Работа с сетью. Сокеты

Цель занятия

После освоения темы вы:

- узнаете термины, которые используются в работе с сетью, и основные сетевые протоколы;
- сможете использовать основные функции, методы, модули и библиотеки Python для написания собственных клиентских и серверных приложений.

План занятия

1. Сетевые протоколы
2. Сокеты, программа клиент-сервер
3. Тайм-ауты и обработка сетевых ошибок
4. Одновременная обработка нескольких соединений
5. Простой HTTP-сервер
6. Основные библиотеки для работы HTTP

Используемые термины

Протокол — набор определенных правил или соглашений, который определяет обмен данными между устройствами или программами.

Сокет — конструкция, состоящая из IP-адреса и номера порта.

Конспект занятия

1. Сетевые протоколы

Протокол связи — набор правил, который позволяет двум или более устройствам осуществлять соединение и обмен информацией посредством какого-либо физического сигнала.

Модель OSI (Open System Interconnection model) разделяет все протоколы на 7 уровней. Смысл такого разделения — построить слоеную модель абстракции, чтобы перейти от задачи прикладного уровня (например, стриминг-видео) к физическому уровню (видео нужно каким-то образом передавать с помощью физического сигнала, например, электрического тока в проводах или радиоволны). Разрыв между этими задачами слишком высок, чтобы решать задачу передачи видео сразу в терминах электрических сигналов или радиоволн.

Таблица 1. Структура модели OSI.

Уровень		Тип данных (PDU)	Функции	Примеры
7	Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, POP3, SMTP, WebSocket
6	Представления (presentation)		Представление и шифрование данных	ASCII, EBCDIC, JPEG, MIDI
5	Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP, gRPC
4	Транспортный (transport)	Сегменты	Прямая связь между конечными пунктами и надежность	TCP, UDP, SCTP, Порты
3	Сетевой (network)	Пакеты	Определение маршрута и логическая адресация	IPv4, IPv6, Ipsec, ICMP
2	Канальный (data link)	Биты / кадры	Физическая адресация	PPP, IEEE 802.22, Ethernet, DSL, ARP
1	Физический (physical)	Биты	Работа со средой передачи, сигналами и двоичными данными	«Витая пара», коаксиальный, оптоволоконный, радиоканал

На самом нижнем уровне модели OSI находятся протоколы физического уровня — стандарты. Они описывают, как должен быть устроен провод, по которому идет сигнал, либо параметры радиоволны — частоту, мощность. Нам, как программистам, этот уровень не интересен.

Поверх физического уровня выделяется канальный уровень — здесь мы не просто имеем дело с физическим сигналом. Мы понимаем, что в сети есть узлы, и они должны взаимодействовать. К этому уровню относят протокол Ethernet — обычный провод, который идет от компьютера к роутеру.

Сетевой уровень отвечает за определение маршрута, по которому идет сигнал.

После этого следуют еще несколько уровней, которые не всегда все представлены.

На практике могут быть нужны только некоторые из этих протоколов описанной модели. Связано это с тем, что модель OSI охватывает все многообразие компьютерных сетей, в том числе Bluetooth-соединения, специфические протоколы обмена данными датчиков и контроллеров.

Далее перейдем к более простой модели и рассмотрим **стек протоколов TCP/IP** — некое подмножество рассмотренных ранее протоколов. Это те протоколы, которые используются в Интернете.

Таблица 2. Стек протоколов TCP/IP.

Прикладной уровень (Application layer)	HTTP, FTP, DNS, SMTP, SSH…
Транспортный уровень (Transport Layer)	TCP, UDP, SCTP, DCCP…
Сетевой / Межсетевой уровень (Network Layer)	IP
Уровень сетевого доступа / Канальный (Link Layer)	Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring…

В этой модели количество уровней сокращено до 4. Остаются:

- канальный уровень — он нужен для того, чтобы передавать физический сигнал;

- сетевой уровень — фактически представлен протоколом IP и некоторыми его разновидностями;
- транспортный уровень — основные протоколы TCP и UDP;
- прикладной уровень — представлен множеством протоколов.

IP (Internet Protocol) — протокол, благодаря которому небольшие локальные сети удалось объединить в глобальную сеть Интернет. Наиболее важная функция протокола IP — адресация.

Существуют две версии протокола IP: IPv4 (появилась в 1981) и IPv6 (появилась в 1996). Несмотря на то что версия IPv6 появилась достаточно давно, по информации Google, на ноябрь 2022 года протокол IPv6 поддерживает около 40% устройств.

Это связано с тем, что сфера сетевых технологий является достаточно консервативной, и внедрение нового протокола затруднительно. Используется большое количество различных устройств во всем мире, которые работают по старым протоколам. Поэтому мы должны учитывать, что есть ряд пользователей и устройств, которые работают по старым протоколам, и должны их поддерживать.

При внедрении протокола IPv6 планировалось, что переход к новой версии будет плавным, и что будут одновременно существовать и IPv4, и IPv6.

Основная проблема IPv4 — недостаток адресов. Адрес протокола IPv4 — 4 байта (32 бита), что дает 4294967296 адресов. Адрес обычно записывается в виде 4 чисел от 0 до 255, разделенных точкой, например: 213.180.204.242. На сегодняшний день все региональные интернет-регистраторы уже исчерпали свои наборы адресов. Поэтому появилась потребность перехода к следующей версии.

Адрес IPv6 — 16 байт (128 бит), что дает порядка 10^{38} адресов.

Адрес обычно записывается в шестнадцатеричном виде группами по 4 цифры (2 байта), например: fe80:0000:0000:0000:0200:f8ff:fe21:67cf.

Старшие нули в группе могут быть опущены: fe80:0:0:0:200:f8ff:fe21:67cf.

Большое количество нулевых групп также может быть опущено: fe80::200:f8ff:fe21:67cf.

Адреса бывают **глобальными** и **локальными**. Чтобы был возможен доступ к серверу из сети, он должен иметь глобальный адрес. Обычный пользовательский компьютер, как правило, не имеет глобального адреса.

Локальный адрес делится на адрес сети и адрес узла. Количество бит, выделенное на адрес сети, указывается через «/». Рассмотрим пример:

- 10.0.0.0/8: первые 8 бит (число 10) — адрес сети;
- остальные 24 бита (3 числа) — адрес узла.

Есть некоторые наборы IP-адресов, выделенные для использования в локальных сетях:

- 10.0.0.0/8, 172.16.0.0/12 и 192.168.0.0/16 — используются только в локальных сетях;
- 0.0.0.0/8 — «эта» сеть (из которой идет запрос);
- 0.0.0.0 — «этот» компьютер;
- 127.0.0.0/8 — адреса, используемые «внутри» одного компьютера (запрос обрабатывается операционной системой без участия сетевой карты);
- 127.0.0.1 — адрес «этого» компьютера (без обращения к сетевой карте).

По протоколу IP данные передаются пакетами размером до 64 KB. К каждому пакету протокол дописывает заголовок, который показывает, откуда и куда идет пакет, а также заголовок содержит некоторые дополнительные данные. Размер заголовка в IPv4 — обычно 20 байт, в IPv6 — 40 байт.

На данном этапе необязательно понимать, как устроены заголовки.

Принципиально важно деление данных на пакеты — это позволит в дальнейшем понять, как работают сетевые соединения. Если бы не существовало деление данных на пакеты, то было бы невозможно обрабатывать несколько соединений сразу. Например, при загрузке большого файла нет необходимости отключать все остальные соединения. Также это необходимо для большей надёжности соединений, — например, когда происходит потеря пакета. В этом случае необходимо передать лишь потерянный пакет.

Протокол IP не описывает, какие именно данные передаются. Он описывает, как найти маршрут от одного узла к другому, и как передать пакет.

Поверх протокола IP существуют протоколы транспортного уровня, которые непосредственно занимаются передачей конкретных данных.

Рассмотрим протоколы TCP и UDP. Поддержка этих протоколов обычно «зашита» в ядро операционной системы. Важные понятия, используемые в этих протоколах:

- **порт** — специальный номер, выданный операционной системой и связанный с определенным процессом, осуществляющим сетевое соединение. При получении пакета данных операционная система по номеру порта (входит в заголовки протоколов TCP и UDP) определяет, какому процессу этот пакет предназначен.
- **сокет** — пара ip-адрес и порт.

Рассмотрим отличие протоколов TCP и UDP.

TCP (Transmission Control Protocol) — основной протокол, поверх которого реализуется большая часть протоколов прикладного уровня. Он обеспечивает контроль корректной передачи данных. Протокол предназначен обеспечить надежность передачи данных и содержит хеш-суммы передаваемых данных, механизмы подтверждения полученных данных.

UDP (User Datagram Protocol) — более простой и быстрый протокол, не гарантирующий корректное получение данных.

На практике чаще используется протокол TCP. Протокол UDP используется, когда на первое место выходит скорость соединения.

Протоколы прикладного уровня, как правило, строятся поверх протокола TCP. Чаще всего эти протоколы описывают формат передаваемых данных. Протоколы прикладного уровня абстрагируются от того, как и куда будут отправлены данные.

Примеры протоколов прикладного уровня:

- HTTP — Hyper-text transfer protocol;
- HTTPS — HTTP secure;
- FTP — File transfer protocol;
- SMTP — Simple mail transfer protocol.

На самом деле существует большее количество протоколов. Помимо широко используемых протоколов, существуют и другие малоизвестные. Производители сетевых устройств также могут описывать свои протоколы.

2. Сокеты, программа клиент-сервер

В стандартной библиотеке Python есть модуль `socket`, который предоставляет низкоуровневые функции для работы с сетевыми соединениями. Интерфейс модуля практически буквально копирует интерфейс системных вызовов Unix.

Основной класс — класс `socket`. При создании класс принимает набор параметров: два протокола (канального и транспортного уровня), номер протокола, номер файла (Unix при создании сетевого соединения выделяет файловый дескриптор — некоторое число):

```
class socket.socket(
    family=AF_INET,   # IPv4
    type=SOCK_STREAM, # TCP
    proto=0,     # Номер протокола,
                 # не имеет значения для TCP/IP
    fileno=None # Можно создать объект socket
                 # из файлового дескриптора
)
```

Чаще всего мы будем вызывать конструктор класса `socket` без параметров, так как те параметры, которые установлены по умолчанию, нам подходят.

Рассмотрим пример echo-сервера, который получает сообщение и отвечает этим же сообщением в ответ:

```
import socket

server = socket.socket()
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, True)
server.bind(('127.0.0.1', 5000))
server.listen()

try:
    while True:
        client, addr = server.accept()
        print(client, addr)
        request = client.recv(4096).decode('utf-8')
        print(request)
        client.send(request.encode('utf-8'))
        client.close()
finally:
    server.close()
```

В программе мы создаем сокет. Для того чтобы создать сервер, нужно сделать несколько вызовов: открыть соединение, привязать соединение к адресу с помощью функции `bind`, запустить сервер с помощью функции `listen`. Номер порта выбран произвольно, но лучше указывать порт больше 1000, так как порты с небольшими номерами могут быть зарезервированы. Строка `server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, True)` в программе взята из документации Unix и необходима для того, чтобы мы могли использовать порт заново сразу же, если мы запустим программу еще раз. Эта строка будет полезна при отладке программы.

Блок `try-finally` обеспечивает то, что сокет будет закрыт. Если закрытие не будет выполнено, далее скорее всего мы не сможем использовать сокет даже с учетом выполненных ранее настроек.

В цикле `while` мы ожидаем входящего подключения с помощью метода `accept`. Метод `accept` «подвиснет» и будет ждать, пока кто-нибудь подключится. Когда будет соединение, метод выдаст еще один объект сокет, который будет клиентом, и его адрес. И программа это распечатает.

С помощью метода `recv` (сокращенно от recieve) мы получаем данные, в скобках указаны параметры буфера: 4096 байт = 4 KB. Размер буфера должен быть степенью числа 2, в примере 4096 = 2^{12}. Получаемый набор байт необходимо декодировать. Мы считаем, что это текст в кодировке UTF-8, и с помощью метода `decode` переводим его в строку.

Рассмотренный код — опасный. Мы можем получить данные, не являющиеся строкой в UTF-8, что приведет к ошибке в программе. В рассмотренном примере дополнительные случаи обработки ошибок опущены, и показана лишь основная логика работы программы.

Далее выполняется отправка данных клиенту с помощью метода `send`. Метод `send` отправляет порцию данных и возвращает в качестве результата, сколько он смог отправить. То есть для корректной работы программы нужно контролировать отправку. Существует также метод `sendall`, который отправляет все сообщение. Если сообщение не поместилось в один пакет, происходит отправка нескольких пакетов. При этом снижается контроль, сколько пакетов было отправлено.

Отправив данные, мы закрываем соединение.

Более лаконичная версия рассмотренной программы:

```
import socket

with socket.create_server(
            ('127.0.0.1', 5000),
            reuse_port=True) as server:
    while True:
        client, addr = server.accept()
        print(client, addr)
        request = client.recv(4096).decode('utf-8')
        print(request)
        client.send(request.encode('utf-8'))
        client.close()
```

Функция `create_server` выполняет все действия по созданию сервера: создает сокет, привязывает его к адресу и порту.

Важно! IP-адрес указывается как строка в кавычках. IP-адрес и порт задаются как кортеж.

Параметр `reuse_port` делает настройки такими же, как и в предыдущем примере (см. описание строки `server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, True)`).

Далее логика программы аналогична рассмотренной ранее. Полученный объект `socket` — контекстный менеджер, и мы можем использовать его в блоке `with`, гарантируя, что объект будет закрыт.

Рассмотрим теперь программу для клиента:

```
import socket

with socket.create_connection(
             ('127.0.0.1', 5000)) as s:
    s.send('Привет!'.encode('utf-8'))
    answer = s.recv(4096).decode('utf-8')
    print(answer)
```

Клиентское приложение будет соединяться с сервером. Воспользуемся лаконичной функцией `create_connection`, которая создает клиентское соединение. Программа отправляет сообщение на сервер, затем получает ответ и его распечатывает.

Модуль `socket` содержит вспомогательные функции. Функция `getaddrinfo` может получить информацию об адресе. Адрес указывается текстом, порт указывается числом после адреса:

```
>>> socket.getaddrinfo('yandex.ru', 80)
[(<AddressFamily.AF_INET: 2>, 0, 0, '', ('5.255.255.70', 80)),
 ...
 (<AddressFamily.AF_INET: 2>, 0, 0, '', ('77.88.55.88', 80))]
# 80 - стандартный порт для протокола HTTP

>>> socket.getaddrinfo('yandex.ru', 443)
[(<AddressFamily.AF_INET: 2>, 0, 0, '', ('5.255.255.70', 443)),
 ...
 (<AddressFamily.AF_INET: 2>, 0, 0, '', ('77.88.55.88', 443))]
# 443 - стандартный порт для протокола HTTPS
```

Функция `gethostbyaddr` выполняет обратное преобразование:

```
>>> socket.gethostbyaddr('5.255.255.70')
('yandex.ru', [], ['5.255.255.70'])
```

Стандартная библиотека Python содержит более удобный модуль для создания TCP-серверов — модуль `socketserver`, который содержит описание базовых классов. Для создания сервера нам нужно описать класс, отвечающий за обработку входящих сообщений. Этот класс должен наследоваться от `BaseRequestHandler` и должен содержать метод `handle`. Метод должен обработать запрос, который хранится в атрибуте `self.request` — сокет присоединившегося клиента.

Код в классе ниже также демонстрирует echo-сервер. Отличие от ранее рассмотренных программ в том, что ответное сообщение выдается в верхнем регистре.

После описания класса мы создаем сервер с помощью класса `TCPServer`, которому передаем IP-адрес и порт, а также класс обработчика запроса `MyTCPHandler`. У сервера запускается единственный метод `serve_forever`. После этого сервер будет принимать запросы до тех пор, пока мы его не остановим.

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        self.data =
self.request.recv(1024).strip().decode('utf-8')
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        self.request.send(self.data.upper().encode('utf-8'))

if __name__ == "__main__":
    HOST, PORT = "localhost", 5000
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as
server:
        server.serve_forever()
```

3. Тайм-ауты и обработка сетевых ошибок

Сетевые подключения могут генерировать множество различных ошибок. В любой серьезной программе, работающей с сетью, обработке ошибок следует уделить достаточно много внимания. Далее рассмотрим примеры ошибок, которые могут возникать.

Ошибка разрешения адреса — обращение к адресу, которого не существует:

```
>>> import socket
>>> s = socket.socket()
>>> s.connect(('bad adress', 0))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
socket.gaierror: [Errno -2] Name or service not known
# gaierror - get_addr_info_error
```

Попытка использовать занятый сокет — то есть использовать тот же самый номер порта:

```
>>> s
<socket.socket ... laddr=('127.0.0.1', 32852)>

>>> s2 = socket.socket()
>>> s2.bind(('127.0.0.1', 32852))
Traceback (most recent call last):
  ...
OSError: [Errno 98] Address already in use
```

Тайм-аут. Иногда требуется самим сгенерировать ошибку, — например, когда сообщение очень долго «висит». По умолчанию тайм-аута на соединениях нет:

```
>>> s2.settimeout(3.0)
>>> s2.connect(('yandex.ru', 80))
>>> s2
<socket.socket ... laddr=('192.169.3.8', 55768),
raddr=('5.255.255.80', 80)>
```

Далее мы отправили начало HTTP-запроса, но сам запрос не закончен. Сервер ждет, когда мы продолжим передачу данных дальше. Но вместо этого получаем ошибку тайм-аута, которую установили ранее:

```
>>> s2.sendall(b'GET / HTTP/1.1')
>>> s2.recv(4096)
Traceback (most recent call last):
  ...
TimeoutError: timed out
```

Попытка нескольких подключений с одним сокетом — используем один и тот же сокет для нескольких соединений.

В примере ниже мы сначала используем порт 80. А затем, используя этот же сокет, пытаемся подключиться к порту 81 — получаем сообщение об ошибке `Connection refused`. При этом полученная ошибка относится к предыдущему соединению. Первое соединение закончилось тайм-аутом, и мы не дождались ответа:

```
>>> s.settimeout(2.0)
>>> s.connect(('127.0.0.1', 80))
Traceback (most recent call last):
  ...
TimeoutError: [Errno 110] Connection timed out

>>> s.connect(('127.0.0.1', 81))
Traceback (most recent call last):
```

```
  ...
ConnectionRefusedError: [Errno 111] Connection refused
```

Если мы попытаемся еще раз подключиться, то получим ошибку `ConnectionAbortedError`, которая говорит, что некоторое программное обеспечение прекратило соединение. В данном случае имеется в виду программное обеспечение внутри самой операционной системы:

```
>>> s.connect(('127.0.0.1', 82))
Traceback (most recent call last):
  ...
ConnectionAbortedError: [Errno 103] Software caused connection
abort
```

Не всегда смысл выдаваемых ошибок может быть понятен. Но рассмотренный сценарий иногда встречается, и полезно о нем знать.

Соединение через несуществующий сокет. В примере выполняется попытка обращения к уже закрытому соединению. `Bad file descriptor` в сообщении означает, что запрос к операционной системе шел с номером файлового дескриптора, который неизвестен операционной системе:

```
>>> s.close()
>>> s.connect(('127.0.0.1', 82))
Traceback (most recent call last):
  ...
OSError: [Errno 9] Bad file descriptor
```

При работе с сетью может происходить много разных ошибок. Несколько важных моментов:

- все исключения в `socket` являются подтипами `OSError`;
- обработку исключений нужно начинать с более специфических.

Разберем пример последовательности обработки ошибок. Сначала обрабатываются более специфические ошибки, так как при наличии нескольких блоков `except` сработает первый подходящий из них:

```
s.settimeout(3.0)
try:
    s.connect((host, port))
    s.sendall(data)
    answer = s.recv(4096)
except socket.gaierror:
    print('Wrong adress')
except TimeoutError:
    print('Connection timed out')
except OSError as e:
```

```
    # Все остальные ошибки
    print('Connection error:', e.errno)
```

4. Одновременная обработка нескольких соединений

Необходимость обработки нескольких соединений может возникать по следующим причинам:

1. Сетевые соединения медленные. Если сервер будет обрабатывать только одно соединение за раз, сервер будет крайне неэффективным. Проведем аналогию с компьютерными технологиями. Если представить, что один такт процессора — это одна секунда, то стандартная загрузка данных из Интернета может занять до 7 лет.
2. Создание нового соединения занимает время, поэтому лучше соединение не разрывать.

Рассмотрим пример клиента, который не разрывает соединение:

```
import socket

with socket.create_connection(('127.0.0.1', 5000)) as sock:
    while True:
        msg = input('[MESSAGE:] ')
        if not msg:
            break
        sock.sendall(msg.encode('utf-8'))
        answer = sock.recv(4096).decode('utf-8')
        print(answer)
```

В программе мы в бесконечном цикле запрашиваем от пользователя сообщение, отправляем его на сервер и получаем ответ. Соединение не разрывается, программа ожидает получения следующего сообщения.

Сервер должен также не разрывать соединение, а обрабатывать его, пока клиент не отключится:

```
def accept_connection(server_socket):
    client_socket, addr = server_socket.accept()
    print(f'Connection from {addr}.')
    try:
        while True:
            try:
                data = client_socket.recv(4096)
            except OSError as e:
                print(f'connection from {addr} caused an error:',
e)
                return
            if not data:
```

```
                return
            print(f'{addr}: {data.decode("utf-8").strip()}')
            client_socket.sendall(data.upper())
    finally:
        client_socket.close()
        print(addr, 'closed')

with socket.create_server(('127.0.0.1', 5000)) as server:
    while True:
        accept_connection(server)
```

С помощью метода `accept` программа получает сокет клиента, далее в бесконечном цикле мы считываем данные от клиента. В программе написана обработка ошибки: клиент может неожиданно отключиться, и мы ничего не получим. Если клиент отключился нормальным способом, метод `recv` вернет пустую строку. Этот случай тоже нужно обработать.

Далее мы распечатываем сообщение и отправляем его, преобразовав все символы в верхний регистр.

После запуска сервера мы должны запустить функцию `accept_connection` на каждое следующее соединение также в бесконечном цикле.

Проблема состоит в следующем: если подключается второй клиент, он должен ждать, пока не отключится первый. И это может быть достаточно долго.

Возможные решения проблемы:

1. Поддерживать пул соединений и обрабатывать по мере поступления данных.
2. Обрабатывать соединения в нескольких потоках.
3. Использовать асинхронный фреймворк.

На практике чаще используются решения 1 и 2.

Разделим логику на две части. Во-первых, мы должны принять соединение от клиента и сохранить клиентский сокет. Поэтому мы напишем соответствующую функцию:

```
def accept_connection(server_socket):
    client_socket, addr = server_socket.accept()
    print(f'Connection from {addr}.')
    return client_socket
```

Во-вторых, мы должны считать сообщение и его обработать:

```
def proceed_message(client_socket):
    addr = client_socket.getpeername()
```

```
    try:
        data = client_socket.recv(4096)
    except OSError:
        return False
    if not data:
        print(f'{addr} left us')
        return False

    print(f'{addr}: {data.decode("utf-8").strip()}')
    client_socket.sendall(data.upper())
    return True
```

Нового кода здесь не появилось, было только выполнено разделение на две функции. Отличие в функции `proceed_message`: мы получаем имя клиента с помощью функции `getpeername` (первая строка в функции). Сервер может обрабатывать несколько соединений, и мы должны знать, от какого именно клиента пришло сообщение.

Но разбиение кода на несколько функций не решает проблему. Проблема в том, что методы `server_socket.accept` и `client_socket.recv` — блокирующие. Так же как функция `input`, они приостанавливают выполнение программы, пока не появятся данные. Поэтому возникает вопрос: а можно ли вызывать их только в тот момент, когда уже точно известно, что данные есть?

С функцией `input` задача реализации неблокирующего ввода данных будет нетривиальной. К сетевым соединениям такой подход вполне применим — есть готовый инструмент, который позволяет это сделать.

Нам понадобится функция `select` из модуля `select`. В Unix-системах эта функция может работать с любыми потоками, в том числе с потоком стандартного ввода, то есть с ее помощью можно сделать неблокирующей функцию `input`. В Windows функция работает только с сетевыми соединениями. С сетевыми соединениями функция работает одинаково во всех операционных системах.

В программе мы вызываем функцию `select` и передаем ей три списка:

1. Потоки, от которых мы ожидаем данные на чтение.
2. Соединения, которые готовы к записи данных (в рассматриваемой программе это не принципиально, поэтому мы передаем пустой список).
3. Список соединений, которые выдают какие-либо исключения (тоже не понадобятся в текущей программе).

На выходе функция `select` выдает три списка:

1. Соединения, готовые к чтению данных.
2. Соединения, готовые к записи данных.
3. Исключительные ситуации.

Второй и третий списки нам не нужны, поэтому мы их сбрасываем, используя переменную со знаком «_».

Далее мы проходим по списку `ready` и смотрим, кто готов. Если готов сервер, то это значит, что есть новое соединение и мы вызываем функцию `accept_connection`. Нового клиента мы записываем в список ожиданий.

Если готов кто-то другой — это клиент. В этом случае вызывается функция `proceed_message`, и мы смотрим, что вернула эта функция. Если сокет уже закрылся, мы закрываем его и удаляем из списка:

```
from select import select
with socket.create_server(('127.0.0.1', 5000)) as server:
    monitoring = [server]
    while True:
        ready, *_ = select(monitoring, [], [])
        for sock in ready:
            if sock is server:
                new_client = accept_connection(sock)
                monitoring.append(new_client)
            else:
                if not proceed_message(sock):
                    sock.close()
                    monitoring.remove(sock)
```

Функция `select` также является блокирующей, но она будет заблокирована до того момента, пока не появятся данные где-нибудь. А нам это и нужно.

Функцию `select` можно настроить так, чтобы она работала с некоторым тайм-аутом.

Рассмотрим, как сделать многопоточный сервер с помощью `socketserver`. В коде реализовано множественное наследование. Во-первых, мы создаем специальный класс `MyTCPHandler` — обработчик запросов. Помимо этого, создаем свой класс сервера `MyServer`, который должен унаследовать от двух классов — `ThreadingMixIn` и `TCPServer`.

`MixIn` — это понятие из объектно-ориентированного программирования. Означает классы для множественного наследования, которые позволяют частично переопределить методы второго класса. Поэтому важно класс `ThreadingMixIn` записать первым.

Множественное наследование используется, поскольку мы можем создать не только TCP-сервер, но и, например, UDP. Поэтому `ThreadingMixIn` нужен как дополнительный инструмент, который мы можем «приклеить» к тому или к другому классу.

В классе `MyServer` можно ничего не описывать, достаточно прописать само наследование. В рассматриваемом примере дополнительно указано, что потоки должны запускаться как демоны. И записан атрибут, позволяющий переиспользовать адрес заново.

Далее мы также запускаем сервер и используем метод `serve_forever`:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        while True:
            self.data =
self.request.recv(4096).strip().decode('utf-8')
            if not self.data:
                break
            print("{} wrote:".format(self.client_address[0]))
            print(self.data)

self.request.sendall(self.data.upper().encode('utf-8'))

class MyServer(socketserver.ThreadingMixIn,
socketserver.TCPServer):
    daemon_threads = True
    allow_reuse_address = True

if __name__ == "__main__":
    HOST, PORT = "localhost", 5000
    with MyServer((HOST, PORT), MyTCPHandler) as server:
        server.socket.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEPORT, True)
        server.serve_forever()
```

5. Простой HTTP-сервер

Разберем простой пример работы с протоколом HTTP. На практике такую задачу редко придется выполнять руками, но сейчас интересно посмотреть, как это работает.

Если мы запустим написанный нами ранее TCP-сервер и попробуем обратиться к нему из браузера, сам сервер получит примерно такой запрос:

```
GET / HTTP/1.1
```

```
Host: 127.0.0.1:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:107.0)
Gecko/20100101 Firefox/107.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,i
mage/webp,*/*;q=0.8
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
```

В первой строке запроса `GET / HTTP/1.1` содержится:

- название метода для загрузки данных (в запросе это метод GET, существует также метод POST);
- название ресурса, который мы хотим получить (в запросе за это отвечает символ / — получение корневой страницы);
- версия протокола HTTP.

Со следующей строки следуют заголовки запросов. Заголовки аналогичны словарю: сначала следует название заголовка, а затем соответствующий ему текст.

Принципиально важный заголовок — `Host`, который описывает, к какому сайту мы обращаемся. То есть то, что записано в адресной строке браузера. Физически один сервер может обрабатывать запросы к разным сайтам. И он должен понимать, что у него спрашивают.

Остальные заголовки не принципиальны, часть из них может отсутствовать. Рассмотрим некоторые заголовки:

- `User-Agent` описывает, из какого браузера выполняется запрос;
- `Accept, Accept-Language, Accept-Encoding` описывает то, что браузер ожидает.

Возьмем за основу рассмотренный запрос, сымитируем его и попробуем отправить серверу. В примере запрос отправляется на сервер `cs.mipt.ru/python`, имя сервера указано в заголовке `Host: cs.mipt.ru` и в первой строчке запроса `/python`:

```
request = b'''GET /python HTTP/1.1
Host: cs.mipt.ru
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:107.0)
Gecko/20100101 Firefox/107.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
image/webp,*/*;q=0.8
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1

'''.replace(b'\n', b'\r\n')
```

Важно! Стандарт HTTP требует, чтобы строки в заголовках запроса были разделены двумя символами переноса строки. Для этого в запрос добавлена инструкция `replace(b'\n', b'\r\n')`, которая заменяет переход на следующую строку (`\n`) возвратом в начало строки переходом на следующую строку (`\r\n`). Такой подход сохранился со времен печатных машинок, в которых для перехода на следующую строку нужно было вернуться в начало строки и потом перейти на следующую строку.

Все запросы идут в байтах, поэтому в инструкции `replace` в скобках присутствует буква `b`.

Далее мы загружаем данные. Поскольку ожидается, что данных будет больше, чем 4KB, мы загружаем данные кусочками и соединяем их вместе в цикле:

```
data = b''
with socket.create_connection(('cs.mipt.ru', 80)) as con:
    con.sendall(request)
    while True:
        chunk = con.recv(4096)
        if not chunk:
            break
        data += chunk

headers, data = data.split(b'\r\n\r\n', 1)
print(headers.decode())
```

Заголовки и сами данные в запросе отделяются двумя пустыми строками. Поэтому, чтобы выделить заголовки, используется вот такая инструкция (два переноса строки):

```
headers, data = data.split(b'\r\n\r\n', 1)
```

Рассмотрим, что ответит сервер на такой запрос:

```
HTTP/1.1 200 OK
Date: Sat, 19 Nov 2022 11:13:02 GMT
Server: nginx/1.4.6 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Last-Modified: Wed, 28 Sep 2022 02:29:20 GMT
Content-Encoding: gzip
Connection: close
Transfer-Encoding: chunked
```

Стандартный ответ протокола HTTP содержит версию протокола HTTP, затем код ответа и название кода.

Существует соглашение — первая цифра кода означает, из какой он группы:

- первая цифра «1» — информационные коды (редко используются на практике);
- первая цифра «2» — все хорошо, ответ на запрос получен;
- первая цифра «3» — перенаправление;
- первая цифра «4» — ошибка в запросе клиента;
- первая цифра «5» — ошибка сервера.

Сервер также сообщает дату ответа, движок сервера (в примере — nginx поставленный на Ubuntu), тип ответа (текст UTF-8), дату изменения запрашиваемого файла (имеет смысл при кешировании файлов).

Сделаем простейший HTTP-сервер, который сможет отвечать на запросы из браузера. Для этого нужно прочитать запрос и выдать ответ, похожий на ранее рассмотренный формат.

Здесь реализован упрощенный подход — отправляется меньшее количество заголовков:

- название протокола,
- код ответа и его описание (описание необязательно должно соответствовать коду, так как браузер считывает и обрабатывает лишь код);
- вид отправляемых данных (текст в кодировке UTF-8);
- длина ответа.

В программе реализуется функция, которая подставит в шаблон нужные параметры:

```
response = '''HTTP/1.1 {code} {code_name}
Content-Type: text/html; charset=utf-8
Content-Length: {byte_len}
```

```
'''.replace('\n', '\r\n')

def generate_response(code, text):
    codes = {
        200: 'OK',
        403: 'Forbidden'
        404: 'Not Found',
    }
    content = text.encode('utf-8')
    return response.format(
        code=code,
        code_name=codes[code],
        byte_len=len(content)
    ).encode('utf-8') + content
```

Очень упрощенный код сервера:

```
with socket.create_server(('0.0.0.0', 5000)) as server:
    while True:
        client, addr = server.accept()
        request = client.recv(4096).decode('utf-8')
        print(request)
        method, resource, *other = request.split()
        if resource == '/':
            code, text = 200, 'Привет!'
        elif resource == '/admin':
            code, text = 403, 'Сюда нельзя'
        else:
            code, text = 404, 'Такого нет'
        client.sendall(generate_response(code, text))
```

Мы получаем клиентское соединение и считываем данные. Из данных вычленяем метод, ресурс и другие параметры, а затем проверяем, какой ресурс запрашивает клиент:

- если корневая папка — код 200;
- если доступ в админку — код 403;
- если что-то другое — код 404.

После этого программа отправляет ответ клиенту.

6. Основные библиотеки для работы HTTP

Встроенные модули для работы с HTTP на практике нужны только создателям библиотек и фреймворков. Прикладные программисты их напрямую не используют.

Пакет `http` состоит из нескольких модулей и содержит базовые функции для работы с протоколом HTTP:

- `http.client`;
- `http.server`;
- `http.cookies`;
- `http.cookiejar`.

Cookies (куки) — представляют собой нечто вроде заголовков запросов, но которые хранятся постоянно у клиента. Cookies необходимы, чтобы идентифицировать клиента.

В стандартной библиотеке есть модуль `ssl`, который поддерживает криптографический протокол `ssl`.

Пакет `urllib` предназначен для работы с URL-адресами, содержит модули:

- `urllib.request`;
- `urllib.error`;
- `urllib.parse`;
- `urllib.robotparser`.

Пакет `wsgiref` — WSGI (Web Server Gateway Interface) — специальный интерфейс в Python, который создан для работы с разными серверными приложениями.

На практике эти модули используются редко, сейчас достаточно иметь представление о наличии этих модулей.

Более известные и удобные библиотеки для загрузки данных из Интернета:

- `requests` — наиболее известная и «почти» стандартная библиотека;
- `httpx` — более современная библиотека с поддержкой асинхронных запросов.

Рассмотрим базовый пример использования библиотеки `httpx`. Из библиотеки мы вызываем функцию `get` и передаем ей URL-адрес, который должен быть полным, включая указание протокола. Эта функция выдает объект `Response`, который содержит все данные об ответе: статус-код, в отдельном атрибуте `headers` хранится словарь с заголовками:

```
>>> import httpx
>>> r = httpx.get('https://www.example.org/')
>>> r
<Response [200 OK]>
```

```
>>> r.status_code
200
>>> r.headers['content-type']
'text/html; charset=UTF-8'
```

Если ответ пришел в виде текста, его можно получить из атрибута `text`:

```
>>> r.text
'<!doctype html>\n<html>\n<head>\n<title>Example
Domain</title>...'
```

Основные функции библиотеки `httpx` дублируют соответствующие HTTP-запросы:

- `httpx.get`;
- `httpx.post`;
- `httpx.put`;
- `httpx.delete`;
- `httpx.head`;
- `httpx.options`;
- `httpx.request`.

Запрос `get` — самый частый, используется для загрузки данных. Запрос `post` используется для отправки данных на сервер. Запрос `put` аналогичен по функциональности запросу `post`, разница — `put` используется для редактирования данных, которые есть на сервере, `post` — для загрузки новых данных. Технически обычно используют `post`.

Запрос `delete` используют для удаления данных с сервера. Запрос `head` загружает только заголовки. Запрос `option` сообщает, какие параметры поддерживает ресурс.

Существует общая функция `request`, которая первым аргументом принимает используемый метод.

Основные параметры запроса:

- `method` — метод;
- `url` — URL-адрес (полный);
- `params` — параметры запроса (в URL следуют после знака «?»);
- `headers` — заголовки (можно не указывать, библиотека добавит заголовки сама);
- `cookies` — куки;
- `follow_redirects` — автоматически следовать перенаправлениям (по умолчанию `False`).

Дополнительные параметры запроса:

- `content` — содержимое запроса (для запроса post);
- `data`, `files`, `json` — возможность отправить данные в более удобном формате;
- `timeout` — по умолчанию 5 секунд (можно перенастроить);
- и другие.

С помощью этой библиотеки можно создать постоянное соединение. Для этого мы создаем экземпляр класса `httpx.Client`, и далее с его помощью обращаемся к разным URL-адресам. Настройки соединения (заголовки, cookies и другие) выполняются при этом один раз:

```
url = 'http://httpbin.org/headers'
headers = {'user-agent': 'my-app/0.0.1'}
with httpx.Client(headers=headers) as client:
    r = client.get(url)
```

При получении ответа нужно проверить, что в нем пришло. Можно проверить статус-код. Если статус-код равен 200, то все хорошо:

```
>>> r = httpx.get('http://example.org')
>>> r.status_code
200
```

Есть удобная функция `raise_for_status` — если получен ответ, начинающийся с 2, функция ничего не делает. Если получен другой код, сообщающий об ошибке, функция создает исключение:

```
>>> r.raise_for_status()
>>> r = httpx.get('http://ya.ru/gjhsiuyefj',
follow_redirects=True)
>>> r.status_code
404

>>> r.raise_for_status()
Traceback (most recent call last):
  ...
httpx.HTTPStatusError: Client error '404 Not Found' for url
'https://ya.ru/gjhsiuyefj'
For more information check: https://httpstatuses.com/404
```

Другие атрибуты и методы объекта `Response`:

- `.content: bytes` — если данные пришли не в текстовом виде, они будут храниться в этом атрибуте;
- `.text: str`;

- `.encoding: str` — если ответ пришел не в UTF-8, можем это увидеть;
- `.request: Request`.
- `def .json()` — получить словарь из json;
- `def .iter_bytes([chunk_size])` — читать байты по частям;
- `def .iter_text([chunk_size])` — читать текст по частям.

Рассмотрим кратко пример создание HTTP-сервера на фреймворке Flask.

Фреймворк означает каркас. Фреймворк нужно отличать от библиотеки. **Библиотека** — набор функций, которые вы можете встраивать в свое приложение. Библиотека никак не ограничивает программиста, который сам пишет свое предложение полностью с нуля. Из библиотеки можно взять только нужные функции.

Фреймворк — по сути, готовое приложение, в которое вы должны встроить свою логику.

Рассмотрим пример простого приложения на фреймворке Flask. Мы импортируем класс `Flask`, который уже содержит готовое серверное приложение. В программе создается экземпляр класса `app` (от слова application), далее выполняется настройка его функций, которые будут отвечать на определенные запросы.

При создании экземпляра класса `Flask` ему передается параметр `__name__`. Сам фреймворк импортирует модуль, и приложение создается с тем названием, как мы назвали файл:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

Нужно пояснить работу декоратора, поскольку она отличается от того, что мы рассматривали ранее. Декоратор в примере никак не меняет работу функции — он ее просто запоминает. Сама функция остается без изменений.

Чтобы запустить сервер, нужно из терминала дать следующую команду:

```
$ flask --app hello --debug run
 * Serving Flask app "hello"
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with inotify reloader
 * Debugger is active!
```

```
 * Debugger PIN: 223-456-919
```

Если бы файл назывался `app.py`, то было бы достаточно запустить команду `flask run`. Мы же предполагаем, что файл называется `hello.py`, поэтому должны указать название приложения с помощью флага `--app`. Полезный флаг `--debug` позволяет отслеживать Flask изменения в коде. Если файл изменился, сервер запускается заново. Это полезно для отладки.

Фреймворк нужен для того, чтобы обработать запрос и подставить в ответ динамические данные.

Пример из документации Flask:

```
from markupsafe import escape

@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return f'User {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return f'Post {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return f'Subpath {escape(subpath)}'
```

Мы можем сами указывать адреса запросов с некоторыми дополнительными параметрами, которые будут переданы в нашу функцию в качестве аргумента.

В примере используется функция `escape` для того, чтобы некоторые специальные символы отображались как символы, а не HTML-теги.

Flask — довольно простой фреймворк, он имеет большую базу плагинов. Один из примеров доступен по ссылке.

Дополнительные материалы для самостоятельного изучения

1. Internet Protocols and Support