**Модуль 4: Основи Web**

**Заняття 2: Побудова Web програми та Socket**

**Простий Web застосунок**

У попередньому розділі ми розглянули основу роботи http.server. Тепер підемо далі та створимо найпростіший сайт. Розглянемо основні моменти функціонування та роботи вебзастосунку. Наш застосунок складатиметься з двох сторінок - великої index.html та сторінки Надіслати повідомлення contact.html.

Оскільки сторінок дві, ми зможемо розібрати питання маршрутизації в нашому застосунку.

**index.html**

За допомогою форми, на сторінці надсилання повідомлення, ми розберемо момент передачі даних на сервер методом POST протоколу HTTP. Для стилізації застосунку ми будемо використовувати популярну бібліотеку [Bootstrap](https://getbootstrap.com/) (https://getbootstrap.com/)

**contact.html**

Ми не розбиратимемо самі моменти стилізації, а зосередимося на питаннях побудови серверного застосунку.

**Маршрутизація в застосунку**

Визначимо для двох наших сторінок маршрути:

* '/' - Головна сторінка
* '/contact' - сторінка з надсиланням повідомлення

Для цього змінимо в HTML коді значення атрибутів href у посилань переходу сторінок

<ul class="navbar-nav">

<li class="nav-item">

<a class="nav-link" aria-current="page" href="/">Home</a>

</li>

<li class="nav-item">

<a class="nav-link" href="/contact">Write to me</a>

</li>

</ul>

Тепер ці маршрути потраплять у змінну self.path класу HttpHandler. Наведемо повний приклад поточного коду застосунку та розглянемо його детальніше.

main.py

from http.server import HTTPServer, BaseHTTPRequestHandler

import urllib.parse

class HttpHandler(BaseHTTPRequestHandler):

def do\_GET(self):

pr\_url = urllib.parse.urlparse(self.path)

if pr\_url.path == '/':

self.send\_html\_file('index.html')

elif pr\_url.path == '/contact':

self.send\_html\_file('contact.html')

else:

self.send\_html\_file('error.html', 404)

def send\_html\_file(self, filename, status=200):

self.send\_response(status)

self.send\_header('Content-type', 'text/html')

self.end\_headers()

with open(filename, 'rb') as fd:

self.wfile.write(fd.read())

def run(server\_class=HTTPServer, handler\_class=HttpHandler):

server\_address = ('', 8000)

http = server\_class(server\_address, handler\_class)

try:

http.serve\_forever()

except KeyboardInterrupt:

http.server\_close()

if \_\_name\_\_ == '\_\_main\_\_':

run()

Наприклад, при зверненні до маршруту /contact у змінній self.path знаходитиметься значення /contact. І, в принципі, для нашого простого прикладу цього достатньо. Але правильнішим буде підхід використання спеціальної функції urllib.parse.urlparse, яка повертає об'єкт ParseResult(scheme='', netloc='', path='/contact', params='', query='', fragment=''). Вона може отримувати з маршруту також, наприклад, 'query' запити. Якщо ми звернемося за адресою /contact?name=Test у змінній self.path буде знаходитись /contact?name=Test, а ось urllib.parse.urlparse поверне нам те, що потрібно ParseResult(scheme='', netloc='', path='/contact', params='', query='name=Test', fragment='') та в атрибуті path буде маршрут '/contact', а 'query' містить значення 'name=Test', куди воно і мало потрапити.

Для відповіді браузеру ми використовуємо метод send\_html\_file:

def send\_html\_file(self, filename, status=200):

self.send\_response(status)

self.send\_header('Content-type', 'text/html')

self.end\_headers()

with open(filename, 'rb') as fd:

self.wfile.write(fd.read())

Метод зчитує файл filename та відправляє його браузеру з кодом за замовчуванням 200.

Сама маршрутизація у нас виконана за допомогою вкладених інструкцій if.

if pr\_url.path == '/':

self.send\_html\_file('index.html')

elif pr\_url.path == '/contact':

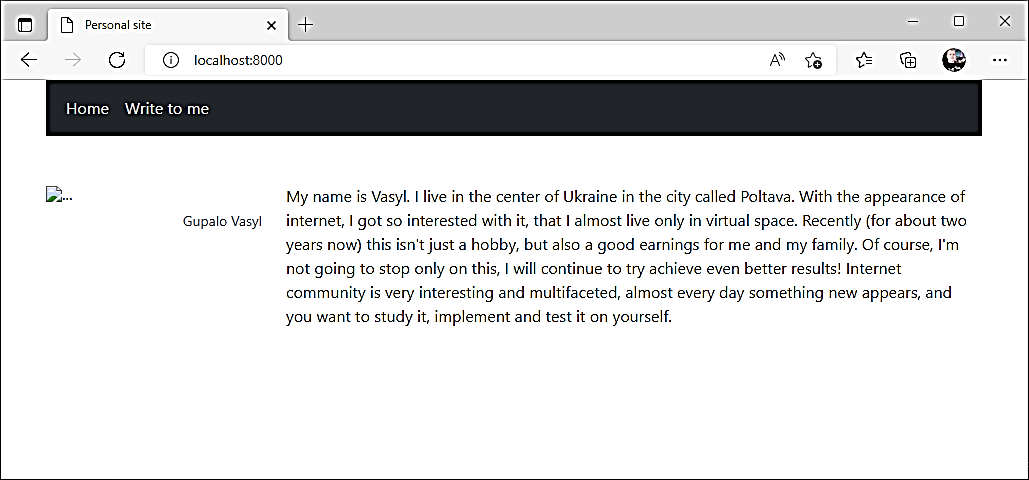
self.send\_html\_file('contact.html')

else:

self.send\_html\_file('error.html', 404)

Якщо жоден маршрут не визначено, то ми повертаємо спеціальну сторінку для помилки 404 Not Found.

Запустимо наш застосунок за допомогою функції run і звернемося за адресою <http://localhost:8000/> (). Побачимо, що маршрути застосунку працюють, але відсутні зображення і не застосовуються до html стилі з файлу style.css. Ми не бачимо колір фону на сторінці.



Всі ці файли, які повинні повертатися з сервера, але не є файлами HTML, називають загальним словом статичні ресурси. Давайте підключимо їх.

**Статичні ресурси**

Якщо зараз звернутися до логів застосунку, ми побачимо, що під час запиту до головної сторінки браузер просить надіслати йому файл стилів style.css та файл зображення gupalo.jpg.

127.0.0.1 - - [16/Sep/2022 18:20:06] "GET / HTTP/1.1" 200 -

127.0.0.1 - - [16/Sep/2022 18:20:07] "GET /style.css HTTP/1.1" 404 -

127.0.0.1 - - [16/Sep/2022 18:20:07] "GET /images/gupalo.jpg HTTP/1.1" 404 -

Він знаходить їх на сторінці index.html і просить наш вебзастосунок їх надіслати.

<link rel="stylesheet" href="style.css" />

...

<img

src="images/gupalo.jpg"

class="figure-img img-fluid rounded"

alt="..."

width="300"

height="300"

/>

Але наш застосунок нічого не знає про ці файли і відповідає помилкою 404 - нічого не знайдено.

Найпростішим рішенням буде перевіряти, чи існує вказаний файл на сервері, наприклад /images/gupalo.jpg, і надсилати його браузеру. Правда незрозумілий тип контенту, що відправляється. При надсиланні html файлу ми повідомили браузеру контент надсиланням наступного заголовка self.send\_header('Content-type', 'text/html'). Це називається [MIME types](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types) (https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics\_of\_HTTP/MIME\_types/Common\_types) файлу. Так ми повідомляємо браузеру тип даних, які можуть бути передані за допомогою HTTP протоколу.

Для визначення MIME types файлів у Python існує окремий модуль [mimetypes](https://docs.python.org/3/library/mimetypes.html) (https://docs.python.org/3/library/mimetypes.html)

Функція для надсилання статичних ресурсів буде мати наступний вигляд:

def send\_static(self):

self.send\_response(200)

mt = mimetypes.guess\_type(self.path)

if mt:

self.send\_header("Content-type", mt[0])

else:

self.send\_header("Content-type", 'text/plain')

self.end\_headers()

with open(f'.{self.path}', 'rb') as file:

self.wfile.write(file.read())

Якщо модуль визначив тип файлу, то надсилаємо його self.send\_header("Content-type", mt[0]). У разі невдачі ми вважаємо тип файлу простим текстом self.send\_header("Content-type", 'text/plain').

Останній штрих — додамо до коду маршрутизації обробку статичних ресурсів.

if pr\_url.path == '/':

self.send\_html\_file('index.html')

elif pr\_url.path == '/contact':

self.send\_html\_file('contact.html')

else:

if pathlib.Path().joinpath(pr\_url.path[1:]).exists():

self.send\_static()

else:

self.send\_html\_file('error.html', 404)

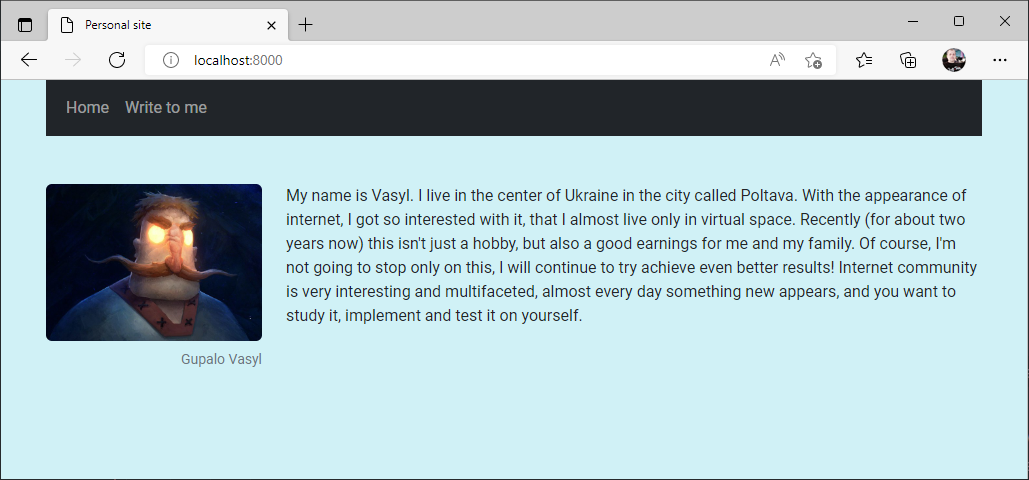
Тепер у логах повний порядок:

127.0.0.1 - - [19/Sep/2022 03:06:56] "GET / HTTP/1.1" 200 -

127.0.0.1 - - [19/Sep/2022 03:06:57] "GET /style.css HTTP/1.1" 200 -

127.0.0.1 - - [19/Sep/2022 03:06:57] "GET /images/gupalo.jpg HTTP/1.1" 200 -

А при запуску нашого застосунку ми бачимо і зображення, і стилі з файлу style.css.



Наступний крок — це обробка запитів POST від форми на сторінці відправлення повідомлень.

**Робота з формою**

Обробка форми виконується функцією do\_POST. Оскільки в нашому застосунку лише одна форма, то ми не писатимемо всередині цієї функції обробку маршрутів. У self.path завжди буде /contact, який прописаний в атрибуті action нашої форми на сторінці contact.html.

<form action="/contact" method="POST" enctype="application/x-www-form-urlencoded" accept-charset="UTF-8">

<h1 class="h3 mb-3 font-weight-normal">Write to me</h1>

<div class="row mb-3">

<label class=""> Your name </label>

<input class="form-control" name="username" type="text"/>

</div>

<div class="row mb-3">

<label class=""> Email address </label>

<input class="form-control" name="email" type="email"/>

</div>

<div class="row mb-3">

<label class=""> Your text message </label>

<textarea class="form-control" name="message"></textarea>

</div>

<div class="d-flex justify-content-evenly">

<button class="btn btn-primary col-4" name="" type="submit">

Send

</button>

<button class="btn btn-secondary col-4" name="" type="reset">

Reset

</button>

</div>

</form>

Сама функція do\_POST досить проста.

def do\_POST(self):

data = self.rfile.read(int(self.headers['Content-Length']))

print(data)

data\_parse = urllib.parse.unquote\_plus(data.decode())

print(data\_parse)

data\_dict = {key: value for key, value in [el.split('=') for el in data\_parse.split('&')]}

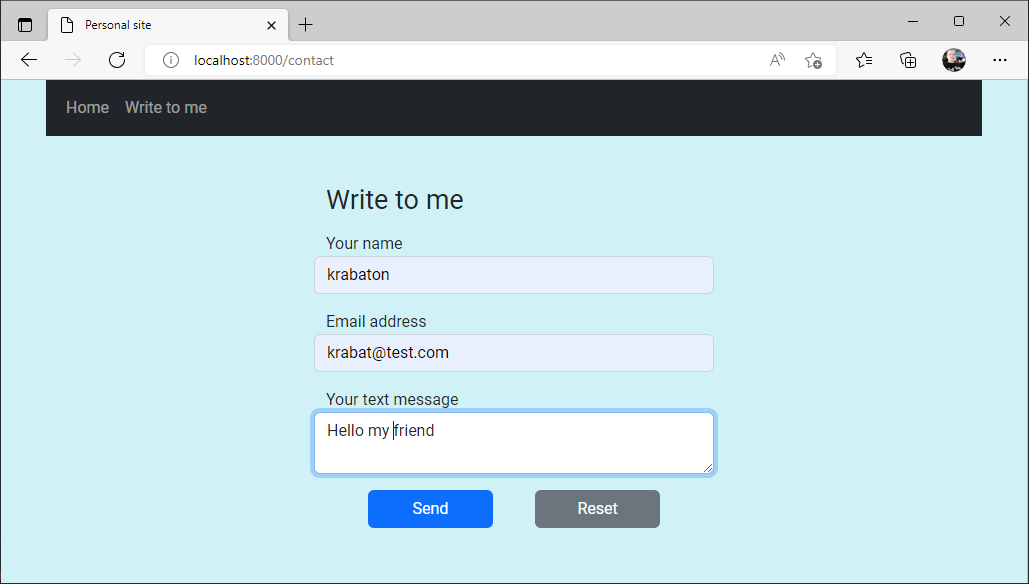
print(data\_dict)

self.send\_response(302)

self.send\_header('Location', '/')

self.end\_headers()

Давайте детально розглянемо кожен рядок роботи цієї функції. Нехай ми ввели у формі такі дані, як на малюнку нижче, і натиснули кнопку Send. Браузер сформує дані та надішле їх нашому застосунку. Він збирає дані з тегів input, textarea тощо. Формує рядки зі значенням атрибуту name та атрибуту value, розділяючи їх знаком дорівнює. Наприклад, для <input class="form-control" name="username" type="text" /> у нашому випадку буде рядок виду username=krabaton. Остаточний рядок виходить з'єднанням через знак & всіх отриманих рядків з елементів введення.



Для отримання даних у застосунку з форми ми використовуємо функцію self.rfile.read, яка читає байт-рядок певного розміру. Розмір даних, що передаються, у байтах браузер (клієнт) передає через заголовок Content-Length. Тому наступним рядком коду ми отримуємо дані від браузера:

data = self.rfile.read(int(self.headers['Content-Length']))

print(data)

Це байт-рядок виду:

b'username=krabaton&email=krabat%40test.com&message=Hello+my+friend'

Для форми з enctype="application/x-www-form-urlencoded" пробіли повинні бути замінені на "+", а також браузер застосовує до рядка метод [encodeURIComponent](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent) (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent>). Щоб повернути дані до початкового вигляду, нам потрібно застосувати метод urllib.parse.unquote\_plus

data\_parse = urllib.parse.unquote\_plus(data.decode())

print(data\_parse)

username=krabaton&email=krabat@test.com&message=Hello my friend

Після цього рядок можна перетворити на словник наступним виразом:

data\_dict = {key: value for key, value in [el.split('=') for el in data\_parse.split('&')]}

print(data\_dict)

Остаточний результат буде таким:

{'username': 'krabaton', 'email': 'krabat@test.com', 'message': 'Hello my friend'}

І цей результат ми вже можемо зберегти, наприклад, у базу даних.

Далі ми виконуємо редирект на головну сторінку. Для цього відправляємо статус 302 та встановлюємо заголовок Location: /. Коли браузер отримає нашу відповідь, він зрозуміє за заголовком Location: /, що йому потрібно перейти на головну сторінку.

self.send\_response(302)

self.send\_header('Location', '/')

self.end\_headers()

Остаточний варіант нашого простого вебзастосунку можна подивитися в живому прикладі.

Після запуску прикладу, застосунок буде доступний за цим посиланням <https://http-server.krabaton.repl.co/>

**Sockets**

Ми розглянули роботу вебзастосунків за протоколом HTTP. Але Python також дозволяє нам виконувати передачу повідомлень між застосунками на нижчому рівні — за допомогою протоколу TCP/IP. Для цього він використовує сокети.

**Сокет** (Socket) — це програмний інтерфейс для забезпечення інформаційного обміну між процесами. Існують клієнтські та серверні сокети. Серверний сокет прослуховує певний порт, а клієнтський підключається до сервера. Щойно було встановлено з'єднання, починається обмін даними. Після успішного встановлення з'єднання сервер і клієнт починають обмінюватися інформацією. Наприклад, сервер посилає вітання та пропозицію ввести будь-яку команду. Клієнт зі свого боку вводить команду, сервер її аналізує, виконує необхідні операції та віддає клієнту результат.

У Python для роботи з сокетами використовується модуль socket:

import socket

sock = socket.socket()

Ця частина є спільною і для клієнтських, і для серверних сокетів, ми створили сокет. У Linux усі системні ресурси реалізують інтерфейс файлів. І сокети не виняток.

З точки зору застосунку у вас є спеціальний файл-сокет. Ви можете в нього писати bytes і читати з нього теж bytes.

Далі почнемо писати серверний сокет. Це той, який слухає на деякому порту і чекає на підключення клієнта.

**Серверна частина**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/socket_05#%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B0-%D1%87%D0%B0%D1%81%D1%82%D0%B8%D0%BD%D0%B0)Для початку скажемо системі, на якому порту та адресі прийматимемо з'єднання:

sock.bind(('', 9090))

Якщо залишити аргумент для адреси порожнім рядком, то сокет буде доступним для всіх інтерфейсів. Порт ми вказали 9090.

Тепер у нас все готове, щоб приймати з'єднання. За допомогою методу listen ми запустимо для цього сокету режим прослуховування. Метод приймає один аргумент — максимальна кількість підключень у черзі.

sock.listen(1)

Тепер ми можемо прийняти підключення за допомогою методу accept, який повертає кортеж із двома елементами: новий сокет та адреса клієнта. Саме цей сокет і буде використовуватися для приймання та відправлення даних клієнту.

conn, addr = sock.accept()

Тепер ми встановили зв'язок з клієнтом і можемо з ним «спілкуватися». Ми не можемо точно знати, що і в яких обсягах клієнт нам надішле, тому ми будемо отримувати дані від нього невеликими порціями. Щоб отримати дані, потрібно скористатися методом **recv**, який як аргумент приймає кількість байт для читання. Ми читатимемо порціями по 1024 байт (або 1 кб):

while True:

data = conn.recv(1024)

if not data:

break

conn.send(data.upper())

Ми в нескінченному циклі приймаємо 1024 байт даних за допомогою методу **recv.** Якщо даних більше немає, цей метод нічого не повертає. Таким чином, ми можемо отримувати від клієнта будь-яку кількість даних.

Далі у нашому прикладі для наочності ми щось зробимо з отриманими даними та відправимо їх назад клієнту. Наприклад, за допомогою методу upper у рядків повернемо клієнту рядок у верхньому регістрі.

Тепер можна і закрити з'єднання:

conn.close()

Сокет — це системний ресурс і, як і файл, його потрібно повернути системі (закрити), що б не сталося. Для цього, як і у файлів, ми можемо скористатися менеджером контексту. Ми також можемо закривати з'єднання за допомогою менеджера контексту, щоб гарантовано не залишати його відкритим, навіть у разі помилки.

Крім цього, ще корисно вказати системі, що якщо застосунок не закрив з'єднання, то потрібно дозволити повторно відкрити тому самому порту. Для цього налаштуємо сокет:

sock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

Об'єднаємо все в одну функцію:

import socket

def echo\_server(host, port):

with socket.socket() as s:

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

s.bind((host, port))

s.listen(1)

conn, addr = s.accept()

print(f"Connected by {addr}")

with conn:

while True:

data = conn.recv(1024)

print(f'From client: {data}')

if not data:

break

conn.send(data.upper())

Тепер функція **echo\_server** створює сервер, який завжди очікує з'єднання.

Далі розглянемо клієнтську частину.

**Клієнтська частина**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/socket_05#%D0%BA%D0%BB%D1%96%D1%94%D0%BD%D1%82%D1%81%D1%8C%D0%BA%D0%B0-%D1%87%D0%B0%D1%81%D1%82%D0%B8%D0%BD%D0%B0)Клієнтський застосунок простіший — нам потрібно створити сокет, підключитися до сервера, надіслати йому дані, прийняти дані та закрити з'єднання.

from time import sleep

def simple\_client(host, port):

with socket.socket() as s:

while True:

try:

s.connect((host, port))

s.sendall(b'Hello, world')

data = s.recv(1024)

print(f'From server: {data}')

break

except ConnectionRefusedError:

sleep(0.5)

Єдине нове тут – це метод **connect**, за допомогою якого ми підключаємось до сервера. З'єднання може не встановлюватися з першого разу. Щоб обробити помилку з'єднання, ми помістили клієнтський код у нескінченний цикл для повторення спроб з'єднатися.

**Клієнт-серверний застосунок на потоках**

[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/socket_05#%D0%BA%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B8%D0%B9-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA-%D0%BD%D0%B0-%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B0%D1%85)Тепер об'єднаємо клієнтський та серверний застосунок в один і запустимо їх у різних потоках:

import threading

HOST = '127.0.0.1'

PORT = 55555

server = threading.Thread(target=echo\_server, args=(HOST, PORT))

client = threading.Thread(target=simple\_client, args=(HOST, PORT))

server.start()

client.start()

server.join()

client.join()

print('Done!')

У консолі побачимо:

Connected by ('127.0.0.1', 58268)

From client: b'Hello, world'

From server: b'HELLO, WORLD'

Done!

Операції із сокетами — за замовчуванням блокуючі. Це означає, що виклики connect, accept, recv, sendall не дадуть застосунку продовжувати роботу, доки не завершаться успішно. Але цю поведінку можна змінити. Якщо ви, наприклад, хочете перевіряти, чи не підключився новий клієнт, і робити ще щось корисне в цьому ж потоці. У такому разі виклик accept відразу поверне виняток і не буде очікувати на підключення. Для цього можна встановити прапорець:

socket.setblocking(0)

Асинхронні сокети **asyncio** реалізовані саме за допомогою обробки результатів неблокуючих викликів при **socket.setblocking(0)**

Ви також можете скористатися фреймворком, який допомагає у створенні власних серверів [socketserver](https://docs.python.org/3.9/library/socketserver.html)

(https://docs.python.org/3.9/library/socketserver.html).

**Побудова простих ехо-серверів на Socket**

Відповідно до способів взаємодії мережею, програми можна поділити на дві категорії:

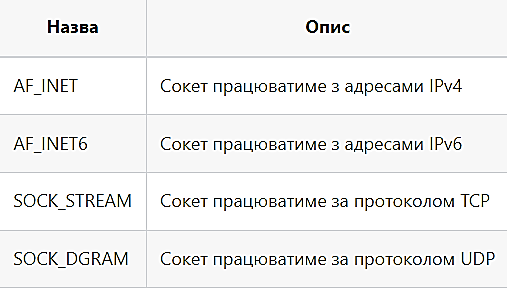
* без встановлення з'єднання (протокол **UDP**)
* із встановленням з'єднання (протокол **TCP**)

Протоколи **UDP** та **TCP** працюють поверх мережевого **IP**-протоколу. Вони більше відомі під загальною назвою — стек протоколів[**TCP/IP**](https://ru.wikipedia.org/wiki/TCP/IP) (https://ru.wikipedia.org/wiki/TCP/IP)

При передачі даних мережею завжди потрібно пам'ятати, що обмін відбувається сокетами (байтовими даними) і необхідно виконувати кодування та декодування даних.

У Python для обміну даними мережею використовують модуль socket, який надає інтерфейс для роботи із сокетами.

Цей модуль надає великий набір констант для конфігурації екземплярів сокетів. Найзначущі наступні константи



Детальніше про модуль socket дивіться [документацію](https://docs.python.org/3/library/socket.html)

(https://docs.python.org/3/library/socket.html)

**Ехо-сервер: UDP протокол**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/socket_06#%D0%B5%D1%85%D0%BE-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80-udp-%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB)Алгоритми роботи застосунків мережею, без встановленого з'єднання, наступний.

Сервера:

1. Створюємо сокет типу socket.SOCK\_DGRAM
2. За допомогою методу bind зв'язуємо сокет із адресою служби
3. Повторюємо наступні дії у нескінченному циклі:

* запит у клієнта сокету даних методом recvfrom (блокуюча операція)
* обробка даних
* надсилання результату клієнту за допомогою методу sendto

Клієнта:

1. Створюємо сокет типу socket.SOCK\_DGRAM
2. Надсилання повідомлення серверу методом sendto
3. Очікування відповіді від сервера за допомогою методу recvfrom
4. Використання отриманих даних далі у програмі клієнта

Код сервера:

import socket

UDP\_IP = '127.0.0.1'

UDP\_PORT = 8080

def run\_server(ip, port):

sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

server = ip, port

sock.bind(server)

try:

while True:

data, address = sock.recvfrom(1024)

print(f'Received data: {data.decode()} from: {address}')

sock.sendto(data, address)

print(f'Send data: {data.decode()} to: {address}')

except KeyboardInterrupt:

print(f'Destroy server')

finally:

sock.close()

if \_\_name\_\_ == '\_\_main\_\_':

run\_server(UDP\_IP, UDP\_PORT)

Код клієнта:

import socket

UDP\_IP = '127.0.0.1'

UDP\_PORT = 8080

MESSAGE = "Python Web development"

def run\_client(ip, port):

sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

server = ip, port

for line in MESSAGE.split(' '):

data = line.encode()

sock.sendto(data, server)

print(f'Send data: {data.decode()} to server: {server}')

response, address = sock.recvfrom(1024)

print(f'Response data: {response.decode()} from address: {address}')

sock.close()

if \_\_name\_\_ == '\_\_main\_\_':

run\_client(UDP\_IP, UDP\_PORT)

Спочатку запустимо код сервера, а потім код клієнта. Результат виконання повинен бути схожим на цей:

Виведення клієнта:

Send data: Python to server: ('127.0.0.1', 8080)

Response data: Python from address: ('127.0.0.1', 8080)

Send data: Web to server: ('127.0.0.1', 8080)

Response data: Web from address: ('127.0.0.1', 8080)

Send data: development to server: ('127.0.0.1', 8080)

Response data: development from address: ('127.0.0.1', 8080)

Виведення сервера:

Received data: Python from: ('127.0.0.1', 57422)

Send data: Python to: ('127.0.0.1', 57422)

Received data: Web from: ('127.0.0.1', 57422)

Send data: Web to: ('127.0.0.1', 57422)

Received data: development from: ('127.0.0.1', 57422)

Send data: development to: ('127.0.0.1', 57422)

Наш приклад — звичайний ехо-сервер, який відправляє і отримує одні й ті самі дані на сервер. Сервер відкритий на порту 8080 і отримує сокети даних у розмірі 1024 байт.

**Ехо-сервер: TCP протокол**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/socket_06#%D0%B5%D1%85%D0%BE-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80-tcp-%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB)Алгоритми роботи для сервера мережею, із встановленим з'єднанням, наступний:

1. Створюємо сокет типу socket.SOCK\_STREAM
2. За допомогою методу bind зв'язуємо сокет із адресою служби
3. Запуск прослуховування запитів на створення з'єднання, метод listen
4. Повторюємо наступні дії у нескінченному циклі:

* очікування запиту від клієнта створення з'єднання, метод accept
* створення асинхронного потоку управління для обробки нового з'єднання
* взаємодія з клієнтом у новому потоці через функцію handle за допомогою методів send та recv

Алгоритми роботи для клієнта:

1. Створюємо сокет типу socket.SOCK\_STREAM
2. Встановлення з'єднання з сервером метод connect
3. Взаємодія з сервером за допомогою методів recv та send
4. Використання отриманих даних далі у програмі клієнта

Код сервера:

import socket

from concurrent import futures as cf

TCP\_IP = 'localhost'

TCP\_PORT = 8080

def run\_server(ip, port):

def handle(sock: socket.socket, address: str):

print(f'Connection established {address}')

while True:

received = sock.recv(1024)

if not received:

break

data = received.decode()

print(f'Data received: {data}')

sock.send(received)

print(f'Socket connection closed {address}')

sock.close()

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind((ip, port))

server\_socket.listen(10)

print(f'Start echo server {server\_socket.getsockname()}')

with cf.ThreadPoolExecutor(10) as client\_pool:

try:

while True:

new\_sock, address = server\_socket.accept()

client\_pool.submit(handle, new\_sock, address)

except KeyboardInterrupt:

print(f'Destroy server')

finally:

server\_socket.close()

if \_\_name\_\_ == '\_\_main\_\_':

run\_server(TCP\_IP, TCP\_PORT)

У нашому коді робота сервера організована за принципом очікування підключення клієнта, а після отримання виконується зворотне пересилання повідомлень клієнту доти, доки він не закриє з'єднання.

Код клієнта:

import socket

TCP\_IP = 'localhost'

TCP\_PORT = 8080

MESSAGE = "Python Web development"

def run\_client(ip: str, port: int):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

server = ip, port

sock.connect(server)

print(f'Connection established {server}')

for line in MESSAGE.split(' '):

print(f'Send data: {line}')

sock.send(line.encode())

response = sock.recv(1024)

print(f'Response data: {response.decode()}')

print(f'Data transfer completed')

if \_\_name\_\_ == '\_\_main\_\_':

run\_client(TCP\_IP, TCP\_PORT)

**Побудова запитів HTTP**

Для зручності використання створені абстракції для роботи з протоколом HTTP. Вони побудовані поверх протоколу TCP/IP так, щоб не потрібно було використовувати сокети напряму. При надсиланні всіх видів HTTP-запитів у Python можна використовувати зовнішній модуль request або вбудований пакет urllib, та його метод urllib.request.

**Вбудований пакет urllib**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/request_07#%D0%B2%D0%B1%D1%83%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82-urllib)

Вбудований у Python пакет **urllib** містить модуль **request** з набором примітивів для запитів в мережу по протоколу HTTP:

import urllib.request

with urllib.request.urlopen('https://www.python.org/') as f:

print(f.read(300))

У відповідь може прийти щось на зразок такого html документа:

b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html

xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n

<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n

<title>Python Programming '

Це перші 300 байт HTML документа на головній сторінці [**https://www.python.org/**](https://www.python.org/)**.**

Модуль **request**містить все необхідне для роботи з HTTP. Ви можете вказати метод запиту, заголовок, додати дані в тіло запиту, отримати файл мережею, перевірити статус-код відповіді тощо.

**Модуль requests**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/request_07#%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C-requests)

Проте, все одно залишаються типові проблеми, з якими доводиться постійно стикатися:

* зберігати кукі відкритих сесій;
* перевикористовувати відкриті TCP з'єднання, щоб заощадити час та ресурси на створення нових для кожного запиту;
* обробку таймаутів;
* повторення запиту у разі обриву мережі.

Ці проблеми вирішені у сторонньому та дуже популярному пакеті requests.

Встановлення пакету:

pip install requests

Пакет добре документований, і це одна з головних причин його популярності. [Основна документація](Основна%20документація) (<https://requests.readthedocs.io/en/latest/>) з прикладами для різних окремих випадків дуже детальна і зрозуміла. Інтерфейс пакету багато в чому повторює urllib.request.

import requests

r = requests.get('https://api.github.com/events')

print(r.text[:300])

Перші 300 байт відповіді:

[{"id":"16602799523","type":"DeleteEvent","actor":{"id":49699333,"login":"dependabot[bot]","display\_login":"dependabot","gravatar\_id":"","url":"https://api.github.com/users/dependabot[bot]","avatar\_url":"https://avatars.githubusercontent.com/u/49699333?"},"repo":{"id":111928778,"name":"akigugale/tre

Пакет "вміє" працювати з архівованими відповідями, JSON даними, різним кодуванням. Більшість "фіч" реалізовані всередині і просто працюють без вашої участі або налаштування. Наприклад, не потрібно вказувати кодування відповіді, воно автоматично буде розпізнано із заголовка відповіді. Не потрібно вказувати, що тіло відповіді стиснене, потрібний алгоритм стиснення буде застосований автоматично.

**Приклад запиту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-04/module-04-02/request_07#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%83)Наприклад, нам потрібно дізнатися про поточні курси валют від банку. Як приклад візьмемо публічний API [**Приватбанку**](https://api.privatbank.ua/) (https://api.privatbank.ua/)**.** Відповідно до документації нам потрібно виконати GET запит за наступною [адресою](https://api.privatbank.ua/p24api/pubinfo?exchange&json&coursid=11) (https://api.privatbank.ua/p24api/pubinfo?exchange&json&coursid=11)

import requests

response = requests.get('https://api.privatbank.ua/p24api/pubinfo?exchange&json&coursid=11')

exchange\_rate = response.json()

print(exchange\_rate)

Відповіддю буде готовий список із даними:

[

{

"ccy": "USD",

"base\_ccy": "UAH",

"buy": "36.56860",

"sale": "37.45318"

},

{

"ccy": "EUR",

"base\_ccy": "UAH",

"buy": "36.43000",

"sale": "37.73585"

},

{

"ccy": "RUR",

"base\_ccy": "UAH",

"buy": "0.32000",

"sale": "0.35001"

},

{

"ccy": "BTC",

"base\_ccy": "USD",

"buy": "17902.7321",

"sale": "19787.2303"

}

]

Ось так просто, за допомогою пакету requests, ми одразу отримали готовий результат.

У цьому модулі ми вивчили основи роботи вебзастосунків. Наступним кроком — ми розберемо асинхронну роботу застосунків у веб.