Асинхронне програмування: Теорія та Концепції

1.1. Що таке Асинхронне програмування?

Асинхронне програмування — це парадигма програмування, яка дозволяє програмі виконувати завдання, які можуть зайняти багато часу (наприклад, мережеві запити, операції з файлами, запити до бази даних), не блокуючи основний потік виконання. Замість того, щоб чекати завершення однієї операції, програма може переключитися на іншу, а потім повернутися до першої, коли вона буде готова.

Ключові ідеї:

- **Неблокуючий ввід/вивід (Non-blocking I/O):** Коли програма виконує операцію вводу/виводу, вона не "зависає", чекаючи на результат. Вона "відпускає" управління, дозволяючи іншим частинам коду виконуватися.
- **Однопотокова конкурентність:** Асинхронний код зазвичай виконується в одному потоці, але створює ілюзію паралелізму, ефективно використовуючи час очікування на I/O.

1.2. Синхронне vs. Асинхронне програмування

- **Синхронне (Synchronous) програмування:**
  - Код виконується послідовно, рядок за рядком.
  - Кожна операція повинна завершитися, перш ніж почнеться наступна.
  - **Блокуюче:** Якщо одна операція (наприклад, мережевий запит) займає багато часу, вся програма "зависає", чекаючи на її завершення.
  - *Приклад:* Виклик зовнішнього API, який займає 5 секунд, заблокує виконання всього коду на ці 5 секунд.
- **Асинхронне (Asynchronous) програмування:**
  - Код може "призупиняти" виконання однієї операції, перемикатися на іншу, а потім повертатися до призупиненої, коли вона готова.
  - **Неблокуюче:** Дозволяє ефективно використовувати час, який витрачається на очікування (наприклад, відповіді від мережі).
  - *Приклад:* Виклик зовнішнього API може бути ініційований, а поки відповідь чекає, програма може обробляти інші запити або виконувати інші завдання.

1.3. Asyncio vs. Багатопоточність (Multithreading)

Обидва підходи використовуються для досягнення конкурентності, але вони працюють по-різному:

- **Багатопоточність (threading):**
  - Використовує кілька потоків виконання, які можуть працювати паралельно (навіть якщо в Python це обмежено GIL для CPU-bound завдань).
  - Потоки ділять спільну пам'ять, що вимагає механізмів синхронізації (Locks, Semaphores) для уникнення станів гонки.
  - Ефективна для I/O-bound завдань, оскільки GIL звільняється під час операцій вводу/виводу.
- **Асинхронне програмування (asyncio):**
  - Виконується в **одному потоці** (зазвичай).
  - Досягає конкурентності за допомогою **перемикання контексту** між "корутинами" (coroutines), коли одна корутина чекає на I/O.
  - Не має проблем зі станами гонки, пов'язаними зі спільною пам'яттю між потоками (оскільки потік один), але все ще потребує уваги до спільних ресурсів, якщо вони модифікуються.
  - **Ідеально підходить для I/O-bound завдань**, оскільки Python GIL не є перешкодою, адже ми не намагаємося виконувати CPU-інтенсивні операції паралельно.

1.4. Цикл подій (Event Loop)

Цикл подій (Event Loop) — це серце будь-якої асинхронної програми. Це нескінченний цикл, який:

1. Чекає на події (наприклад, дані надійшли з мережі, файл готовий до читання, таймер спрацював).
2. Коли подія відбувається, він передає управління відповідній "корутині" (coroutine).
3. Коли корутина "призупиняється" (наприклад, чекаючи на I/O), цикл подій перемикається на іншу готову корутину.

1.5. Ключові слова async та await

Python 3.5+ представив синтаксис async та await для визначення та керування корутинами.

- **async def**: Визначає **корутину (coroutine)**. Це функція, яка може бути призупинена та відновлена.
  - Корутини не виконуються одразу при виклику; вони повертають об'єкт корутини, який потрібно "запустити" в циклі подій.
- **await**: Використовується **всередині корутини** для "призупинення" її виконання, доки не завершиться інша "очікувана" (awaitable) операція (наприклад, інша корутина, asyncio.sleep, мережевий запит).

    - Коли await призупиняє поточну корутину, цикл подій може переключитися на виконання інших корутин.

Приклад:

```
import asyncio

async def my_coroutine():
    print("Починаю корутину...")
    await asyncio.sleep(1) # Призупиняю виконання на 1 секунду
    print("Корутина завершена.")

# Щоб запустити корутину, потрібен цикл подій
# asyncio.run(my_coroutine())
```

1.6. Модуль asyncio

Модуль asyncio є вбудованою бібліотекою Python для написання конкурентного коду за допомогою синтаксису async/await.

Основні функції та класи:

- **asyncio.run(coroutine)**:
    - Це основна точка входу для запуску асинхронної програми.
    - Вона створює новий цикл подій, запускає передану корутину, а потім закриває цикл подій після завершення корутини.
    - Викликається звичайним (синхронним) кодом.
- **await asyncio.sleep(delay)**:
    - Це асинхронна версія time.sleep().
    - Вона призупиняє поточну корутину на вказану кількість секунд, але **не блокує** весь потік виконання. Цикл подій може виконувати інші корутини в цей час.
- **asyncio.create_task(coroutine)**:
    - Запускає корутину як "завдання" (Task) у фоновому режимі.
    - Завдання виконуються конкурентно з іншими завданнями в циклі подій.
    - Повертає об'єкт Task, який можна await для отримання результату або для очікування завершення.
- **await asyncio.gather(*coroutines_or_tasks)**:
    - Запускає кілька корутин або завдань конкурентно і чекає, доки всі вони завершаться.

    - Повертає список результатів у порядку, в якому були передані корутини/завдання.
    - Дуже корисний для паралельного виконання кількох I/O-операцій.

1.7. Переваги asyncio для веб-розробників

- **Висока конкурентність:** Можливість обробляти тисячі одночасних клієнтських з'єднань з мінімальними ресурсами.
- **Ефективність I/O:** Ідеально підходить для веб-додатків, які багато взаємодіють з базою даних, зовнішніми API, файловою системою, оскільки ці операції є I/O-bound.
- **Масштабованість:** Дозволяє будувати сервіси, які можуть легко масштабуватися для обробки великого навантаження.
- **Простота коду:** Хоча синтаксис async/await вимагає звикання, він часто робить конкурентний код більш читабельним та легким для розуміння, ніж багатопотоковий код з локами та іншими примітивами синхронізації.
- **Сучасні фреймворки:** Багато сучасних Python веб-фреймворків (FastAPI, Starlette, Sanic, aiohttp) побудовані на asyncio.

Практичні Приклади Асинхронного програмування

```
import asyncio
import time
import datetime
import random
```

Приклад 1: Базове використання async/await та asyncio.run()

Визначення простої корутини та її запуск.

```
async def say_hello_async(name):
    """Асинхронна функція для привітання."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Привіт,
{name}! (Початок)")
    await asyncio.sleep(1) # Неблокуюча затримка
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Привіт,
{name}! (Кінець)")
    return f"Привітання для {name} завершено."

async def main_example_1():
    """Основна корутина для Прикладу 1."""
    print("Запускаю Приклад 1...")
    result = await say_hello_async("Світ")
    print(f"Результат з корутини: {result}")
    print("Приклад 1 завершено.")

asyncio.run(main_example_1())
```

Завдання до Прикладу 1: Базове async/await

Створіть асинхронну функцію `greet_with_delay(name, delay)`, яка виводить привітання, чекає `delay` секунд, а потім виводить повідомлення про завершення.

Приклад 2: Запуск кількох корутин конкурентно з asyncio.gather()

Демонструє, як кілька I/O-bound завдань виконуються "паралельно".

```
async def perform_io_task(task_id, duration):
    """Імітує I/O-операцію з певною тривалістю."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Завдання
{task_id}: Починаю (тривалість {duration}с).")
    await asyncio.sleep(duration)
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Завдання
{task_id}: Завершено.")
    return f"Результат завдання {task_id}"

async def main_example_2():
    """Основна корутина для Прикладу 2."""
    print("Запускаю Приклад 2 (конкурентно)...")
    start_time = time.perf_counter()

    # Запускаємо три корутини конкурентно
    results = await asyncio.gather(
```

```
        perform_io_task(1, 2), # Завдання 1 займе 2 секунди
        perform_io_task(2, 1), # Завдання 2 займе 1 секунду
        perform_io_task(3, 1.5) # Завдання 3 займе 1.5 секунди
    )

    end_time = time.perf_counter()
    print(f"Всі завдання Прикладу 2 завершено за {end_time -
start_time:.2f} секунд.")
    print(f"Отримані результати: {results}")

# Для порівняння, як би це було послідовно:
async def main_example_2_sequential():
    """Основна корутина для Прикладу 2 (послідовно)."""
    print("\nЗапускаю Приклад 2 (послідовно)...")
    start_time = time.perf_counter()

    await perform_io_task(1, 2)
    await perform_io_task(2, 1)
    await perform_io_task(3, 1.5)

    end_time = time.perf_counter()
    print(f"Всі завдання Прикладу 2 завершено послідовно за {end_time
- start_time:.2f} секунд.")
```

```
asyncio.run(main_example_2())
asyncio.run(main_example_2_sequential())
```

Завдання до Прикладу 2: Запуск кількох корутин конкурентно

Імітуйте 3 асинхронні "API-виклики", кожен з яких має випадкову затримку від 0.5 до 2.0 секунд. Використайте `asyncio.gather()` для їх конкурентного виконання. Виведіть загальний час виконання.

Приклад 3: Імітація конкурентних веб-запитів (без aiohttp)

Використання `asyncio.sleep` для імітації затримки мережі при завантаженні кількох URL.

```
async def fetch_url_mock(url):
    """Імітує асинхронне завантаження даних з URL."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Починаю завантаження: {url}")
    await asyncio.sleep(random.uniform(0.5, 3.0)) # Імітація мережевої затримки
```

```
    data = f"Отримано дані з {url}"
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Завершив завантаження: {url}")
    return data

async def main_example_3():
    """Основна корутина для Прикладу 3."""
    print("Запускаю Приклад 3 (конкурентні веб-запити)...")
    urls_to_fetch = [
        "https://api.example.com/users",
        "https://api.example.com/products",
        "https://api.example.com/orders",
        "https://api.example.com/dashboard"
    ]
    start_time = time.perf_counter()

    # Створюємо список завдань
    tasks = [fetch_url_mock(url) for url in urls_to_fetch]

    # Чекаємо на завершення всіх завдань
    results = await asyncio.gather(*tasks)

    end_time = time.perf_counter()
    print(f"Всі веб-запити Прикладу 3 завершено за {end_time -
start_time:.2f} секунд.")
    print("Отримані дані:")
    for res in results:
        print(f"  - {res}")

asyncio.run(main_example_3())
```

Завдання до Прикладу 3: Імітація конкурентних веб-запитів

Імітуйте конкурентне отримання даних про 4 користувачів з "бази даних". Кожен запит до БД має імітувати затримку від 0.2 до 1.5 секунд. Виведіть повідомлення про початок та завершення отримання даних для кожного користувача, а також загальний час виконання.

Приклад 4: Імітація асинхронного клієнта, що взаємодіє з "API"

Демонструє, як асинхронний клієнт може надсилати кілька запитів.

```
# Імітація асинхронного "API-сервісу"
async def mock_api_service(request_data):
    """Імітує асинхронну обробку запиту на стороні сервера."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] API
Service: Отримано запит: {request_data['type']}")
    await asyncio.sleep(random.uniform(0.1, 0.8)) # Імітація обробки
на сервері
    response_data = {"status": "success", "message": f"Запит
'{request_data['type']}' оброблено."}
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] API
Service: Запит {request_data['type']} оброблено.")
    return response_data

async def send_api_request(request_type, payload):
    """Асинхронна функція для надсилання запиту до імітованого API."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Клієнт:
Надсилаю запит '{request_type}'...")
    response = await mock_api_service({"type": request_type,
"payload": payload})
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Клієнт:
Отримано відповідь для '{request_type}': {response['status']}")
    return response

async def main_example_4():
    """Основна корутина для Прикладу 4."""
```

```
    print("Запускаю Приклад 4 (асинхронний клієнт)...")
    requests_to_send = [
        ("login", {"username": "user1"}),
        ("get_profile", {"user_id": 123}),
        ("update_settings", {"setting": "value"}),
        ("logout", {})
    ]
    start_time = time.perf_counter()

    tasks = [send_api_request(req_type, payload) for req_type,
payload in requests_to_send]
    responses = await asyncio.gather(*tasks)

    end_time = time.perf_counter()
    print(f"Всі запити Прикладу 4 завершено за {end_time -
start_time:.2f} секунд.")
    print("Отримані відповіді:")
    for resp in responses:
        print(f"  - {resp}")

asyncio.run(main_example_4())
```

Завдання до Прикладу 4: Імітація асинхронного клієнта

Створіть асинхронну функцію `log_event_to_service(event_name, data)`, яка імітує надсилання лог-події до зовнішнього сервісу. Кожна відправка має займати випадковий час від 0.1 до 0.5 секунд. Запустіть 5 таких функцій конкурентно для різних подій.

```

```