**Модуль 3: Багатопоточність і процеси в Python**

**Заняття 1: Багатопоточність у Python**

**Вступ до багатопотоковості**

**Сфера застосування**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/intro#%D1%81%D1%84%D0%B5%D1%80%D0%B0-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)Що робить застосунок, доки чекає відповіді від сервера або чекає, доки файл прочитається із жорсткого диска? Зазвичай у такі моменти застосунок нічого не робить і просто чекає на результат операції. Це не є проблемою, тільки, якщо вам не потрібно прискорити роботу вашого застосунку. Особливо це критично для ситуацій, коли застосунок робить багато дуже схожих блокуючих викликів (запитів на віддалений сервер, операцій читання/запису файлів). У такі моменти, звичайно, хочеться оптимізувати роботу застосунку та виконувати кілька блокуючих операцій паралельно. Код, що виконує паралельно кілька завдань, називається асинхронним. Найпростіший спосіб реалізувати асинхронність — це виконувати завдання в окремих потоках всередині одного процесу.

**Процес** — область пам'яті (віртуальна) + набір ресурсів + 1 і більше потоків.

**Потік** — послідовність інструкцій та системних викликів всередині процесу.

Всі потоки мають доступ до всіх ресурсів свого процесу. Усі процеси ізольовані один від одного, і будь-яка міжпроцесна взаємодія відбувається тільки через операції введення/виведення (системні виклики).

**IO and CPU bound tasks**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/intro#io-and-cpu-bound-tasks)Завдання, які виконують операції введення/виведення (читання/запис файлів, запити в мережі тощо), називаються**IO (Input Output)-bound** завданнями. Домогтися паралелізму виконання IO завдань у Python можна, використовуючи потоки.

Однак потрібно пам'ятати, що асинхронний код завжди на порядок складніший для розуміння та відлагодження. Для багатопотокових застосунків складно писати тести та складно перевіряти всілякі ситуації, які відбуваються рідко та залежать від порядку викликів у різних потоках. Загальне правило для програмування будь-якою мовою: якщо є можливість обійтися синхронним кодом, то так і потрібно зробити.

Інший тип блокуючих викликів — це важкі з точки зору обчислень операції.

Реальний застосунок завжди повинен якимось чином реагувати на дії користувача і, якщо ваш застосунок під час виконання складних обчислень перестає відповідати на запити, то користувач може вирішити, що застосунок просто завис. Виходить, що для зручності користувача застосунок повинен відповідати на запити, навіть коли робить якісь складні та довгі обчислення.

Такі завдання називаються **CPU-bound** завданнями. Як і для **IO-bound** завдань, можна винести виконання блокуючих операцій (складних обчислень) в окремий потік, щоб застосунок продовжував взаємодіяти з користувачем, здійснюючи обчислення.

Загалом, операційна система передає управління потокам (як і передача управління процесами). Це означає, що будь-якої миті, перед будь-яким викликом ОС (Операційна Система) може призупинити виконання коду потоку та розпочати виконувати код іншого потоку, щоб потім так само далі призупинити і його для передачі управління.

До появи багатоядерних процесорів справжній паралелізм був неможливим. Звичайно, коли управління передається різним потокам по кілька тисяч разів на секунду, з погляду користувача це виглядає як паралельне виконання кількох завдань. У сучасних процесорах зазвичай є мінімум два ядра і тепер ми можемо писати код, який виконується справді паралельно. Це з однієї сторони додає можливостей, але й додає складнощів, оскільки тепер потрібно бути ще уважнішими при написанні асинхронного коду, адже припуститися помилки ще простіше.

https://medium.com/swift-india/concurrency-parallelism-threads-processes-async-and-sync-related-39fd951bc61d про асинхронний код.

**Global Interpreter Lock (GIL)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/intro#global-interpreter-lock-gil)Потоки можуть виконуватися дійсно паралельно (якщо ядер процесора більше 1), процеси — тим більше. **Але** у Python є механізм, який примусово блокує виконання коду різними потоками одного Python процесу в один і той самий час.

1. Тільки один потік всередині процесу Python виконується, всі інші (якщо такі є) знаходяться в режимі 'Sleep'.
2. Операції, пов'язані з введенням/виведенням (системні виклики) не блокуються GIL, але не їх послідовність.

Це означає, що якщо ви зробите кілька IO викликів у різних потоках, то вам не гарантується черговість завершення цих потоків, але гарантується, що коли виконується код будь-якого з потоків, всі інші потоки чекають черги і нічого не роблять. Це буде так, навіть якщо код виконується на сучасному процесорі з кількома ядрами

**Чому в Python є GIL?**

1. Простий і зрозумілий збирач сміття.
2. Виключає можливість одночасного доступу до ресурсів/пам'яті. Немає потреби враховувати особливості конкретної ОС для обробки таких ситуацій.
3. Це спадщина епохи одноядерних процесорів, коли додаткові потоки/процеси уповільнювали виконання програми.

Python розроблявся в епоху одноядерних процесорів і навіть теоретично ніхто тоді не міг припустити дійсне одночасне виконання коду в різних потоках. Через це було зроблено низку архітектурних рішень, які вже не змінити, і на Python накладено обмеження GIL.

**Як обійти GIL:**

1. Написати частину коду, яку потрібно запускати паралельно, на Cython і використовувати потоки.
2. Використовувати Multiprocessing.

**Чому не потрібно цього робити:**

1. Python — скриптова мова і швидкість роботи не її сильна сторона. Якщо потрібна швидкість, то, можливо, є сенс розглянути інший інструмент.
2. Створення процесів використовує деяку кількість ресурсів системи (пам'ять та процесорний час).
3. Перемикання між процесами також використовує процесорний час.

**Створення потоків у Python**

https://docs.python.org/3/library/concurrency.html про доступні в Python механізми написання потокового коду.

**Потік як клас**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/create#%D0%BF%D0%BE%D1%82%D1%96%D0%BA-%D1%8F%D0%BA-%D0%BA%D0%BB%D0%B0%D1%81)Щоб створити потік, найпростіше імпортувати клас Thread з модуля threading і наслідуватись від цього класу. Далі вам потрібно визначити метод run у вашого класу, цей метод буде виконуватись в окремому потоці. Щоб розпочати виконання коду в окремому потоці, потрібно викликати метод start, який визначений у Thread. Давайте напишемо клас MyThread, що в окремому потоці спить вказаний час і після цього виводить у консоль 'Wake up!':

from threading import Thread

import logging

from time import sleep

class MyThread(Thread):

def \_\_init\_\_(self, group=None, target=None, name=None, args=(), kwargs=None, \*, daemon=None):

super().\_\_init\_\_(group=group, target=target, name=name, daemon=daemon)

self.args = args

self.kwargs = kwargs

def run(self) -> None:

sleep(2)

logging.debug('Wake up!')

logging.debug(f"args: {self.args}")

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

for i in range(5):

thread = MyThread(args=(f"Count thread - {i}",))

thread.start()

print('Usefull message')

Якщо виконати цей скрипт, то в консолі ви побачите:

Usefull message

Thread-5 Wake up!

Thread-5 args: ('Count thread - 4',)

Thread-3 Wake up!

Thread-3 args: ('Count thread - 2',)

Thread-2 Wake up!

Thread-2 args: ('Count thread - 1',)

Thread-1 Wake up!

Thread-1 args: ('Count thread - 0',)

Thread-4 Wake up!

Thread-4 args: ('Count thread - 3',)

Це означає, що основний потік застосунку спочатку вивів 'Usefull message' і після нього через 2 секунди п'ять потоків MyThread вивели своє 'Wake up!', і тільки після цього скрипт завершився.

Запустіть для демонстрації живий приклад:

**Потік як функтор**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/create#%D0%BF%D0%BE%D1%82%D1%96%D0%BA-%D1%8F%D0%BA-%D1%84%D1%83%D0%BD%D0%BA%D1%82%D0%BE%D1%80)Є інший спосіб виконати код окремого потоку. Для цього потрібно, щоб код виконання був функтором (функцією або класом, який має метод \_\_call\_\_). Тоді об'єкт можна передати як іменований аргумент target у Thread:

from threading import Thread

from time import sleep

import logging

class UsefulClass():

def \_\_init\_\_(self, second\_num):

self.delay = second\_num

def \_\_call\_\_(self):

sleep(self.delay)

logging.debug('Wake up!')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

t2 = UsefulClass(2)

thread = Thread(target=t2)

thread.start()

print('Some stuff')

Результат:

Some stuff

Thread-1 Wake up!

**Потік у функції**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/create#%D0%BF%D0%BE%D1%82%D1%96%D0%BA-%D1%83-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97)У процесі створення екземпляра класу Thread можна передати аргументу target функцію та передати їй аргументи:

from threading import Thread

from time import sleep

import logging

def example\_work(delay):

sleep(delay)

logging.debug('Wake up!')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

for i in range(5):

thread = Thread(target=example\_work, args=(i,))

thread.start()

Результат буде:

Thread-1 Wake up!

Thread-2 Wake up!

Thread-3 Wake up!

Thread-4 Wake up!

Thread-5 Wake up!

Зверніть увагу, що аргументи, які потрібно передати у функцію, передаються як кортеж args у Thread. Іменовані аргументи для функції можна так само передати як словник kwargs у Thread.

**Очікування виконання потоку**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/create#%D0%BE%D1%87%D1%96%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D1%83)

Коли потрібно в основному застосунку дочекатися виконання потоку, можна скористатися блокуючим методом join.

from threading import Thread

import logging

from time import sleep

def example\_work(params):

sleep(params)

logging.debug('Wake up!')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

logging.debug('Start program')

threads = []

for i in range(5):

thread = Thread(target=example\_work, args=(i,))

thread.start()

threads.append(thread)

[el.join() for el in threads]

logging.debug('End program')

У консолі ви побачите:

MainThread Start program

Thread-1 Wake up!

Thread-2 Wake up!

Thread-3 Wake up!

Thread-4 Wake up!

Thread-5 Wake up!

MainThread End program

Основний потік дочекався [el.join() for el in threads], доки завершаться всі потоки thread зі списку threads, і тільки потім вивів End program.

Ви також можете перевірити — чи виконується потік, викликавши метод is\_alive:

from threading import Thread

from time import sleep

import logging

class UsefulClass:

def \_\_init\_\_(self, second\_num):

self.delay = second\_num

def \_\_call\_\_(self):

sleep(self.delay)

logging.debug('Wake up!')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

t2 = UsefulClass(2)

thread = Thread(target=t2)

thread\_locking = Thread(target=t2)

thread.start()

print(thread.is\_alive(), thread\_locking.is\_alive())

thread\_locking.start()

thread.join()

thread\_locking.join()

print(thread.is\_alive(), thread\_locking.is\_alive())

print('After all...')

True False

Thread-2 Wake up!

Thread-1 Wake up!

False False

After all...

Це може бути корисним, якщо ви хочете перевіряти стан потоку самостійно і не блокувати застосунок в очікуванні завершення.

Запустіть для демонстрації живий приклад:

**Потоки Timer**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/create#%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8-timer)Екземпляри класу Timer починають працювати з деякою затримкою, яку визначає програміст. Крім того, ці потоки можна скасувати будь-якої миті в період затримки. Наприклад, ви передумали стартувати певний потік.

from threading import Timer

import logging

from time import sleep

def example\_work():

logging.debug('Start!')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

first = Timer(0.5, example\_work)

first.name = 'First thread'

second = Timer(0.7, example\_work)

second.name = 'Second thread'

logging.debug('Start timers')

first.start()

second.start()

sleep(0.6)

second.cancel()

logging.debug('End program')

Тут ми запланували виконання двох потоків, через 0.5 та 0.7 секунд. Але потім через 0.6 секунди скасували виконання другого потоку second.cancel()

Отримаємо таке виведення:

MainThread Start timers

First thread Start!

MainThread End program

**Контроль доступу до ресурсів**

Оскільки ОС може на будь-якому виклику перервати виконання потоку та передати контроль іншому потоку, ви не можете бути впевненим, що робота із загальним ресурсом буде коректно завершеною і ресурс не опиниться в невизначеному стані.

**Блокування**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/control#%D0%B1%D0%BB%D0%BE%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)Для цього є механізм блокування. У Python є два примітива блокувань: Lock та RLock. Lock трохи швидший і більш низькорівневий, але він не рекурсивний і може бути ситуація потрапляння в DeadLock, коли виконання коду заблокується, кілька потоків чекатимуть, доки хтось віддасть Lock, а його ніхто ніколи вже не віддасть. Це і є ситуація, коли програма "зависла".

RLock трохи повільніший, зате виключає взаємне блокування. Рекомендується завжди використовувати саме його, якщо немає вагомих причин використовувати Lock.

NOTE

Якщо провести аналогію з життя, то Lock це коли у кожного потоку один і той самий ключ і будь-який потік може відкрити замок, хто б його не закрив із потоків. З RLock ситуація трохи інша, у кожного потоку свій ключ і свій замок.

Потік може відкрити лише свій замок своїм ключем і не відкриє замок, якщо його закрив інший потік.

from threading import Thread, RLock

import logging

from time import time, sleep

lock = RLock()

def func(locker, delay):

timer = time()

locker.acquire()

sleep(delay)

locker.release()

logging.debug(f'Done {time() - timer}')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

t1 = Thread(target=func, args=(lock, 2))

t2 = Thread(target=func, args=(lock, 2))

t1.start()

t2.start()

logging.debug('Started')

У цьому прикладі ми запустили два потоки і один загальний RLock. У консолі ви побачите:

MainThread Started

Thread-1 Done 2.0003550052642822

Thread-2 Done 4.000735521316528

Таке виведення означає, що один із потоків "взяв" lock і поки він його не "відпустив", інший чекав доки lock звільниться. Блокування ресурсу досягається виконанням команди locker.acquire(). Це робиться, щоб загальним ресурсом міг користуватися лише один потік на один момент часу, і лише коли потік закінчить роботу із загальним ресурсом, він відпускає lock, у нашому випадку команда locker.release(), і хтось інший зможе попрацювати з ресурсом. Так гарантується, що загальний ресурс не потрапить у невизначений стан, коли хтось почав із ним роботу та не закінчив, а хтось інший почав, і так далі.

Але найчастіше для блокування використовують контекст виконання:

from threading import Thread, RLock

import logging

from time import time, sleep

lock = RLock()

def func(locker, delay):

timer = time()

with locker:

sleep(delay)

logging.debug(f'Done {time() - timer}')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

t1 = Thread(target=func, args=(lock, 2))

t2 = Thread(target=func, args=(lock, 2))

t1.start()

t2.start()

logging.debug('Started')

Результат буде тим самим:

MainThread Started

Thread-1 Done 2.0014190673828125

Thread-2 Done 4.002039194107056

Запустіть для демонстрації живий приклад:

**Семафори**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/control#%D1%81%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80%D0%B8)

Другий примітив синхронізації — це *семафори*.

Семафори підходять до блокування іншим шляхом та вказують, що кілька потоків можуть користуватися ресурсом одночасно і цим обмежують кількість потоків. Наприклад, ми не хочемо надсилати десятки тисяч запитів до мережі одночасно, щоб не створювати навантаження на обладнання і вкажемо семафор, щоб не більше ста потоків могли одночасно надсилати запити. Щойно якийсь із потоків закінчить роботу і семафор його відпустить, то наступний потік із черги очікування зможе зробити свій запит.

Як приклад розглянемо виконання 10 потоків і обмежимо виконання за допомогою семафору до двох одночасно:

from threading import Semaphore, Thread

import logging

from time import sleep

def worker(condition):

with condition:

logging.debug(f'Got semaphore')

sleep(1)

logging.debug(f'finished')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

pool = Semaphore(2)

for num in range(10):

thread = Thread(name=f'Th-{num}', target=worker, args=(pool, ))

thread.start()

У цьому прикладі ми створили семафор, що обмежує до 2. Виконавши код, побачимо:

Th-0 Got semaphore

Th-1 Got semaphore

Th-1 finished

Th-0 finished

Th-2 Got semaphore

Th-3 Got semaphore

Th-3 finished

Th-2 finished

Th-4 Got semaphore

Th-5 Got semaphore

Th-5 finished

Th-4 finished

Th-6 Got semaphore

Th-7 Got semaphore

Th-7 finished

Th-6 finished

Th-8 Got semaphore

Th-9 Got semaphore

Th-9 finished

Th-8 finished

Результат у вас може відрізнятися, це справа випадку, коли і який потік візьме семафор. Але суть буде та сама, один потік чекатиме своєї черги, доки семафор звільниться. Крім того, якщо потоки одночасно почнуть писати в консоль їх виведення може перемішатися і матиме вигляд, начебто дві людини одночасно намагаються набрати повідомлення на одній клавіатурі

Запустіть для демонстрації живий приклад:

Наступним кроком ми розглянемо синхронізацію роботи потоків за допомогою умов та подій.

**Синхронізація потоків**

**Condition**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/sync#condition)Є примітиви синхронізації, які дозволяють потокам очікувати сигнал від інших потоків — це Condition. Створимо дві різні функції: одна master повідомлятиме, що worker може продовжити роботу. При цьому вони будуть виконуватись у різних потоках.

from threading import Thread, Condition

import logging

from time import sleep

def worker(condition: Condition):

logging.debug('Worker ready to work')

with condition:

condition.wait()

logging.debug('The worker can do the work')

def master(condition: Condition):

logging.debug('Master doing some work')

sleep(2)

with condition:

logging.debug('Informing that workers can do the work')

condition.notify\_all()

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

condition = Condition()

master = Thread(name='master', target=master, args=(condition,))

worker\_one = Thread(name='worker\_one', target=worker, args=(condition, ))

worker\_two = Thread(name='worker\_two', target=worker, args=(condition,))

worker\_one.start()

worker\_two.start()

master.start()

logging.debug('End program')

Спочатку master у потоці виконує якусь роботу. Після цього він виконує метод condition.notify\_all() чим дозволяє запустити інші потоки, доки вони чекають виконання. Вони очікують на виконання в точці виклику методу condition.wait().

Після виконання коду прикладу, у консолі ви побачите:

worker\_one Worker ready to work

worker\_two Worker ready to work

master Master doing some work

MainThread End program

master Informing that workers can do the work

worker\_one The worker can do the work

worker\_two The worker can do the work

Якщо ж master повинен дозволити роботу лише одному з worker, можна викликати метод condition.notify(), тоді тільки один з тих, хто очікує дозволу worker продовжить роботу. Другий чекатиме, доки не буде виконано наступне condition.notify.

**Event**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/sync#event)Інший примітив синхронізації — це потокобезпечний прапорець класу Event. Клас Event має внутрішній прапорець, який можуть встановлювати або скидати інші потоки. Для цього використовують метод set, щоб встановити прапор та метод clear для скидання. Методу wait класу Event зупиняє роботу потоку до того часу, доки інший потік не встановить прапор методом set. Є можливість перевірити, чи встановлено прапор методом is\_set.

Таким чином, master може встановити прапорець класу Event, і всі worker потоки продовжать роботу тільки після отримання дозволу.

from threading import Thread, Event

import logging

from time import sleep

def worker(event: Event):

logging.debug('Worker ready to work')

event.wait()

logging.debug('The worker can do the work')

def master(event: Event):

logging.debug('Master doing some work')

sleep(2)

logging.debug('Informing that workers can do the work')

event.set()

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

event = Event()

master = Thread(name='master', target=master, args=(event, ))

worker\_one = Thread(name='worker\_one', target=worker, args=(event, ))

worker\_two = Thread(name='worker\_two', target=worker, args=(event,))

worker\_one.start()

worker\_two.start()

master.start()

logging.debug('End program')

Виведення буде таке саме, як і для Condition:

worker\_one Worker ready to work

worker\_two Worker ready to work

master Master doing some work

MainThread End program

master Informing that workers can do the work

worker\_one The worker can do the work

worker\_two The worker can do the work

Виникає закономірне питання, навіщо, якщо результат той самий? Справа в тому, що ми можемо керувати виконанням, перезапуском та зупинкою роботи потоків через клас Event. Наприклад, у наступному прикладі ми перериваємо виконання потоку, який працює в нескінченному циклі та інакше просто ніколи не завершиться.

from threading import Thread, Event

import logging

from time import sleep

def example\_work(event\_for\_exit: Event):

while True:

sleep(1)

logging.debug('Run event work')

if event\_for\_exit.is\_set():

break

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

event = Event()

thread = Thread(target=example\_work, args=(event,))

thread.start()

sleep(5)

event.set()

logging.debug('End program')

Виведення:

Thread-1 Run event work

Thread-1 Run event work

Thread-1 Run event work

Thread-1 Run event work

MainThread End program

Thread-1 Run event work

Запустіть для демонстрації живий приклад:

**Barrier**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-01/sync#barrier)Останній примітив синхронізації, який ми розглянемо в Python — це бар'єр Barrier. Він дозволяє задати умову, щоб кілька потоків продовжили роботу лише після того, як задане число потоків добереться у виконанні коду до цього "бар'єру".

Використовується, коли вам потрібно, щоб робота застосунку продовжилася лише після того, як усі потоки зроблять якусь частину своєї роботи та дійдуть до деякої точки, з якою можна знову продовжувати.

Розглянемо наступний приклад:

from random import randint

from threading import Thread, Barrier

import logging

from time import sleep, ctime

def worker(barrier: Barrier):

logging.debug(f'Start thread: {ctime()}')

sleep(randint(1, 3)) # Simulate some work

r = barrier.wait()

logging.debug(f'count: {r}')

logging.debug(f'Barrier overcome: {ctime()}')

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

barrier = Barrier(5)

for num in range(10):

thread = Thread(name=f'Th-{num}', target=worker, args=(barrier, ))

thread.start()

Виведення:

Th-0 Start thread: Tue Oct 18 12:13:21 2022

Th-1 Start thread: Tue Oct 18 12:13:21 2022

Th-2 Start thread: Tue Oct 18 12:13:21 2022

Th-3 Start thread: Tue Oct 18 12:13:21 2022

Th-4 Start thread: Tue Oct 18 12:13:21 2022

Th-5 Start thread: Tue Oct 18 12:13:21 2022

Th-6 Start thread: Tue Oct 18 12:13:21 2022

Th-7 Start thread: Tue Oct 18 12:13:21 2022

Th-8 Start thread: Tue Oct 18 12:13:21 2022

Th-9 Start thread: Tue Oct 18 12:13:21 2022

Th-4 count: 4

Th-0 count: 0

Th-1 count: 1

Th-6 count: 2

Th-3 count: 3

Th-4 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-0 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-1 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-6 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-3 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-7 count: 4

Th-2 count: 0

Th-9 count: 3

Th-8 count: 1

Th-5 count: 2

Th-7 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-2 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-9 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-8 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-5 Barrier overcome: Tue Oct 18 12:13:24 2022

Потік може дістатися бар'єру і чекати його за допомогою функції wait(). Це блокуючий виклик, який повернеться, коли решта потоків (попередньо налаштована кількість barrier = Barrier(5)) дістануться бар'єру.

Функція очікування wait() повертає ціле число, яка вказує на кількість учасників, що залишилися до бар'єру. Якщо потік був останнім, що прибув, то повернене значення буде нульовим.

Як бачимо в нашому прикладі, спочатку виконалися потоки

Th-4 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-0 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-1 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-6 Barrier overcome: Tue Oct 18 12:13:22 2022

Th-3 Barrier overcome: Tue Oct 18 12:13:22 2022

А потім наступні п'ять потоків

Th-7 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-2 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-9 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-8 Barrier overcome: Tue Oct 18 12:13:24 2022

Th-5 Barrier overcome: Tue Oct 18 12:13:24 2022

Головна вимога, щоб кількість потоків, що запускаються, в нашому випадку 10 - range(10), була кратною кількості бар'єру, у нашому випадку 5 - Barrier(5).

Запустіть для демонстрації живий приклад (порядок виведення може відрізнятися):

**Пул потоків**

В Python існує ще один механізм написання асинхронного коду. Ви можете скористатися пакетом [concurrent.futures](https://docs.python.org/3/library/concurrent.futures.html). Він дозволяє піднятися на вищий рівень абстракції, коли вам просто потрібно паралельно виконати ряд однотипних завдань і немає необхідності вдаватися до низькорівневих деталей реалізації.

Основна ідея полягає у використанні реалізації абстрактного класу Executor. У concurrent.futures є дві реалізації цього абстрактного базового класу: ProcessPoolExecutor — для виконання коду окремих процесів (з ним ми познайомимося пізніше) та ThreadPoolExecutor — для виконання в окремих потоках.

Кожен такий Executor приховує набір потоків або процесів, яким ви можете дати роботу та отримати результат її виконання. Вам не потрібно вручну управляти створенням потоків та їх коректним завершенням.

Звичайно, все ще потрібно пам'ятати про доступ до загальних ресурсів та примітиви синхронізації.

import concurrent.futures

import logging

from random import randint

from time import sleep

def greeting(name):

logging.debug(f'greeting for: {name}')

sleep(randint(0, 3))

return f"Hello {name}"

arguments = (

"Bill",

"Jill",

"Till",

"Sam",

"Tom",

"John",

)

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

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s %(message)s')

with concurrent.futures.ThreadPoolExecutor(max\_workers=2) as executor:

results = list(executor.map(greeting, arguments))

logging.debug(results)

У цьому прикладі ми створюємо ThreadPoolExecutor та задаємо, що робота буде виконуватися не більше ніж 2 потоками. Пул можна створити в менеджері контексту як у прикладі, щоб бути впевненим, що всі ресурси будуть коректно повернуті до системи після завершення. Але це не обов'язково, можна створити пул і потім викликати у нього метод shutdown, щоб завершити всі потоки та повернути ресурси системі.

Далі ми передаємо в пул функцію, яку потрібно виконати у кілька потоків, та набір аргументів цієї функції — кожен для виконання в окремому потоці. В executor є метод map, він використовується, коли потрібно паралельно виконати функцію з різними вхідними аргументами в окремих потоках.

Результатом виклику map буде ітератор за результатами виконання в окремих потоках.

У консолі ви побачите щось схоже:

ThreadPoolExecutor-0\_0 greeting for: Bill

ThreadPoolExecutor-0\_1 greeting for: Jill

ThreadPoolExecutor-0\_0 greeting for: Till

ThreadPoolExecutor-0\_1 greeting for: Sam

ThreadPoolExecutor-0\_0 greeting for: Tom

ThreadPoolExecutor-0\_1 greeting for: John

MainThread ['Hello Bill', 'Hello Jill', 'Hello Till', 'Hello Sam', 'Hello Tom', 'Hello John']

Результат, звичайно, може відрізнятися через випадкові затримки.

Зверніть увагу, що код паралельно виконується не більше ніж двома потоками — ThreadPoolExecutor-0\_0 та ThreadPoolExecutor-0\_1.

Результат повертається лише після того, як усі вхідні дані оброблені.