ММП 2025/2026

Викладач Канцедал Георгій Олегович

Асинхронна парадигма

Асинхронне програмування – підхід до оброблення вводу/виводу, що дозволяє програмі працювати з мережевими пристроями та обслуговувати запити мережі, або працювати з базами даних, поки процесор продовжує виконувати програму, не очікуючи на відповідь від мережі або бази даних.

Ключові особливості:
Паралельне виконання коду
Відсутність чіткої послідовності кроків

Асинхронна парадигма

Переваги:

- **Підвищена продуктивність**
  - Завдання виконуються **одночасно**, не блокуючи програму.
  - Особливо ефективне для **I/O-операцій** (робота з мережею, файлами, базами даних).
- **Ефективне використання ресурсів**
  - Асинхронний код дозволяє використовувати **менше потоків**, що економить пам’ять та CPU.
  - Завдяки asyncio не потрібно створювати окремі **потоки (threads)** чи процеси.
- **Швидке виконання мережевих запитів**
  - Наприклад, отримання **даних з API або завантаження файлів** відбувається швидше, оскільки очікування відповіді не блокує програму.
- **Краща підтримка масштабованості**
  - Використовується у **серверних застосунках**, які обробляють велику кількість запитів (FastAPI, aiohttp). **Приклад:** Сервер FastAPI може обробляти **тисячі** запитів одночасно завдяки асинхронному підходу.

Недоліки:

- **Вища складність коду**
  - async/await вимагає **іншого мислення** порівняно з синхронним кодом.
  - Деякі програмісти знаходять асинхронний підхід **менш інтуїтивним**.
- **Не всі бібліотеки підтримують async**
  - Деякі популярні Python-бібліотеки (наприклад, pymysql, sqlite3) **не підтримують асинхронні виклики**.
  - Доводиться використовувати асинхронні аналоги (aiomysql, aiosqlite).
- **Нестабільна взаємодія з багатопотоковістю**
  - Python **не поєднує добре** asyncio з потоками (threading).
  - Якщо потрібна багатопоточність + асинхронність, доводиться **ретельно планувати** структуру коду.
- **Перевантаження подійного циклу**
  - Якщо в асинхронному коді є **важкі обчислення** (наприклад, обробка великих файлів), це **може заблокувати** виконання інших завдань.
  - У таких випадках краще використовувати **окремі процеси (multiprocessing)**.

async/await

- Перед визначенням функцій з’явився префікс **async**. Він повідомляє інтерпретатору, що функція повинна виконуватися асинхронно.
- Замість звичного **time.sleep** ми використали **asyncio.sleep**. Це **"неблокуючий sleep"**. У межах функції він поводиться так само, як традиційний, але **не зупиняє** виконання всієї програми.
- Перед викликом асинхронних функцій з’явився префікс **await**. Він повідомляє інтерпретатору приблизно таке: *"Я тут, можливо, трохи затримаюсь, але ти мене не чекай – нехай виконується інший код, а коли в мене буде настрій продовжити, я тобі повідомлю."*
- На основі функцій ми за допомогою **asyncio.create_task** створили **задачі** (що це таке, розглянемо пізніше) і запустили їх за допомогою **asyncio.run**.

```python
import asyncio
import time


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    await task1
    await task2


print(time.strftime('%X'))

asyncio.run(main())

print(time.strftime('%X'))
```

async/await

Як це працює:

- Виконалася швидка частина функції **fun1**.
- **fun1** сказала інтерпретатору:
  - *"Іди далі, я посплю 3 секунди."*
- Виконалася швидка частина функції **fun2**.
- **fun2** сказала інтерпретатору:
  - *"Іди далі, я посплю 3 секунди."*
- Інтерпретатору **більше нічого робити**, тому він **чекає**, поки перша "прокинута" функція повідомить про завершення.
- На **долі мілісекунди раніше** прокинулася **fun1** (адже вона й заснула трохи раніше) і **повідомила про завершення**.
- Те ж саме зробила **fun2**.

```python
import asyncio
import time


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    await task1
    await task2


print(time.strftime('%X'))

asyncio.run(main())

print(time.strftime('%X'))
```

async/await

Клас функції **не змінився**, але завдяки ключовому слову **async** вона тепер повертає не <class 'NoneType'>, а <class 'coroutine'>. Ніщо перетворилося на **щось**! На сцену виходить нова сутність — **корутина**.

Що нам потрібно знати про корутину?

- Пам’ятаєте, як у Python влаштований **генератор**? Це така функція, яка починає **повертати значення частинами**, якщо замість return у ній використати yield.
- Так от, **корутина — це різновид генератора**. **Корутина дає інтерпретатору можливість відновити виконання функції, яка була призупинена в місці розміщення ключового слова await.**

```python
import asyncio
import time


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    await task1
    await task2


print(type(fun1))

print(type(fun1(4)))
```

async/await

Часто **корутиною називають саму функцію**, що містить await.
Формально це неправильно.

- **Корутина — це не функція, а те, що ця функція повертає!**
- **Відчуваєте різницю між f та f()?**
- f — це корутинна функція
- f() — це **корутина**, яка з'являється **після виклику цієї функції**

```python
import asyncio
import time


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    await task1
    await task2


print(type(fun1))

print(type(fun1(4)))
```

async/await

Наступним кроком для нас є побачити прекрасний об'єкт таск який як не дивно має тип <class '_asyncio.Task'> (похідний від <class '_asyncio.Future'>).

- **Футура** (якщо зовсім спростити) — це **обгортка** для деякої **асинхронної сутності**, яка дозволяє виконувати її *"начебто одночасно"* з іншими асинхронними сутностями, **перемикаючись між ними в точках, позначених ключовим словом await**.
- Крім того, **футура** має внутрішню змінну **"результат"**, яка:
  - Доступна через **.result()**
  - Встановлюється через **.set_result(value)**

Задача — це **окремий випадок футури**, призначений для **обгортання корутини**

```python
import asyncio


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    print(type(task1))
    print(task1.__class__.__bases__)

    await task1
    await task2


asyncio.run(main())
```

async/await

І так що ж відбувалось в нас в коді:

- **Корутину** асинхронної функції fun1 **обгорнули в задачу** task1.
- **Корутину** асинхронної функції fun2 **обгорнули в задачу** task2.
- В асинхронній функції main **позначили точку переключення** до задачі task1.
- В асинхронній функції main **позначили точку переключення** до задачі task2.
- **Корутину** асинхронної функції main **передали у функцію** asyncio.run

```python
import asyncio


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    task1 = asyncio.create_task(fun1(4))
    task2 = asyncio.create_task(fun2(4))

    print(type(task1))
    print(task1.__class__.__bases__)

    await task1
    await task2


asyncio.run(main())
```

async/await простий приклад

Скільки секунд буде працювати?

```python
import asyncio
import time


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    await fun1(4)
    await fun2(4)


print(time.strftime('%X'))

asyncio.run(main())

print(time.strftime('%X'))
```

async/await простий приклад

6.
У asyncio.run потрібно передавати **асинхронну функцію** з await **на задачі**, а не безпосередньо на корутини.
Інакше **"не злетить"**. Тобто **програма працюватиме**, але **строго послідовно**.

```python
import asyncio
import time


async def fun1(x):
    print(x**2)
    await asyncio.sleep(3)
    print('fun1 завершена')


async def fun2(x):
    print(x**0.5)
    await asyncio.sleep(3)
    print('fun2 завершена')


async def main():
    await fun1(4)
    await fun2(4)


print(time.strftime('%X'))

asyncio.run(main())

print(time.strftime('%X'))
```

Async with

Наступним кроком пропоную переглянути асинхронний менеджер контектсу в папці додаткові (менеджер контектсту це структура яка виконує певні дії при її виклику/вході в неї та при закінчені роботи з нею – виході.

```
# визначення асинхронного менеджера контексту
class AsyncContextManager:
    # вхід в асинхронний менеджер контексту
    async def __aenter__(self):
        # виведення повідомлення
        print('>вхід у контекстний менеджер')
        # блокування на деякий час
        await asyncio.sleep(0.5)

    # вихід з асинхронного менеджера контексту
    async def __aexit__(self, exc_type, exc, tb):
        # виведення повідомлення
        print('>вихід з контекстного менеджера')
        # блокування на деякий час
        await asyncio.sleep(0.5)
```

```
# SuperFastPython.com
# приклад роботи з асинхронним менеджером контексту та async with
import asyncio

# визначення асинхронного менеджера контексту
class AsyncContextManager:
    # вхід в асинхронний менеджер контексту
    async def __aenter__(self):
        # виведення повідомлення
        print('>вхід у контекстний менеджер')
        # блокування на деякий час
        await asyncio.sleep(0.5)

    # вихід з асинхронного менеджера контексту
    async def __aexit__(self, exc_type, exc, tb):
        # виведення повідомлення
        print('>вихід з контекстного менеджера')
        # блокування на деякий час
        await asyncio.sleep(0.5)

# визначення простої корутини
async def custom_coroutine():
    # створення та використання асинхронного менеджера контексту
    async with AsyncContextManager() as manager:
        # виведення результатного повідомлення
        print(f'всередині менеджера')

# запуск asyncio-програми
asyncio.run(custom_coroutine())
```

Асинхронні comprehension-вирази

Comprehension-вирази — такі, як **спискові включення** та **компактні вирази для створення словників** — це одна з особливостей Python, яка виглядає як щось дуже **"пітонічне"**. Ці вирази є різновидом циклів, які **відрізняються від традиційних циклів**, що застосовуються в багатьох інших мовах.
Асинхронні comprehension-вирази дозволяють обходити **асинхронні генератори та ітератори**, використовуючи конструкцію async for
Модуль asyncio підтримує два типи асинхронних comprehension-виразів:

- Вирази, у яких використовується **async for**
- Вирази, у яких використовується **await**

```
# створення списку за допомогою спискового включення
result = [a * 2 for a in range(100)]

# створення словника за допомогою comprehension-виразу
result = {a: i for a, i in zip(['a', 'b', 'c'], range(3))}

# створення множини за допомогою comprehension-виразу
result = {a for a in [1, 2, 3, 2, 3, 1, 5, 4]}
```

Асинхронні comprehension-вирази

При використанні цієї конструкції створюється **список результатів**, отриманих після **послідовного очікування** завершення кожного з об'єктів, які підтримують await.
Важливий нюанс:

- Поточна **корутина призупиняється**, щоб дочекатися виконання кожного awaitable-об'єкта **по черзі**. Це **відрізняється** від конкурентного виконання через asyncio.gather(), яке дозволяє **запускати всі об'єкти паралельно**. Через це **асинхронне списокове включення може працювати повільніше**, ніж asyncio.gather() (перший варіант)

```
# асинхронне списокове включення, у якому використовується асинхронний генератор
result = [a async for a in agenerator]

# списокове включення з виразом await та обробкою колекції об'єктів, які підтримують очікування
results = [await a for a in awaitables]
```

Істина багатопоточність

Модуль multiprocessing у Python дозволяє використовувати справжній паралелізм, тобто створювати окремі процеси, які виконуються повністю незалежно один від одного.

- Відмінність від threading у тому, що потоки використовують спільну пам’ять і обмежені GIL (Global Interpreter Lock), тоді як multiprocessing створює окремі процеси, які мають власну пам’ять і можуть використовувати всі ядра процесора.
- Процеси не мають спільної пам’яті та не можуть просто так змінювати однакові змінні. Бажано уникати міжпроцесного обміну даними, якщо це можливо, але для складних задач він може бути необхідним, хоча ускладнює реалізацію.

```python
import time
import multiprocessing

# Функція, яка виконує обчислювальне завдання
def task(n=100_000_000):
    while n:
        n -= 1

if __name__ == '__main__':
    # Запускаємо таймер для вимірювання часу виконання
    start = time.perf_counter()

    # Створюємо два окремі процеси
    p1 = multiprocessing.Process(target=task)
    p2 = multiprocessing.Process(target=task)

    # Запускаємо процеси
    p1.start()
    p2.start()

    # Очікуємо завершення процесів
    p1.join()
    p2.join()

    # Фіксуємо час завершення
    finish = time.perf_counter()

    # Виводимо час виконання
    print(f'Виконання зайняло {finish-start: .2f} секунд.')
```

Істина багатопоточність

Як це працює?

- Створюємо два процеси та передаємо кожному з них функцію task().
- Конструктор Process() повертає новий об'єкт Process.
- Викликаємо метод start(), щоб запустити процеси.
- Чекаємо на завершення процесів за допомогою методу join()

```python
import time
import multiprocessing

# Функція, яка виконує обчислювальне завдання
def task(n=100_000_000):
    while n:
        n -= 1

if __name__ == '__main__':
    # Запускаємо таймер для вимірювання часу виконання
    start = time.perf_counter()

    # Створюємо два окремі процеси
    p1 = multiprocessing.Process(target=task)
    p2 = multiprocessing.Process(target=task)

    # Запускаємо процеси
    p1.start()
    p2.start()

    # Очікуємо завершення процесів
    p1.join()
    p2.join()

    # Фіксуємо час завершення
    finish = time.perf_counter()

    # Виводимо час виконання
    print(f'Виконання зайняло {finish-start: .2f} секунд.')
```

Кінець

Нажаль зробити лабораторну на асинхронність коли сама перевірка є асинхронною доволі важко,
але для допиливих оригінальні матеріали за посиланням: https://superfastpython.com/python-asyncio/