Python. Лекция 9. Сетевые приложения на Python.

Работа с сокетами

Применяемая в IP-сетях архитектура клиент-сервер использует IP-пакеты для коммуникации между клиентом и сервером. Клиент отправляет запрос серверу, на который тот отвечает. В случае с TCP/IP между клиентом и сервером устанавливается соединение (обычно с двусторонней передачей данных), а в случае с UDP/IP - клиент и сервер обмениваются пакетами (дейтаграммамми) с негарантированной доставкой.

Каждый сетевой интерфейс IP-сети имеет уникальный в этой сети адрес (**IP-адрес**). Упрощенно можно считать, что каждый компьютер в сети Интернет имеет собственный IP-адрес. При этом в рамках одного сетевого интерфейса может быть несколько сетевых **портов**. Для установления сетевого соединения приложение клиента должно выбрать свободный порт и установить соединение с серверным приложением, которое слушает (listen) порт с определенным номером на удаленном сетевом интерфейсе. Пара IP-адрес и порт характеризуют **сокет** (гнездо) - начальную (конечную) точку сетевой коммуникации. Для создания соединения TCP/IP необходимо два сокета: один на локальной машине, а другой - на удаленной. Таким образом, каждое сетевое соединение имеет IP-адрес и порт на локальной машине, а также IP-адрес и порт на удаленной машине.

Модуль `socket` обеспечивает возможность работать с сокетами из Python. Сокеты используют транспортный уровень согласно семиуровневой модели OSI (Open Systems Interconnection, взаимодействие открытых систем), то есть относятся к более низкому уровню, чем большинство описываемых в этом разделе протоколов.

Уровни модели OSI:

Физический

Поток битов, передаваемых по физической линии. Определяет параметры физической линии.

Канальный (Ethernet, PPP, ATM и т.п.)

Кодирует и декодирует данные в виде потока битов, справляясь с ошибками, возникающими на физическом уровне в пределах физически единой сети.

Сетевой (IP)

Маршрутизирует информационные пакеты от узла к узлу.

Транспортный (TCP, UDP и т.п.)

Обеспечивает прозрачную передачу данных между двумя точками соединения.

Сеансовый

Управляет сеансом соединения между участниками сети. Начинает, координирует и завершает соединения.

Представления

Обеспечивает независимость данных от формы их представления путем преобразования форматов. На этом уровне может выполняться прозрачное (с точки зрения вышележащего уровня) шифрование и дешифрование данных.

Приложений (HTTP, FTP, SMTP, NNTP, POP3, IMAP и т.д.)

Поддерживает конкретные сетевые приложения. Протокол зависит от типа сервиса.

Каждый сокет относится к одному из коммуникационных доменов. Модуль `socket` поддерживает домены UNIX и Internet. Каждый домен подразумевает свое семейство протоколов и адресацию. Данное изложение будет затрагивать только домен Internet, а именно протоколы TCP/IP и UDP/IP, поэтому для указания коммуникационного домена при создании сокета будет указываться константа `socket.AF_INET`.

В качестве примера следует рассмотреть простейшую клиент-серверную пару. Сервер будет принимать строку и отвечать клиенту. Сетевое устройство иногда называют хостом (host), поэтому будет употребляться этот термин по отношению к компьютеру, на котором работает сетевое приложение.

Сервер:

```
import socket, string

def do_something(x):
  lst = map(None, x);
  lst.reverse();
  return string.join(lst, "")

HOST = ""                 # localhost
PORT = 33333
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.bind((HOST, PORT))
while 1:
  print "Слушаю порт 33333"
  srv.listen(1)
  sock, addr = srv.accept()
  while 1:
    pal = sock.recv(1024)
    if not pal:
      break
    print "Получено от %s:%s:" % addr, pal
    lap = do_something(pal)
    print "Отправлено %s:%s:" % addr, lap
    sock.send(lap)
  sock.close()
```

Клиент:

```
import socket

HOST = ""                 # удаленный компьютер (localhost)
PORT = 33333              # порт на удаленном компьютере
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST, PORT))
sock.send("ПАЛИНДРОМ")
result = sock.recv(1024)
```

```
sock.close()
print "Получено:", result
```

Примечание:

В примере использованы русские буквы: необходимо указывать кодировку.

Прежде всего, нужно запустить сервер. Сервер открывает сокет на локальной машине на порту 33333, и адресе 127.0.0.1. После этого он слушает (`listen()`) порт. Когда на порту появляются данные, принимается (`accept()`) входящее соединение. Метод `accept()` возвращает пару - Socket-объект и адрес удаленного компьютера, устанавливающего соединение (пара - IP-адрес, порт на удаленной машине). После этого можно применять методы `recv()` и `send()` для общения с клиентом. В `recv()` задается число байтов в очередной порции. От клиента может прийти и меньшее количество данных.

Код программы-клиента достаточно очевиден. Метод `connect()` устанавливает соединение с удаленным хостом (в приведенном примере он расположен на той же машине). Данные передаются методом `send()` и принимаются методом `recv()` - аналогично тому, что происходит на сервере.

Модуль `socket` имеет несколько вспомогательных функций. В частности, функции для работы с системой доменных имен (**DNS**):

```
>>> import socket
>>> socket.gethostbyname('www.onego.ru')
('www.onego.ru', [], ['195.161.136.4'])
>>> socket.gethostbyaddr('195.161.136.4')
('www.onego.ru', [], ['195.161.136.4'])
>>> socket.gethostname()
'rnd.onego.ru'
```

В новых версиях Python появилась такая функция как `socket.getservbyname()`. Она позволяет преобразовывать наименования Интернет-сервисов в общепринятые номера портов:

```
>>> for srv in 'http', 'ftp', 'imap', 'pop3', 'smtp':
...   print socket.getservbyname(srv, 'tcp'), srv
...
80 http
21 ftp
143 imap
110 pop3
25 smtp
```

Модуль также содержит большое количество констант для указания протоколов, типов сокетов, коммуникационных доменов и т.п. Другие функции модуля `socket` можно при необходимости изучить по документации.

Модуль smtplib

Сообщения электронной почты в Интернете передаются от клиента к серверу и между серверами в основном по протоколу **SMTP** (**S** imple **M** ail **T** ransfer **P** rotocol, простой протокол передачи почты). Протокол SMTP и ESMTP (расширенный вариант SMTP) описаны в RFC 821 и RFC 1869. Для работы с SMTP в стандартной библиотеке модулей имеется модуль `smtplib`. Для того чтобы начать SMTP-соединение с сервером электронной почты, необходимо в начале создать объект для управления SMTP-сессией с помощью конструктора класса `SMTP`:

```
smtplib.SMTP([host[, port]])
```

Параметры `host` и `port` задают адрес и порт SMTP-сервера, через который будет отправляться почта. По умолчанию, `port=25`. Если `host` задан, конструктор сам установит соединение, иначе придется отдельно вызывать метод `connect()`. Экземпляры класса `SMTP` имеют методы для всех распространенных команд SMTP-протокола, но для отправки почты достаточно вызова конструктора и методов `sendmail()` и `quit()`:

```
# -*- coding: cp1251 -*-
from smtplib import SMTP
fromaddr = "student@mail.ru"     # От кого
toaddr = "rnd@onego.ru"          # Кому
message = """From: Student <%(fromaddr)s>
To: Lecturer <%(toaddr)s>
Subject: From Python course student
MIME-Version: 1.0
Content-Type: text/plain; charset=Windows-1251
Content-Transfer-Encoding: 8bit

Здравствуйте! Я изучаю курс по языку Python и
отправляю письмо его автору.
"""
connect = SMTP('mail.onego.ru')
connect.set_debuglevel(1)
connect.sendmail(fromaddr, toaddr, message % vars())
connect.quit()
```

Следует заметить, что `toaddr` в сообщении (в поле `To`) и при отправке могут не совпадать. Дело в том, что получатель и отправитель в ходе SMTP-сессии передается командами SMTP-протокола. При запуске указанного выше примера на экране появится отладочная информация (ведь уровень отладки задан равным 1):

```
send: 'ehlo rnd.onego.ru\r\n'
reply:  '250-mail.onego.ru  Hello  as3-042.dialup.onego.ru  [195.161.147.4],
pleased to meet you\r\n'
send: 'mail FROM:<student@mail.ru> size=270\r\n'
reply: '250 2.1.0 <student@mail.ru>... Sender ok\r\n'
send: 'rcpt TO:<rnd@onego.ru>\r\n'
reply: '250 2.1.5 <rnd@onego.ru>... Recipient ok\r\n'
send: 'data\r\n'
reply: '354 Enter mail, end with "." on a line by itself\r\n'
send: 'From: Student <student@mail.ru>\r\n . . . '
reply: '250 2.0.0 iBPFgQ7q028433 Message accepted for delivery\r\n'
send: 'quit\r\n'
reply: '221 2.0.0 mail.onego.ru closing connection\r\n'
```

Из этой (несколько сокращенной) отладочной информации можно увидеть, что клиент отправляет (send) команды SMTP-серверу (EHLO, MAIL FROM, RCPT TO, DATA, QUIT), а тот выполняет команды и отвечает (reply), возвращая код возврата.

В ходе одной SMTP-сессии можно отправить сразу несколько писем подряд, если не вызывать `quit()`.

В принципе, команды SMTP можно подавать и отдельно: для этого у объекта-соединения есть методы (`helo()`, `ehlo()`, `expn()`, `help()`, `mail()`, `rcpt()`, `vrfy()`, `send()`, `noop()`, `data()`), соответствующие одноименным командам SMTP-протокола.

Можно задать и произвольную команду SMTP-серверу с помощью метода `docmd()`. В следующем примере показан простейший сценарий, который могут использовать те, кто время от времени принимает почту на свой сервер по протоколу SMTP от почтового сервера, на котором хранится очередь сообщений для некоторого домена:

```
from smtplib import SMTP
connect = SMTP('mx.abcde.ru')
connect.set_debuglevel(1)
connect.docmd("ETRN rnd.abcde.ru")
connect.quit()
```

Этот простенький сценарий предлагает серверу `mx.abcde.ru` попытаться связаться с основным почтовым сервером домена `rnd.abcde.ru` и переслать всю накопившуюся для него почту.

При работе с классом `smtplib.SMTP` могут возбуждаться различные исключения. Назначение некоторых из них приведено ниже:

`smtplib.SMTPException`
Базовый класс для всех исключений модуля.

`smtplib.SMTPServerDisconnected`
Сервер неожиданно прервал связь (или связь с сервером не была установлена).

`smtplib.SMTPResponseException`
Базовый класс для всех исключений, которые имеют код ответа SMTP-сервера.

`smtplib.SMTPSenderRefused`
Отправитель отвергнут

`smtplib.SMTPRecipientsRefused`
Все получатели отвергнуты сервером.

`smtplib.SMTPDataError`
Сервер ответил неизвестным кодом на данные сообщения.

`smtplib.SMTPConnectError`
Ошибка установления соединения.

`smtplib.SMTPHeloError`
Сервер не ответил правильно на команду `HELO` или отверг ее.

Модуль poplib

Еще один протокол - **POP3** (**P** ost **O** ffice **P** rotocol, почтовый протокол) - служит для приема почты из почтового ящика на сервере (протокол определен в RFC 1725).

Для работы с почтовым сервером требуется установить с ним соединение и, подобно рассмотренному выше примеру, с помощью SMTP-команд получить требуемые сообщения. Объект-соединение POP3 можно установить посредством конструктора класса `POP3` из модуля `poplib`:

```
poplib.POP3(host[, port])
```

Где `host` - адрес POP3-сервера, `port` - порт на сервере (по умолчанию 110), `pop_obj` - объект для управления сеансом работы с POP3-сервером.

Следующий пример демонстрирует основные методы для работы с POP3-соединением:

```
import poplib, email
# Учетные данные пользователя:
SERVER = "pop.server.com"
USERNAME = "user"
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 (если оно небольшое)

# этап обновления
print p.quit()
```

Примечание:

Разумеется, чтобы пример сработал корректно, необходимо внести реальные учетные данные.

При выполнении сценарий выведет на экран примерно следующее.

```
+OK POP3 pop.server.com server ready
+OK User name accepted, password please
+OK Mailbox open, 68 messages
+OK Mailbox scan listing follows
Сообщение 1 имеет длину 4202
UIDL = 4152a47e00000004
* От: online@kaspersky.com
* Кому: user@server.com
* Тема: KL Online Activation

...

+OK Sayonara
```

Эти и другие методы экземпляров класса `POP3` описаны ниже:

Метод	Команда POP3	Описание
`getwelcome()`		Получает строку `s` с приветствием POP3-сервера
`user(name)`	USER name	Посылает команду `USER` с указанием имени пользователя `name`. Возвращает строку с ответом сервера
`pass_(pwd)`	PASS pwd	Отправляет пароль пользователя в команде PASS. После этой команды и до выполнения команды QUIT почтовый ящик блокируется
`apop(user, secret)`	APOP user secret	Идентификация на сервере по APOP
`rpop(user)`	RPOP user	Идентификация по методу RPOP
`stat()`	STAT	Возвращает кортеж с информацией о почтовом ящике. В нем

		m - количество сообщений, l - размер почтового ящика в байтах
list([num])	LIST [num]	Возвращает список сообщений в формате (resp, ['num octets', ...]), если не указан num, и "+OK num octets", если указан. Список lst состоит из строк в формате "num octets".
retr(num)	RETR num	Загружает с сервера сообщение с номером num и возвращает кортеж с ответом сервера (resp, lst, octets)
dele(num)	DELE num	Удаляет сообщение с номером num
rset()	RSET	Отменяет пометки удаления сообщений
noop()	NOOP	Ничего не делает (поддерживает соединение)
quit()	QUIT	Отключение от сервера. Сервер выполняет все необходимые изменения (удаляет сообщения) и снимает блокировку почтового ящика
top(num, lines)	TOP num lines	Команда аналогична RETR, но загружает только заголовок и lines строк тела сообщения. Возвращает кортеж (resp, lst, octets)
uidl([num])	UIDL [num]	Сокращение от "unique-id listing" (список уникальных идентификаторов сообщений). Формат результата: (resp, lst, octets), если num не указан, и "+OK num uniqid", если указан. Список lst состоит из строк вида "+OK num uniqid"

В этой таблице num обозначает номер сообщения (он не меняется на протяжении всей сессии), resp -- ответ сервера, возвращается для любой команды, начинается с "+OK " для успешных операций (при неудаче возбуждается исключение poplib.proto_error). Параметр octets обозначает количество байт в принятых данных. uniqid - идентификатор сообщения, генерируемый сервером.

Работа с POP3-сервером состоит из трех фаз: идентификации, транзакций и обновления. На этапе идентификации сразу после создания POP3-объекта разрешены только команды USER, PASS (иногда APOP и RPOP). После идентификации сервер получает информацию о пользователе и наступает этап транзакций. Здесь уместны остальные команды. Этап обновления вызывается командой QUIT, после которой POP3-сервер обновляет почтовый ящик пользователя в соответствии с поданными командами, а именно - удаляет помеченные для удаления сообщения.

Модули для клиента WWW

Стандартные средства языка Python позволяют получать из программы доступ к объектам WWW как в простых случаях, так и при сложных обстоятельствах, в частности при необходимости передавать данные формы, идентификации, доступа через прокси и т.п.

Стоит отметить, что при работе с WWW используется в основном протокол HTTP, однако WWW охватывает не только HTTP, но и многие другие схемы (FTP, gopher, HTTPS и т.п.). Используемая схема обычно указана в самом начале URL.

Функции для загрузки сетевых объектов

Простой случай получения WWW-объекта по известному URL показан в следующем примере:

```
import urllib
doc = urllib.urlopen("http://python.onego.ru").read()
print doc[:40]
```

Функция `urllib.urlopen()` создает файлоподобный объект, который читает методом `read()`. Другие методы этого объекта: `readline()`, `readlines()`, `fileno()`, `close()` работают как и у обычного файла, а также есть метод `info()`, который возвращает соответствующий полученному с сервера Message-объект. Этот объект можно использовать для получения дополнительной информации:

```
>>> import urllib
>>> f = urllib.urlopen("http://python.onego.ru")
>>> print f.info()
Date: Sat, 25 Dec 2004 19:46:11 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.10
Content-Type: text/html; charset=windows-1251
Content-Length: 4291
>>> print f.info()['Content-Type']
text/html; charset=windows-1251
```

С помощью функции `urllib.urlopen()` можно делать и более сложные вещи, например, передавать web-серверу данные формы. Как известно, данные заполненной web-формы могут быть переданы на web-сервер с использованием метода GET или метода POST. Метод GET связан с кодированием всех передаваемых параметров после знака `"?"` в URL, а при методе POST данные передаются в теле HTTP-запроса. Оба варианта передачи представлены ниже:

```
import urllib
data = {"search": "Python"}
enc_data = urllib.urlencode(data)

# метод GET
f = urllib.urlopen("http://searchengine.com/search" + "?" + enc_data)
print f.read()

# метод POST
f = urllib.urlopen("http://searchengine.com/search", enc_data)
print f.read()
```

В некоторых случаях данные имеют повторяющиеся имена. В этом случае в качестве параметра `urllib.urlencode()` можно использовать вместо словаря последовательность пар имя-значение:

```
>>> import urllib
>>> data = [("n", "1"), ("n", "3"), ("n", "4"), ("button", "Привет"),]
>>> enc_data = urllib.urlencode(data)
>>> print enc_data
n=1&n=3&n=4&button=%F0%D2%C9%D7%C5%D4
```

Модуль `urllib` позволяет загружать web-объекты через прокси-сервер. Если ничего не указывать, будет использоваться прокси-сервер, который был задан принятым в конкретной ОС способом. В Unix прокси-серверы задаются в переменных окружения `http_proxy`, `ftp_proxy` и т.п., в Windows прокси-серверы записаны в реестре, а в Mac OS они берутся из конфигурации Internet. Задать прокси-сервер можно и как именованный параметр `proxies` к `urllib.urlopen()`:

```
# Использовать указанный прокси
proxies = {'http': 'http://www.proxy.com:3128'}
f = urllib.urlopen(some_url, proxies=proxies)
# Не использовать прокси
f = urllib.urlopen(some_url, proxies={})
# Использовать прокси по умолчанию
f = urllib.urlopen(some_url, proxies=None)
f = urllib.urlopen(some_url)
```

Функция `urlretrieve()` позволяет записать заданный URL сетевой объект в файл. Она имеет следующие параметры:

```
urllib.urlretrieve(url[, filename[, reporthook[, data]]])
```

Здесь `url` - URL сетевого объекта, `filename` - имя локального файла для помещения объекта, `reporthook` - функция, которая будет вызываться для сообщения о состоянии загрузки, `data` - данные для метода POST (если он используется). Функция возвращает кортеж `(filepath, headers)` , где `filepath` - имя локального файла, в который закачан объект, `headers` - результат метода `info()` для объекта, возвращенного `urlopen()`.

Для обеспечения интерактивности функция `urllib.urlretrieve()` вызывает время от времени функцию, заданную в `reporthook()`. Этой функции передаются три аргумента: количество принятых блоков, размер блока и общий размер принимаемого объекта в байтах (если он неизвестен, этот параметр равен -1).

В следующем примере программа принимает большой файл и, чтобы пользователь не скучал, пишет процент от выполненной загрузки и предполагаемое оставшееся время:

```
FILE = 'boost-1.31.0-9.src.rpm'
URL = 'http://download.fedora.redhat.com/pub/fedora/linux/core/3/SRPMS/' +
FILE

def download(url, file):
  import urllib, time
  start_t = time.time()

  def progress(bl, blsize, size):
    dldsize = min(bl*blsize, size)
    if size != -1:
      p = float(dldsize) / size
      try:
        elapsed = time.time() - start_t
        est_t = elapsed / p - elapsed
      except:
        est_t = 0
      print "%6.2f %% %6.0f s %6.0f s %6i / %-6i bytes" % (
          p*100, elapsed, est_t, dldsize, size)
    else:
      print "%6i / %-6i bytes" % (dldsize, size)

  urllib.urlretrieve(URL, FILE, progress)

download(URL, FILE)
```

Эта программа выведет примерно следующее (процент от полного объема закачки, прошедшие секунды, предполагаемое оставшееся время, закачанные байты, полное количество байтов):

```
0.00 %      1 s      0 s      0 / 6952309 bytes
0.12 %      5 s   3941 s   8192 / 6952309 bytes
0.24 %      7 s   3132 s  16384 / 6952309 bytes
0.35 %     10 s   2864 s  24576 / 6952309 bytes
0.47 %     12 s   2631 s  32768 / 6952309 bytes
0.59 %     15 s   2570 s  40960 / 6952309 bytes
0.71 %     18 s   2526 s  49152 / 6952309 bytes
0.82 %     20 s   2441 s  57344 / 6952309 bytes
...
```

Функции для анализа URL

Согласно документу RFC 2396 URL должен строиться по следующему шаблону:

```
scheme://netloc/path;parameters?query#fragment
```

где

`scheme`
Адресная схема. Например: http, ftp, gopher.

`netloc`
Местонахождение в сети.

`path`
Путь к ресурсу.

`params`
Параметры.

`query`
Строка запроса.

`fragment`
Идентификатор фрагмента.

Одна из функций уже использовалась для формирования URL - `urllib.urlencode()`. Кроме нее в модуле `urllib` имеются и другие функции:

`quote(s, safe='/')`
Функция экранирует символы в URL, чтобы их можно было отправлять на web-сервер. Она предназначена для экранирования пути к ресурсу, поэтому оставляет `'/'` как есть. Например:

```
>>> urllib.quote("rnd@onego.ru")
'rnd%40onego.ru'
>>> urllib.quote("a = b + c")
'a%20%3D%20b%20%2B%20c'
>>> urllib.quote("0/1/1")
'0/1/1'
>>> urllib.quote("0/1/1", safe="")
'0%2F1%2F1'
```

`quote_plus(s, safe='')`
Функция экранирует некоторые символы в URL (в строке запроса), чтобы их можно было отправлять на web-сервер. Аналогична `quote()`, но заменяет пробелы на плюсы.

`unquote(s)`
Преобразование, обратное `quote()`. Пример:

```
>>> urllib.unquote('a%20%3D%20b%20%2B%20c')
'a = b + c'
```

Преобразование, обратное `quote_plus()`. Пример:

```
>>> urllib.unquote_plus('a+=+b+%2B+c')
'a = b + c'
```

Для анализа URL можно использовать функции из модуля `urlparse`:

```
urlparse(url, scheme='', allow_fragments=1)
```

Разбирает URL в 6 компонентов (сохраняя экранирование символов): `scheme://netloc/path;params?query#frag`

```
urlsplit(url, scheme='', allow_fragments=1)
```

Разбирает URL в 5 компонентов (сохраняя экранирование символов): `scheme://netloc/path?query#frag`

```
urlunparse((scheme, netloc, url, params, query, fragment))
```

Собирает URL из 6 компонентов.

```
urlunsplit((scheme, netloc, url, query, fragment))
```

Собирает URL из 5 компонентов.

Пример:

```
>>> from urlparse import urlsplit, urlunsplit
>>> URL = "http://google.com/search?q=Python"
>>> print urlsplit(URL)
('http', 'google.com', '/search', 'q=Python', '')
>>> print urlunsplit(
...     ('http', 'google.com', '/search', 'q=Python', ''))
http://google.com/search?q=Python
```

Еще одна функция того же модуля `urlparse` позволяет корректно соединить две части URL - базовую и относительную:

```
>>> import urlparse
>>> urlparse.urljoin('http://python.onego.ru', 'itertools.html')
'http://python.onego.ru/itertools.html'
```

Возможности urllib2

Функциональности модулей `urllib` и `urlparse` хватает для большинства задач, которые решают сценарии на Python как web-клиенты. Тем не менее, иногда требуется больше. На этот случай можно использовать модуль для работы с протоколом HTTP - `httplib` - и создать собственный класс для HTTP-запросов (в лекциях модуль `httplib` не рассматривается). Однако вполне вероятно, что нужная функциональность уже имеется в модуле `urllib2`.

Одна из полезных возможностей этих модулей - доступ к web-объектам, требующий авторизации. Ниже будет рассмотрен пример, который не только обеспечит доступ с авторизацией, но и обозначит основную идею модуля `urllib2`: использование обработчиков (handlers), каждый из которых решает узкую специфическую задачу.

Следующий пример показывает, как создать собственный открыватель URL с помощью модуля `urllib2` (этот пример взят из документации по Python):

```
import urllib2

# Подготовка идентификационных данных
authinfo = urllib2.HTTPBasicAuthHandler()
authinfo.add_password('My page', 'localhost', 'user1', 'secret')

# Доступ через прокси
proxy_support = urllib2.ProxyHandler({'http' : 'http://localhost:8080'})

# Создание нового открывателя с указанными обработчиками
opener = urllib2.build_opener(proxy_support,
                              authinfo,
                              urllib2.CacheFTPHandler)
```

```
# Установка поля с названием клиента
opener.addheaders = [('User-agent', 'Mozilla/5.0')]

# Установка нового открывателя по умолчанию
urllib2.install_opener(opener)

# Использование открывателя
f = urllib2.urlopen('http://localhost/mywebdir/')
print f.read()[:100]
```

В этом примере получен доступ к странице, которую охраняет mod_python (см. предыдущую лекцию). Первый аргумент при вызове метода `add_password()` задает область действия (realm) идентификационных данных (он задан директивой `AuthName "My page"` в конфигурации web-сервера). Остальные параметры достаточно понятны: имя хоста, на который нужно получить доступ, имя пользователя и его пароль. Разумеется, для корректной работы примера нужно, чтобы на локальном web-сервере был каталог, требующий авторизации.

В данном примере явным образом затронуты всего три обработчика: `HTTPBasicAuthHandler`, `ProxyHandler` и `CacheFTPHandler`. В модуле `urllib2` их более десятка, назначение каждого можно узнать из документации к используемой версии Python. Есть и специальный класс для управления открывателями: `OpenerDirector`. Именно его экземпляр создала функция `urllib2.build_opener()`.

Модуль `urllib2` имеет и специальный класс для воплощения запроса на открытие URL. Называется этот класс `urllib2.Request`. Его экземпляр содержит состояние запроса. Следующий пример показывает, как получить доступ к каталогу с авторизацией, используя добавление заголовка в HTTP-запрос:

```
import urllib2, base64
req = urllib2.Request('http://localhost/mywebdir')
b64 = base64.encodestring('user1:secret').strip()
req.add_header('Authorization', 'Basic %s' % b64)
req.add_header('User-agent', 'Mozilla/5.0')
f = urllib2.urlopen(req)
print f.read()[:100]
```

Как видно из этого примера, ничего загадочного в авторизации нет: web-клиент вносит (закодированные base64) идентификационные данные в поле `Authorization` HTTP-запроса.

Примечание:

Приведенные два примера почти эквивалентны, только во втором примере прокси-сервер не назначен явно.

XML-RPC сервер

До сих пор высокоуровневые протоколы рассматривались с точки зрения клиента. Не менее просто создавать на Python и их серверные части. Для иллюстрации того, как разработать программу на Python, реализующую сервер, был выбран протокол XML-RPC. Несмотря на свое название, конечному пользователю необязательно знать XML (об этом языке разметки говорилось на одной из предыдущих лекций), так как он скрыт от него. Сокращение **RPC** (**R** emote **P** rocedure **C** all, вызов удаленной процедуры) объясняет суть дела: с помощью XML-RPC можно вызывать процедуры на удаленном хосте. Причем при помощи XML-RPC можно абстрагироваться от конкретного языка программирования за счет использования общепринятых типов данных (строки, числа, логические значения и т.п.). В языке Python вызов удаленной функции по синтаксису ничем не отличается от вызова обычной функции:

```
import xmlrpclib

# Установить соединение
req = xmlrpclib.ServerProxy("http://localhost:8000")

try:
    # Вызвать удаленную функцию
    print req.add(1, 3)
except xmlrpclib.Error, v:
    print "ERROR",
```

А вот как выглядит XML-RPC-сервер (для того чтобы попробовать пример выше, необходимо сначала запустить сервер):

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
srv = SimpleXMLRPCServer(("localhost", 8000))  # Запустить сервер
srv.register_function(pow)  # Зарегистрировать функцию
srv.register_function(lambda x,y: x+y, 'add')  # И еще одну
srv.serve_forever()  # Обслуживать запросы
```

С помощью XML-RPC (а этот протокол достаточно "легковесный" среди других подобных протоколов) приложения могут общаться друг с другом на понятном им языке вызова функций с параметрами основных общепринятых типов и такими же возвращаемыми значениями. Преимуществом же Python является удобный синтаксис вызова удаленных функций.

Внимание!

Разумеется, это только пример. При реальном использовании необходимо позаботиться, чтобы XML-RPC сервер отвечал требованиям безопасности. Кроме того, сервер лучше делать многопоточным, чтобы он мог обрабатывать несколько потоков одновременно. Для многопоточности (она будет обсуждаться в отдельной лекции) не нужно многое переделывать: достаточно определить свой класс, скажем, `ThreadingXMLRPCServer`, в котором вместо `SocketServer.TCPServer` использовать `SocketServer.ThreadingTCPServer`. Это предлагается в качестве упражнения. Наводящий вопрос: где находится определение класса `SimpleXMLRPCServer`?

Заключение

В этой лекции на практических примерах и сведениях из документации были показаны возможности, которые дает стандартный Python для работы в Интернете. Из сценария на Python можно управлять соединением на уровне сокетов, а также использовать модули для конкретного сетевого протокола или набора протоколов. Для работы с сокетами служит модуль `socket`, а модули для высокоуровневых протоколов имеют такие названия как `smtplib`, `poplib`, `httplib` и т.п. Для работы с системой WWW можно использовать модули `urllib`, `urllib2`, `urlparse`. Указанные модули рассмотрены с точки зрения типичного применения. Для решения нестандартных задач лучше обратиться к другим источникам: документации, исходному коду модулей, поиску в Интернете. В этой лекции говорилось и о серверной составляющей высокоуровневых сетевых протоколов. В качестве примера приведена клиент-серверная пара для протокола XML-RPC. Этот протокол создан на основе HTTP, но служит специальной цели.