**План лекції: Детальніше про асинхронне програмування**

**1. Вступ в асинхронне програмування**

* Що таке асинхронне програмування?
* Переваги асинхронного програмування (підвищення ефективності при роботі з I/O, зменшення часу очікування).

**2. Основи асинхронного програмування в Python**

* Коротке повторення основ:
  + async def і await.
  + asyncio як стандартна бібліотека для асинхронного програмування.
* Принципи роботи з асинхронними функціями та задачами.

**3. Асинхронні генератори**

* Що таке асинхронні генератори?
* Синтаксис і використання асинхронних генераторів:
  + async def і yield.
  + Взаємодія з асинхронними генераторами.
* Приклад коду: створення асинхронного генератора для читання даних з файлів або мережі.

**4. Асинхронні контекстні менеджери**

* Що таке асинхронні контекстні менеджери?
* Синтаксис і використання асинхронних контекстних менеджерів:
  + async def \_\_aenter\_\_ і async def \_\_aexit\_\_.
* Приклад коду: реалізація асинхронного контекстного менеджера для роботи з асинхронними ресурсами (наприклад, з’єднаннями з базами даних).

**5. Паралелізм за допомогою asyncio**

* Огляд модуля asyncio.
* Основні компоненти:
  + asyncio.run(), asyncio.create\_task(), asyncio.gather().
* Приклади:
  + Запуск асинхронних задач паралельно.
  + Збір результатів виконання кількох асинхронних задач.

**6. Обробка помилок в асинхронному коді**

* Як обробляти виключення в асинхронних функціях.
* Використання try/except з await.
* Приклади коду: обробка помилок в асинхронних функціях.

**7. Практичні приклади і кейси**

* Розгляд реальних прикладів використання асинхронного програмування:
  + Веб-скрейпінг.
  + Робота з API.
  + Асинхронне читання і запис даних.

**8. Тестування асинхронного коду**

* Як тестувати асинхронні функції і задачі.
* Використання бібліотеки pytest для асинхронного тестування:
  + pytest-asyncio.

**9. Проблеми і обмеження асинхронного програмування**

* Поширені помилки і проблеми.
* Обмеження асинхронного програмування і його вплив на продуктивність.

**10. Запитання і відповіді**

* Відповіді на запитання студентів.
* Обговорення можливих питань і труднощів, з якими можна зіткнутися.

**11. Домашнє завдання**

* Реалізувати кілька асинхронних функцій та генераторів.
* Написати тестові кейси для асинхронного коду.

**Матеріали та ресурси:**

* Офіційна документація Python для асинхронного програмування.
* Приклади коду і вправи для самостійної роботи.
* Ресурси для поглибленого вивчення (блоги, статті, відео).

https://habr.com/ru/articles/830720/

Обзор генераторов

Если вы уже знакомы с генераторами, пропустите эту часть, но если нет, то именно на них построен asyncio, поэтому очень важно понимать, как они работают.

Прежде всего, причина, по которой генераторы существуют, заключается в том, что они позволяют сделать ваш код более эффективным с точки зрения памяти. Представьте, что у вас есть следующий цикл:

for i in range(100\_000\_000):

print(i)

Если бы range был не генератором, а функцией, возвращающей список для последующего его перебора, код, подобный вышеприведенному примеру, был бы очень неэффективным в плане использования памяти, поскольку вы бы создали список из 100 миллионов элементов... Однако, поскольку range это генератор, по крайней мере в Python 3+, вы генерируете числа только по мере необходимости, одно за другим, не сохраняя всю последовательность в памяти.

Есть несколько способов создания генераторов, но мы сосредоточимся на функциях генераторов. Эти генераторы определяются как любые другие функции, но используют для возврата данных оператор yield. Этот оператор превращает обычную функцию в генератор, которая вместо того, чтобы выполняться сразу, может приостанавливать и возобновлять свое состояние при вызове next(iterator).

Возьмем, к примеру, следующую функцию-генератор:

def generator():

yield 'hello'

yield 'world'

iterator = generator()

Когда вы вызываете генератор, Python видит ключевое слово yield и вместо того, чтобы запустить код внутри функции, как он это обычно делает — возвращает объект генератора. Как только у нас есть объект генератора, мы можем вызвать next(iterator) и он запустит код функции до первого/следующего оператора yield:

print(next(iterator)) # Output: hello

print(next(iterator)) # Output: world

Если мы попробуем вызвать next(iterator) еще раз, генератор вызовет исключение StopIteration, поскольку в функции генератора больше нет операторов yield.

Еще одна интересная функция генераторов Python — это yield from — возможность из генератора вызывать подгенератор или итерируемый объект, что позволяет вам создавать цепочку генераторов!

def generator():

yield 'hello'

def another\_generator():

yield from generator()

iterable = another\_generator()

print(next(iterable)) # Output: hello

Генераторы — это нечто большее, чем я говорю, например, генераторные включения, которые похожи на списочные включения, но создаются с круглыми скобками вместо квадратных, и возможность отправлять данные генераторам с помощью iterator.send(value). Однако для данной статьи важно помнить о генераторах, что они позволяют запускать и останавливать функцию, сохраняя ее состояние!

Цикл событий

Цикл событий, который отвечает за запуск и управление всеми текущими задачами, является ядром asyncio и первое, что мы воссоздадим с помощью генераторов. Хотя цикл событий asyncio написан на языке C, его проще всего представить в виде списка, содержащего все текущие задачи. Пока что представьте эти задачи просто как объекты-генераторы. Менеджер цикла событий будет проходить по каждой задаче в списке и использовать функцию next(task) для запуска каждой из них. Затем эта задача будет запущена, и когда она будет выполнять работу, связанную с вводом-выводом, например, спать, она будет использовать ключевое слово yield чтобы приостановить свое выполнение и вернуть управление этому циклу событий, который затем перейдет к следующей задаче в цикле.

Вот пример этого — у нас есть две задачи, обе печатают свой номер и затем приостанавливают свое выполнение. Поскольку менеджер цикла событий — это тот, кто вызывает next(), после того как задача приостановлена, он получает управление обратно и затем переходит к выполнению следующей задачи в цикле.

def task1():

while True:

print('Task 1')

yield

def task2():

while True:

print('Task 2')

yield

event\_loop = [task1(), task2()]

while True:

for task in event\_loop:

next(task)

Впоследствии вывод этого кода будет выглядеть следующим образом и будет продолжаться до бесконечности, поскольку обе функции генератора никогда не завершатся из-за циклов while True.

Task 1

Task 2

Task 1

Task 2

…

Sleep

Если мы возьмем тот же код сверху, мы можем добавить подгенераторы к нашим задачам с помощью yield from. Ниже я добавил спящий генератор, который приостановит выполнение задач, пока не истечет указанное время. Это работает, потому что sleepбудет продолжать выдавать результат, пока не пройдет определенное количество секунд, после чего он выйдет из цикла while. Поскольку в sleep больше нет операторов yield, возникает исключение StopIteration, которое сигнализирует оператору yield from в функциях задач о необходимости перейти к следующей строке кода.

import time

def sleep(seconds):

start\_time = time.time()

while time.time() - start\_time < seconds:

yield

def task1():

while True:

print('Task 1')

yield from sleep(1)

def task2():

while True:

print('Task 2')

yield from sleep(5)

event\_loop = [task1(), task2()]

while True:

for task in event\_loop:

next(task)

Выход:

Task 1

Task 2

Task 1

Task 1

Task 1

Task 1

Task 2

Task 1

…

От yield к await

Теперь мы можем взять приведенный выше код и перейти от использования yield к await с помощью dunder-метода \_\_await\_\_ и ключевого слова async. Когда класс имеет метод \_\_await\_\_, мы можем использовать ключевое слово await перед экземпляром класса, чтобы вызвать его. В asyncio вы обычно работаете с объектами Task через функцию, например asyncio.create\_task. Эти объекты Task наследуются от объекта asyncio Future, у которого есть метод \_\_await\_\_ . Мы также можем использовать await перед корутиной, которая является объектом, созданным при вызове функции с ключевым словом async перед ней. Корутины похожи на функции-генераторы в том смысле, что выполнение корутины также можно приостанавливать и возобновлять.

Можете думать о ключевом слове await как о простом синониме yield from с некоторыми дополнительными правилами проверки. Таким образом, при написании кода await object вы в основном говорите либо yield из метода \_\_await\_\_ в экземпляре класса «object», ИЛИ «object» может быть другой корутиной (вроде подгенератора).

На самом деле вы можете взглянуть на исходный код Asyncio и увидеть, что метод \_\_await\_\_ внутри объекта Future по сути просто вызывает yield если Future (или Task) не завершено:

Исходный код Asyncio. Метод \_\_await\_\_ в объекте Future

Чтобы перенести код, который мы написали в разделе выше, на использование async и await, нам сначала нужно создать свой собственный класс Task, поскольку функция не может иметь dunder-метод \_\_await\_\_. Ниже приведена простая версия, которую я придумал:

from queue import Queue

event\_loop = Queue()

class Task():

def init(self, generator):

self.iter = generator

self.finished = False

def done(self):

return self.finished

def await(self):

while not self.finished:

yield self

def create\_task(generator):

task = Task(generator)

event\_loop.put(task)

return task

На этот раз вместо использования списка Python для создания цикла событий мы используем очередь, что имеет немного более осмысленно, поскольку мы хотим иметь возможность добавлять и удалять задачи из цикла за постоянное время.

Для нашего класса Task мы сохраняем объект генератора в self.iter а также устанавливаем self.finished в False, который будет отслеживать, закончил ли генератор работу (он завершает работу, когда вызывается StopIteration). У нашего объекта Task также есть dunder-метод \_\_await\_\_, который будет просто продолжать возвращать управление циклу событий, пока задача не будет завершена. Наконец, после создания объекта Task с помощью вспомогательной функции create\_task мы добавляем его в цикл событий, который планирует его запуск.

Теперь давайте создадим менеджер цикла событий, который будет запускать задачи:

def run(main):

event\_loop.put(Task(main))

while not event\_loop.empty():

task = event\_loop.get()

try:

task.iter.send(None)

except StopIteration:

task.finished = True

else:

event\_loop.put(task)

Можете заметить, что это начинает имитировать фактический API asyncio, поскольку для запуска цикла событий нам нужно вызвать начальную функцию run. Функция сначала оборачивает основную функцию в объект Task и добавляет ее в цикл событий. Затем будет запущен цикл While, и для каждого прогона будем получать через очередь

следующую задачу для запуска. Вместо использования next(task.iter) теперь нам нужно использовать task.iter.send(None), что является просто странной особенностью работы с ключевыми словами async/await, но делает то же самое. Также мы хотим обернуть этот вызов в блок try-except, поскольку если StopIteration выбрасывается исключение, мы можем установить task.finished в True, но если никаких исключений не возникает,

код перейдет к оператору else, который добавляет задачу обратно в цикл событий для повторного запуска.

Далее нам нужно сделать функцию sleep асинхронно-совместимой. Раньше мы использовали функцию-генератор с циклом while и одну yield для управления засыпанием. Мне нравится этот подход, но вы не можете использовать ключевое слово await в сочетании с функцией-генератором — это должен быть объект с dunder-методом \_\_await\_\_ или функция корутины. Поэтому, чтобы решить эту проблему, я переместил код в другую функцию, и теперь фактическая функция sleep создает объект задачи, а затем ожидает его. Этот await вызывает метод \_\_await\_\_ внутри объекта Task, который затем выдаст результат, позволяя циклу событий перейти к другой задаче. Когда цикл событий доберется до новой задачи \_sleep, он проверит время, и если прошло недостаточно времени, также вызовет yield чтобы вернуть управление циклу событий. Если спящая задача снова вызывается через цикл событий, подобно тому, как генератор сохраняет свое состояние, корутина все равно будет ожидать возврата sleep, и поскольку sleep все еще будет ожидать завершения задачи \_sleep, dunder-метод задачи \_\_await\_\_ снова будет вызван, а поскольку задача не завершена, будет вызван метод yield в dunder-методе.

import time

def \_sleep(seconds):

start\_time = time.time()

while time.time() - start\_time < seconds:

yield

async def sleep(seconds):

task = create\_task(\_sleep(seconds))

return await task

Вот весь код вместе:

from queue import Queue

import time

event\_loop = Queue()

def \_sleep(seconds):

start\_time = time.time()

while time.time() - start\_time < seconds:

yield

async def sleep(seconds):

task = create\_task(\_sleep(seconds))

return await task

class Task():

def \_\_init\_\_(self, generator):

self.iter = generator

self.finished = False

def done(self):

return self.finished

def \_\_await\_\_(self):

while not self.finished:

yield self

def create\_task(generator):

task = Task(generator)

event\_loop.put(task)

return task

def run(main):

event\_loop.put(Task(main))

while not event\_loop.empty():

task = event\_loop.get()

try:

task.iter.send(None)

except StopIteration:

task.finished = True

else:

event\_loop.put(task)

Теперь, когда мы создали цикл событий, способ создания задач и функцию sleep, мы можем импортировать файл (называемый «jacobio.py») и взять код из прошлой версии, когда мы использовали yield-ы. Заменим все операторы yield from на await, добавим ключевое слово async к функциям с await, чтобы обозначить, что эти функции могут быть ожидающими, а затем создать основную функцию, как в asyncio, чтобы добавить задачи в цикл событий:**import** jacobio  
  
**async** **def** task1():  
    **for** \_ **in** range(2):  
        print('Task 1')  
        **await** jacobio.sleep(1)  
  
**async** **def** task2():  
    **for** \_ **in** range(3):  
        print('Task 2')  
        **await** jacobio.sleep(0)  
  
**async** **def** main():  
    one = jacobio.create\_task(task1())  
    two = jacobio.create\_task(task2())  
  
    **await** one  
    **await** two  
      
    print('done')  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
    jacobio.run(main())

Выход:

Task 1  
Task 2  
Task 2  
Task 2  
Task 1  
done

## **Ожидание с AsyncIO**

Теперь мы можем взять наш код выше и заменить все вхождения «jacobio» на «asyncio», и теперь мы полностью используем пакет asyncio!

**import** asyncio  
**async** **def** task1():  
    **for** \_ **in** range(2):  
        print('Task 1')  
        **await** asyncio.sleep(1)  
**async** **def** task2():  
    **for** \_ **in** range(3):  
        print('Task 2')  
        **await** asyncio.sleep(0)  
**async** **def** main():  
    one = asyncio.create\_task(task1())  
    two = asyncio.create\_task(task2())  
    **await** one  
    **await** two  
      
    print('done')  
**if** name == 'main':  
    asyncio.run(main())

What's a Task

Before talking about tasks, it’s important to understand how Asyncio coroutines work since a task object is just a coroutine wrapper that can run asynchronously.

Coroutines

To create a coroutine object, all you need to do is add the async keyword to the front of a function/method definition. This signifies that the function is able to be stopped and started via the event loop (if the coroutine has its own await keywords). When calling a coroutine function, instead of the actual function being run, a coroutine object is created. This object then needs to be waited upon with the await keyword, which will then run the code in the coroutine.

Below is an example of how to create a coroutine and then run the code inside of the coroutine object using await:

import asyncio

async def my\_function():

print(‘Hello World’)

async def main():

coro = my\_function()

print(type(coro))

await coro

asyncio.run(main())

In the example, we first call my\_function, which creates the coroutine object, which we can see via the print statement. Then to print “Hello World,” we use the await keyword to pause the execution of main and run my\_function. The full output is:

<class ‘coroutine’>

Hello World

Scheduled Coroutines

After making a coroutine, you generally want to wrap it in an asyncio.Task object. Creating a task will automatically schedule the coroutine to be run by appending the task object to the event loop (basically just a list of task objects). To create a task object, you can use the asyncio.create\_task(coro, \*, name=None, context=None) function. This function takes a coroutine object as well as two optional keyword arguments. The first optional argument is name, which lets you give the task object a name so that you can remember what it does. You can also set the name after creating the task object via Task.set\_name(name) and then get the task name via Task.get\_name(). The other keyword argument, added in 3.11, is context, which lets you give the task a context variable to implement local storage inside the task (sort of similar to the Threading.local() object but for async tasks). The name argument can be pretty useful, but context is pretty rarely used.

Another important thing to note is that the event loop only stores weak references to the task objects. So, if you just write asyncio.create\_task(my\_function()), the garbage collector will destroy the task. Instead, you need to store a non-weak reference to all tasks by storing the task object returned from create\_task in either a variable or other object.

Here’s a basic example of using a task object:

import asyncio

async def my\_function():

print(‘Hello World’)

async def main():

task = asyncio.create\_task(my\_function())

print(type(task))

await task

asyncio.run(main())

Output:

<class ‘\_asyncio.Task’>

Hello World

Besides just waiting for the task to finish, you can also cancel it with Task.cancel(), add a callback function to be called when the task finishes with Task.add\_done\_callback(cb), manually check if the coroutine is done running with Task.done(), or get the result of the wrapped coroutine when the task is done with Task.result(); checkout the complete list of Task methods in Python’s documentation.

Here’s the same example from above, but with some of these task methods in use:

import asyncio

async def my\_function():

return ‘Hello World!‘

async def main():

task = asyncio.create\_task(my\_function())

print(task.done()) # Will print False

await task

print(task.done()) # Will print True

print(task.result()) # Will print Hello World!

asyncio.run(main())

While we generally create tasks and then use some method to wait for them to be done, if you want to create a task and then simply forget about it, you can utilize the following pattern. This is straight out of the Asyncio documentation; it creates tasks, adds them to a set so that there is a reference to them, and then when the task is done, it is discarded from the set via a callback.

background\_tasks = set()

for \_ in range(10):

task = asyncio.create\_task(some\_coro())

background\_tasks.add(task)

task.add\_done\_callback(background\_tasks.discard)

Wait for a Single Task

Now that we've discussed coroutines and task objects, we're ready to talk about ways to manage them more elegantly. While the await keyword is fundamental for pausing the current coroutine until the awaitable (such as a coroutine, task, or future) being waited upon with await completes, it inherently focuses on one operation at a time. This article is leading towards using Asyncio's built-in functions to aggregate multiple tasks into a single awaitable object and then use await on that single object.

While most of Asyncio’s functions are for waiting on multiple tasks, there’s one function meant to wait for a single awaitable called wait\_for. Let’s talk about that one first:

asyncio.wait\_for

The next step up from a simple await is the wait\_for function.

asyncio.wait\_for(aw, timeout)

This function takes in a single awaitable object (it automatically wraps coroutines in a task object so that it can be run on the event loop) and waits for it to be done. But unlike a simple await, It also allows you to add a timeout - if the task takes longer than the allotted time to finish, a TimeoutError will be raised, and the task inside of wait\_for is canceled.

async def slow\_function():

await asyncio.sleep(100)

async def main():

try:

await asyncio.wait\_for(slow\_function(), timeout=5.0)

except TimeoutError:

print(‘Function was too slow :(‘)

asyncio.run(main())

Because the coroutine function tried to sleep for 100 seconds, a TimeoutError was raised since the timeout in wait\_for was only set to 5 seconds:

Function was too slow :(

Wait for Multiple Tasks

Now, let’s get to the interesting stuff - awaiting multiple tasks! There are three main ways to wait for a collection of tasks; each has its pros and cons and can be helpful in different scenarios.

asyncio.wait

Our first option is similar to wait\_for but for a collection (such as a list, tuple, or set) of either task or lower-level future objects.

asyncio.wait(collection\_of\_tasks, \*, timeout=None, return\_when=ALL\_COMPLETED)

This function returns a tuple containing two sets; the first set is the finished tasks, and the second are the ones that aren’t done yet. Tasks that are finished before the timeout or the return\_when directive go into the finished task set, and ones that aren’t are put in the second set, often called pending or \_ if you don’t plan on using them. However, unlike ‘asyncio.wait`, the unfinished tasks are not canceled when a timeout occurs.

The return\_when argument lets you tell asyncio.wait to return when one of the following three things happen:

FIRST\_COMPLETED returns when the first task finishes or is canceled.

FIRST\_EXCEPTION returns when one of the tasks causes an exception or they all finish.

ALL\_COMPLETED is the default and will return when all futures are finished or canceled.

Let’s see this in action:

import asyncio

import random

async def job():

await asyncio.sleep(random.randint(1, 5))

async def main():

tasks = [

asyncio.create\_task(job(), name=index)

for index in range(1, 5)

]

done, pending = await asyncio.wait(tasks, return\_when=asyncio.FIRST\_COMPLETED)

print(f’The first task completed was {done.pop().get\_name()}’)

asyncio.run(main())

Output:

The first task completed was 4

asyncio.gather

Next, let’s take a look at asyncio.gather(\*aws, return\_exceptions=False).

Unlike wait\_for, which takes a collection of only tasks or future objects, gather takes any number of tasks, futures, or even coroutine objects as a bunch of positional arguments. However, any coroutine objects passed into the function are automatically scheduled as task objects so that they can run on the event loop. Once all the tasks are finished, all return values, obtained via Task.result(), are returned as a single list. One of the nicest things about gather is that the list being returned holds the tasks in the exact order in which they were passed into the function!

Another nice thing about gather is that it’s the only one of the three functions that can gracefully return exceptions. If the return\_exceptions keyword argument is set to True, the returned list will contain any exception caused by the tasks in place of where the task’s result value would have been.

Let’s look at an example of how this works:

import asyncio

import random

async def job(id):

print(f’Starting job {id}’)

await asyncio.sleep(random.randint(1, 3))

print(f’Finished job {id}’)

return id

async def main():

# create a list of worker tasks

coros = [job(i) for i in range(4)]

# gather the results of all worker tasks

results = await asyncio.gather(\*coros)

# print the results

print(f’Results: {results}’)

asyncio.run(main())

First, we make a list of coroutine objects and then use the \* symbol to unpack them into positional arguments for the gather function to accept. Awaiting the object returned by gather will then start running the tasks until they are eventually all done. Notice how the resulting list of return values matches the order in which the tasks were put into gather despite the tasks finishing in a random order due to await asyncio.sleep(random.randint(1,3)).

Starting job 0

Starting job 1

Starting job 2

Starting job 3

Finished job 3

Finished job 0

Finished job 1

Finished job 2

Results: [0, 1, 2, 3]

In the next example, I’m putting two coroutines directly into gather and are setting return\_exceptions to True, which gracefully returns the exceptions in the same result list:

import asyncio

async def task1():

raise ValueError()

async def task2():

raise KeyError()

async def main():

results = await asyncio.gather(task1(), task2(), return\_exceptions=True)

print(results) # Will print [ValueError(), KeyError()]

asyncio.run(main())

One last feature of asyncio.gather is that just like how you can cancel an individual task with Task.cancel(), the object that gather returns (to then be awaited) has its own cancel() method which will loop through all of the tasks that it’s managing and cancel all of them.

asyncio.as\_completed

The next one is a bit different from the above two; Instead of returning a set or list of the results all at once, as\_completed returns an iterable object that lets you handle results as they finish! It accepts all types of awaitables (generally either coroutines, tasks, and futures) and like many of the other methods, also has a timeout keyword argument, which will raise a TimeoutError if the tasks haven’t finished by the specified time.

asyncio.as\_completed(aws, \*, timeout=None)

The following is an example of how as\_completed works:

import asyncio

async def my\_task(id):

return f’I am number {id}’

async def main():

tasks = [my\_task(id) for id in range(5)]

for coro in asyncio.as\_completed(tasks):

result = await coro

print(result)

asyncio.run(main())

We can see that the output is random, as it just prints out whichever finishes first:

I am number 3

I am number 0

I am number 4

I am number 2

I am number 1

Wait for a Group of Tasks

The newest addition to the Asyncio world in Python 3.11 was asyncio.TaskGroup, which simplified the process of managing a group of tasks in the form of a context manager. It ensures that if one task within the group fails, all other tasks are canceled, helping to maintain robust error handling within async code.

Consider a scenario where you have two coroutines, each responsible for calling a different API. Once you get the API responses from both coroutines, you want to enter all the data into a database, but if one API fails, you don’t want to insert anything. Well, this would be a perfect use-case for a TaskGroup since you want both coroutines to finish, but if one fails, you might as well cancel the other one right away.

You can use the tg.create\_task() method to add tasks to a task group. The first time any of the task group tasks fail, all of the other tasks in the group will be canceled. An exception will then bubble up to the coroutine with the task group as an ExceptionGroup or BaseExceptionGroup.

Here's an example of how to use a task group:

import asyncio

async def do\_something():

return 1

async def do\_something\_else():

return 2

async def main():

async with asyncio.TaskGroup() as tg:

task1 = tg.create\_task(do\_something())

task2 = tg.create\_task(do\_something\_else())

print(f’Everything done: {task1.result()}, {task2.result()}’)

asyncio.run(main())

Output:

Everything done: 1, 2

Summary

That’s a lot of ways to handle awaitables, so here’s a summary of what we covered:

await is the most basic form of waiting. You can put this keyword in front of any awaitable object and it will run the code inside of said awaitable; however, await doesn’t let you directly handle multiple tasks at a time.

asyncio.wait\_for handles a single awaitable object like await, but also lets you set a timeout to handle long-running tasks.

asyncio.wait Is like asyncio.wait\_for but accepts a collection of either task or future objects. You can specify a timeout and also when you want to return, whether that’s when they are all completed, the first one is completed, or on the first exception.

asyncio.gather takes any number of awaitable objects in the form of positional arguments. The nice part of gather is that it returns a list in the same order as the positional arguments that were passed in and can also return tasks that encountered exceptions.

asyncio.as\_completed is an iterable that takes in any awaitable object and lets you handle tasks as they finish instead of all at once. It also has a timeout argument.

asyncio.TaskGroup is the newest addition to Python 3.11; It lets you handle a group of tasks and will either return all or none of them depending on whether any of the tasks raise an exception.

### Завдання 1: Асинхронний веб-скрейпінг

\*\*Опис:\*\* Напишіть асинхронний скрипт для збору заголовків новин з кількох веб-сайтів новин. Використовуйте бібліотеку `aiohttp` для асинхронних HTTP-запитів і `BeautifulSoup` для парсингу HTML.

- \*\*Ціль:\*\* Зібрати заголовки новин з трьох різних новинних сайтів.

- \*\*Вимоги:\*\*

- Використання `asyncio` і `aiohttp` для асинхронних запитів.

- Асинхронне парсинг HTML з допомогою `BeautifulSoup`.

- Вивести результати на екран або зберегти у файл.

import asyncio

import aiohttp

from bs4 import BeautifulSoup

# URL-адреси новинних сайтів

NEWS\_SITES = [

'https://example-news-site1.com',

'https://example-news-site2.com',

'https://example-news-site3.com'

]

async def fetch\_page(session, url):

async with session.get(url) as response:

return await response.text()

async def parse\_titles(html, site):

soup = BeautifulSoup(html, 'html.parser')

# Залежно від сайту, змініть селектори

if site == 'https://example-news-site1.com':

return [title.get\_text() for title in soup.select('h2.headline')]

elif site == 'https://example-news-site2.com':

return [title.get\_text() for title in soup.select('div.news-title')]

elif site == 'https://example-news-site3.com':

return [title.get\_text() for title in soup.select('h3.article-title')]

return []

async def fetch\_and\_parse\_titles(session, site):

html = await fetch\_page(session, site)

titles = await parse\_titles(html, site)

return site, titles

async def main():

async with aiohttp.ClientSession() as session:

tasks = [fetch\_and\_parse\_titles(session, site) for site in NEWS\_SITES]

results = await asyncio.gather(\*tasks)

# Виведення результатів

for site, titles in results:

print(f"Заголовки з {site}:")

for title in titles:

print(f"- {title}")

print()

# Запуск асинхронної програми

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

asyncio.run(main())

### Завдання 2: Асинхронне читання файлів

\*\*Опис:\*\* Реалізуйте асинхронну функцію, яка читає кілька великих файлів паралельно і обробляє їх вміст. Наприклад, функція може рахувати кількість слів у кожному файлі.

- \*\*Ціль:\*\* Реалізувати асинхронне читання і обробку файлів.

- \*\*Вимоги:\*\*

- Використання `asyncio` і асинхронних файлових операцій.

- Обробка кількох файлів паралельно.

- Вивести результат обробки для кожного файлу.

### Завдання 3: Асинхронний генератор

\*\*Опис:\*\* Напишіть асинхронний генератор, який генерує числові послідовності з затримкою між числами. Генератор повинен працювати з певним інтервалом часу між генераціями.

- \*\*Ціль:\*\* Реалізувати асинхронний генератор з затримкою.

- \*\*Вимоги:\*\*

- Використання `async def` і `await` з `yield`.

- Генерація чисел з затримкою (наприклад, кожні 1 секунду).

- Продовження генерації до досягнення певної умови (наприклад, до досягнення певної кількості чисел).

### Завдання 4: Асинхронний контекстний менеджер

\*\*Опис:\*\* Реалізуйте асинхронний контекстний менеджер для роботи з асинхронними ресурсами, такими як асинхронні з'єднання з базою даних або API. Контекстний менеджер повинен автоматично відкривати і закривати ресурс.

- \*\*Ціль:\*\* Створити асинхронний контекстний менеджер.

- \*\*Вимоги:\*\*

- Реалізація методів `\_\_aenter\_\_` і `\_\_aexit\_\_`.

- Використання контекстного менеджера для роботи з асинхронними ресурсами.

- Приклад використання для з'єднання з API або базою даних.

### Завдання 5: Паралельне виконання завдань

\*\*Опис:\*\* Реалізуйте асинхронну функцію, яка виконує кілька завдань паралельно і збирає їх результати. Завдання можуть бути, наприклад, запити до API, обробка даних або інші асинхронні операції.

- \*\*Ціль:\*\* Реалізувати паралельне виконання завдань з використанням `asyncio.gather()`.

- \*\*Вимоги:\*\*

- Використання `asyncio.gather()` для паралельного виконання.

- Обробка результатів виконання всіх завдань.

- Вивести або зберегти результати.

Ці завдання охоплюють різні аспекти асинхронного програмування, від роботи з HTTP-запитами до створення асинхронних генераторів і контекстних менеджерів, що допоможе студентам глибше зрозуміти і застосувати асинхронне програмування в Python.

### Завдання 1: Асинхронне завантаження зображень

\*\*Опис:\*\* Напишіть асинхронний скрипт для завантаження зображень з кількох URL-адрес і збереження їх на диску.

- \*\*Ціль:\*\* Завантажити і зберегти зображення з інтернету.

- \*\*Вимоги:\*\*

- Використання `aiohttp` для асинхронних HTTP-запитів.

- Асинхронне збереження файлів за допомогою `aiofiles`.

- Обробка помилок (наприклад, для непрацюючих URL-адрес).

### Завдання 2: Асинхронне читання CSV-файлів

\*\*Опис:\*\* Реалізуйте асинхронний скрипт для читання даних з кількох великих CSV-файлів і обробки даних.

- \*\*Ціль:\*\* Зчитати і обробити дані з CSV-файлів.

- \*\*Вимоги:\*\*

- Використання `asyncio` і асинхронних операцій для читання файлів.

- Збір і обробка даних (наприклад, підрахунок кількості рядків).

- Використання бібліотеки `pandas` у асинхронному контексті (через `dask` або `modin`).

### Завдання 3: Асинхронне API з кешуванням

\*\*Опис:\*\* Створіть асинхронний клієнт для роботи з API, який реалізує кешування відповідей для зменшення кількості запитів до сервера.

- \*\*Ціль:\*\* Реалізувати кешування API-відповідей.

- \*\*Вимоги:\*\*

- Використання `aiohttp` для асинхронних запитів.

- Реалізація простого кешу у пам’яті.

- Обробка кешованих і не кешованих запитів.

### Завдання 4: Асинхронний таймер

\*\*Опис:\*\* Напишіть асинхронний скрипт, який використовує таймер для виконання певних завдань через задані інтервали часу.

- \*\*Ціль:\*\* Реалізувати таймери для асинхронних завдань.

- \*\*Вимоги:\*\*

- Використання `asyncio.sleep()` для затримок.

- Визначення інтервалів часу для виконання завдань.

- Виведення результатів на екран через певний інтервал.

### Завдання 5: Асинхронне оброблення черги завдань

\*\*Опис:\*\* Створіть асинхронний скрипт для обробки черги завдань, де завдання додаються в чергу і обробляються паралельно.

- \*\*Ціль:\*\* Реалізувати асинхронну обробку черги завдань.

- \*\*Вимоги:\*\*

- Використання `asyncio.Queue` для створення черги завдань.

- Запуск декількох асинхронних виконавців для обробки завдань.

- Виведення результатів обробки.

### Завдання 6: Асинхронний чат-клієнт

\*\*Опис:\*\* Розробіть простий асинхронний чат-клієнт, який може підключатися до чат-сервера, відправляти і отримувати повідомлення в реальному часі.

- \*\*Ціль:\*\* Реалізувати простий чат-клієнт з асинхронним обробленням повідомлень.

- \*\*Вимоги:\*\*

- Використання `asyncio` і `aiohttp` для підключення до сервера.

- Асинхронне отримання і відправлення повідомлень.

- Інтерфейс командного рядка для взаємодії з користувачем.

Ці завдання допоможуть поглибити розуміння асинхронного програмування і нададуть можливість практично застосувати отримані знання.

Ось детальний опис матеріалів для теми про модуль `concurrent.futures` в Python, зокрема про використання `ThreadPoolExecutor` і `ProcessPoolExecutor`, а також про розподілені обчислення і асинхронне виконання:

---

## Тема 3: Модуль `concurrent.futures`

### 1. Вступ до модуля `concurrent.futures`

\*\*Модуль `concurrent.futures`\*\* забезпечує інтерфейси для виконання паралельних завдань за допомогою пулів потоків і процесів. Основні класи, що використовуються для цього, — це `ThreadPoolExecutor` та `ProcessPoolExecutor`.

### 2. `ThreadPoolExecutor`

`ThreadPoolExecutor` дозволяє використовувати пул потоків для виконання завдань паралельно.

#### Основні аспекти:

- \*\*Створення пулу потоків:\*\*

```python

from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max\_workers=5) as executor:

# Ваш код тут

```

`max\_workers` визначає максимальну кількість потоків у пулі.

- \*\*Подання завдань:\*\*

```python

future = executor.submit(task\_function, \*args, \*\*kwargs)

result = future.result() # Отримання результату завдання

```

- \*\*Масове подання завдань:\*\*

```python

futures = [executor.submit(task\_function, arg) for arg in args\_list]

results = [future.result() for future in futures]

```

- \*\*Використання `map`:\*\*

```python

results = list(executor.map(task\_function, args\_list))

```

#### Приклад:

```python

from concurrent.futures import ThreadPoolExecutor

def square(n):

return n \* n

with ThreadPoolExecutor(max\_workers=4) as executor:

numbers = [1, 2, 3, 4, 5]

results = list(executor.map(square, numbers))

print(results)

```

### 3. `ProcessPoolExecutor`

`ProcessPoolExecutor` використовує пул процесів для паралельного виконання завдань. Це корисно для обчислень, які потребують багато ресурсів ЦП.

#### Основні аспекти:

- \*\*Створення пулу процесів:\*\*

```python

from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor(max\_workers=5) as executor:

# Ваш код тут

```

- \*\*Подання завдань та отримання результатів:\*\*

```python

future = executor.submit(task\_function, \*args, \*\*kwargs)

result = future.result() # Отримання результату завдання

```

- \*\*Масове подання завдань та `map`:\*\*

```python

results = list(executor.map(task\_function, args\_list))

```

#### Приклад:

```python

from concurrent.futures import ProcessPoolExecutor

def compute(x):

return x \* x

with ProcessPoolExecutor(max\_workers=4) as executor:

numbers = [1, 2, 3, 4, 5]

results = list(executor.map(compute, numbers))

print(results)

```

### 4. Розподілені обчислення

\*\*Розподілені обчислення\*\* стосуються використання кількох обчислювальних ресурсів (процесорів, машин) для вирішення задачі.

#### Основні аспекти:

- \*\*Масштабування завдань:\*\* Розподіл завдань між декількома процесами або машинами.

- \*\*Реалізація з `ProcessPoolExecutor`:\*\* Використовуйте `ProcessPoolExecutor` для розподілу завдань між кількома процесами на одній машині або між кількома машинами в кластері.

### 5. Асинхронне виконання

\*\*Асинхронне виконання\*\* дозволяє одночасно виконувати кілька завдань без блокування основного потоку.

#### Основні аспекти:

- \*\*Паралельне виконання:\*\* `ThreadPoolExecutor` та `ProcessPoolExecutor` виконують завдання паралельно, але для асинхронних операцій використовується `asyncio`.

- \*\*Змішане використання:\*\* Для асинхронних I/O-операцій використовують `asyncio`, а для CPU-bound операцій — `ProcessPoolExecutor`.

#### Приклад змішаного використання:

```python

import asyncio

from concurrent.futures import ProcessPoolExecutor

def cpu\_bound\_task(x):

return x \* x

async def main():

loop = asyncio.get\_running\_loop()

with ProcessPoolExecutor() as pool:

result = await loop.run\_in\_executor(pool, cpu\_bound\_task, 10)

print(f'Result: {result}')

asyncio.run(main())

```

### 6. Порівняння `ThreadPoolExecutor` і `ProcessPoolExecutor`

- \*\*`ThreadPoolExecutor`:\*\* Підходить для I/O-bound завдань, таких як мережеві запити або читання/запис файлів.

- \*\*`ProcessPoolExecutor`:\*\* Підходить для CPU-bound завдань, таких як важкі обчислення, де є необхідність у паралельній обробці на рівні процесів.

---

Ці матеріали допоможуть студентам зрозуміти і використовувати `concurrent.futures` для реалізації паралельних обчислень у Python.

Ось кілька завдань для класної роботи по темі модулю `concurrent.futures`, які допоможуть студентам краще зрозуміти його використання:

### Завдання 1: Обчислення факторіалів

\*\*Опис:\*\* Напишіть програму, яка обчислює факторіали для декількох чисел паралельно за допомогою `ThreadPoolExecutor`.

- \*\*Вимоги:\*\*

- Створіть функцію для обчислення факторіала.

- Використовуйте `ThreadPoolExecutor` для паралельного обчислення факторіалів для списку чисел.

- Виведіть результати для кожного числа.

```python

from concurrent.futures import ThreadPoolExecutor

from math import factorial

def compute\_factorial(n):

return factorial(n)

def main():

numbers = [5, 10, 15, 20]

with ThreadPoolExecutor(max\_workers=4) as executor:

results = list(executor.map(compute\_factorial, numbers))

for num, fact in zip(numbers, results):

print(f'Факторіал числа {num} = {fact}')

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

main()

```

### Завдання 2: Паралельне читання файлів

\*\*Опис:\*\* Напишіть програму, яка асинхронно читає вміст кількох великих файлів паралельно за допомогою `ProcessPoolExecutor` і підраховує кількість рядків у кожному файлі.

- \*\*Вимоги:\*\*

- Створіть функцію для підрахунку кількості рядків у файлі.

- Використовуйте `ProcessPoolExecutor` для паралельного виконання цієї функції для декількох файлів.

- Виведіть кількість рядків для кожного файлу.

```python

from concurrent.futures import ProcessPoolExecutor

def count\_lines(file\_path):

with open(file\_path, 'r') as file:

return len(file.readlines())

def main():

files = ['file1.txt', 'file2.txt', 'file3.txt']

with ProcessPoolExecutor(max\_workers=3) as executor:

results = list(executor.map(count\_lines, files))

for file, line\_count in zip(files, results):

print(f'Файл: {file}, Кількість рядків: {line\_count}')

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

main()

```

### Завдання 3: Масштабоване виконання завдань

\*\*Опис:\*\* Напишіть програму, яка використовує `ThreadPoolExecutor` для паралельного виконання численних завдань (наприклад, обробка зображень або виконання запитів до API).

- \*\*Вимоги:\*\*

- Створіть функцію для обробки завдання (наприклад, симуляція обробки зображення).

- Використовуйте `ThreadPoolExecutor` для паралельного виконання цих завдань.

- Виведіть інформацію про виконання кожного завдання.

```python

from concurrent.futures import ThreadPoolExecutor

import time

def process\_task(task\_id):

time.sleep(1) # Симуляція обробки

return f'Завдання {task\_id} завершено'

def main():

tasks = range(10) # 10 завдань

with ThreadPoolExecutor(max\_workers=4) as executor:

results = list(executor.map(process\_task, tasks))

for result in results:

print(result)

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

main()

```

### Завдання 4: Обробка завдань у фоновому режимі

\*\*Опис:\*\* Напишіть програму, яка використовує `ProcessPoolExecutor` для обробки довготривалих завдань у фоновому режимі. Завдання можуть бути будь-якими обчисленнями, які потребують значних ресурсів.

- \*\*Вимоги:\*\*

- Створіть функцію для обчислення важкого завдання (наприклад, числовий розрахунок).

- Запустіть цю функцію в `ProcessPoolExecutor` і відстежуйте виконання.

- Виведіть результати обробки завдань.

```python

from concurrent.futures import ProcessPoolExecutor

import math

def heavy\_computation(x):

return math.sqrt(x \*\* 2 + x \*\* 3)

def main():

numbers = [10000, 20000, 30000, 40000]

with ProcessPoolExecutor(max\_workers=4) as executor:

results = list(executor.map(heavy\_computation, numbers))

for num, result in zip(numbers, results):

print(f'Результат обчислення для {num} = {result}')

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

main()

```

### Завдання 5: Синхронізація з `concurrent.futures`

\*\*Опис:\*\* Реалізуйте програму, яка використовує `ThreadPoolExecutor` для паралельного виконання завдань, але також контролює і синхронізує їх виконання.

- \*\*Вимоги:\*\*

- Створіть функцію, яка виконуватиме завдання з затримкою.

- Використовуйте `ThreadPoolExecutor` разом з механізмами синхронізації (`Lock`, `Event`, тощо) для управління виконанням завдань.

- Виведіть інформацію про те, як завдання були синхронізовані.

```python

from concurrent.futures import ThreadPoolExecutor

import threading

import time

lock = threading.Lock()

def task\_with\_lock(task\_id):

with lock:

print(f'Завдання {task\_id} починає виконання')

time.sleep(2) # Симуляція тривалого виконання

print(f'Завдання {task\_id} завершено')

def main():

tasks = range(5) # 5 завдань

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

executor.map(task\_with\_lock, tasks)

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

main()

```

### Завдання 6: Порівняння `ThreadPoolExecutor` і `ProcessPoolExecutor`

\*\*Опис:\*\* Напишіть програму, яка порівнює швидкість виконання завдань за допомогою `ThreadPoolExecutor` і `ProcessPoolExecutor`.

- \*\*Вимоги:\*\*

- Створіть функцію для виконання обчислень.

- Запустіть функцію за допомогою обох типів пулів.

- Виміряйте і порівняйте час виконання.

```python

from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor

import time

import math

def computation(x):

return math.factorial(x)

def time\_execution(executor\_class, numbers):

start\_time = time.time()

with executor\_class(max\_workers=4) as executor:

list(executor.map(computation, numbers))

end\_time = time.time()

return end\_time - start\_time

def main():

numbers = range(20, 30)

thread\_time = time\_execution(ThreadPoolExecutor, numbers)

process\_time = time\_execution(ProcessPoolExecutor, numbers)

print(f'Час виконання з ThreadPoolExecutor: {thread\_time:.2f} секунд')

print(f'Час виконання з ProcessPoolExecutor: {process\_time:.2f} секунд')

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

main()

```

Ці завдання охоплюють різні аспекти використання `concurrent.futures`, включаючи як `ThreadPoolExecutor`, так і `ProcessPoolExecutor`, що допоможе студентам краще зрозуміти і застосувати ці інструменти у своїх проектах.

Ось шість завдань для домашньої роботи, що використовують клієнт-серверну взаємодію, з акцентом на практичне застосування `concurrent.futures`:

### Завдання 1: Асинхронний сервер для обробки запитів

\*\*Опис:\*\* Реалізуйте асинхронний сервер, який приймає запити від клієнтів і відповідає на них певним повідомленням. Використовуйте `asyncio` і `aiohttp` для реалізації сервера та обробки запитів.

- \*\*Вимоги:\*\*

- Реалізуйте сервер, який приймає HTTP-запити і відповідає простим текстовим повідомленням.

- Використовуйте `asyncio` і `aiohttp` для реалізації сервера.

- Створіть клієнтську програму, яка надсилає запити до сервера і виводить отримані відповіді.

\*\*Примітка:\*\* Можна реалізувати сервер, який обробляє запити з параметрами і повертає результат у відповідь.

### Завдання 2: Паралельне виконання запитів до API

\*\*Опис:\*\* Напишіть програму, яка використовує `ThreadPoolExecutor` для паралельного виконання запитів до різних API і збирає дані з відповідей.

- \*\*Вимоги:\*\*

- Реалізуйте функцію для надсилання запиту до API і обробки відповіді.

- Використовуйте `ThreadPoolExecutor` для надсилання запитів до декількох API паралельно.

- Виведіть зібрані дані на екран або збережіть у файл.

\*\*Примітка:\*\* Можна використовувати бібліотеку `requests` для HTTP-запитів до API.

### Завдання 3: Сервер для обробки завдань

\*\*Опис:\*\* Реалізуйте сервер, який отримує завдання від клієнтів, обробляє їх і відправляє результати назад клієнтам. Використовуйте `concurrent.futures` для обробки завдань.

- \*\*Вимоги:\*\*

- Реалізуйте сервер, який отримує завдання у вигляді HTTP-запитів, обробляє їх і надсилає результати назад.

- Використовуйте `ProcessPoolExecutor` для обробки завдань на сервері.

- Створіть клієнтську програму, яка надсилає завдання серверу і отримує результати.

\*\*Примітка:\*\* Завдання можуть бути простими обчислювальними або текстовими обробками.

### Завдання 4: Розподілений лічильник

\*\*Опис:\*\* Реалізуйте розподілену систему лічильника, де кілька клієнтів можуть підвищувати лічильник на сервері. Використовуйте `asyncio` для асинхронних запитів.

- \*\*Вимоги:\*\*

- Реалізуйте сервер, який має змогу приймати запити на збільшення лічильника і повертати його значення.

- Реалізуйте клієнтську програму, яка надсилає запити на збільшення лічильника паралельно з декількох клієнтів.

- Виведіть кінцеве значення лічильника після завершення всіх запитів.

\*\*Примітка:\*\* Використовуйте асинхронний підхід для обробки запитів та оновлення лічильника.

### Завдання 5: Асинхронний чат-сервер

\*\*Опис:\*\* Реалізуйте асинхронний чат-сервер, який дозволяє кільком клієнтам підключатися і обмінюватися повідомленнями в режимі реального часу.

- \*\*Вимоги:\*\*

- Реалізуйте сервер, який обробляє повідомлення від клієнтів і транслює їх усім підключеним клієнтам.

- Використовуйте `asyncio` і `aiohttp` для реалізації чат-сервера.

- Реалізуйте клієнтську програму для підключення до сервера та надсилання/отримання повідомлень.

\*\*Примітка:\*\* Можна додати можливість підключення декількох клієнтів одночасно і відображення повідомлень у реальному часі.

### Завдання 6: Веб-скрапінг з паралельними запитами

\*\*Опис:\*\* Напишіть програму для веб-скрапінгу, яка паралельно збирає інформацію з кількох веб-сайтів і агрегує результати.

- \*\*Вимоги:\*\*

- Реалізуйте функцію для скрапінгу інформації з одного веб-сайту.

- Використовуйте `ThreadPoolExecutor` для паралельного збору даних з кількох веб-сайтів.

- Обробіть і виведіть зібрані дані.

\*\*Примітка:\*\* Використовуйте бібліотеки для роботи з HTTP-запитами (наприклад, `requests`) і парсингом HTML (наприклад, `BeautifulSoup`).

Ці завдання допоможуть студентам практично застосувати знання з клієнт-серверної взаємодії, паралельного і асинхронного виконання завдань.

**План Лекції: Розробка Розширень і Модулів на C/C++**

**Вступ (10 хв)**

* **Огляд Теми:**
  + Що таке розширення і модулі для Python.
  + Переваги використання C/C++ для розширення Python.
  + Основні бібліотеки: ctypes, cffi, і створення розширень на C/C++.

**Розділ 1: Використання ctypes (20 хв)**

* **Що таке ctypes:**
  + Опис бібліотеки ctypes для взаємодії Python з C-бібліотеками.
  + Як ctypes дозволяє викликати функції C і маніпулювати даними.
* **Основи використання ctypes:**
  + Імпорт C-бібліотек у Python.
  + Виклик функцій C з Python.
  + Оголошення типів даних і параметрів.
* **Приклад:**
  + Написання простого C-коду, компіляція в DLL (Windows) або shared library (Linux).
  + Використання ctypes для завантаження і виклику функцій з C-бібліотеки.

c

Copy code

// simple.c

#include <stdio.h>

void hello\_world() {

printf("Hello, World!\n");

}

python

Copy code

# simple.py

import ctypes

# Завантаження бібліотеки

lib = ctypes.CDLL('./simple.so') # На Windows використовувати 'simple.dll'

# Виклик функції

lib.hello\_world()

**Розділ 2: Використання cffi (20 хв)**

* **Що таке cffi:**
  + Опис бібліотеки cffi для роботи з C-кодом з Python.
  + Переваги cffi в порівнянні з ctypes.
* **Основи використання cffi:**
  + Інтерфейс cffi для створення C-розширень.
  + Оголошення C-функцій і типів даних.
* **Приклад:**
  + Написання простого C-коду.
  + Використання cffi для взаємодії з C-кодом.

c

Copy code

// example.c

int add(int a, int b) {

return a + b;

}

python

Copy code

# example.py

from cffi import FFI

ffi = FFI()

ffi.cdef("int add(int, int);")

# Завантаження бібліотеки

lib = ffi.dlopen('./example.so')

# Виклик функції

result = lib.add(3, 4)

print(result)

**Розділ 3: Розробка Розширень для Python на C/C++ (30 хв)**

* **Що таке розширення на C/C++:**
  + Як розширення на C/C++ інтегруються з Python.
  + Опис механізму Python C API.
* **Створення простого розширення на C:**
  + Написання C-коду для створення Python-модуля.
  + Опис макросів і функцій для інтеграції з Python.
* **Приклад:**
  + Реалізація простого модуля на C, компіляція, і використання в Python.

c

Copy code

// mymodule.c

#include <Python.h>

static PyObject\* my\_add(PyObject\* self, PyObject\* args) {

int a, b;

if (!PyArg\_ParseTuple(args, "ii", &a, &b)) {

return NULL;

}

return PyLong\_FromLong(a + b);

}

static PyMethodDef MyMethods[] = {

{"add", my\_add, METH\_VARARGS, "Add two numbers"},

{NULL, NULL, 0, NULL}

};

static struct PyModuleDef mymodule = {

PyModuleDef\_HEAD\_INIT,

"mymodule",

NULL,

-1,

MyMethods

};

PyMODINIT\_FUNC PyInit\_mymodule(void) {

return PyModule\_Create(&mymodule);

}

python

Copy code

# setup.py

from setuptools import setup, Extension

setup(

name="mymodule",

version="0.1",

ext\_modules=[Extension("mymodule", ["mymodule.c"])],

)

bash

Copy code

# Компіляція і установка

python setup.py build\_ext --inplace

python

Copy code

# Використання модуля

import mymodule

print(mymodule.add(3, 4))

**Розділ 4: Дебаг і Оптимізація (10 хв)**

* **Дебаг розширень:**
  + Інструменти для налагодження C-коду (GDB, LLDB).
  + Техніки для відладки розширень.
* **Оптимізація:**
  + Техніки оптимізації C-коду для покращення продуктивності.
  + Використання профілювальників для вимірювання продуктивності.

**Заключення (10 хв)**

* **Підведення підсумків:**
  + Основні моменти використання ctypes, cffi, і створення розширень на C/C++.
  + Обговорення потенційних застосувань і переваг розширень.
* **Питання та відповіді:**
  + Відповіді на питання студентів.
  + Обговорення практичних кейсів і додаткових ресурсів для самостійного навчання.

Цей план лекції охоплює основні аспекти розробки розширень і модулів на C/C++ для Python, включаючи приклади коду і практичні завдання

Ось кілька завдань для класної роботи по темі розробки розширень і модулів на C/C++, що використовують `ctypes`, `cffi`, і розширення для Python:

### Завдання 1: Використання `ctypes` для взаємодії з C-функціями

\*\*Опис:\*\* Напишіть просту C-бібліотеку, яка містить функцію для обчислення факторіала числа. Потім використовуйте `ctypes`, щоб викликати цю функцію з Python і перевірити її коректність.

- \*\*Вимоги:\*\*

- Напишіть C-код, що реалізує функцію для обчислення факторіала.

- Скомпілюйте C-код у бібліотеку (DLL або shared library).

- Напишіть Python-скрипт, що використовує `ctypes` для виклику функції і виводить результат.

\*\*Примітка:\*\* Можна додати перевірку на невірні параметри (негативні числа).

### Завдання 2: Використання `cffi` для роботи з C-структурами

\*\*Опис:\*\* Реалізуйте C-функцію, яка обробляє просту структуру з двома полями (наприклад, `x` і `y` типу `int`). Напишіть Python-скрипт з використанням `cffi`, щоб створити структуру, передати її в C-функцію і отримати результат.

- \*\*Вимоги:\*\*

- Напишіть C-код, що включає структуру і функцію, яка працює з цією структурою.

- Напишіть Python-скрипт, що використовує `cffi` для взаємодії з C-кодом.

\*\*Примітка:\*\* Можна реалізувати функцію, яка обчислює суму полів структури.

### Завдання 3: Розширення на C для Python (математичні функції)

\*\*Опис:\*\* Напишіть C-модуль для Python, що реалізує функцію для обчислення квадратного кореня і функцію для обчислення кубічного кореня. Напишіть Python-скрипт, що використовує цей модуль і перевіряє правильність роботи функцій.

- \*\*Вимоги:\*\*

- Реалізуйте модуль на C з функціями для обчислення квадратного і кубічного кореня.

- Напишіть `setup.py` для компіляції модуля.

- Напишіть Python-скрипт для тестування функцій.

\*\*Примітка:\*\* Перевірте функції з різними вхідними значеннями.

### Завдання 4: Взаємодія між Python і C через `ctypes`

\*\*Опис:\*\* Реалізуйте C-функцію, яка приймає масив чисел і повертає їхню суму. Напишіть Python-скрипт, що використовує `ctypes` для передачі масиву з Python у C-функцію і виводу результату.

- \*\*Вимоги:\*\*

- Реалізуйте C-функцію для обробки масиву чисел.

- Скомпілюйте C-код у бібліотеку.

- Напишіть Python-скрипт для передачі масиву чисел у C-функцію.

\*\*Примітка:\*\* Можна реалізувати перевірку на правильність обробки різних типів даних.

### Завдання 5: Динамічне завантаження бібліотек

\*\*Опис:\*\* Напишіть C-бібліотеку, яка реалізує кілька функцій (наприклад, арифметичні операції). Напишіть Python-скрипт, який динамічно завантажує цю бібліотеку і викликає різні функції з неї.

- \*\*Вимоги:\*\*

- Реалізуйте C-код з кількома функціями.

- Скомпілюйте C-код у бібліотеку.

- Напишіть Python-скрипт, який динамічно завантажує бібліотеку і викликає її функції.

\*\*Примітка:\*\* Можна додати перевірку на успішне завантаження бібліотеки та обробку можливих помилок.

### Завдання 6: Розширення на C з використанням Python C API

\*\*Опис:\*\* Реалізуйте C-розширення для Python, яке реалізує клас, що представляє собою просту математичну точку з методами для обчислення відстані до іншої точки. Напишіть Python-скрипт, що використовує цей клас.

- \*\*Вимоги:\*\*

- Напишіть C-код, що реалізує Python-клас з методами для обчислення відстані.

- Напишіть `setup.py` для компіляції розширення.

- Напишіть Python-скрипт для тестування класу і його методів.

\*\*Примітка:\*\* Можна реалізувати перевірку на правильність обчислень з різними координатами точок.

Ці завдання допоможуть студентам краще зрозуміти, як інтегрувати C/C++ з Python, використовуючи різні бібліотеки і підходи

Ось 6 завдань для домашньої роботи по темі розробки розширень і модулів на C/C++, що включають використання `ctypes`, `cffi` та створення розширень для Python:

### Завдання 1: Параметризація C-функції через `ctypes`

\*\*Опис:\*\* Напишіть C-функцію, яка приймає параметри різних типів (int, double, char). Функція повинна повернути рядок, що складається з результатів обчислень. Використовуйте `ctypes` для виклику цієї функції з Python.

- \*\*Вимоги:\*\*

- Реалізуйте C-функцію, яка обробляє параметри різних типів і повертає рядок.

- Скомпілюйте C-код у бібліотеку.

- Напишіть Python-скрипт, що використовує `ctypes` для виклику цієї функції і виводу результату.

### Завдання 2: Робота з масивами через `cffi`

\*\*Опис:\*\* Реалізуйте C-функцію, яка приймає масив цілих чисел і обчислює їх середнє арифметичне. Використовуйте `cffi` для взаємодії з цією функцією з Python.

- \*\*Вимоги:\*\*

- Напишіть C-код, що реалізує функцію для обчислення середнього арифметичного масиву.

- Скомпілюйте C-код у бібліотеку.

- Напишіть Python-скрипт, що використовує `cffi` для виклику функції і виводу результату.

### Завдання 3: Розширення для Python на C для роботи з матрицями

\*\*Опис:\*\* Розробіть C-розширення для Python, яке реалізує функції для обробки матриць: додавання, віднімання і множення. Напишіть Python-скрипт для тестування цих функцій.

- \*\*Вимоги:\*\*

- Реалізуйте C-код для функцій обробки матриць.

- Напишіть `setup.py` для компіляції розширення.

- Напишіть Python-скрипт для тестування функцій з різними матрицями.

### Завдання 4: Створення C-розширення з Python C API для обробки строк

\*\*Опис:\*\* Реалізуйте C-розширення для Python, яке включає функцію для підрахунку кількості символів у рядку. Напишіть Python-скрипт для тестування цієї функції.

- \*\*Вимоги:\*\*

- Напишіть C-код, що реалізує функцію для підрахунку кількості символів у рядку.

- Напишіть `setup.py` для компіляції розширення.

- Напишіть Python-скрипт для тестування функції.

### Завдання 5: Динамічне завантаження функцій з C-бібліотеки

\*\*Опис:\*\* Реалізуйте C-бібліотеку з кількома функціями для математичних обчислень (наприклад, обчислення синуса, косинуса). Напишіть Python-скрипт, який динамічно завантажує бібліотеку і викликає ці функції.

- \*\*Вимоги:\*\*

- Реалізуйте C-код з кількома математичними функціями.

- Скомпілюйте C-код у бібліотеку.

- Напишіть Python-скрипт для динамічного завантаження і виклику функцій.

### Завдання 6: Модуль на C/C++ для обробки структур

\*\*Опис:\*\* Напишіть C-модуль для Python, який реалізує функцію для створення та обробки складної структури даних (наприклад, структура з полями `name` і `age`). Напишіть Python-скрипт для створення таких структур і виклику функцій, які працюють з ними.

- \*\*Вимоги:\*\*

- Напишіть C-код для роботи зі структурою.

- Напишіть `setup.py` для компіляції розширення.

- Напишіть Python-скрипт для створення структур і тестування функцій.

Ці завдання допоможуть студентам набути практичних навичок у створенні розширень для Python з використанням C/C++, включаючи роботу з різними бібліотеками та підходами.

### План Лекції: Оптимізація Продуктивності

#### Вступ (10 хв)

- \*\*Огляд Теми:\*\*

- Важливість оптимізації продуктивності коду.

- Основні поняття: профілювання, оптимізація.

- Цілі лекції: ознайомлення з інструментами для профілювання і оптимізації коду.

#### Розділ 1: Вступ до Профілювання Коду (15 хв)

- \*\*Що таке профілювання:\*\*

- Означення профілювання.

- Чому профілювання важливе для оптимізації коду.

- \*\*Основні терміни:\*\*

- Вимірювання часу виконання.

- Визначення вузьких місць у коді.

- Профіль викликів.

#### Розділ 2: Інструменти для Профілювання Коду (20 хв)

- \*\*`cProfile`:\*\*

- Опис інструменту `cProfile`.

- Як використовувати `cProfile` для збору статистики про виконання коду.

- Інтерпретація результатів профілювання.

\*\*Приклад використання `cProfile`:\*\*

```python

import cProfile

import pstats

def my\_function():

# Деякий код для профілювання

pass

profiler = cProfile.Profile()

profiler.enable()

my\_function()

profiler.disable()

stats = pstats.Stats(profiler).sort\_stats(pstats.SortKey.TIME)

stats.print\_stats()

```

- \*\*`timeit`:\*\*

- Опис інструменту `timeit`.

- Як використовувати `timeit` для вимірювання часу виконання окремих частин коду.

- Як отримати точні результати.

\*\*Приклад використання `timeit`:\*\*

```python

import timeit

def test\_function():

# Деякий код для вимірювання часу

pass

duration = timeit.timeit(test\_function, number=1000)

print(f"Час виконання: {duration} секунд")

```

- \*\*Інші Інструменти:\*\*

- Огляд альтернативних інструментів для профілювання (наприклад, `line\_profiler`, `memory\_profiler`).

- Порівняння та випадки використання.

#### Розділ 3: Основи Оптимізації Коду (20 хв)

- \*\*Принципи Оптимізації:\*\*

- Важливість зменшення складності алгоритмів.

- Оптимізація пам’яті і часу виконання.

- \*\*Загальні Техніки Оптимізації:\*\*

- Зменшення кількості ітерацій.

- Оптимізація доступу до даних.

- Використання більш ефективних алгоритмів і структур даних.

- \*\*Кейси Оптимізації:\*\*

- Приклади практичних завдань, що потребують оптимізації (наприклад, обробка великих даних, високонавантажені операції).

#### Розділ 4: Практична Робота (30 хв)

- \*\*Аналіз Коду:\*\*

- Розглядання прикладів коду, де необхідна оптимізація.

- Використання `cProfile` і `timeit` для аналізу і профілювання коду.

\*\*Приклад завдання:\*\*

- Знайти і проаналізувати вузькі місця у наступному коді:

```python

def slow\_function():

total = 0

for i in range(10000):

for j in range(10000):

total += i \* j

return total

```

- \*\*Оптимізація і Рефакторинг:\*\*

- Рефакторинг коду на основі отриманих даних профілювання.

- Перевірка ефективності змін за допомогою `timeit`.

#### Заключення (10 хв)

- \*\*Підведення Підсумків:\*\*

- Основні висновки з лекції.

- Значення профілювання для досягнення оптимальної продуктивності.

- \*\*Питання та Відповіді:\*\*

- Обговорення запитань і проблемних моментів студентів.

- Рекомендації для подальшого навчання і самостійного вдосконалення.

### Додаткові Ресурси

- Рекомендації по літературі та онлайн-ресурсам для поглибленого вивчення теми.

Цей план лекції охоплює ключові аспекти профілювання та оптимізації продуктивності, включаючи інструменти та техніки, що дозволяють ефективно аналізувати і покращувати продуктивність коду.

Ось кілька завдань для розгляду в аудиторії по темі оптимізації продуктивності коду:

### Завдання 1: Аналіз і Оптимізація Функції Обчислення Фібоначчі

\*\*Опис:\*\* Розгляньте наступну реалізацію функції для обчислення чисел Фібоначчі:

```python

def fibonacci(n):

if n <= 1:

return n

return fibonacci(n-1) + fibonacci(n-2)

```

- \*\*Завдання:\*\*

1. Використовуйте `cProfile` для профілювання цієї функції з різними значеннями `n`.

2. Проаналізуйте результати профілювання і вкажіть на вузькі місця.

3. Оптимізуйте функцію, використовуючи динамічне програмування або інший метод.

4. Виміряйте продуктивність оптимізованої функції з `timeit` і порівняйте з початковою версією.

### Завдання 2: Оптимізація Обробки Даних

\*\*Опис:\*\* Розгляньте наступний код, який обробляє великий список чисел, підраховуючи кількість парних чисел:

```python

def count\_even\_numbers(numbers):

count = 0

for num in numbers:

if num % 2 == 0:

count += 1

return count

```

- \*\*Завдання:\*\*

1. Профілюйте цю функцію, використовуючи `cProfile` і великі списки чисел.

2. Проаналізуйте результати і визначте, чи є можливості для оптимізації.

3. Оптимізуйте код, використовуючи векторизацію з NumPy або інший метод.

4. Перевірте продуктивність до і після оптимізації за допомогою `timeit`.

### Завдання 3: Профілювання і Оптимізація Розділу Коду

\*\*Опис:\*\* Дано наступний код, який виконує кілька вкладених циклів:

```python

def nested\_loops(n):

total = 0

for i in range(n):

for j in range(n):

total += i \* j

return total

```

- \*\*Завдання:\*\*

1. Використовуйте `cProfile` для профілювання цієї функції з різними значеннями `n`.

2. Проаналізуйте результати профілювання і виявите, де можна зменшити витрати на обчислення.

3. Оптимізуйте код, наприклад, за допомогою алгебраїчних спрощень.

4. Оцініть ефективність змін з `timeit` і порівняйте результати.

### Завдання 4: Оптимізація Функції Читання Файлів

\*\*Опис:\*\* Розгляньте наступний код для читання великого текстового файлу та підрахунку кількості рядків:

```python

def count\_lines(file\_path):

with open(file\_path, 'r') as file:

lines = file.readlines()

return len(lines)

```

- \*\*Завдання:\*\*

1. Профілюйте цю функцію з різними файлами різного розміру.

2. Проаналізуйте результати профілювання і визначте можливості для покращення.

3. Оптимізуйте код, використовуючи методи для поступового читання файлу (наприклад, читання рядків по черзі).

4. Перевірте ефективність оптимізованої версії за допомогою `timeit` і порівняйте з початковою версією.

Ці завдання допоможуть студентам краще зрозуміти процес профілювання та оптимізації коду, а також практично застосувати різні техніки для покращення продуктивності програм.

Профілювання багатопоточного та асинхронного коду має свої особливості, оскільки ці типи коду часто є більш складними в плані вимірювання продуктивності. Ось як можна профілювати багатопоточний та асинхронний код в Python:

### Профілювання Багатопоточного Коду

Багатопоточний код може бути складним для профілювання через одночасне виконання декількох потоків. Ось кілька підходів для профілювання багатопоточного коду:

1. \*\*Використання `cProfile` для Профілювання:\*\*

`cProfile` може бути використаний для профілювання багатопоточного коду, але важливо розуміти, що він профілює тільки головний потік за замовчуванням. Для профілювання всіх потоків потрібно використовувати додаткові бібліотеки або спеціальні підходи.

```python

import cProfile

import threading

def worker():

# Код для виконання в потоці

pass

def main():

threads = [threading.Thread(target=worker) for \_ in range(10)]

for t in threads:

t.start()

for t in threads:

t.join()

cProfile.run('main()')

```

2. \*\*Використання `py-spy`:\*\*

`py-spy` є зовнішнім профайлером, який може профілювати багатопотокові програми без змінювання коду.

```bash

py-spy top --pid <pid>

```

3. \*\*Використання `line\_profiler`:\*\*

`line\_profiler` дозволяє профілювати конкретні функції, навіть у багатопоточному середовищі.

```python

from line\_profiler import LineProfiler

profiler = LineProfiler()

profiler.add\_function(worker)

profiler.enable()

main()

profiler.disable()

profiler.print\_stats()

```

### Профілювання Асинхронного Коду

Асинхронний код виконується з використанням `asyncio` та інших асинхронних бібліотек. Для профілювання асинхронного коду використовуються спеціалізовані інструменти та методи:

1. \*\*Використання `cProfile` з Асинхронним Кодом:\*\*

`cProfile` можна використовувати для профілювання асинхронних функцій, але важливо правильно інтегрувати профайлер з асинхронним контекстом.

```python

import asyncio

import cProfile

import pstats

async def async\_worker():

# Асинхронний код

await asyncio.sleep(1)

async def main():

await asyncio.gather(\*[async\_worker() for \_ in range(10)])

profiler = cProfile.Profile()

profiler.enable()

asyncio.run(main())

profiler.disable()

stats = pstats.Stats(profiler).sort\_stats(pstats.SortKey.TIME)

stats.print\_stats()

```

2. \*\*Використання `py-spy` для Асинхронного Коду:\*\*

Як і для багатопоточного коду, `py-spy` може бути використаний для профілювання асинхронних програм.

```bash

py-spy top --pid <pid>

```

3. \*\*Використання `asyncio`-специфічних Інструментів:\*\*

- \*\*`aiomonitor`:\*\* Додатковий інструмент для моніторингу та профілювання асинхронного коду на основі `asyncio`.

```python

import asyncio

from aiomonitor import Monitor

async def async\_worker():

await asyncio.sleep(1)

async def main():

async with Monitor():

await asyncio.gather(\*[async\_worker() for \_ in range(10)])

asyncio.run(main())

```

4. \*\*Використання `tracemalloc`:\*\*

`tracemalloc` може бути корисний для вимірювання використання пам'яті у асинхронних програмах.

```python

import tracemalloc

import asyncio

async def async\_worker():

await asyncio.sleep(1)

async def main():

tracemalloc.start()

await asyncio.gather(\*[async\_worker() for \_ in range(10)])

snapshot = tracemalloc.take\_snapshot()

top\_stats = snapshot.statistics('lineno')

for stat in top\_stats[:10]:

print(stat)

asyncio.run(main())

```

### Загальні Рекомендації

- \*\*Збирати дані з кількох інструментів:\*\* Використання кількох інструментів профілювання дозволяє отримати комплексну картину продуктивності.

- \*\*Зосереджуватися на вузьких місцях:\*\* Аналізуйте і оптимізуйте найбільш ресурсоємні частини коду.

- \*\*Перевіряти ефективність змін:\*\* Після оптимізації проводьте повторне профілювання для підтвердження покращень.

Ці методи дозволяють ефективно профілювати як багатопотокові, так і асинхронні програми, допомагаючи знайти і усунути проблеми з продуктивністю.