# Конспект лекції (18)

## ▼ Взаємодія з WEB

### ▼ Основні протоколи

Термін "WEB" складається зі слів "World Wide Web", що в перекладі з англійської означає "Всесвітня павутина". Він використовується для опису мережі інтернет-ресурсів, які доступні для користувачів по всьому світу.

Взаємодія з Вебом може відбуватися за допомогою різних протоколів. Ось деякі з найбільш поширених протоколів, які використовуються для взаємодії з веб-серверами:

1. **HTTP (Hypertext Transfer Protocol)**: Це основний протокол для передачі даних через Інтернет. Він використовується для відправки та отримання різноманітних веб-ресурсів, таких як веб-сторінки, зображення, відео та інше.
2. **HTTPS (Hypertext Transfer Protocol Secure)**: Це захищена версія протоколу HTTP, яка використовує шифрування TLS/SSL для захисту конфіденційності та цілісності даних під час передачі через мережу Інтернет.
3. **FTP (File Transfer Protocol)**: Це протокол для передачі файлів між комп'ютерами через мережу Інтернет. Він часто використовується для завантаження або завантаження файлів на або з веб-сервера.
4. **WebSocket**: Це протокол для двостороннього зв'язку між клієнтом та сервером через одне з'єднання TCP. Він дозволяє веб-сторінці отримувати миттєві оновлення з сервера без необхідності постійних запитів.

### ▼ Основні HTTP-операції

У данній лекці мирозглянемо взаємодію з web через протоколи **HTTP/HTTPS** які є найбільш поширеними

Основні HTTP-операції включають в себе:

1. **GET:** Використовується для отримання даних з сервера за певним URI. GET-запити не змінюють стану сервера та повинні бути безпечними, тобто не повинні впливати на дані на сервері. У методу GET тіло запиту не використовується, оскільки дані передаються через параметри URL. Всі дані у методі GET передаються у рядку запиту, розташованому після знаку "?" в URL-адресі.
2. **POST:** Використовується для відправлення даних на сервер для обробки. Цей метод часто використовується для створення нових записів або виконання інших змін, що впливають на стан сервера. Якщо дані присутні вони передаються через тіло запиту.
3. **PUT:** Використовується для оновлення існуючих даних на сервері за певним URI. Він надсилає повний набір даних для заміни на сервері. Якщо дані присутні вони передаються через тіло запиту.
4. **DELETE:** Використовується для видалення даних на сервері за певним URI. Дані згідно специфікації HTTP передаються через параметри ULR адреси

## ▼ Основні HTTP-коди відповідей

HTTP-код (іноді також відомий як код статусу HTTP) - це числовий код, який повертається сервером у відповідь на HTTP-запит. Цей код показує стан виконання запиту та вказує на успішність, помилку або потребу у подальшій дії.

Ось таблиця основних HTTP-кодів

```
+------+----------------------+-------------------------
-----------------+
| Код  | Значення             | Пояснення
|
+------+----------------------+-------------------------
-----------------+
| 2xx  | Успішна відповідь    | Ресурс успішно знайдений
та оброблений   |
|      |                      | сервером.
|
|      |                      |
```

```
|
| 200  | OK (Успішно)         | Успішна відповідь на зап
ит.              |
| 201  | Created (Створено)   | Ресурс успішно створено.
|
| 204  | No Content           | Запит успішний, але нема
є вмісту для     |
|      |                      | відображення.
|
|      |                      |
|
+------+----------------------+-------------------------
-----------------+
| 3xx  | Перенаправлення      | Потрібно зробити додатко
ві дії для       |
|      |                      | завершення запиту.
|
| 301  | Moved Permanently    | Ресурс було переміщено н
азавжди на інший |
|      |                      | URL.
|
| 302  | Found                | Ресурс знайдено, але теп
ер знаходиться   |
|      |                      | під іншим URL.
|
| 304  | Not Modified         | Кешований вміст не був з
мінений.         |
|      |                      |
|
+------+----------------------+-------------------------
-----------------+
| 4xx  | Помилка клієнта      | Помилка в запиті або дос
тупі до ресурсу. |
|      |                      |
|
| 400  | Bad Request          | Некоректний або неповний
```

```
запит.          |
| 401  | Unauthorized         | Потрібна аутентифікація
для доступу до   |
|      |                      | ресурсу.
|
| 403  | Forbidden            | Доступ до ресурсу заборо
нено.            |
| 404  | Not Found            | Ресурс не знайдено.
|
| 405  | Method Not Allowed   | Використано непідтримува
ний HTTP-метод.  |
|      |                      |
|
+------+----------------------+-------------------------
-----------------+
| 5xx  | Помилка сервера      | Помилка в обробці запиту
на сервері.     |
|      |                      |
|
| 500  | Internal Server Error| Помилка в обробці запиту
на сервері.     |
| 502  | Bad Gateway          | Помилка у взаємодії з пр
оміжним сервером.|
| 503  | Service Unavailable  | Сервер тимчасово недосту
пний через       |
|      |                      | перевантаження або обслу
говування.       |
+------+----------------------+-------------------------
-----------------+
```

# ▼ Модуль urllib

## ▼ Введення в urllib

`urllib` - це модуль вбудованої бібліотеки Python, який надає інструменти для роботи з URL-адресами та мережевими операціями.

Основне предназначення `urllib` - це забезпечення можливостей взаємодії з різними ресурсами в Інтернеті через протокол HTTP.

**Основні функції та компоненти `urllib` включають:**

1. **urllib.request:** Цей підмодуль дозволяє виконувати HTTP-запити, отримувати відповіді та обробляти їх. Це використовується для завантаження веб-сторінок, взаємодії з API, а також інших мережевих операцій.
2. **urllib.parse:** Цей підмодуль надає інструменти для розбору та конструювання URL-адрес. Він допомагає управляти компонентами URL, включаючи параметри запиту, шляхи та інші елементи.
3. **urllib.error:** Модуль `urllib.error` містить класи винятків, які можуть виникнути при роботі з `urllib`, наприклад, у випадку помилок HTTP.

**Сфера застосування `urllib` включає:**

1. **Завантаження вмісту з Інтернету:** `urllib` дозволяє вам взаємодіяти з веб-сайтами, отримувати HTML-код сторінок та інші ресурси.
2. **Робота з API:** Модуль `urllib` може використовуватися для взаємодії з веб-сервісами та API, відправляти HTTP-запити та обробляти відповіді.
3. **Розбір та конструювання URL:** `urllib.parse` дозволяє розбирати URL-адреси на їх складові та конструювати URL-адреси з окремих компонентів.
4. **Робота з мережевими ресурсами:** Модуль `urllib` використовується для різноманітних мережевих операцій, таких як відправка та отримання даних через протокол HTTP.

## ▼ urllib.request

`urllib.request` надає можливості для взаємодії з ресурсами через протокол HTTP. За допомогою цього модуля можна виконувати HTTP-запити, отримувати та відправляти дані з веб-ресурсами.

**Основні можливості**

1. **Відправлення GET-запиту:**

```
import urllib.request

url = "https://www.example.com"
response = urllib.request.urlopen(url)
data = response.read()
print(data)
```

Ви здійснюєте GET-запит до вказаної URL-адреси та отримуєте відповідь у вигляді байтів. Це послідовність чисел, де кожне число представляє собою відоме значення байту (від 0 до 255). Для перетворення байтів у рядки потрібно робити декодування

2. **Відправлення POST-запиту:**

```
import urllib.request
import urllib.parse

url = "https://www.example.com"
data = {'key1': 'value1', 'key2': 'value2'}
# Кодуємо дані для POST-запиту
data = urllib.parse.urlencode(data).encode('utf-8')
response = urllib.request.urlopen(url, data)
result = response.read()
print(result)
```

Використовується для відправки POST-запиту з вказаними даними до сервера та отримання результату.

3. **Відправлення PUT-запиту з JSON-даними:**

```
import urllib.request
import urllib.parse
import json

url = "https://www.example.com/resource/123"
data_to_send = {'key1': 'value1', 'key2': 'value2'}
```

```
# Кодування даних в формат JSON
data_encoded = json.dumps(data_to_send).encode('utf-
8')

# Відправлення PUT-запиту з даними у форматі JSON
request = urllib.request.Request(url, method='PUT', d
ata=data_encoded, headers={'Content-Type': 'applicati
on/json'})
with urllib.request.urlopen(request) as f:
    result = f.read()

print(result)
```

4. **Відправлення DELETE-запиту:**

```
import urllib.request
import urllib.parse

url = "https://www.example.com/resource/123"

# Відправлення DELETE-запиту
response = urllib.request.Request(url, method='DELET
E')
with urllib.request.urlopen(response) as f:
    result = f.read()

print(result)
```

5. **Робота з заголовками:**

```
import urllib.request

url = "https://www.example.com"
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.
```

```
0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/58.0.3029.110 Safari/537.3'}

request = urllib.request.Request(url, headers=header
s)
response = urllib.request.urlopen(request)
data = response.read()
print(data)
```

Додає заголовки до запиту, що дозволяє визначити характеристики запиту, такі як User-Agent.

6. **Розбір заголовків і вилучення JSON-даних із відповіді**

   Для розбору заголовків і вилучення JSON-даних із відповіді можна використовувати бібліотеку `json` для розбору JSON та методи, доступні у класі `http.client.HTTPMessage`, для розбору заголовків. Ось простий приклад:

```
import urllib.request
import json

url = "https://www.example.com/api/data"

# Відправка GET-запиту
response = urllib.request.urlopen(url)

# Отримання заголовків відповіді
headers = response.headers

# Отримання JSON-даних з відповіді
json_data = json.loads(response.read().decode('utf-
8'))

# Виведення заголовків та JSON-даних
print("Заголовки:")
print(headers)
```

```
print("\nJSON-дані:")
print(json_data)
```

У цьому прикладі `response.headers` повертає об'єкт типу `http.client.HTTPMessage`, а `json.loads()` використовується для розбору JSON-даних із тіла відповіді.

## ▼ urllib.parse

`urllib.parse` - це модуль, який надає різноманітні функції для роботи з URL-адресами. Ось декілька типових прикладів використання методів `urllib.parse` в Python:

1. **Розбір URL-адресу:**

```
from urllib.parse import urlparse

url = "https://www.example.com/path/to/resource?param
1=value1&param2=value2"

parsed_url = urlparse(url)

print("Схема:", parsed_url.scheme)
print("Мережева адреса:", parsed_url.netloc)
print("Шлях:", parsed_url.path)
print("Параметри:", parsed_url.params)
print("Запити:", parsed_url.query)
print("Фрагмент:", parsed_url.fragment)
```

2. **Складання URL-адреси:**

```
from urllib.parse import urlunparse

scheme = "https"
netloc = "www.example.com"
path = "/path/to/resource"
```

```
params = "param1=value1"
query = "param2=value2"
fragment = "section"

composed_url = urlunparse((scheme, netloc, path, para
ms, query, fragment))

print("Складений URL:", composed_url)
```

3. **Кодування та декодування параметрів для URL:**

   Кодування URL (URL encoding або percent encoding) - це процес перетворення спеціальних символів та нелатинських символів у кодовий формат, який може бути безпечно використаний у URL-адресі. Кодування URL дозволяє вставляти символи, які мають спеціальне значення у контексті URL, такі як пробіл, знак питання, амперсанд і т.д.

   В кодуванні URL використовується спеціальний формат, де кожен символ, що потребує кодування, представляється символом "%" та його ASCII-кодом у шістнадцятковій системі.

   Наприклад, символ пробілу (' ') у URL-кодуванні стане "%20", а символ "&" буде представлений як "%26".

```
from urllib.parse import urlencode

params = {'param1': 'value 1', 'param2': 'value&2',
'param3': 'value3'}

encoded_params = urlencode(params)

print("Закодовані параметри:", encoded_params)
```

   URL декодування (URL decoding або percent decoding) - це процес перетворення закодованих символів у їх оригінальний вигляд в контексті URL-адресу. Коли URL містить закодовані символи

(наприклад, `%20` замість пробілу або `%26` замість символу "&"), URL декодування використовується для відновлення оригінального вигляду символів.

```
from urllib.parse import quote, unquote

original_string = "This is a string with spaces and s
pecial characters !@#$%^&*"

# Кодування
encoded_string = quote(original_string)
print("Закодований рядок:", encoded_string)

# Декодування
decoded_string = unquote(encoded_string)
print("Декодований рядок:", decoded_string)
```

4. **Розбір рядка запиту в словник:**

```
from urllib.parse import parse_qs

query_string = "param1=value1&param2=value2&param3=va
lue3"

query_params = parse_qs(query_string)

print("Розбиті параметри:", query_params)
```

## ▼ urllib.error

типові приклади використання модуля `urllib.error` в Python для обробки помилок та статус кодів відповіді HTTP:

1. **Обробка HTTPError:**

```
from urllib.request import urlopen, Request
from urllib.error import HTTPError
```

```
url = "https://www.example.com/nonexistent_page"

try:
    response = urlopen(url)
except HTTPError as e:
    print(f'HTTP Error: {e.code} {e.reason}')
    # Тут можливо виконати додаткові дії в залежності
від коду помилки
```

2. **Обробка URLError:**

```
from urllib.request import urlopen, Request
from urllib.error import URLError

url = "https://www.example-nonexistent-domain.com"

try:
    response = urlopen(url)
except URLError as e:
    print(f'URL Error: {e.reason}')
    # Тут можливо виконати додаткові дії в
    # залежності від причини помилки
```

Помилка `URLError` генерується в різних ситуаціях, коли сталася помилка при взаємодії з URL (Unified Resource Locator) в мові програмування Python. Ось декілька типових випадків, коли може виникнути `URLError`:

- Помилка з'єднання
- Таймаут з'єднання
- Невірна URL-адреса
- Відсутність з'єднання з Інтернетом

3. **Обробки HTTP-кодів, які не є успішними (коди помилок):**

Ви можете використовувати модуль `urllib.error` для обробки HTTP-кодів, які не є успішними (коди помилок). Основний клас, який ви можете використовувати для обробки цих помилок, - це `HTTPError`. Цей клас генерується, коли сервер повертає HTTP-код відмови (4xx або 5xx).

Ось приклад, як ви можете обробляти HTTP-коди помилок за допомогою `urllib.error.HTTPError`:

```
pythonCopy code
from urllib.request import urlopen, Request
from urllib.error import HTTPError

url = "https://www.example.com/nonexistent_page"

try:
    response = urlopen(url)
except HTTPError as e:
    print(f'HTTP Error: {e.code} {e.reason}')

    # Обробка конкретних кодів помилок
    if e.code == 404:
        print("Сторінка не знайдена.")
    elif e.code == 403:
        print("Доступ заборонено.")
    else:
        print("Інша HTTP-помилка.")
```

У цьому прикладі, якщо сервер повертає HTTP-код помилки (наприклад, 404 для "сторінка не знайдена"), ви можете визначити свою обробку для цього конкретного коду помилки.

Також, можна розширити обробку помилок для включення кодів, які не є успішними, через змінну `e.code` у блоку `except HTTPError`.

## ▼ Робота з самопідписаними сертифікатами

При використанні бібліотеки `urllib` для взаємодії з HTTPS-ресурсами, вам не потрібно явно виконувати перевірку сертифікатів. За замовчуванням, бібліотека `urllib` використовує системний магазин сертифікатів для перевірки достовірності SSL/TLS-з'єднань.

Проте, якщо ви стикаєтеся з ситуацією, де потрібно докласти додаткових зусиль для перевірки сертифікатів (наприклад, у випадках власного самопідписаного сертифіката або специфічних вимог безпеки), ви можете використовувати бібліотеку `ssl` для власноручної конфігурації перевірки сертифікатів.

Ось приклад, як ви можете це зробити:

```
import ssl
from urllib.request import urlopen, Request

# Вимкнення перевірки сертифікатів (не рекомендується в
продакшні)
context = ssl.create_default_context()
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE

url = "https://www.example.com"

try:
    # Вказання контексту SSL при використанні urllib
    response = urlopen(Request(url), context=context)
    print(response.read())
except Exception as e:
    print(f'Помилка: {e}')
```

Цей приклад використовує `ssl.create_default_context()` для отримання об'єкту контексту SSL. Параметри `check_hostname` та `verify_mode` встановлюються так, щоб вимкнути перевірку сертифікатів.

> 💡 Однак, важливо зауважити, що вимкнення перевірки сертифікатів - це потенційна безпека, і в цілому, рекомендується використовувати системний набір сертифікатів. У реальному проекті рекомендується використовувати коректний та довірений сертифікат для забезпечення безпеки передачі даних.

## ▼ Обмеження у використанні

`urllib` в Python має деякі обмеження та особливості, які слід враховувати при його використанні:

1. **Одночасність:**
   В
   `urllib` вбудовано підтримку обмеженого числа одночасних з'єднань. Це означає, що якщо ви виконуєте багато запитів паралельно, може виникнути блокування.
2. **Базовий функціонал:** `urllib` надає базовий функціонал для роботи з URL-адресами, відкриття з'єднань та отримання відповідей. Однак в порівнянні з більш високорівневими бібліотеками, такими як `requests`, може бути менше зручний та повільний.
3. **Спрощений інтерфейс:**
   Інтерфейс
   `urllib` може виглядати менш зручним порівняно з іншими бібліотеками, такими як `requests`, які надають більше зручний та зрозумілий API.
4. **Відсутність автоматичного редіректу:**
   В
   `urllib` вам потрібно самостійно обробляти редіректи, якщо вони виникають. В більш високорівневих бібліотеках, наприклад у `requests`, це здійснюється автоматично.
5. **Не повністю сумісний з Python 2:**
   Деякі функції
   `urllib` можуть відрізнятися або взагалі відсутні у версії Python 2. Рекомендується використовувати його в середовищі Python 3.

6. **Недостатня підтримка cookies:**
   Робота з cookies в
   `urllib` менш зручна та гнучка порівняно з іншими бібліотеками, інколи ви можете зіткнутися з обмеженнями при роботі з cookies.

# ▼ Модуль requests

## ▼ Введення в requests

Бібліотека `requests` в мові програмування Python є потужним інструментом для виконання HTTP-запитів і взаємодії з веб-ресурсами. У контексті автоматичного тестування, ця бібліотека використовується для надання можливостей тестування веб-застосунків шляхом взаємодії з їхнім API або іншими веб-сервісами. Основні ролі бібліотеки `requests` в автоматичному тестуванні включають:

1. **Виконання HTTP-запитів:** Бібліотека `requests` дозволяє легко виконувати різноманітні HTTP-запити, такі як GET, POST, PUT, DELETE і багато інших. Це особливо корисно при тестуванні веб-сервісів, де тести відправляють HTTP-запити для спостереження і перевірки поведінки сервера.
2. **Налаштування параметрів запиту:** Бібліотека дозволяє визначати різноманітні параметри запиту, такі як заголовки, параметри URL, тіло запиту і т.д. Це особливо корисно для тестування різноманітних сценаріїв і перевірки реакції сервера на різні умови.
3. **Обробка відповідей сервера:** За допомогою бібліотеки `requests` можна легко отримувати і аналізувати відповіді від сервера. Це включає в себе перевірку статус-кодів, аналіз тіла відповіді, роботу з заголовками та інші аспекти взаємодії з сервером.
4. **Сесії та кукіси:** Бібліотека підтримує використання сесій для збереження стану між різними запитами, а також дозволяє працювати з кукісами. Це особливо важливо при виконанні послідовних запитів або автентифікації на веб-ресурсах під час тестування.
5. **Обробка помилок:** За допомогою `try/except` конструкцій, бібліотека дозволяє обробляти помилки, які можуть виникнути під час

виконання запитів. Це спрощує написання стійких тестів, які можуть адекватно реагувати на різноманітні ситуації.

Враховуючи ці можливості, бібліотека `requests` стає потужним інструментом для автоматизованого тестування web-apps, дозволяючи розробникам легко створювати та виконувати тести, що взаємодіють з веб-сервісами.

> 💡 Щоб використовувати бібліотеку `requests`, ви повинні встановити її за допомогою `pip install requests`.

При роботі

- Звертайте увагу на статус-коди відповідей сервера для адекватної обробки помилок.
- Для отримання та відправлення даних у форматі JSON, можна використовувати методи `json()` бібліотеки `requests`.

## ▼ Методи GET / POST / PUT / DELETE

Приклади використання бібліотеки `requests` для різних типів запитів:

GET, POST, PUT та DELETE.

1. **GET-запит:**

```
import requests

url = 'http://jsonplaceholder.typicode.com/posts/1/co
mments'
response = requests.get(url)

# Перевірка статус-коду
if response.status_code == 200:
    data = response.json()  # отримання даних у форма
ті JSON
    print('Отримано дані:', data)
else:
```

```
    print('Помилка. Статус-код:', response.status_cod
e)
```

**GET-запиті з параметрами**

```
import requests

url = 'http://jsonplaceholder.typicode.com/posts/1/co
mments'
params = {'postId': 1, 'email': 'Nikita@garfield.bi
z'}

response = requests.get(url, params=params)

if response.status_code == 200:
    data = response.json()
    print(data)
else:
    print('Помилка запиту:', response.status_code)
```

2. **POST-запит:**

```
import requests

url = 'http://jsonplaceholder.typicode.com/posts'
data_to_send = {'userId': 10, 'id': 101, 'title': 'So
me title'}

response = requests.post(url, data=data_to_send)

# Перевірка статус-коду
if response.status_code == 201:
    created_data = response.json()  # отримання даних
у форматі JSON
    print('Створено дані:', created_data)
```

```
else:
    print('Помилка. Статус-код:', response.status_cod
e)
```

3. **PUT-запит:**

```
import requests

url = 'http://jsonplaceholder.typicode.com/posts/1'
data_to_send = {'userId': 10, 'id': 101, 'title': 'Ne
w title'}

response = requests.put(url, data=data_to_send)

# Перевірка статус-коду
if response.status_code == 200:
    updated_data = response.text  # отримання даних у
форматі текст
    print('Оновлено дані:', updated_data)
else:
    print('Помилка. Статус-код:', response.status_cod
e)
```

4. **DELETE-запит:**

```
import requests

url = 'http://jsonplaceholder.typicode.com/posts/1'
params = {'userId': 10, 'id': 101, 'title': 'New titl
e'}
response = requests.delete(url, params=params)

# Перевірка статус-коду
if response.status_code == 200:
    print('Дані успішно видалено')
```

```
else:
    print('Помилка. Статус-код:', response.status_cod
e)
```

## ▼ Формування запитів

Для формування запитів за допомогою бібліотеки `requests` в Python ви можете використовувати різні методи та аргументи для відправки різних типів запитів. Ось детальний огляд того, як формувати запити та які дані та параметри можна додавати:

1. **Використання методів запитів:**
   - `get(url, params=None, **kwargs)`: Відправляє GET-запит на вказаний URL з опціональними параметрами.
   - `post(url, data=None, json=None, **kwargs)`: Відправляє POST-запит на вказаний URL з опціональними даними форми або JSON-даними.
   - `put(url, data=None, **kwargs)`: Відправляє PUT-запит на вказаний URL з опціональними даними.
   - `delete(url, **kwargs)`: Відправляє DELETE-запит на вказаний URL.
2. **Додавання даних до запитів:**
   - `params`: Дані параметрів для GET-запитів.
   - `data`: Дані форми для POST- або PUT-запитів у текстовому вигляді.
   - `json`: JSON-дані для POST- або PUT-запитів.
   - `files`: Файли для відправлення на сервер.
3. **Модифікація параметрів запиту:**
   - `headers`: Заголовки запиту, такі як User-Agent, Content-Type і т.д.
   - `timeout`: Максимальний час очікування відповіді сервера.
   - `auth`: Аутентифікаційні дані для авторизації на сервері.
   - Інші параметри, такі як `cookies`, `proxies`, `allow_redirects` та інші, що дозволяють налаштовувати різні аспекти запитів.

### User-Agent

Заголовок "User-Agent"  відправляється серверу веб-сайту, щоб ідентифікувати програму або агента, який відправляє запит. Цей заголовок містить інформацію про тип програми, її версію та операційну систему, за допомогою якої вона працює. Використання цього заголовка допомагає серверам веб-сайтів відповідати на запити користувачів, а також адаптувати вміст відповідно до характеристик агента, що надіслав запит. Деякі веб-сайти можуть надавати різний контент або блокувати запити від певних типів клієнтів

### Content-Type

Заголовок "Content-Type" вказує на тип медіа-вмісту, який відправляється . Цей заголовок вказує серверу або клієнту, який отримує відповідь, який тип даних очікується. Наприклад, тип може бути `"text/html"` для веб-сторінки HTML або `"application/json"` для обміну даними у форматі JSON.

Використання заголовка "Content-Type" допомагає визначити, як правильно обробляти вміст запиту.

Приклад POST-запиту з модифікацією заголовків, відправленням файлу та коригуванням timeout:

```
import requests

url = 'https://api.example.com/data'
headers = {'User-Agent': 'Мій агент', 'Content-Type': 'a
pplication/json'}
file = {'file': open('filename.txt', 'rb')}
timeout = 10

resp = requests.post(url, headers=headers, files=file, t
imeout=timeout)
```

У цьому прикладі використовуються такі параметри:

- `headers` - для встановлення заголовків запиту, включаючи "User-Agent" та "Content-Type".
- `files` - для відправлення файлу на сервер.
- `timeout` - для встановлення максимального часу очікування відповіді в 10 секунд.

## ▼ Обробка помилок

При використанні бібліотеки `requests` в Python можуть виникати різні види помилок під час взаємодії з сервером. Ось деякі типові помилки, які можуть виникати, та як їх обробляти:

1. **HTTP помилки:**
   - **HTTPError:** Ця помилка виникає, коли отримуємо неправильну відповідь від сервера, наприклад, статусний код не успішний (не 2xx).

```
import requests
from requests.exceptions import HTTPError

try:
    response = requests.get('https://example.com')
    response.raise_for_status()  # Викликає виняток,
якщо код не 2xx
except HTTPError as e:
    print('HTTP Помилка:', e)
```

2. **Помилки з'єднання:**
   - **ConnectionError:** Виникає, коли не вдається підключитися до сервера.

```
import requests
from requests.exceptions import ConnectionError
```

```
try:
    response = requests.get('https://example.com')
except ConnectionError as e:
    print('Помилка з\'єднання:', e)
```

3. **Часові помилки:**
   - **Timeout:** Виникає, коли перевищено час очікування відповіді від сервера.

```
import requests
from requests.exceptions import Timeout

try:
        # Встановлення таймауту
    response = requests.get('https://example.com', ti
meout=5)
except Timeout as e:
    print('Часова помилка:', e)
```

4. **Інші помилки:**
   Різні інші помилки можуть виникати залежно від конкретного контексту. Для їх обробки можна використовувати загальний блок обробки винятків або спеціальні блоки для конкретних типів помилок.

## ▼ Обробка відповідей

Щоб обробляти відповіді від сервера за допомогою бібліотеки `requests` в Python, ви можете скористатися різними методами та властивостями, які вона надає. Ось деякі з них:

1. **Статус коду відповіді:**
   Ви можете отримати статус коду відповіді за допомогою властивості
   `status_code`. Це дозволяє вам перевірити успішність або неуспішність запиту. Наприклад:

```
import requests

response = requests.get('https://api.example.com/dat
a')
if response.status_code == 200:
    print('Успішний запит!')
else:
    print('Не вдалося зробити запит. Код помилки:', r
esponse.status_code)
```

2. **Вміст відповіді:**
   Ви можете отримати вміст відповіді за допомогою властивості `text`, `content` або `json`, залежно від формату вмісту. Наприклад:

```
import requests

response = requests.get('https://api.example.com/dat
a')
if response.status_code == 200:
    data = response.json()
    print('Отримані дані:', data)
else:
    print('Не вдалося отримати дані. Код помилки:', r
esponse.status_code)
```

3. **Особливості користування методом `json()`**

   `json()` у бібліотеці `requests` в Python дозволяє легко отримувати та відправляти дані у форматі JSON при роботі з HTTP-запитами.

> 💡 Метод `json()` автоматично викликається для відповіді у вигляді JSON, він використовує вбудовану бібліотеку `json` для перетворення рядка JSON у структуру даних Python (наприклад, словник). Саме тому потрібно робити додаткову обробку помилок при `json` серіалізації/десеріалізації даних

Приклад:

```
import requests
import json

url = 'https://api.example.com/data'

# Спроба відправити GET-запит та отримати JSON-відпов
ідь
try:
    response = requests.get(url)
    response.raise_for_status()  # Викидає виняток, я
кщо статус-код не в діапазоні 2xx

    # Спроба отримати JSON дані з відповіді
    try:
        data = response.json()
        print('Отримано дані:', data)
    except json.JSONDecodeError as e:
        print('Помилка при серіалізації JSON:', e)

except requests.exceptions.RequestException as e:
    print('Помилка при виконанні запиту:', e)
```

## ▼ request.Session

Клас `requests.Session` використовується для створення сесій при взаємодії з веб-сайтами за допомогою бібліотеки `requests` в Python. Він

надає зручний спосіб для виконання кількох запитів до одного сервера, зберігаючи при цьому параметри сеансу, такі як cookies, заголовки та інші.

Основні переваги використання `requests.Session` включають:

1. **Збереження cookies:** Сесія автоматично зберігає та передає cookies між різними запитами, що дозволяє підтримувати авторизацію та інші параметри сеансу.
2. **Збереження заголовків:** Усі налаштування, встановлені для сесії, залишаються активними для кожного запиту, що дозволяє зручно налаштовувати заголовки та інші параметри.
3. **Ефективне використання ресурсів:** Створення сесії дозволяє уникнути зайвого з'єднання з сервером для кожного запиту, що поліпшує продуктивність та швидкість взаємодії з сервером.

Ось приклад використання `requests.Session`:

```
import requests

# Створення сесії
session = requests.Session()

# Встановлення cookies для сесії
cookies = {'user_id': '12345', 'session_id': 'abcdef'}
session.cookies.update(cookies)

# Виконання запитів за допомогою сесії
response1 = session.get('https://example.com/page1')
response2 = session.get('https://example.com/page2')
```

Отже, `requests.Session` дозволяє зберігати та обмінюватися даними сесії між різними запитами, що робить взаємодію з веб-сайтами більш ефективною та зручною.

### HTTPAdapter

Клас `requests.adapters.HTTPAdapter` використовується для налаштування адаптерів, які використовуються для виконання HTTP-запитів за допомогою бібліотеки `requests` в Python.

Ось декілька прикладів його використання разом з `requests.Session`:

- **Налаштування таймауту:**
  Встановлення таймауту для HTTP-запитів, який вказує максимальний час очікування відповіді від сервера.

```
import requests
from requests.adapters import HTTPAdapter

# Створення сесії
session = requests.Session()

# Налаштування адаптера з таймаутом 10 секунд
adapter = HTTPAdapter(timeout=10)

# Додавання адаптера до сесії
session.mount('http://', adapter)
session.mount('https://', adapter)

# Виконання запиту з встановленим таймаутом
response = session.get('https://example.com')
```

- **Налаштування кількості спроб:**
  Встановлення кількості спроб повторити HTTP-запит у випадку невдалого підключення або інших помилок.

```
import requests
from requests.adapters import HTTPAdapter

# Створення сесії
session = requests.Session()

# Налаштування адаптера з кількістю спроб 3
```

```
adapter = HTTPAdapter(max_retries=3)

# Додавання адаптера до сесії
session.mount('http://', adapter)
session.mount('https://', adapter)

# Виконання запиту з автоматичною повторною спробою
response = session.get('https://example.com')
```

## ▼ Аутентифікація

У бібліотеці `requests` в Python є кілька способів виконання аутентифікації при взаємодії з веб-серверами. Ось декілька способів налаштування аутентифікації:

1. **Basic Auth:**
   Використовується для передачі ім'я користувача та пароля у заголовку запиту.

```
import requests

response = requests.get('https://api.example.com', au
th=('username', 'password'))
```

2. **Digest Auth:**
   Використовується для передачі ім'я користувача та пароля у вигляді хешу у заголовку запиту.

```
import requests

from requests.auth import HTTPDigestAuth

response = requests.get('https://api.example.com', au
th=HTTPDigestAuth('username', 'password'))
```

3. **Bearer Token:**
   Використовується для передачі токена доступу у заголовку запиту.

```
import requests
my_token = 'my_secret_token'
headers = {'Authorization': f'Bearer {my_token}'}
response = requests.get('https://api.example.com', he
aders=headers)
```

## ▼ Відправка та скачування файлів

Щоб виконати відправка файлу (file upload) або скачати файл (file download) за допомогою бібліотеки `requests` в Python, вам потрібно використовувати методи `post()` або `get()` відповідно. Ось як це можна зробити:

- **File Upload (Відправка файлу):**

```
import requests

# Відкриття файлу для завантаження
with open('file_to_upload.txt', 'rb') as file:
    files = {'file': file}

    # Виконання POST-запиту з файлом
    response = requests.post('https://example.com/uploa
d', files=files)

# Обробка відповіді
print(response.text)
```

- **File Download (Скачування файлу):**

```
import requests

# Виконання GET-запиту для скачування файлу
response = requests.get('https://example.com/download')

# Запис файлу на диск
```

```
with open('downloaded_file.txt', 'wb') as file:
    file.write(response.content)
```

У першому прикладі, файл `file_to_upload.txt` відкривається у режимі бінарного читання (rb) для завантаження. Він передається у POST-запиті як частина `files`. Після виконання запиту, ви можете обробити відповідь за необхідності.

У другому прикладі, ви виконуєте GET-запит для отримання вмісту файлу з веб-сервера. Після отримання відповіді, ви записуєте отриманий вміст у файл `downloaded_file.txt` на вашому локальному диску.

### ▼ Робота з самопідписаними сертифікатами

Для роботи з самопідписаними сертифікатами  вам необхідно передати параметр `verify` зі значенням `False` при виклику методу `requests.get()` або `requests.post()`. Це дозволить ігнорувати перевірку сертифікатів при взаємодії з сервером.

Наприклад:

```
import requests

response = requests.get('https://your-server.com', verif
y=False)
print(response.text)
```

Проте, важливо мати на увазі, що використання параметра `verify=False` може створювати потенційні ризики з точки зору безпеки, оскільки не перевіряється достовірність сертифікатів сервера.

## ▼ Порівняння requests vs urlib

| **Характеристика** | `requests` | `urllib` |
|---|---|---|
| **Простота використання** | + Зручний та лаконічний синтаксис | + Більше коду для виконання тих самих завдань |
| **Підтримка сесій** | + Підтримка сесій для збереження стану між | - Потребує додаткового коду для реалізації сесій |

| | | |
|---|---|---|
| | запитами | |
| **Обробка Cookies** | + Зручна робота з Cookies | - Вимагає вручну обробку Cookies |
| **Робота з URL-параметрами** | + Проста передача параметрів через `params` | - Вимагає вручну складання URL-параметрів |
| **Робота з JSON** | + Вбудована підтримка JSON | - Вимагає додаткового коду для роботи з JSON |
| **Обробка винятків HTTP** | + Зручне керування помилками HTTP | - Може бути складніше управління помилками |
| **SSL-підтримка** | + Вбудована підтримка SSL | + Потребує додаткового коду для SSL |
| **Розширені можливості** | + Широкий функціонал, такий як автоматична перевірка SSL-сертифікатів, потокове завантаження | + Більш розширені можливості, але вимагає більше коду |
| **Швидкодія** | + Зазвичай швидший через оптимізації та вбудовані оптимізації | - Може бути менш ефективним через більше коду та менші оптимізації. +Але для простих випадків може бути трохи швидше, оскільки вона вбудована безпосередньо у стандартну бібліотеку Python |

Вважливо відзначити, що бібліотека `requests` надає багато зручностей, що дозволяє легше і зручніше виконувати різноманітні завдання з HTTP, та в багатьох випадках вона може бути більш зручною для роботи.

Але існують випадки, коли використання бібліотеки `urllib` може бути обгрунтованим в порівнянні із бібліотекою `requests`. Ось деякі з них:

1. **Вбудовані можливості Python**: `urllib` входить до стандартної бібліотеки Python, що означає, що вона буде доступна за замовчуванням при встановленні Python. Якщо проект обмежений використанням стандартної бібліотеки, то `urllib` може бути вибором.

2. **Легкість інтеграції з іншими модулями**: У вас може бути ситуація, коли необхідно легко інтегрувати роботу з HTTP-запитами в існуючий код, і використання `urllib` може бути вигідним з цієї точки зору.
3. **Обмеження ресурсів / швидкодія при простих запитах** ： Для оптимізації швидкодії у випадках великої кількості простих запитів або обмежених ресурсів може бути розумним вибрати `urllib` оскільки вона вбудована безпосередньо у стандартну бібліотеку Python і не потребує встановлення додаткових модулей