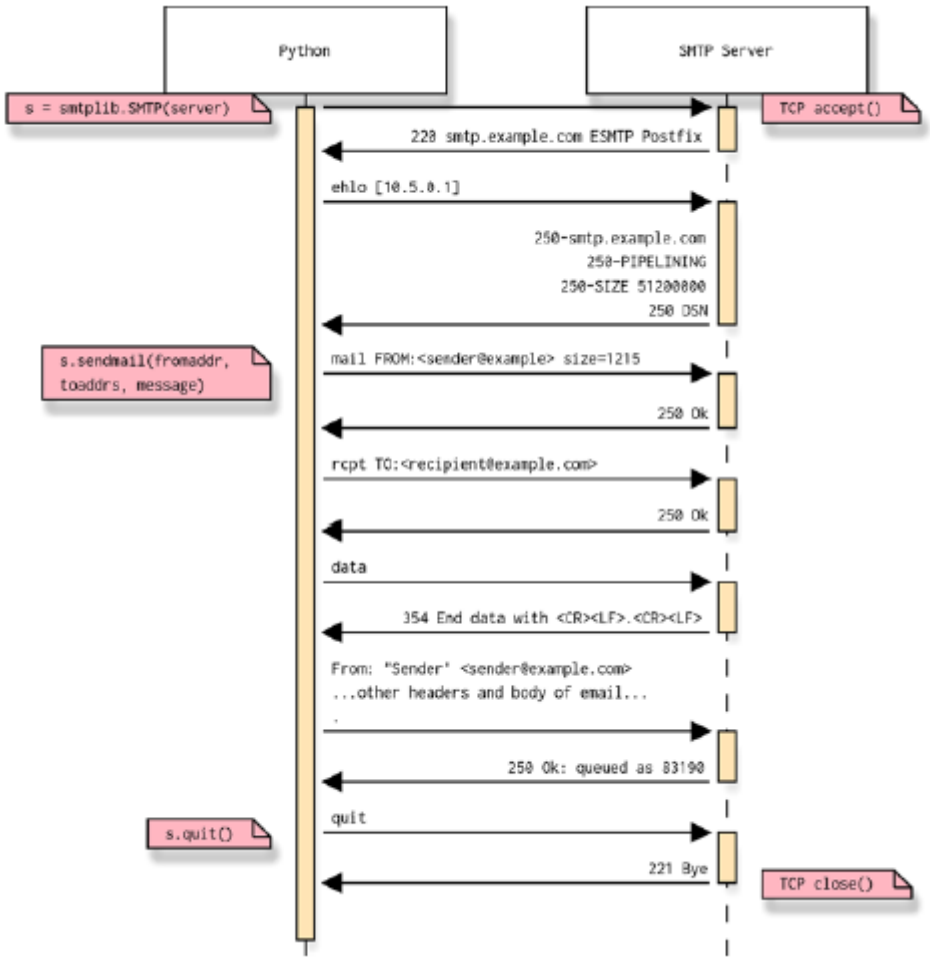

Figure *An example of a Python-driven SMTP session*

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

POP

POP, the Post Office Protocol, is a simple protocol for downloading e-mail from a server. It is typically used through an e-mail client like Thunderbird or Outlook.

The most common implementation of POP is version 3, commonly referred to as POP3.

The Python Standard Library provides the poplib module, which provides a convenient interface for using POP.

Figure illustrates a very simple POP conversation driven from Python.

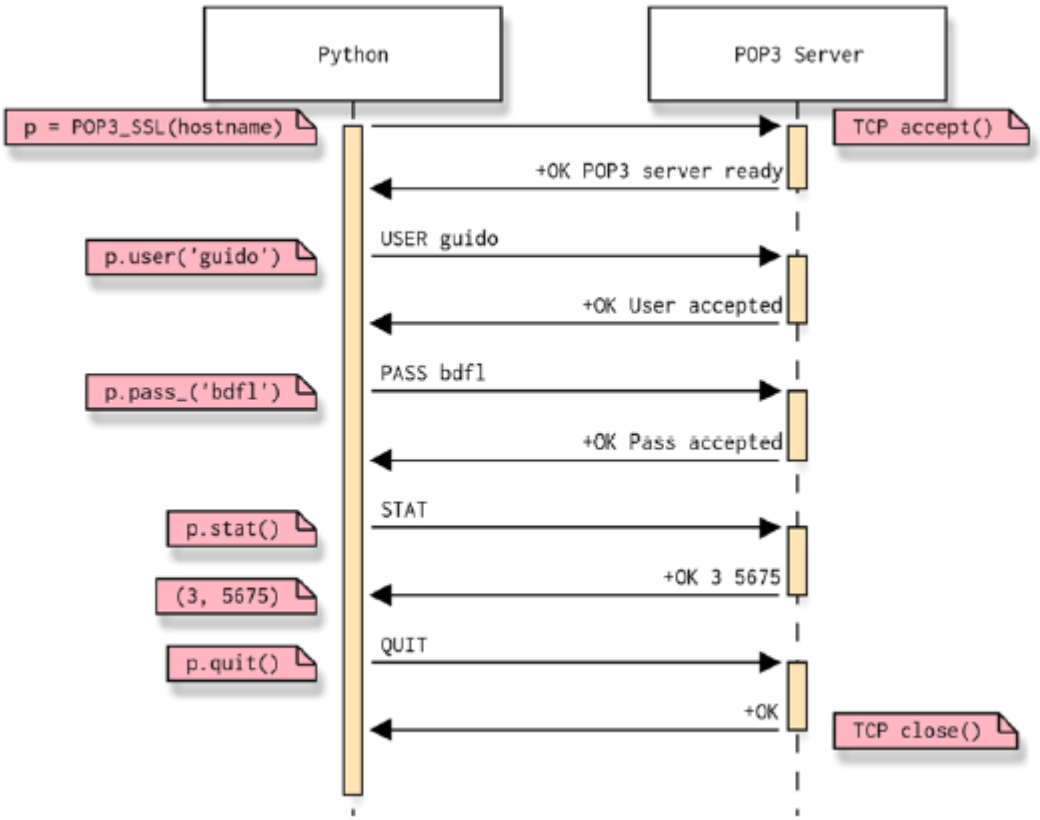

Figure *A simple conversation using POP*

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

IMAP

IMAP is a robust protocol for accessing e-mail messages stored on a remote server. Many IMAP libraries exist for Python; imaplib is built into the Python Standard Library, but it requires you to do all sorts of low-level response parsing by yourself. A far better choice is IMAPClient by Menno Smits, which you can install from the Python Package Index.

Figure provides a sample conversation between Python and an IMAP server.

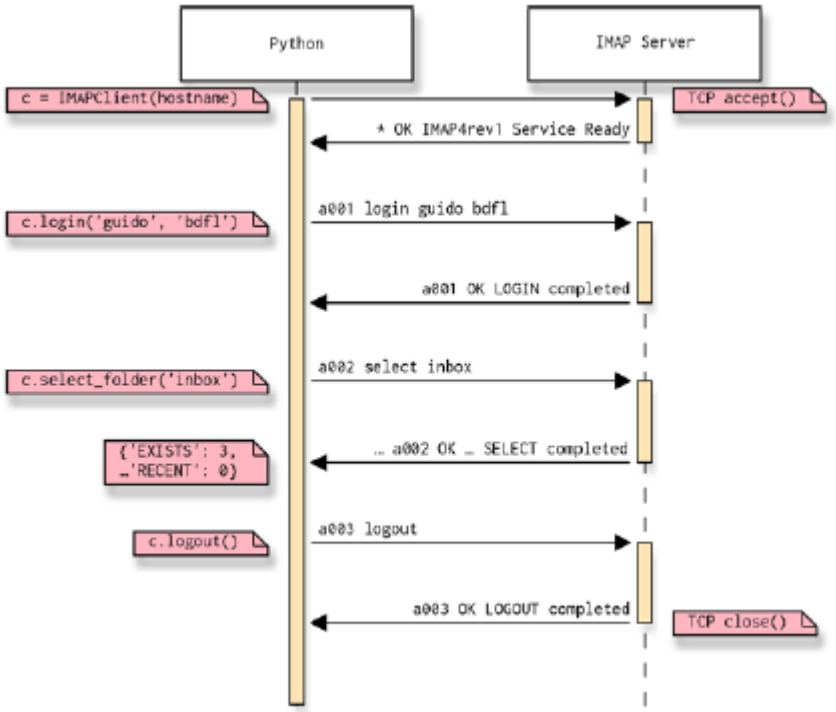

Figure *An example conversation between Python and an IMAP server*

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Telnet and SSH

Logging Into a Remote Host Using Telnet

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter16/telnet_login.py
# Connect to localhost, watch for a login prompt, and try logging in

import argparse, getpass, telnetlib

def main(hostname, username, password):
t = telnetlib.Telnet(hostname)
# t.set_debuglevel(1) # uncomment to get debug messages
t.read_until(b'login:')
t.write(username.encode('utf-8'))
t.write(b'\r')
t.read_until(b'assword:') # first letter might be 'p' or 'P'
t.write(password.encode('utf-8'))
t.write(b'\r')
n, match, previous_text = t.expect([br'Login incorrect', br'\$'], 10)
if n == 0:
print('Username and password failed - giving up')
else:
t.write(b'exec uptime\r')
print(t.read_all().decode('utf-8')) # read until socket closes

if __name__ == '__main__':
parser = argparse.ArgumentParser(description='Use Telnet to log in')
parser.add_argument('hostname', help='Remote host to telnet to')
parser.add_argument('username', help='Remote username')
args = parser.parse_args()
password = getpass.getpass('Password: ')
main(args.hostname, args.username, password)
```

FTP

In Python, the ftplib library is used to talk to FTP servers.

The Python module ftplib is the primary interface to FTP for Python programmers. It handles the details of establishing the various connections for you, and it provides convenient ways to automate common commands.

Making a Simple FTP Connection

```
#!/usr/bin/env python3
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter17/connect.py

from ftplib import FTP

def main():
ftp = FTP('ftp.ibiblio.org')
print("Welcome:", ftp.getwelcome())
ftp.login()
print("Current working directory:", ftp.pwd())
ftp.quit()

if __name__ == '__main__':
main()
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

RPC

Making XML-RPC Calls

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Foundations of Python Network Programming, Third Edition
# https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter18/xmlrpc_client.py
# XML-RPC client

import xmlrpc.client

def main():
proxy = xmlrpc.client.ServerProxy('http://127.0.0.1:7001')
print(proxy.addtogether('x', 'ÿ', 'z'))
print(proxy.addtogether(20, 30, 4, 1))
print(proxy.quadratic(2, -4, 0))
print(proxy.quadratic(1, 2, 1))
print(proxy.remote_repr((1, 2.0, 'three')))
print(proxy.remote_repr([1, 2.0, 'three']))
print(proxy.remote_repr({'name': 'Arthur',
'data': {'age': 42, 'sex': 'M'}}))
print(proxy.quadratic(1, 0, 1))

if __name__ == '__main__':
main()
```

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP. It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.