Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра Компьютерных систем и программных технологий

Отчет по лабораторной работе

по дисциплине «Сети ЭВМ и телекоммуникации»

на тему: «POP3 – клиент на языке Python»

Работу выполнил:

студент гр. 43501/3 Родина В.В.

Руководитель Вылегжанина К.Д.

«___» _____ 2016 г

Санкт-Петербург

2016

1. Постановка задачи

Реализовать POP3 – клиент на языке программирования Python для ОС Windows.

Для тестирования программы использовать готовый POP3 – сервер.

2. Ход работы

Итак, приведем немного теории, на основе которой написан клиент, а именно про протокол POP3.

Протокол POP3 (Post Office Protocol — version 3, третья версия протокола почтового отделения) является наиболее распространенным протоколом получения электронной почты с почтового сервера. Для этих целей также используется IMAP. В IMAP предусмотрено больше возможностей, чем в POP3, но зато POP3 намного проще. На рис.1 и 2 изображены модели клиент-сервер по протоколу POP. Сервер POP находится между агентом пользователя и почтовыми ящиками.

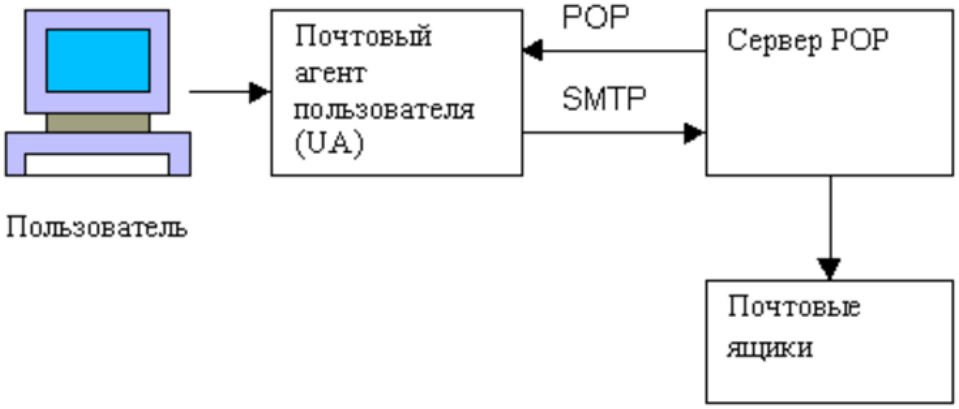

Рис. 1. Конфигурация модели клиент – сервер по протоколу POP

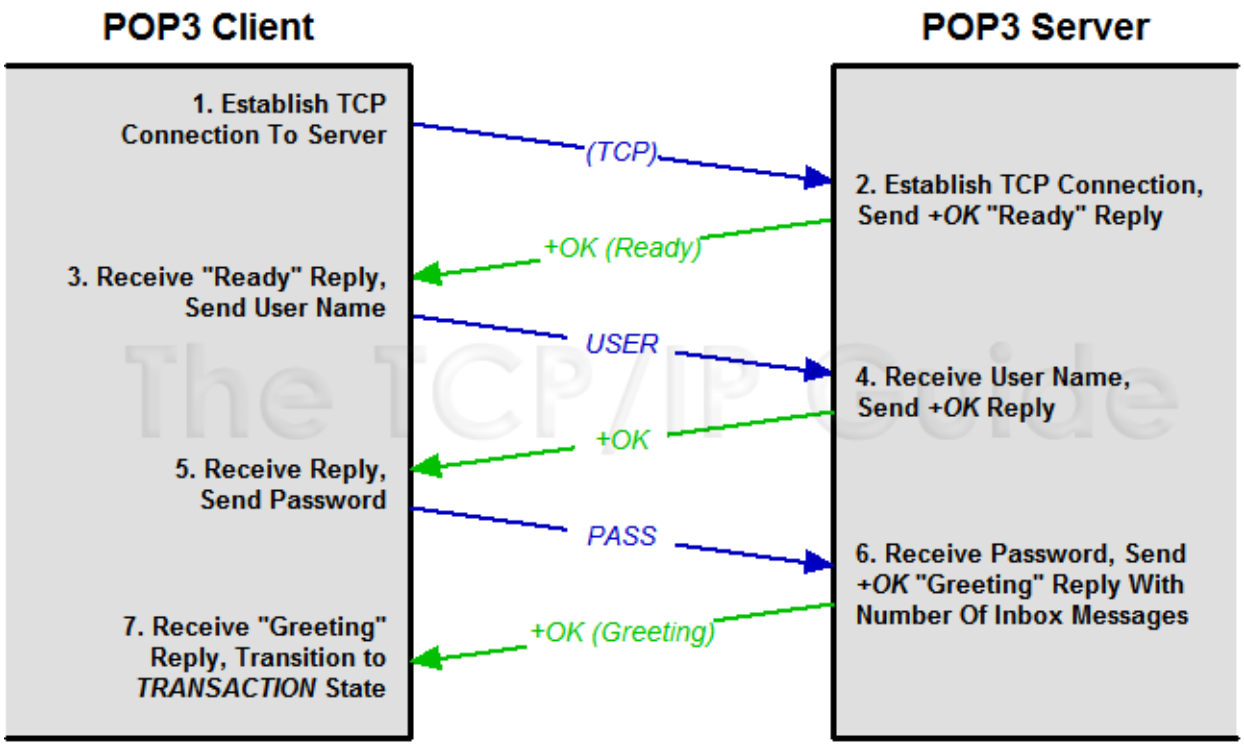

Рис. 2. Клиент – сервер POP3

Конструкция протокола POP3 обеспечивает возможность пользователю обратиться к своему почтовому серверу и изъять накопившуюся для него почту. Пользователь может получить доступ к POP-серверу из любой точки доступа к Интернет. При этом он должен запустить специальный почтовый агент (UA), работающий по протоколу POP3, и настроить его для работы со своим почтовым сервером. Итак, во главе модели POP находится отдельный персональный компьютер, работающий исключительно в качестве клиента почтовой системы (сервера). Сообщения доставляются клиенту по протоколу POP, а посылаются по-прежнему при помощи SMTP. То есть на компьютере пользователя существуют два отдельных агента-интерфейса к почтовой системе - доставки (POP) и отправки (SMTP). Разработчики протокола POP3 называет такую ситуацию "раздельные агенты".

В протоколе POP3 оговорены три стадии процесса получения почты: авторизация, транзакция и обновление. После того как сервер и клиент POP3 установили соединение, начинается стадия авторизации. На стадии авторизации клиент идентифицирует себя для сервера. Если авторизация прошла успешно, сервер открывает почтовый ящик клиента и начинается стадия транзакции. В ней клиент либо запрашивает у сервера информацию (например, список почтовых сообщений), либо просит его совершить определенное действие (например, выдать почтовое сообщение). Наконец, на стадии обновления сеанс связи заканчивается. На Рис. 3 перечислены

команды протокола POP3, обязательные для работающей в Интернет реализации минимальной конфигурации.

Команда	Описание
USER	Идентифицирует пользователя с указанным именем
PASS	Указывает пароль для пары клиент-сервер
QUIT	Закрывает TCP-соединение
STAT	Сервер возвращает количество сообщений в почтовом ящике плюс размер почтового ящика
LIST	Сервер возвращает идентификаторы сообщений вместе с размерами сообщений (параметром команды может быть идентификатор сообщения)
RETR	Извлекает сообщение из почтового ящика (требуется указывать аргумент-идентификатор сообщения)
DELE	Отмечает сообщение для удаления (требуется указывать аргумент - идентификатор сообщения)
NOOP	Сервер возвращает положительный ответ, но не совершает никаких действий
LAST	Сервер возвращает наибольший номер сообщения из тех, к которым ранее уже обращались
RSET	Отменяет удаление сообщения, отмеченного ранее командой DELE

Рис. 3. Команды протокола POP3 для минимальной конфигурации

В протоколе POP3 определено несколько команд, но на них дается только два ответа: +OK (позитивный, аналогичен сообщению-подтверждению ACK) и -ERR (негативный, аналогичен сообщению "не подтверждено" NAK). Оба ответа подтверждают, что обращение к серверу произошло и что он вообще отвечает на команды. Как правило, за каждым ответом следует его содержательное словесное описание. В RFC 1225 есть образцы нескольких типичных сеансов POP3.

Итак, написанный почтовый клиент протестирован двумя способами. Первый способ не позволил протестировать программу полностью на все функции, я использовала свой реальный почтовый ящик на Яндексе. С помощью настройки на самом Яндексе на «режим POP3» удалось «вытащить» нужное мне письмо из моей почты, так как сам протокол не позволяет отправлять письмо, а только извлечь (скачать) его, прочесть или удалить. Таким образом, выполнены все эти действия. На консоли в программе клиента мы просто увидели текст «вытащенного из почты» сообщения почтового ящика valerirodina@ya.ru.

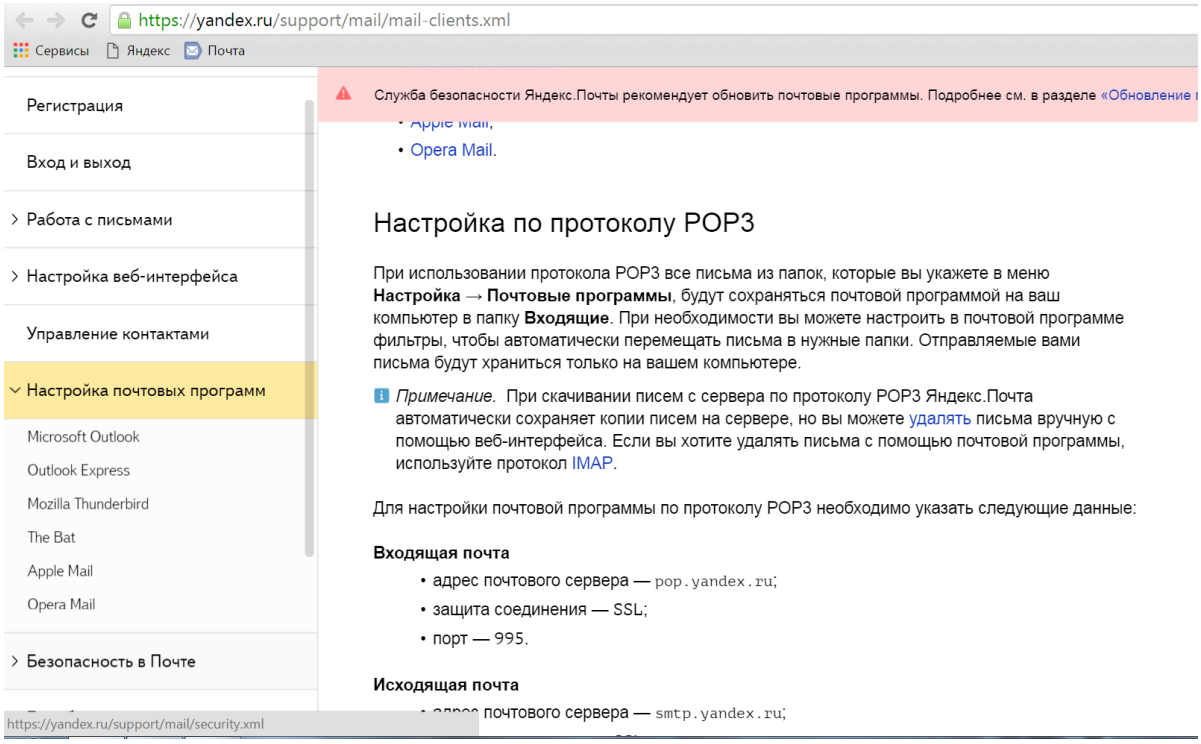

Рис. 4. Первый способ тестирования с использованием почтового сервера Яндекс

Второй способ производился с помощью консоли среды разработки PyCharm, так как в Python есть готовые библиотеки для использования почты, как и в Java, что значительно облегчило задачу написания кода самого почтового клиента. Итак, далее разберем по шагам и функциям, проверенным данным способом. То есть, проведем простейший сеанс взаимодействия POP3 – клиента и сервера.

POP3 сессия состоит из нескольких режимов. Как только соединение с сервером было установлено и сервер отправил приглашение, то сессия переходит в режим AUTHORIZATION (Авторизация). В этом режиме клиент должен идентифицировать себя на сервере. После успешной идентификации сессия переходит в режим TRANSACTION (Передача). В этом режиме клиент запрашивает сервер выполнить определённые команды. Когда клиент отправляет команду QUIT, сессия переходит в режим UPDATE. В этом режиме POP3 сервер освобождает все занятые ресурсы и завершает работу. После этого TCP соединение закрывается.

Авторизация: как только будет установлено TCP соединение с POP3 сервером, он отправляет приглашение - S: +OK POP3 server ready.

Теперь POP3 сессия находится в режиме AUTHORIZATION. Клиент должен идентифицировать себя на сервере, используя команды USER и PASS.

Сначала надо отправить команду USER, после которой в качестве аргумента следует имя пользователя. Если сервер отвечает положительно, то теперь необходимо отправить команду PASS, за которой следует пароль. Если после отправки команды USER или PASS сервер отвечает негативно, то можно попробовать авторизоваться снова или выйти из сессии с помощью команды QUIT. После успешной авторизации сервер открывает и блокирует maildrop (почтовый ящик). В ответе на команду PASS сервер сообщает сколько сообщений находится в почтовом ящике и передаёт их общий размер. Теперь сессия находится в режиме TRANSACTION. Команды, которые были введены и проверены:

1. Команда: USER. Нужно отослать строку, указывающая имя почтового ящика и команда передаст серверу имя пользователя. Ответа два:

+OK name is a valid mailbox

-ERR never heard of mailbox name

2. Команда: PASS – передать серверу пароль для почтового ящика. Отв:

+OK maildrop locked and ready

-ERR invalid password

-ERR unable to lock maildrop

3. Команда: QUIT, у нее нет аргументов; сервер завершает POP3 сессию и переходит в режим UPDATE (обновляется). Ответ:

+OK

После успешной идентификации пользователя на сервере POP3 сессия переходит в режим TRANSACTION, где пользователь может передавать ниже следующие команды. После каждой из таких команд следует ответ сервера. Доступные команды в этом режиме:

4. Команда: STAT. В ответ на вызов команды сервер выдаёт положительный ответ "+OK", за которым следует количество сообщений в почтовом ящике и их общий размер в символах. Сообщения, которые помечены для удаления не учитываются в ответе сервера.

+OK n s

5. Команда: LIST. Если был передан аргумент, то сервер выдаёт информацию о указанном сообщении. Если аргумент не был передан, то сервер выдаёт информацию о всех сообщениях, находящихся в почтовом ящике. Сообщения, помеченные для удаления не перечисляются.

+OK scan listing follows

-ERR no such message

6. Команда:RETR. После положительного ответа сервер передаёт содержание сообщения.

+OK message follows

-ERR no such message

7. Команда: DELE. POP3 сервер помечает указанное сообщение как удалённое, но не удаляет его, пока сессия не перейдёт в режим UPDATE.

+OK message deleted

-ERR no such message

8. Команда: NOOP. POP3 сервер ничего не делает и всегда отвечает положительно.

+OK

9. Команда: RSET. Если какие - то сообщения были помечены для удаления, то с них снимается эта метка.

+OK

C: RSET

S: +OK maildrop has 2 messages (320 octets)

Итак, приведем результат работы при данном способе тестирования:

```
> telnet pop.example.com pop3
telnet: Trying 192.0.2.2...
telnet: Connected to pop.example.com.
telnet: Escape character is '^]'.
server: +OK InterMail POP3 server ready.
client: USER MyUsername
server: +OK please send PASS command
client: PASS MyPassword
server: +OK MyUsername is welcome here
client: LIST
server: +OK 1 messages
server: 1 1801
server: .
client: RETR 1
server: +OK 1801 octets
server: Return-Path: sender@example.com
server: Received: from client.example.com ([192.0.2.1])
server:         by mx1.example.com with ESMTP

server: From: sender@example.com
server: Subject: Test message
server: To: recipient@example.com
server: Message-Id: <20040120203404.CCCC18555.mx1.example.com@client.example.com>
server:
server: This is a test message.
server: .
client: DELE 1
server: +OK
client: quit
server: +OK MyUsername InterMail POP3 server signing off.
```

Рис. 5. Тестовый сеанс связи в PyCharm

Также результаты работы клиента можно проверить и через командную строку:

S: <создаём новое TCP соединение с POP3 сервером>

S: +OK POP3 server ready

C: USER valeriarodina

S: +OK User valeriarodina is exists

C: PASS mymail

S: +OK valeriarodina's maildrop has 2 messages (320 octets)

C: STAT

S: +OK 2 320

C: LIST

S: +OK 2 messages (320 octets)

S: 1 120

S: 2 200

S: .

C: RETR 1

S: +OK 120 octets

S:

S: .

C: DELE 1

S: +OK message 1 deleted

C: RETR 2

S: +OK 200 octets

S:

S: .

C: DELE 2

S: +OK message 2 deleted

C: QUIT

S: +OK POP3 server signing off (maildrop empty)

C: <закрываем соединение>

Таким образом, видим, клиент работает исправно.

3. Выводы

Итак, удалось реализовать POP3 – клиент. С написанием программы особых сложностей не было, так как в Python существует готовая библиотека для использования почты. Но также я познакомилась с новым для себя языком программирования Python, так как раньше его не изучала, удалось освоить основы языка применительно к клиент – серверной архитектуре.

4. Листинги

INIT.PY

```
import poplib, email
# Учетные данные пользователя:
SERVER = "pop.yandex.ru"
USERNAME = "valeriarodina"
USERPASSWORD = "secretword"
p = poplib.POP3(SERVER)
```

```
print(p.getwelcome())
# этап идентификации
print(p.user(USERNAME))
print(p.pass_(USERPASSWORD))
# этап транзакций
response, lst, octets = p.list()
print(response)
for msgnum, msgsize in [i.split() for i in lst]:
    print("Сообщение %(msgnum)s имеет длину %(msgsize)s" % vars())
    print("UIDL =", p.uidl(int(msgnum)).split()[2])
if int(msgsize) > 32000:
    (resp, lines, octets) = p.top(msgnum, 0)
else:
    (resp, lines, octets) = p.retr(msgnum)
msgtxt = "\n".join(lines)+"\n\n"
msg = email.message_from_string(msgtxt)
print("* От: %(from)s\n* Кому: %(to)s\n* Тема: %(subject)s\n" % msg)
# msg содержит заголовки сообщения или все сообщение


print(p.quit())
```

POPLIB.PY (основной класс, сформировался сам после построения проекта)

```
"""A POP3 client class.

Based on the J. Myers POP3 draft, Jan. 96
"""

# Author: David Ascher <david_ascher@brown.edu>
#         [heavily stealing from nntplib.py]
# Updated: Piers Lauder <piers@cs.su.oz.au> [Jul '97]
# String method conversion and test jig improvements by ESR, February 2001.
# Added the POP3_SSL class. Methods loosely based on IMAP_SSL. Hector Urtubia
<urtubia@mrbook.org> Aug 2003

# Example (see the test function at the end of this file)

# Imports

import errno
import re
import socket

try:
    import ssl
    HAVE_SSL = True
except ImportError:
    HAVE_SSL = False

__all__ = ["POP3","error_proto"]

# Exception raised when an error or invalid response is received:

class error_proto(Exception): pass

# Standard Port
POP3_PORT = 110

# POP SSL PORT
POP3_SSL_PORT = 995
```

```
# Line terminators (we always output CRLF, but accept any of CRLF, LFCR, LF)
CR = b'\r'
LF = b'\n'
CRLF = CR+LF

# maximal line length when calling readline(). This is to prevent
# reading arbitrary length lines. RFC 1939 limits POP3 line length to
# 512 characters, including CRLF. We have selected 2048 just to be on
# the safe side.
_MAXLINE = 2048


class POP3:

    """This class supports both the minimal and optional command sets.
    Arguments can be strings or integers (where appropriate)
    (e.g.: retr(1) and retr('1') both work equally well.

    Minimal Command Set:
            USER name               user(name)
            PASS string             pass_(string)
            STAT                    stat()
            LIST [msg]              list(msg = None)
            RETR msg                retr(msg)
            DELE msg                dele(msg)
            NOOP                    noop()
            RSET                    rset()
            QUIT                    quit()

    Optional Commands (some servers support these):
            RPOP name               rpop(name)
            APOP name digest        apop(name, digest)
            TOP msg n               top(msg, n)
            UIDL [msg]              uidl(msg = None)
            CAPA                    capa()
            STLS                    stls()
            UTF8                    utf8()

    Raises one exception: 'error_proto'.

    Instantiate with:
            POP3(hostname, port=110)

    NB:     the POP protocol locks the mailbox from user
            authorization until QUIT, so be sure to get in, suck
            the messages, and quit, each time you access the
            mailbox.

            POP is a line-based protocol, which means large mail
            messages consume lots of python cycles reading them
            line-by-line.

            If it's available on your mail server, use IMAP4
            instead, it doesn't suffer from the two problems
            above.
    """

    encoding = 'UTF-8'

    def __init__(self, host, port=POP3_PORT,
                 timeout=socket._GLOBAL_DEFAULT_TIMEOUT):
        self.host = host
```

```
        self.port = port
        self._tls_established = False
        self.sock = self._create_socket(timeout)
        self.file = self.sock.makefile('rb')
        self._debugging = 0
        self.welcome = self._getresp()

    def _create_socket(self, timeout):
        return socket.create_connection((self.host, self.port), timeout)

    def _putline(self, line):
        if self._debugging > 1: print('*put*', repr(line))
        self.sock.sendall(line + CRLF)


    # Internal: send one command to the server (through  putline())

    def _putcmd(self, line):
        if self._debugging: print('*cmd*', repr(line))
        line = bytes(line, self.encoding)
        self._putline(line)


    # Internal: return one line from the server, stripping CRLF.
    # This is where all the CPU time of this module is consumed.
    # Raise error_proto('-ERR EOF') if the connection is closed.

    def _getline(self):
        line = self.file.readline(_MAXLINE + 1)
        if len(line) > _MAXLINE:
            raise error_proto('line too long')

        if self._debugging > 1: print('*get*', repr(line))
        if not line: raise error_proto('-ERR EOF')
        octets = len(line)
        # server can send any combination of CR & LF
        # however, 'readline()' returns lines ending in LF
        # so only possibilities are ...LF, ...CRLF, CR...LF
        if line[-2:] == CRLF:
            return line[:-2], octets
        if line[:1] == CR:
            return line[1:-1], octets
        return line[:-1], octets


    # Internal: get a response from the server.
    # Raise 'error proto' if the response doesn't start with '+'.

    def _getresp(self):
        resp, o = self._getline()
        if self._debugging > 1: print('*resp*', repr(resp))
        if not resp.startswith(b'+'):
            raise error_proto(resp)
        return resp


    # Internal: get a response plus following text from the server.

    def _getlongresp(self):
        resp = self._getresp()
        list = []; octets = 0
        line, o = self._getline()
        while line != b'.':
```

```
            if line.startswith(b'..'):
                o = o-1
                line = line[1:]
            octets = octets + o
            list.append(line)
            line, o = self._getline()
        return resp, list, octets


    # Internal: send a command and get the response

    def _shortcmd(self, line):
        self._putcmd(line)
        return self._getresp()


    # Internal: send a command and get the response plus following text

    def _longcmd(self, line):
        self._putcmd(line)
        return self._getlongresp()


    # These can be useful:

    def getwelcome(self):
        return self.welcome


    def set_debuglevel(self, level):
        self._debugging = level


    # Here are all the POP commands:

    def user(self, user):
        """Send user name, return response

        (should indicate password required).
        """
        return self._shortcmd('USER %s' % user)


    def pass_(self, pswd):
        """Send password, return response

        (response includes message count, mailbox size).

        NB: mailbox is locked by server from here to 'quit()'
        """
        return self._shortcmd('PASS %s' % pswd)


    def stat(self):
        """Get mailbox status.

        Result is tuple of 2 ints (message count, mailbox size)
        """
        retval = self._shortcmd('STAT')
        rets = retval.split()
        if self._debugging: print('*stat*', repr(rets))
        numMessages = int(rets[1])
        sizeMessages = int(rets[2])
```

```
        return (numMessages, sizeMessages)


    def list(self, which=None):
        """Request listing, return result.

        Result without a message number argument is in form
        ['response', ['mesg_num octets', ...], octets].

        Result when a message number argument is given is a
        single response: the "scan listing" for that message.
        """
        if which is not None:
            return self._shortcmd('LIST %s' % which)
        return self._longcmd('LIST')


    def retr(self, which):
        """Retrieve whole message number 'which'.

        Result is in form ['response', ['line', ...], octets].
        """
        return self._longcmd('RETR %s' % which)


    def dele(self, which):
        """Delete message number 'which'.

        Result is 'response'.
        """
        return self._shortcmd('DELE %s' % which)


    def noop(self):
        """Does nothing.

        One supposes the response indicates the server is alive.
        """
        return self._shortcmd('NOOP')


    def rset(self):
        """Unmark all messages marked for deletion."""
        return self._shortcmd('RSET')


    def quit(self):
        """Signoff: commit changes on server, unlock mailbox, close
connection."""
        resp = self._shortcmd('QUIT')
        self.close()
        return resp

    def close(self):
        """Close the connection without assuming anything about it."""
        try:
            file = self.file
            self.file = None
            if file is not None:
                file.close()
        finally:
            sock = self.sock
            self.sock = None
```

```
            if sock is not None:
                try:
                    sock.shutdown(socket.SHUT_RDWR)
                except OSError as e:
                    # The server might already have closed the connection
                    if e.errno != errno.ENOTCONN:
                        raise
                finally:
                    sock.close()

    #  del    = quit


    # optional commands:

    def rpop(self, user):
        """Not sure what this does."""
        return self._shortcmd('RPOP %s' % user)


    timestamp = re.compile(br'\+OK.*(<[^>]+>)')

    def apop(self, user, password):
        """Authorisation

        - only possible if server has supplied a timestamp in initial
greeting.

        Args:
                user     - mailbox user;
                password - mailbox password.

        NB: mailbox is locked by server from here to 'quit()'
        """
        secret = bytes(password, self.encoding)
        m = self.timestamp.match(self.welcome)
        if not m:
            raise error_proto('-ERR APOP not supported by server')
        import hashlib
        digest = m.group(1)+secret
        digest = hashlib.md5(digest).hexdigest()
        return self._shortcmd('APOP %s %s' % (user, digest))


    def top(self, which, howmuch):
        """Retrieve message header of message number 'which'
        and first 'howmuch' lines of message body.

        Result is in form ['response', ['line', ...], octets].
        """
        return self._longcmd('TOP %s %s' % (which, howmuch))


    def uidl(self, which=None):
        """Return message digest (unique id) list.

        If 'which', result contains unique id for that message
        in the form 'response mesgnum uid', otherwise result is
        the list ['response', ['mesgnum uid', ...], octets]
        """
        if which is not None:
            return self._shortcmd('UIDL %s' % which)
        return self._longcmd('UIDL')
```

```
    def utf8(self):
        """Try to enter UTF-8 mode (see RFC 6856). Returns server response.
        """
        return self._shortcmd('UTF8')


    def capa(self):
        """Return server capabilities (RFC 2449) as a dictionary
        >>> c=poplib.POP3('localhost')
        >>> c.capa()
        {'IMPLEMENTATION': ['Cyrus', 'POP3', 'server', 'v2.2.12'],
         'TOP': [], 'LOGIN-DELAY': ['0'], 'AUTH-RESP-CODE': [],
         'EXPIRE': ['NEVER'], 'USER': [], 'STLS': [], 'PIPELINING': [],
         'UIDL': [], 'RESP-CODES': []}
        >>>

        Really, according to RFC 2449, the cyrus folks should avoid
        having the implementation split into multiple arguments...
        """
        def _parsecap(line):
            lst = line.decode('ascii').split()
            return lst[0], lst[1:]

        caps = {}
        try:
            resp = self._longcmd('CAPA')
            rawcaps = resp[1]
            for capline in rawcaps:
                capnm, capargs = _parsecap(capline)
                caps[capnm] = capargs
        except error_proto as _err:
            raise error_proto('-ERR CAPA not supported by server')
        return caps


    def stls(self, context=None):
        """Start a TLS session on the active connection as specified in RFC
2595.

                context - a ssl.SSLContext
        """
        if not HAVE_SSL:
            raise error_proto('-ERR TLS support missing')
        if self._tls_established:
            raise error_proto('-ERR TLS session already established')
        caps = self.capa()
        if not 'STLS' in caps:
            raise error_proto('-ERR STLS not supported by server')
        if context is None:
            context = ssl._create_stdlib_context()
        resp = self._shortcmd('STLS')
        self.sock = context.wrap_socket(self.sock,
                                        server_hostname=self.host)
        self.file = self.sock.makefile('rb')
        self._tls_established = True
        return resp


if HAVE_SSL:

    class POP3_SSL(POP3):
```

```
        """POP3 client class over SSL connection

        Instantiate with: POP3_SSL(hostname, port=995, keyfile=None,
certfile=None,
                                   context=None)

                hostname - the hostname of the pop3 over ssl server
                port - port number
                keyfile - PEM formatted file that contains your private key
                certfile - PEM formatted certificate chain file
                context - a ssl.SSLContext

        See the methods of the parent class POP3 for more documentation.
        """

        def __init__(self, host, port=POP3_SSL_PORT, keyfile=None,
certfile=None,
                     timeout=socket._GLOBAL_DEFAULT_TIMEOUT, context=None):
            if context is not None and keyfile is not None:
                raise ValueError("context and keyfile arguments are mutually
"
                                 "exclusive")
            if context is not None and certfile is not None:
                raise ValueError("context and certfile arguments are mutually
"
                                 "exclusive")
            self.keyfile = keyfile
            self.certfile = certfile
            if context is None:
                context = ssl._create_stdlib_context(certfile=certfile,
                                                     keyfile=keyfile)
            self.context = context
            POP3.__init__(self, host, port, timeout)

        def _create_socket(self, timeout):
            sock = POP3._create_socket(self, timeout)
            sock = self.context.wrap_socket(sock,
                                            server_hostname=self.host)
            return sock

        def stls(self, keyfile=None, certfile=None, context=None):
            """The method unconditionally raises an exception since the
            STLS command doesn't make any sense on an already established
            SSL/TLS session.
            """
            raise error_proto('-ERR TLS session already established')

    __all__.append("POP3_SSL")

if __name__ == "__main__":
    import sys
    a = POP3(sys.argv[1])
    print(a.getwelcome())
    a.user(sys.argv[2])
    a.pass_(sys.argv[3])
    a.list()
    (numMsgs, totalSize) = a.stat()
    for i in range(1, numMsgs + 1):
        (header, msg, octets) = a.retr(i)
        print("Message %d:" % i)
        for line in msg:
            print('   ' + line)
```

```
        print('------------------------')
    a.quit()
```