НИИТ

Лекция №9

Работа с сетью

- Сокеты
- Методы сокетов
- TCP-сервер и TCP-клиент
- UDP-сервер и UDP-клиент
- Многопоточный TCP-сервер
- Протокол HTTP
- Структура HTTP-сообщения
- Создание HTTP-сервера
- urllib
- xmlrpc.server и xmlrpc.client
- cherrypy
- smtplib
- Практика

НИИТ

Разминка

Написать приложение, которое непрерывно выводит на экран введенный пользователем символ. Пользователь может задать новый символ, не останавливая вывод старого символа. При это программа должна мгновенно переключиться на вывод нового символа. Пока ни один символ не введен – программа ничего не выводит, как только введен символ ‘q’ – программа завершается.

python

Сокеты

Стандартный модуль socket обеспечивает доступ к интерфейсу сокетов BSD (Berkeley Software Distribution). Сокеты BSD (или сокеты Беркли) – это интерфейс программирования приложений (API), представляющий собой библиотеку для разработки приложений на языке C с поддержкой межпроцессного взаимодействия (IPC), часто применяемый в компьютерных сетях. Модуль socket можно использовать на современных Unix, Windows и MacOS платформах. Он позволяет создавать клиентские и серверные приложения на низком уровне (сетевом уровне модели ISO/OSI), используя возможности операционной системы, для протоколов с установкой и без установки соединения.

Также Python предоставляет библиотеки для работы с прикладными сетевыми протоколами: FTP, HTTP и т.д.

Методы сокетов

Методы серверного сокета:

Метод	Описание
socket.bind((host, port))	Привязывает адрес (имя хоста, номер порта) к сокету.
socket.listen()	Устанавливает и запускает прослушивание TCP.
socket.accept()	Пассивно принимает TCP-подключение клиента, блокирующе ожидает новые подключения.

Методы клиентского сокета:

Метод	Описание
socket.connect((host, port))	Активно инициирует подключение к TCP серверу.

Общие методы сокетов:

Метод	Описание
socket.recv()	Получает TCP сообщение
socket.send()	Отправляет TCP сообщение
socket.recvfrom()	Получает UDP сообщение
socket.sendto()	Отправляет UDP сообщение
socket.close()	Закрывает сокет
socket.gethostname()	Возвращает имя хоста.

Пример TCP сервера

Для написания TCP сервера мы используем функцию socket из модуля socket, которая создает и возвращает сокет-объект (или просто сокет). Вызывая методы этого объекта, мы настраиваем этот сокет как серверный: с помощью bind((hostname, port)) занимаем порт на указанном хосте под наш сервис, вызывая accept, ожидаем подключения клиентов на указанном ранее порту и получаем connection object для доступа к очередному подключению.

```
import socket

# 1-й параметр - семейство адресов, с которыми будет работать сокет
# AF_INET соответствует адресам IPv4
# 2-й параметр - протокол транспортного уровня
# SOCK_STREAM соответствует протоколу TCP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = '127.0.0.1' # '127.0.0.1' соответствует хосту, на котором запускается скрипт
port = 12345
s.bind((host, port))
s.listen(5) # Открываем порт на сервере (не более 5 клиентов одновременно)
while True:
    conn, addr = s.accept()
    print('Server got connection from {}'.format(addr))
    # Преобразуем строку в набор байтов (ascii в utf-8) и отправляем
    conn.send('Thank you for the connection'.encode())
    conn.close()
# s.close()
```

Пример TCP клиента

Клиентское приложение должно подключаться на заданный порт 12345 заданного хоста. Здесь используется тот же самый модуль socket и аналогичный сокет-объект. Метод socket.connect((hostname, port)) этого объекта открывает TCP подключение к хосту и порту. Как только подключение установится, из него можно читать, как из любого объекта ввода/вывода. После работы с сокетом его нужно закрыть так же, как это делается для файла.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = '127.0.0.1'
port = 12345
s.connect((host, port)) # подключаемся к серверу
d = s.recv(1024)  # получаем данные от сервера (1024 байта - размер буфера для данных)
# преобразуем данные из байтового представления в строковое и выводим
# (преобразование из utf-8 в ascii)
print(d.decode())
s.close()
```

Клиент-серверное взаимодействие (TCP)

Запускаем последовательно скрипт сервера tcp_server.py, скрипт клиента tcp_client.py и смотрим результаты.

Вывод сервера:

```
Server got connection from ('127.0.0.1', 50701)
```

Вывод клиента:

```
Thank you for the connection
```

Пример UDP сервера

Поскольку UDP работает без установки соединения, отличие сервера от клиента заключается только в необходимости занять определенный порт на хосте с помощью bind((hostname, port)), на который будут приходить сообщения от клиентов.

```
import socket

# AF_INET соответствует адресам IPv4
# SOCK_DGRAM соответствует протоколу UDP
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
host = '127.0.0.1'
port = 12345
s.bind((host, port))
while True:
    data, addr = s.recvfrom(1024) # размер буфера для данных – 1024 байта
    print('Server got data from client: {}'.format(data.decode()))
    s.sendto('Thank you for the data'.encode(), addr)
# s.close()
```

Пример UDP клиента

Клиентское приложение отправляет и получает данные, зная адрес хоста и порт, на котором ожидает данных сервер. Соединение в данном случае не устанавливается: если отправленные данные не дойдут до сервера, клиент об этом не узнает. То же самое и с ответом сервера: если он не дойдет до клиента, сервер не будет об этом знать.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
host = '127.0.0.1'
port = 12345
# отправляем сообщение серверу без установки соединения
s.sendto('client data'.encode(), (host, port))
# получаем ответ от сервера
data, addr = s.recvfrom(1024)
print(data.decode())
s.close()
```

Клиент-серверное взаимодействие (UDP)

Запускаем последовательно скрипт сервера udp_server.py, скрипт клиента udp_client.py и смотрим результаты.

Вывод сервера:

```
Server got data from client: client data
```

Вывод клиента:

```
Thank you for the data
```

TCP сервер с многопоточной обработкой запросов

Для улучшения производительности обработки сетевых подключений на сервере имеет смысл использовать пул потоков, выделяя под работу с каждым подключением отдельный поток.

```
import threading
import socket

class ClientThread(threading.Thread):
    def __init__(self, conn, addr):
        super().__init__()
        self._connection = conn
        self._address = addr

    def run(self):
        print('Connection from address {}'.format(self._address))
        data = self._connection.recv(1024)
        print('Received {}'.format(data.decode()))
        self._connection.send(data)
        self._connection.close()
        print('Closed connection from {}'.format(self._address))
```

TCP сервер с многопоточной обработкой запросов

```
class TcpServer:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self._socket = None
        self._runnning = False

    def run(self):
        self._socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self._socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self._socket.bind((self.host, self.port))
        self._socket.listen(5)
        self._runnning = True
        print('Server is up')
        while self._runnning:
            conn, addr = self._socket.accept()
            ClientThread(conn, addr).start()

    def stop(self):
        self._runnning = False
        self._socket.close()
        print('Server is down')

if __name__ == '__main__':
    srv = TcpServer(host='127.0.0.1', port=5555)
    try:
        srv.run()
    except KeyboardInterrupt:
        srv.stop()
```

TCP клиент

```
import socket
import random

class TcpClient:
    def __init__(self, host, port, name):
        self.host = host
        self.port = port
        self.name = name
        self._socket = None

    def run(self):
        self._socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self._socket.connect((self.host, self.port))
        self._socket.send(self.name.encode())
        data = self._socket.recv(1024)
        print('Received: {}'.format(data.decode()))
        self._socket.close()

if __name__ == '__main__':
    name = 'Python client ' + str(random.randint(1, 1000))
    myclient = TcpClient(host='127.0.0.1', port=5555, name=name)
    myclient.run()
```

HTTP

Переходим к организации сетевого взаимодействия на прикладном уровне. Один из самых широко используемых протоколов этого уровня - HTTP - HyperText Transfer Protocol (протокол передачи гипертекста). HTTP используется как для получения информации с веб-сайтов, так и в качестве транспорта для других протоколов (SOAP, XML-RPC). Обмен сообщениями идет по схеме “запрос-ответ”в соответствии с уже знакомой нам технологией клиент-сервер. Для идентификации ресурсов HTTP использует глобальные URI (Uniform Resource Identifier). Браузеры, с помощью которых пользователи обращаются к сетевым ресурсам, содержат в себе реализации HTTP-клиентов.

HTTP-сообщение

Каждое HTTP-сообщение состоит из трёх частей, которые передаются в указанном порядке:

- стартовая строка (starting line) — определяет тип сообщения;
- заголовки (headers) — характеризуют тело сообщения, параметры передачи и прочие сведения;
- тело сообщения (message body) — непосредственно данные сообщения (обязательно должно отделяться от заголовков пустой строкой).

Пример запроса:
GET /wiki/HTTP HTTP/1.0
Host: ru.wikipedia.org

Пример ответа сервера:
HTTP/1.0 200 OK

HTTP-сообщение

Стартовая строка запроса клиента:
Метод URI HTTP/Версия

Метод — тип запроса, одно слово заглавными буквами.
URI определяет путь к запрашиваемому документу.
Версия — пара разделённых точкой цифр. Например: 1.0.

Стартовая строка ответа сервера:
HTTP/Версия КодСостояния Пояснение

Версия — пара разделённых точкой цифр, как в запросе;
Код состояния — три цифры. По коду состояния определяется дальнейшее содержимое сообщения и поведение клиента;
Пояснение — текстовое короткое пояснение к коду ответа для пользователя. Никак не влияет на сообщение и является необязательным.

HTTP-сервер средствами Python

Простейший HTTP-сервер на своем хосте можно организовать, просто запустив на хосте соответствующий модуль Python и номер свободного порта.

Для Linux это будет SimpleHTTPServer:

python –m SimpleHTTPServer 8888

Для Windows – http.server:

python –m http.server 8888

После этого в браузере можно набрать http://127.0.0.1:8888/ и изучать содержимое папки, в которой запущен сервер, непосредственно через браузер в формате гипертекста.

Свой HTTP-сервер

Также можно написать свой HTTP-сервер, используя модуль BaseHTTPServer (Linux) или http.server (Windows).

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class MyHandler(BaseHTTPRequestHandler):
    # обработчик GET запросов
    def do_GET(self):
        self.send_response(200) # OK
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        # собственно html сообщение
        self.wfile.write('Hello World!'.encode())

if __name__ == '__main__':
    port = 8080
    server = HTTPServer(('127.0.0.1', port), MyHandler)
    print('Started HTTP server on port: {}'.format(port))
    # бесконечно ожидаем входящие http запросы
    server.serve_forever()
```

После запуска скрипта в браузере можно набрать http://127.0.0.1:8080/ и увидеть тот самый ‘Hello World!’, отправляемый в do_GET.

urllib

Для чтения веб-страниц в скрипте Python для последующей обработки, используется модуль urllib.

```
from urllib import request

req = request.Request('http://google.com')
response = request.urlopen(req)
web_page = response.read()
print(web_page)
```

xmlrpc

XML-RPC – стандарт/протокол вызова удаленных процедур, использующий XML для кодирования своих сообщений и HTTP в качестве транспортного механизма.

XML (Extensible Markup Language) – язык разметки соответствующих документов.

RPC (Remote Procedure Call) – удаленный вызов процедур – класс технологий для запуска процедур на удаленных хостах.

Модуль xmlrpc.server (в Python2 – xmlrpclib) содержит классы для создания собственного кроссплатформенного сервера, работающего по протоколу XML-RPC.

Модуль xmlrpc.client (в Python2 – та же xmlrpclib) позволяет написать программу-клиент для взаимодействия с XML-RPC сервером.

xmlrpc.server

Простой пример сервера, предоставляющего единственную функцию для удаленного вызова. Функция принимает число и возвращает True если оно четное и False в противном случае. Сначала создаем экземпляр SimpleXMLRPCServer и задаем адрес хоста и порт, который он будет прослушивать. Затем регистрируем функцию для удаленного вызова, чтоб сервер знал, как ее вызывать. И наконец, запускаем сервер в бесконечном цикле ожидания и обработки запросов.

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

if __name__ == '__main__':
    server = SimpleXMLRPCServer(("localhost", 8000))
    print( "Listening on port 8000...")
    server.register_function(is_even, "is_even")
    server.serve_forever()
```

xmlrpc.client

Сервер будет доступен по URL http://localhost:8000 через объект класса ServerProxy модуля xmlrpc.client. К процедурам сервера можно обращаться как к методам этого объекта.

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
print( "3 is even: %s" % str(proxy.is_even(3)))
print("100 is even: %s" % str(proxy.is_even(100)))
```

Обращения к методам объекта proxy транслируется внутри xmplrpc.client в запросы к серверу на языке XML через HTTP-метод POST. Сервер выполняет парсинг XML-структуры и определяет, какая function должна быть вызвана, исходя из имени функции, указанной клиентом. Аргументы также извлекаются из XML-структуры и передаются в функцию. Возвращаемое значение функции так же транслируется в XML и передается обратно клиенту.

cherrypy

Для создания собственного веб-сервера (не только для работы по HTTP, но и для запуска веб-приложений можно воспользоваться веб-фреймворком cherrypy, установив соответствующую библиотеку с помощью pip).

```
import cherrypy

class HelloWorld:
    @cherrypy.expose
    def index(self):
        return 'HelloWorld from cherrypy!'

if __name__ == '__main__':
    cherrypy.config.update({'server.socket_port': 8099})
    cherrypy.quickstart(HelloWorld())
```

Cherrypy не занимается такими задачами, как обработка шаблонов для вывода данных, доступ к базе данных, авторизация пользователя. Как правило, этот фреймворк используется для организации доступа к разделяемым ресурсам по внутренней сети компании.

Более сложным и функциональным фреймворком является Django, который будет рассмотрен отдельно.

smtplib

Для отправки сообщений через сервер электронной почты по протоколу SMTP используется библиотека smtplib. Также при формировании сообщения полезна библиотека email.

```
def main():
    sender = 'orlov@gmail.com'
    targets = ['orlov@mera.ru']

    msg_text = 'Hello!'
    msg = MIMEText(msg_text)
    msg['Subject'] = 'Simple subject'
    msg['From'] = sender
    msg['To'] = ', '.join(targets)

    server = SMTP_SSL('smtp.gmail.com', 465)
    server.login('username', 'password')
    server.sendmail(sender, targets, msg.as_string())
    server.quit()

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

Практика

1. Написать клиентское и серверное приложения. Клиент отправляет на сервер список зашифрованных слов, сервер дешифрует слова по словарю и возвращает клиенту список расшифрованных слов. Клиент должен вывести полученный список.
2. * Написать клиентское и серверное приложения. Клиент при установке соединения отправляет на сервер информацию о пользователе (имя, возраст), хранимую в атрибутах объекта класса User. Сервер должен выводить информацию о подключенных пользователях. Клиентское приложение должно быть запущено несколько раз с различными пользователями.
3. * Используя модуль urllib, соберите все ссылки на заданной веб-странице (http://google.com) и проверьте их работоспособность.