**Модуль 5: Асинхронне програмування в Python**

**Заняття 1: Асинхронне програмування**

Найпростіший, найочевидніший та найінтуїтивніший спосіб написання застосунку — це виконання завдань в одному потоці по черзі — *синхронна модель програмування* (*Synchronous model*). Використовуючи такий підхід, ви завжди знаєте, що і за чим виконується, і можете точно сказати, в якому стані знаходиться застосунок.

Однак, як вам уже відомо, істотну частину операцій читання/запису займає очікування відповіді. Під час цього очікування застосунок простоює і не робить нічого корисного.

Щоб якось використовувати час простою застосунку, ви можете виконувати завдання в кількох потоках — *потокова модель програмування* (*Threaded model*).

Виконуючи завдання в окремих потоках, ваш застосунок не витрачає "марно" час процесора, очікуючи відповіді під час операцій читання/запису.

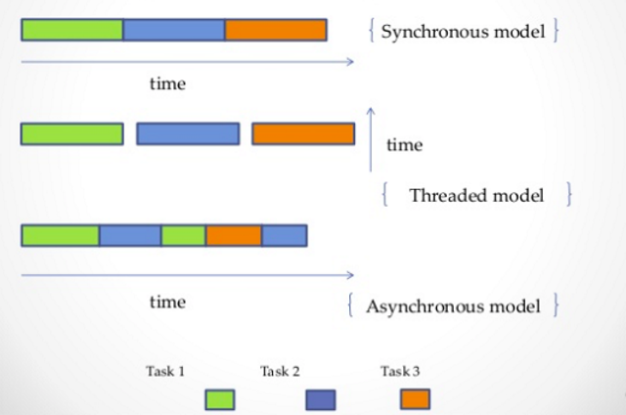
Але такий підхід ускладнює застосунок. Основна причина ускладнення — це втрата контролю за передачею управління. Розробник не управляє тим, коли та якому потоку операційна система (ОС) віддасть контроль. Через це доводиться використовувати примітиви синхронізації (Lock, RLock, Semaphore, Barier та інші).

Крім цього, сам процес передачі управління створює додаткові накладні витрати. ОС повинна зберегти стан пам'яті потоку, вивантажити із швидкої пам'яті процесора (кеша) всі дані цього потоку, завантажити дані іншого потоку, щоб процесор міг працювати з ними.

Насправді ці накладні витрати призводять до того, що виконувати Python застосунок у більше ніж 10 потоках, зазвичай, не призводить до зростання продуктивності. А використання більше 100 потоків може бути повільнішим, ніж виконання тієї самої роботи синхронно в одному потоці.

Є архітектурний підхід, який поєднує в собі простоту та однозначність стану пам'яті однопотокового застосунку, та ефективне використання часу очікування (простою) під час IO операцій читання/запису.

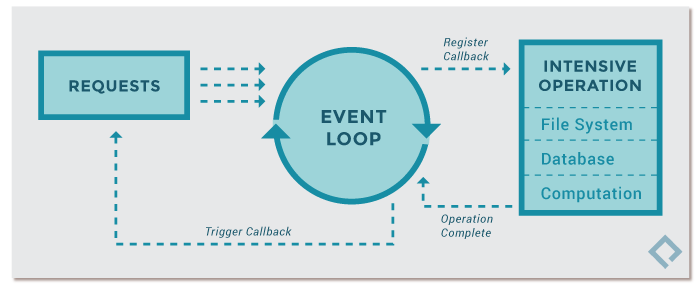
Цей підхід — це цикл подій (**Event loop**) або *асинхронна модель програмування* (*Asynchronous model*).



**Моделі програмування**

Суть підходу:

* застосунок містить один головний обробник подій (цикл подій);
* події, аналог потоків, послідовність операцій (coroutine у Python);
* всі події реалізують єдиний протокол подій (обробник запускає виконання події стандартизованим чином);
* події зберігаються у черзі, звідки їх бере обробник;
* результатом події може бути нова подія, яка вирушає в чергу;



**Діаграма станів циклу подій (Event loop)**

Щоб ефективно використовувати час простою для**IO**(Input-Output) операцій, події повинні передавати контроль **Event loop** одразу після виклику**IO**операції, не блокуючи виконання застосунку, та повертати нову подію, яка повинна перевірити, чи став доступний ресурс (файл прочитаний, отримано відповідь на запит мережі тощо) і, якщо доступний, то продовжити обробку.

В результаті ми отримуємо однопотоковий застосунок, де розробник завжди знає, в якому стані знаходиться пам'ять застосунку, оскільки сам управляє передачею управління між coroutine.

Тут потрібно розуміти, що за такого підходу coroutine не являються справжніми потоками, але виконують ту саму роль з точки зору розробника, об'єднують ланцюжок операцій обробки даних (завдання) в одну послідовність.

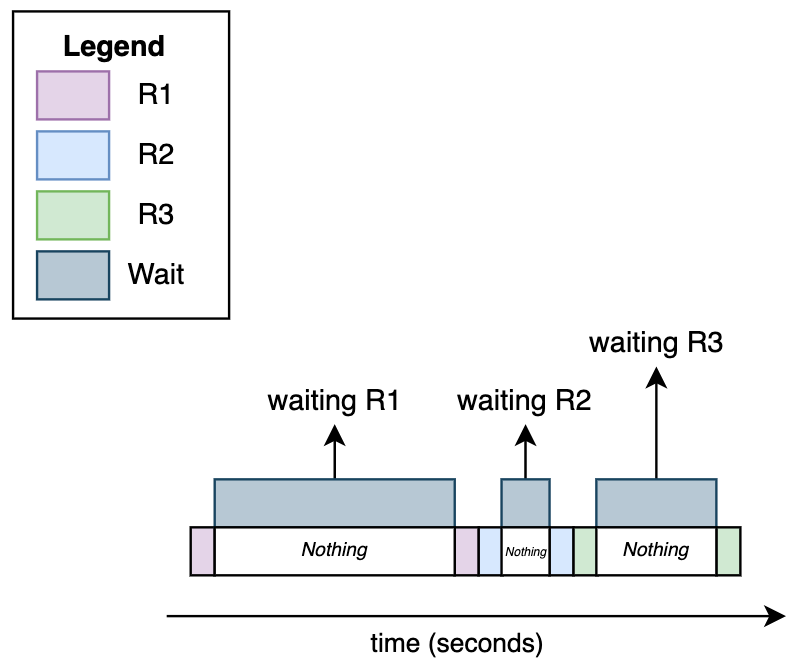
При цьому час простою застосунку зводиться до мінімуму за рахунок того, що блокуючі **IO** операції не блокують застосунок і поки ресурс недоступний, виконується інше завдання із черги завдань.

Крім того, накладні витрати зменшуються за рахунок того, що не потрібно міняти вміст кешу процесора через перемикання потоків застосунку.

**NOTE**

Такий підхід часто називають просто асинхронним, хоча це і не зовсім правильно, оскільки багатопотоковість так само асинхронна, але це вже сталий термін і ми теж далі будемо його дотримуватися.

Ключовим моментом, що дає можливість збільшити продуктивність **асинхронних** застосунків — це передача контролю під час **IO** операцій. Якщо ми порівняємо однопотокову синхронну реалізацію з **асинхронною** для випадку, коли управління не передається і всі**IO** виклики є блокуючими, то зауважимо, що різниці жодної немає.



**Синхронна та асинхронна реалізація з очікуванням**

Виходить, що не реалізувавши неблокуючі аналоги всіх**IO**операцій, не можна досягти приросту продуктивності **асинхронного** коду. Через це **асинхронний** підхід так повільно впроваджувався у Python. Адже операції, що не блокують**IO**, реалізувати набагато складніше своїх синхронних аналогів.

Давайте коротко підіб'ємо порівняльний аналіз багатопотокової та асинхронної моделі програмування

Багатопотокові застосунки:

* необхідно контролювати доступ до загальних ресурсів;
* складно відлагоджувати і легше припуститися помилки;
* перемикання контексту з'їдає ресурс;
* ефективне використання ресурсу під час очікування.

Асинхронні застосунки:

* передбачуваний стан пам'яті;
* простіше відлагоджувати, легше писати;
* практично "безкоштовні" потоки (невеликі накладні витрати);
* ефективне використання ресурсу під час очікування.

На наступному кроці ми розглянемо реалізацію асинхронної моделі програмування у Python

**Основи роботи з asyncio**

У Python, починаючи з версії 3.5, додали нові ключові слова: async та await. Був реалізований на рівні синтаксису цикл подій та код, який поміщений у модуль asyncio.

Модуль асинхронного програмування ayncio представляє інструменти для створення застосунків, заснованих на виконанні паралельних обчислень з використанням співпрограм (*coroutine*).

[**Coroutine**](https://docs.python.org/3/library/asyncio-task.html) **(https://docs.python.org/3/library/asyncio-task.html)** — це аналог потоків для **асинхронного** коду. Якщо викликати *coroutine* напряму, то отримаємо об'єкт coroutine object.

Безпосередньо виконання *coroutine* покладено на Event loop.

Центральне місце у модулі asyncio займає "цикл подій" Event loop — об'єкт першого класу, відповідальний за ефективну обробку подій введення-виведення та зміни контексту застосунку.

Взаємодія застосунку та Event loop відбувається наступним чином. Спочатку виконується реєстрація коду для асинхронного виконання, після чого Event loop виконує необхідні виклики у коді застосунку.

Повернення управління Event loop реалізовано за допомогою *coroutine*. При виклику функції *coroutine* створюється об'єкт, що дозволяє викликаючому коду виконати код цієї функції. Призупинити виконання *coroutine* можна за допомогою ключового слова await спільно з ім'ям іншого *coroutine*. Протягом цієї паузи стан програми зберігається, що дозволяє їй при поверненні результату *coroutine* продовжити виконання з тієї точки, в якій вона була перервана.

Щоб функція виконувалася в асинхронному режимі та виступала в ролі *coroutine*, перед її визначенням необхідно додати ключове слово async.

У модулі asyncio є функція run, яка створює Event loop, поміщає у чергу передану *coroutine* і, коли черга спорожніє, завершує Event loop. Ви можете зробити всі ці операції самостійно, якщо потрібно, але бажаний спосіб — run.

Давайте розглянемо простий приклад:

import asyncio

async def baz() -> str:

print('Before Sleep')

await asyncio.sleep(1)

print('After Sleep')

return 'Hello world'

async def main():

r = baz()

print(r)

result = await r

print(result)

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

asyncio.run(main())

Отримаємо ми наступне виведення:

<coroutine object baz at 0x00000211E82CCDD0>

Before Sleep

After Sleep

Hello world

Це найпростіший приклад на async/await. Виклик print(r) нам повертає об'єкт *coroutine*. Щоб отримати результат від асинхронної функції baz, нам потрібен await. І тільки виконавши result = await r, ми отримаємо у змінній result значення Hello world.

Але головна перевага async/await – це прискорення IO запитів. Для демонстрації розглянемо наступний приклад.

from time import sleep, time

fake\_users = [

{'id': 1, 'name': 'April Murphy', 'company': 'Bailey Inc', 'email': 'shawnlittle@example.org'},

{'id': 2, 'name': 'Emily Alexander', 'company': 'Martinez-Smith', 'email': 'turnerandrew@example.org'},

{'id': 3, 'name': 'Patrick Jones', 'company': 'Young, Pruitt and Miller', 'email': 'alancoleman@example.net'}

]

def get\_user\_sync(uid: int) -> dict:

sleep(0.5)

user, = list(filter(lambda user: user["id"] == uid, fake\_users))

return user

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

start = time()

for i in range(1, 4):

print(get\_user\_sync(i))

print(time() - start)

Виведення:

{'id': 1, 'name': 'April Murphy', 'company': 'Bailey Inc', 'email': 'shawnlittle@example.org'}

{'id': 2, 'name': 'Emily Alexander', 'company': 'Martinez-Smith', 'email': 'turnerandrew@example.org'}

{'id': 3, 'name': 'Patrick Jones', 'company': 'Young, Pruitt and Miller', 'email': 'alancoleman@example.net'}

1.5010051727294922

Тут ми імітуємо запити до бази даних за допомогою функції get\_user\_sync і в циклі запитуємо фейкових користувачів з id від 1 до 3. На це у нас йде приблизно 1.5 секунди. Прискоримо цей процес за допомогою async/await.

import asyncio

from time import time

fake\_users = [

{'id': 1, 'name': 'April Murphy', 'company': 'Bailey Inc', 'email': 'shawnlittle@example.org'},

{'id': 2, 'name': 'Emily Alexander', 'company': 'Martinez-Smith', 'email': 'turnerandrew@example.org'},

{'id': 3, 'name': 'Patrick Jones', 'company': 'Young, Pruitt and Miller', 'email': 'alancoleman@example.net'}

]

async def get\_user\_async(uid: int) -> dict:

await asyncio.sleep(0.5)

user, = list(filter(lambda user: user["id"] == uid, fake\_users))

return user

async def main():

r = []

for i in range(1, 4):

r.append(get\_user\_async(i))

return await asyncio.gather(\*r)

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

start = time()

result = asyncio.run(main())

for r in result:

print(r)

print(time() - start)

Виведення:

{'id': 1, 'name': 'April Murphy', 'company': 'Bailey Inc', 'email': 'shawnlittle@example.org'}

{'id': 2, 'name': 'Emily Alexander', 'company': 'Martinez-Smith', 'email': 'turnerandrew@example.org'}

{'id': 3, 'name': 'Patrick Jones', 'company': 'Young, Pruitt and Miller', 'email': 'alancoleman@example.net'}

0.5019364356994629

Ми отримуємо той самий результат, але втричі швидший. Фактично ми запустили три функції get\_user\_async на паралельне виконання і цей процес зайняв близько 0.5 секунд. Допоміжна функція asyncio.gather потрібна, щоб покласти в чергу кілька співпрограм та дозволити Event loop виконувати їх у будь-якому порядку. Крім того, asyncio.gather повертає результат виконання *coroutine* у тому порядку, в якому вони були викликані.

**Паралельне виконання завдань**

Зазвичай працювати з низькорівневими примітивами asyncio не потрібно і не рекомендується. Але якщо виникає необхідність, то можна відправити завдання до поточного Event loop за допомогою функції create\_task, яка повертає об'єкт Task.

Завдання Task — це один із основних способів взаємодії з циклом подій. Вони є обгортками над співпрограмами та відстежують момент їх завершення. Оскільки завдання є похідним класом від Future, то наслідують частину його функціональності, а саме:

* інші співпрограми можуть очікувати на завершення завдань
* кожне завдання має результат, який може бути отриманий після його завершення

Результуюче завдання буде виконуватися як частина паралельних операцій, які управляються Event loop, доки виконується цикл та співпрограма не поверне управління.

Давайте наведемо приклад:

import asyncio

import random

async def random\_value():

print("start task")

await asyncio.sleep(1)

print("task finished")

return random.random()

async def main():

task = asyncio.create\_task(random\_value())

print("task scheduled")

await task

print(f"result: {task.result()}")

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

asyncio.run(main())

В результаті в консолі ви побачите:

task scheduled

start task

task finished

result: 0.4314553905867432

Зверніть увагу, що доки не було викликано await task, завдання не виконувалося.

**Паралельне виконання CPU-bound завдань**

Оскільки далеко не всі**IO** операції у всіх пакетах реалізують async/await синтаксис, то часто трапляється, що потрібно виконати в асинхронному коді блокуючий **IO**виклик з пакета, де немає async/await аналога.

Для цього можна скористатися вже відомим вам пакетом concurrent.futures та методом run\_in\_executor об'єкта loop (це і є Event loop).

Розглянемо наступний приклад, де у нас є синхронна функція blocks, яка виконує важкі обчислення:

import asyncio

import concurrent.futures

from time import time

def blocks(n):

counter = n

start = time()

while counter > 0:

counter -= 1

return time() - start

async def monitoring():

while True:

await asyncio.sleep(2)

print(f'Monitoring {time()}')

async def run\_blocking\_tasks(executor, n):

loop = asyncio.get\_event\_loop()

print('waiting for executor tasks')

result = await loop.run\_in\_executor(executor, blocks, n)

return result

async def main():

asyncio.create\_task(monitoring())

with concurrent.futures.ThreadPoolExecutor(max\_workers=3) as executor:

futures = [run\_blocking\_tasks(executor, n) for n in [50\_000\_000, 60\_000\_000, 70\_000\_000]]

results = await asyncio.gather(\*futures)

return results

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

result = asyncio.run(main())

for r in result:

print(r)

Виконавши цей код, ви отримаєте схожий результат:

waiting for executor tasks

waiting for executor tasks

waiting for executor tasks

Monitoring 1667170130.4642367

Monitoring 1667170132.527182

4.648142337799072

5.0788047313690186

5.447803974151611

Щоб здійснити блокуючий виклик в Event loop, ми розміщуємо його в Executor і тоді виклик виконується в окремому потоці або процесі. У разі, коли основна частина коду – *асинхронна* (у нашому випадку це функція monitoring) і лише окремий виклик блокує увесь Event loop ( функція blocks), такий спосіб перетворити блокуючу функцію на *асинхронну* цілком виправданий.

Сам процес перетворення синхронного коду в асинхронний досить простий. Ми обгортаємо функцію blocks функцією run\_blocking\_tasks. Всередині неї беремо окремий loop = asyncio.get\_event\_loop() та через виклик run\_in\_executor отримуємо асинхронний результат функції blocks. В якості Executor у нас виступає concurrent.futures.ThreadPoolExecutor

CAUTION

У живому прикладі завдання monitoring може виконатися велику кількість разів, ніж у локальному прикладі. Це пов'язано з тим, що хмарний сервіс ReplIt має набагато слабші обчислювальні можливості, ніж у локальної машини.

INFO

Також можна перетворити CPU завдання, де відбуваються "важкі" обчислення, в асинхронний виклик, використовуючи ProcessPoolExecutor, замість ThreadPoolExecutor, тоді виклики будуть виконуватися в окремих процесах і не блокуватимуть основний Event loop.

**Паралельне виконання IO-bound завдань**

Так само можна перетворити на паралельне виконання мережеві запити requests. Спочатку розглянемо синхронний приклад:

import requests

from time import time

urls = ['http://www.google.com', 'http://www.python.org', 'http://duckduckgo.com']

def preview\_fetch(url):

r = requests.get(url)

return url, r.text[:150]

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

start = time()

for url in urls:

r = preview\_fetch(url)

print(r)

print(time() - start)

Тут ми виконуємо три послідовні мережеві запити:

('http://www.google.com', '<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="uk"><head><meta content="text/html; charset=UTF-8" http-equiv="Content-Ty')

('http://www.python.org', '<!doctype html>\n<!--[if lt IE 7]> <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9"> <![endif]-->\n<!--[if IE 7]> <html class="no-js ie7 lt-ie8 lt-')

('http://duckduckgo.com', '<!DOCTYPE html>\n<!--[if IEMobile 7 ]> <html lang="en-US" class="no-js iem7"> <![endif]-->\n<!--[if lt IE 7]> <html class="ie6 lt-ie10 lt-ie9 lt-ie8 lt-')

0.6619992256164551

Перетворимо їх за допомогою пакету concurrent.futures у паралельне виконання.

import asyncio

import requests

from concurrent.futures import ThreadPoolExecutor

from time import time

urls = ['http://www.google.com', 'http://www.python.org', 'http://duckduckgo.com']

def preview\_fetch(url):

r = requests.get(url)

return url, r.text[:150]

async def preview\_fetch\_async():

loop = asyncio.get\_running\_loop()

with ThreadPoolExecutor(3) as pool:

futures = [loop.run\_in\_executor(pool, preview\_fetch, url) for url in urls]

result = await asyncio.gather(\*futures, return\_exceptions=True)

return result

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

start = time()

r = asyncio.run(preview\_fetch\_async())

print(r)

print(time() - start)

Виведення:

[('http://www.google.com', '<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="uk"><head><meta content="text/html; charset=UTF-8" http-equiv="Content-Ty'), ('http://www.python.org', '<!doctype html>\n<!--[if lt IE 7]> <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9"> <![endif]-->\n<!--[if IE 7]> <html class="no-js ie7 lt-ie8 lt-'), ('http://duckduckgo.com', '<!DOCTYPE html>\n<!--[if IEMobile 7 ]> <html lang="en-US" class="no-js iem7"> <![endif]-->\n<!--[if lt IE 7]> <html class="ie6 lt-ie10 lt-ie9 lt-ie8 lt-')]

0.2819480895996094

Ми робимо функцію обгортку preview\_fetch\_async над функцією preview\_fetch. Всередині беремо поточний виконуваний *Event loop* loop = asyncio.get\_running\_loop() та за допомогою ThreadPoolExecutor поміщаємо функцію preview\_fetch в Executor - [loop.run\_in\_executor(pool, preview\_fetch, url) for url in urls]. Отриманий список об'єктів Futures передаємо в asyncio.gather(\*futures) для отримання остаточного результату.

NOTE

Параметр 'return\_exceptions' відповідає за обробку помилок, за замовчуванням встановлено значення False. Перший згенерований виняток негайно поширюється на завдання, що очікує в gather. Якщо return\_exceptions має значення True, винятки обробляються так само, як і успішні результати, та об'єднуються у списку результатів.

Як бачимо, навіть за такої малої кількості запитів, ми бачимо приріст продуктивності при виконанні async/await операцій.

**Асинхронна робота з файлами**

**Пакет aiofile[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-01/file" \l "%D0%BF%D0%B0%D0%BA%D0%B5%D1%82-aiofile" \t "_blank)**

Для асинхронної роботи з файлами існує низка пакетів. І почнемо ми з [aiofile](https://github.com/mosquito/aiofile) (https://github.com/mosquito/aiofile). Він виконує асинхронні операції за підтримки пакета asyncio.

Замість звичної функції open, необхідно використовувати async\_open. Він повертає файлоподібні об'єкти python з асинхронними методами.

Методи, що підтримуються:

* async def read(length = -1) - читання фрагмента з файлу, за довжини -1 файл буде прочитаний до кінця.
* async def write(data) - записати фрагмент у файл
* def seek(offset) - встановити позицію покажчика файлу
* def tell() - повертає поточну позицію покажчика файлу
* async def readline(size=-1, newline="\n") - читати фрагменти до нового рядка або EOF.
* def \_\_aiter\_\_() -> LineReader - ітератор по рядках.
* def iter\_chunked(chunk\_size: int = 32768) -> Reader — ітератор по чанкам.

**NOTE**

Метод readline неоптимальний для невеликих рядків, оскільки не використовує повторно буфер читання. Якщо ви хочете читати файл порядково, будь ласка, уникайте використання async\_open, замість цього використовуйте LineReader.

Наприклад, наступний код створить нам файл hello.txt:

import asyncio

from aiofile import async\_open

async def main():

async with async\_open("hello.txt", 'w+') as afp:

await afp.write("Hello ")

await afp.write("world\n")

await afp.write("Hello from - async world!")

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

asyncio.run(main())

**hello.txt**

Hello world

Hello from - async world!

Щоб прочитати "hello.txt" файл можна різним асинхронним підходом:

**Підхід await afp.read()**

import asyncio

from aiofile import async\_open

async def main():

async with async\_open("hello.txt", 'r') as afp:

print(await afp.read())

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

asyncio.run(main())

**Підхід async for**

import asyncio

from aiofile import async\_open

async def main():

async with async\_open("hello.txt", 'r') as afp:

async for line in afp:

print(line)

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

asyncio.run(main())

Виведення буде однаковим:

Hello world

Hello from - async world!

LineReader — помічник, який дуже ефективний, коли ви хочете прочитати файл лінійно та рядково. Він містить буфер і зчитуватиме фрагменти файлу частинами в буфер, де намагатиметься знайти рядки. Розмір фрагмента за замовчуванням складає 4 КБ.

import asyncio

from aiofile import AIOFile, LineReader

async def main():

async with AIOFile("hello.txt", 'r') as afp:

async for line in LineReader(afp):

print(line)

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

asyncio.run(main())

Виведення:

Hello world

Hello from - async world!

**Пакет aiopath**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-01/file" \l "%D0%BF%D0%B0%D0%BA%D0%B5%D1%82-aiopath" \t "_blank)

Якщо ви пишете асинхронний код Python і хочете скористатися перевагами pathlib, але не хочете змішувати блокуюче та неблокуюче введення-виведення, ви можете звернутися до [aiopath](https://github.com/alexdelorenzo/aiopath) (<https://github.com/alexdelorenzo/aiopath>). API aiopath прямо збігається з API pathlib, але всі необхідні методи асинхронні.

Наприклад, перевіримо, чи існує файл "hello.txt" у поточній папці:

import asyncio

from aiopath import AsyncPath

async def main():

apath = AsyncPath("hello.txt")

print(await apath.exists())

print(await apath.is\_file())

print(await apath.is\_dir())

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

asyncio.run(main())

Якщо файл є, виведення буде наступним:

True

True

False

**Пакет aioshutil[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-01/file" \l "%D0%BF%D0%B0%D0%BA%D0%B5%D1%82-aioshutil" \t "_blank)**

Бібліотека [aioshutil](https://github.com/kumaraditya303/aioshutil) (<https://github.com/kumaraditya303/aioshutil>) надає асинхронну версію функції модуля Shutil.

Модуль Shutil є синхронним, та його використання в асинхронних застосунках заблокує цикл подій і уповільнить роботу застосунку, aioshutil надає асинхронні дружні версії функцій модуля Shutil.

Для прикладу, давайте ми створимо папку logs та скопіюємо туди наш файл "hello.txt"

import asyncio

from aiopath import AsyncPath

from aioshutil import copyfile

async def main():

apath = AsyncPath("hello.txt")

if await apath.exists():

new\_path = AsyncPath('logs')

await new\_path.mkdir(exist\_ok=True, parents=True)

await copyfile(apath, new\_path / apath)

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

asyncio.run(main())