Course ASYNCIO

Table of Content

[1 Reposytory 4](#_Toc166533900)

[1.1 Основы асинхронного програ 4](#_Toc166533901)

[1.2 Введение в асинхронность 4](#_Toc166533902)

[1.3 Основные преимущества asyncio в Python 6](#_Toc166533903)

[1.4 IO bound и CPU bound 6](#_Toc166533904)

[2 Awaitable объекты 6](#_Toc166533905)

[2.1 Введение в Awaitable объекты 7](#_Toc166533906)

[2.2 Event Loop 10](#_Toc166533907)

[2.3 Ключевое слово await 12](#_Toc166533908)

[2.4 Coroutines / Корутины 17](#_Toc166533909)

[2.5 TASK 21](#_Toc166533910)

[2.6 Future 22](#_Toc166533911)

[3 Good To Know 27](#_Toc166533912)

[3.1 Long polling 27](#_Toc166533913)

[3.2 Polling 29](#_Toc166533914)

[3.3 Mutex 31](#_Toc166533915)

[3.4 Атомарность операций 35](#_Toc166533916)

[3.5 DEADLOCK блокировка 38](#_Toc166533917)

[3.6 Race Condition 45](#_Toc166533918)

[3.7 Race condition example – Bank account 45](#_Toc166533919)

[4 Основы работы с Asyncio 46](#_Toc166533920)

[4.1 Основные методы asyncio 46](#_Toc166533921)

[4.2 asyncio.run() 49](#_Toc166533922)

[4.3 asyncio.sleep() 51](#_Toc166533923)

[4.4 asyncio.create\_task() 53](#_Toc166533924)

[4.5 Предотвращаем "зависание" корутин asyncio.wait\_for() 56](#_Toc166533925)

[4.6 asyncio.gather() 61](#_Toc166533926)

[4.7 Ожидание группы корутин asyncio.wait() 64](#_Toc166533927)

[4.8 Запуск и работа с футурами asyncio.ensure\_future() 66](#_Toc166533928)

[4.9 Получаем ссылку на текущий цикл событий asyncio.get\_event\_loop() 69](#_Toc166533929)

[4.10 новый цикл событий asyncio.new\_event\_loop() 70](#_Toc166533930)

[4.11 Устанавливаем цикл событий по умолчанию asyncio.set\_event\_loop(loop) 72](#_Toc166533931)

[4.12 доступ к текущему циклу событий get\_running\_loop() 74](#_Toc166533932)

[4.13 to\_thread() Запуск блокирующих функций в отдельном потоке 76](#_Toc166533933)

[5 Работа с задачами TASK 80](#_Toc166533934)

[5.1 Введение 80](#_Toc166533935)

[5.2 Основные методы управления задачами TASK 82](#_Toc166533936)

[5.3 Проверка статуса задачи task.done() 84](#_Toc166533937)

[5.4 Получение предварительного результата задачи task.result 85](#_Toc166533938)

[5.5 Работа с текущей задачей asyncio.current\_task() 86](#_Toc166533939)

[5.6 Получение всех задач в цикле событий asyncio.all\_tasks() 87](#_Toc166533940)

[5.7 Установка и получение имени задачи get\_name() и set\_name() 89](#_Toc166533941)

[5.8 Работа с исключениями exception() 90](#_Toc166533942)

[5.9 Отмена задач cancel(), cancelled(), cancelling() и uncancel() 91](#_Toc166533943)

[5.10 состоянии цикла событий и корутин get\_coro() и get\_loop() 96](#_Toc166533944)

[5.11 Работа с асинхронным стеком вызова get\_stack() и print\_stack() 100](#_Toc166533945)

[5.12 Функции callback add\_done\_callback(fn) и remove\_done\_callback(fn) 101](#_Toc166533946)

[5.13 Защита корутины от отмены asyncio.shield() 106](#_Toc166533947)

[5.14 TaskGroup() !!! Python 3.11 110](#_Toc166533948)

[5.15 Task\_group Exceptions CancelledError 112](#_Toc166533949)

[5.16 Taskgrpup exception 113](#_Toc166533950)

[6 Асинхронные конструкции 116](#_Toc166533951)

[6.1 Основы async with 116](#_Toc166533952)

[6.2 async for 119](#_Toc166533953)

[7 Работа с очередями в Asyncio 128](#_Toc166533954)

[7.1 8.1 Введение в очереди 128](#_Toc166533955)

[7.2 Очередь FIFO Простая очередь asyncio.Queue() 130](#_Toc166533956)

[7.3 Очередь LIFO asyncio.LifoQueue 136](#_Toc166533957)

[7.4 Приоритетная очередь asyncio.PriorityQueue() 144](#_Toc166533958)

[8 Базовые примитивы синхронизации asyncio 147](#_Toc166533959)

[8.1 Введение в примитивы 148](#_Toc166533960)

[8.2 Защита ресурса от одновременного доступа asyncio.Lock() 155](#_Toc166533961)

[8.3 Ожидание событий для синхронизации доступа asyncio.Event() 164](#_Toc166533962)

[8.4 Синхронизация задач asyncio.Condition() 170](#_Toc166533963)

[8.5 Принцип простого семафора asyncio.Semaphore() 176](#_Toc166533964)

[8.6 Принцип ограниченного семафора asyncio.BoundedSemaphore() 185](#_Toc166533965)

[8.7 Реализация барьера для защиты одновременного доступа Barrier 188](#_Toc166533966)

[9 Асинхронное чтение файлов aiofiles 195](#_Toc166533967)

[9.1 Introduction 195](#_Toc166533968)

[9.2 Aiofiles начало 196](#_Toc166533969)

[9.3 Основной метод aiofiles.open() 197](#_Toc166533970)

[9.4 Асинхронный файловый объект и доступные методы 210](#_Toc166533971)

|  |
| --- |
| Reposytory <https://github.com/ecrvmal/asyncio_course>  asyncio cheat sheet:  <https://attachments.convertkitcdnn2.com/830932/acbbda84-76f4-48a4-8e2f-400b09da4e1b/cheat-sheet-for-python-asyncio.pdf> |
| Основы асинхронного програ |
| Введение в асинхронностьCode main3: import time import asyncio  start = time.time() # Время начала эксперимента!)  async def sleeping(n):  print(f'start operation# {n}: {time.time() - start:4f}')  await asyncio.sleep(1)  print(f'Long operation# {n} completed')  return f'result operation# : {n}'  async def main():  # run 30 operations  # all\_results = [sleeping(i) for i in range(1, 31)]  task = [sleeping(i) for i in range(1, 31)]  all\_results = await asyncio.gather(\*task)  print(f'completed {len(all\_results)} operations')  print(f'Programm completed in {time.time() - start:.4f} seconds')  # main() asyncio.run(main()) |
| IO bound и CPU bound операции. **CPU bound операции** - операции, ограниченные процессором (также можно встретить термин "счетные операции").  Математические вычисления   * Обработка изображений: * Сортировка больших массивов данных: * Криптографические операции: Шифрование и дешифрование данных. * Симуляции и моделирование   **IO(input / output) bound операции** - операции, ограниченные вводом/выводом.   * Чтение и запись на диск: * Запросы в базы данных: * Сетевые запросы: * Ожидание ввода от пользователя:     asyncio, могут помочь оптимизировать и параллелить IO bound операции. |
| **Программа:**   1. Это набор инструкций, которые описывают, как выполнять определенную задачу. 2. Программа хранится на диске в виде исполняемого файла или исходного кода. 3. Программа не выполняется, пока её не запустят или не интерпретируют.   **Процесс:**   1. Это выполняемый экземпляр программы. 2. Когда программа запускается, операционная система создает новый процесс для этой программы. 3. Процесс имеет свои собственные ресурсы и изолированное окружение выполнения. |
| **Особенности многопроцессности в Python**:  * **Изолированность памяти**: Поскольку каждый из процессов имеет свою собственную область памяти, они не разделяют состояние между собой. Это уменьшает риск состояния [гонки](https://stepik.org/lesson/933669/step/1?unit=939568) и делает синхронизацию проще, чем в многопоточных системах. * **Стоимость создания**: Создание нового процесса обычно требует больше ресурсов, чем создание нового потока. * **Коммуникация между процессами**: Поскольку процессы не разделяют память, им требуется специальная коммуникация (например, через каналы, очереди) для обмена данными между собой. * **Настоящая параллельность**: Благодаря отсутствию общего GIL между процессами, многопроцессные приложения могут эффективно использовать многоядерные процессоры. |
| **Основные преимущества asyncio в Python** <https://stepik.org/lesson/933660/step/1?unit=939559>  **Asyncio** – это встроенный модуль для асинхронного программирования в языке Python  позволяет избегать блокировок, связанных с ожиданием ресурсов.  В **asyncio** приложение может выполнять другие задачи во время ожидания, что в результате повышает производительность приложения.  для обработки нескольких тысяч открытых соединений не требуется тысячи потоков или процессов. Вместо этого asyncio использует мультиплексирование ввода-вывода для обработки всех этих соединений в рамках одного потока или процесса. |
| IO bound и CPU bound **IO(input / output) bound** – это программы, которые занимают большую часть времени на ввод/вывод данных.  В asyncio мы можем управлять **IO bound** задачами с помощью корутин. Корутины - это асинхронные функции, которые могут останавливаться и возобновляться, пока они ждут выполнения IO bound задачи. Это позволяет другим корутинам выполняться в то время, как одна ждет IO bound задачи. |
| Awaitable объекты |
| Введение в Awaitable объекты Awaitable объекты в asyncio - это объекты, которые можно использовать в выражении await.  Есть три типа awaitable объектов:   * Сoroutine , * Tasks и * Futures.   они могут быть переданы в оператор await для асинхронного выполнения кода.  Когда вы пишете ключевое слово await перед awaitable объектом, вы приостанавливаете выполнение текущей корутины до тех пор, пока awaitable объект не вернет результат или не подаст сигнал в цикл событий.  Используя **await** в какой-либо корутине, мы таким образом объявляем, что корутина может отдавать управление обратно в event loop, который, в свою очередь, запустит какую-либо следующую задачу.  **Coroutine** - это функция, которая может приостанавливаться во время выполнения и возобновляться позже, передавая управление другому awaitable объекту.  **Корутины** являются основой асинхронного программирования в Python. Они представляют собой специальные функции, объявленные с ключевым словом async def. Корутины выполняются асинхронно, и при вызове возвращают объект корутины, который можно передать в оператор await.  async def my\_coroutine():  # Код корутины  **Future** - это специальные объекты, которые могут быть ожидаемыми, но они не предназначены для создания пользователем. Они обычно используются, когда низкоуровневый код общается с кодом на уровне asyncio.   * Объекты типа **Future**похожи на объекты **Task**, но предназначены для интеграции с низкоуровневым кодом. Они используются для представления результата асинхронной операции, который может быть доступен в будущем. Создать объект **Future**можно с помощью метода asyncio.ensure\_future(). |
| **Task** - это awaitable объект, который оборачивает корутину в задачу task для дальнейшего планирования и запуска в event loop.  Объекты типа **Task** используются для запуска корутин в event loop (в цикле событий). Они предоставляют удобный способ отслеживания выполнения корутины и их результата. Создать объект **Task**можно с помощью метода asyncio.create\_task().  task = asyncio.create\_task(my\_coroutine())  **Задачи**- это как делегирование задачи сотруднику. Вы отдаёте задачу и ожидаете, что она будет выполнена, пока занимаетесь другими делами. |
| import asyncio  async def coocker(i):  print(f'cooker {i} starts to cook')  await asyncio.sleep(i)  print(f'cooker {i} ends to cook')  return f"dish from coocker {i}"  async def main():  tasks = [asyncio.create\_task(coocker(i)) for i in range(1,4)]  print(await asyncio.gather(\*tasks))   if \_\_name\_\_ == '\_\_main\_\_':  asyncio.run(main())   * Функция cook\_dish(n) - это корутина, которая имитирует поведение повара, готовящего блюдо. Эта функция "спит" (или приостанавливает свое выполнение) на количество секунд, равное номеру повара, что имитирует процесс приготовления блюда. * Функция main() создает задачи для каждого повара (от 1 до 3) с использованием asyncio.create\_task(), которые затем выполняются конкурентно. Результаты всех этих задач собираются с помощью asyncio.gather(\*tasks), который ожидает, пока все задачи не будут выполнены, и затем собирает результаты в список. |
| [sttackowerflow ASYNCIO.Отличие Tasks от Future](https://ru.stackoverflow.com/questions/902586/asyncio-Отличие-tasks-от-future/902762#902762) |
| Coroutine execution async def callee():  print('Hello')  async def caller():  await callee()  print('World') # ждет пока выполнится calle  выполнение caller приостановится до выполнения callee. В этот момент какие-то другие операции в каких-то других сопрограммах могут продолжаться, но caller будет ждать там, где выполнил await. |
| Task execution async def callee():  print('Hello')  async def caller():  asyncio.create\_task(callee())  print('World') # не ждет выполняется сама  caller сразу же продолжит свою работу. Строка "World" будет выведена раньше, чем "Hello". Здесь мы видим, что caller поставил циклу событий **задачу** выполнить сопрограмму callee. |
| Future execution Футура (Future) - будущий результат выполнения сопрограммы. Метод ensure\_future поручает циклу событий выполнить сопрограмму и сразу же, в момент вызова, возвращает футуру, в которой будет значение, но неизвестно когда. Вызывающая сторона может подождать выполнения футуры так же, как ожидало саму сопрограмму  async def callee():  return 'Hello'  async def caller():  loop = asyncio.get\_event\_loop()  future = loop.ensure\_future(callee())  result = await future # резервирует ответ  print(result + ' World')  Или может заняться своими делами, периодически проверяя готовность  async def caller():  loop = asyncio.get\_event\_loop()  future = loop.ensure\_future(callee())  while not future.done(): # резервирует ответ  # Какие-нибудь циклические дела  print(future.result() + ' World') |
| Event Loop **Event loop** - это цикл событий, который последовательно выполняет задачи в одном потоке, а не в нескольких, что делает его более простым и предсказуемым. Цикл событий по большей части всего лишь управляет выполнением различных задач: регистрирует поступление и запускает в подходящий момент задачи(**tasks**) и события(**futures**).  **Event loop**является центральным механизмом, управляющим выполнением асинхронных задач. Он отслеживает задачи, которые готовы выполняться, и выполняет их в порядке появления. |
| #Базовый цикл событий мог бы выглядеть следующим образом:  from collections import deque  messages = deque()  while True:  if messages:  message = messages.pop()  process\_message(message) |
|  |
| # Example Pizza programm  import asyncio from random import randint  class Pizzeria:  def \_\_init\_\_(self, name):  self.name = name   async def cooking\_pizza(self, order\_id):  cook\_time = randint(2,7)  print(f'Pizzeria "{self.name}" satrted to cook order {order\_id}, cooktime: {cook\_time}')  await asyncio.sleep(cook\_time)  print(f'Pizzeria "{self.name}" completed order {order\_id}')   async def main():  pizzeria = Pizzeria('Bread & Cheese')  tasks = [pizzeria.cooking\_pizza(i) for i in range(1,6) ]  await asyncio.gather(\*tasks)  if \_\_name\_\_ == '\_\_main\_\_':  asyncio.run(main()) |
| Ключевое слово await await - это ключевое слово, которое используется для переключения между awaitable объектами. Это позволяет вашей программе продолжать выполняться, в то время как другая асинхронная функция выполняется в фоновом режиме. Когда асинхронная функция завершится, await вернет результат. Чтобы использовать await, ваша функция должна быть объявлена с ключевым словом async def. |
| Test1 – non async code  import asyncio  async def counter():  for i in range(100):  await asyncio.sleep(.1)  print(i)   async def main():  print('we started')  # await asyncio.sleep(1)  # await counter() # non-parallel, occupies 10 seconds   # tasks = [asyncio.create\_task(counter()),] # non-parallel, occupies 10 seconds  # await asyncio.gather(\*tasks)   await asyncio.create\_task(counter()) # non-parallel, occupies 10 seconds   print('we completed')   if \_\_name\_\_ == "\_\_main\_\_":  asyncio.run(main())  we started  0  1  2  ......  .....  94  95  96  97  98  99  100  we completed |
| Test2 Parallel execution  import asyncio  async def counter():  for i in range(100):  await asyncio.sleep(.1)  print(i)  async def informer():  print('we started')  await asyncio.sleep(1)  print('we completed')   async def main():   tasks = [asyncio.create\_task(informer()),  asyncio.create\_task(counter()),  ]  await asyncio.gather(\*tasks)   if \_\_name\_\_ == "\_\_main\_\_":  asyncio.run(main())  D:\GB\pythonProject\asyncio\venv\Scripts\python.exe D:/GB/pythonProject/asyncio/Chapter\_3/3\_3\_await/tmp1.py  we started  0  1  2  3  4  5  6  7  8  we completed  9  10  11  12 |
| Await example 1 Когда вы хотите ожидать завершения асинхронной операции, такой как чтение или запись в сеть, чтение или запись в файл, или выполнение других длительных операций.  import aiohttp  import asyncio  async def fetch\_data(url):  async with aiohttp.ClientSession() as session:  async with session.get(url) as response:  return await response.text()  async def main():  data = await fetch\_data('http://python.org')  print(data)  asyncio.run(main())  *В контексте данного кода await используется перед fetch\_data('http://python.org') и response.text(). Это значит, что выполнение функции main() приостанавливается, пока не будут получены данные с сайта, а затем выполнение функции fetch\_data() приостанавливается, пока не будет прочитан текст ответа от сервера.* |
| Await example2 Когда вы хотите ожидать завершения асинхронной функции, которая возвращает объект, который можно ожидать (Awaitable) или которая вызывается внутри другой асинхронной функции. В примере ниже это функция cook\_pasta()  import asyncio  async def cook\_pasta():   print("Начинаем готовить пасту")   await asyncio.sleep(5)   print("Паста готова")   async def main():   await cook\_pasta()   asyncio.run(main())  В данном коде ключевое слово *await* используется для приостановки выполнения функции до завершения асинхронной операции (в данном случае *asyncio.sleep(5)*). Это позволяет коду оставаться отзывчивым и продолжать выполнять другие асинхронные операции, вместо того чтобы блокировать всю программу, пока не завершится операция с задержкой. |
| Await example3 Когда вы хотите ожидать завершения асинхронной функции, которая вызывается с помощью asyncio.ensure\_future или asyncio.create\_task.  import asyncio  async def cook\_pasta():   print("Начинаем готовить пасту")   await asyncio.sleep(5)   print("Паста готова")   async def main():   task = asyncio.create\_task(cook\_pasta())   await task   asyncio.run(main())    То же самое, что и пример ранее, только вместо корутины мы ждём задачу(task). |
| Await example4 Когда вы хотите ожидать завершения нескольких асинхронных операций с помощью asyncio.gather(\*aws).  import asyncio   async def cook\_pasta():   print("Начинаем готовить пасту")   await asyncio.sleep(5)   print("Паста готова")   async def cook\_sauce():   print("Начинаем готовить соус")   await asyncio.sleep(3)   print("Соус готов")   async def main():   await asyncio.gather(cook\_pasta(), cook\_sauce())   asyncio.run(main())  В этом примере ключевое слово *await* используется для приостановки выполнения текущей функции до тех пор, пока из функции *asyncio.gather(cook\_pasta(), cook\_sauce())* не вернётся результат выполнения *cook\_pasta()*и*cook\_sauce().* |
| Await example5 # Псевдокод который ничего не делает,  # но наглядно демонстрирует применение await  import asyncio  async def fetch\_data(url):   await asyncio.sleep(1)   return f"Data from {url}"   async def main():   data1 = await fetch\_data("https://www.example1.com")   data2 = await fetch\_data("https://www.example2.com")    print(f'{data1=}')   print(f'{data2=}')   asyncio.run(main())  В этом примере у нас есть две корутины fetch\_data() и main().   * Корутина fetch\_data() выполняет асинхронную операцию, которая занимает одну секунду. * Корутина main() дважды вызывает fetch\_data(), и каждый раз она использует оператор await, чтобы приостановить свое выполнение и ждать завершения корутины fetch\_data(). |
| FUTURE Оператор await может быть использован с любой асинхронной операцией, которая возвращает future.  Future - это специальный тип объекта, который представляет результат задачи, которая еще не завершена. Оператор await принимает future в качестве аргумента и блокирует выполнение корутины, пока future не будет готов. |
| Await example 6 Визуальный пример:  import asyncio   async def hello\_world\_message() -> str:  # Имитация выполнения длительной I/O операции (1 секунда)  await asyncio.sleep(1)  return 'Hello World!'   async def main() -> None:  # Приостановка main() до выполнения hello\_world\_message/получения результатов ее работы.  message = await hello\_world\_message()  print(f'{message=}')  asyncio.run(main())  Эта программа ждет 1 сек, а затем печатает сообщение «**Hello World!**». Поскольку hello\_world\_message() – корутина, а мы приостановили ее на 1 сек, у нас появилась одна секунда, в течение которой мог бы конкурентно работать другой код. В последующих примерах мы часто будем использовать asyncio.sleep(). |
| Await example 7 import asyncio async def add\_one(num: int) -> int:  return num + 1  async def hello\_world\_message() -> str:  await asyncio.sleep(1)  return 'Hello World!'   async def main() -> None:  # Приостановка main() до выполнения hello\_world\_message  message = await hello\_world\_message()  # Приостановка main() до выполнения add\_one  one\_plus\_one = await add\_one(1)  print(f'{one\_plus\_one=}')  print(f'{message=}')   asyncio.run(main()) |
| Coroutines / Корутины |
| В Python асинхронный код реализуется с помощью корутин, которые представляют собой особый вид функций, написанных с использованием выражения async def. |
| import asyncio # Такой синтаксис является не верным, т.к. все корутины должны запускаться внутри цикла событий  #НЕ ПРАВИЛЬНО async def example\_coroutine():  print("Hello from coroutine!")  example\_coroutine()   #ПРАВИЛЬНО async def example\_coroutine():  print("Hello from coroutine!")  asyncio.run(example\_coroutine()) |
| print(type(example\_coroutine()))  >>> <class 'coroutine'> |
| Корутина  может принимать аргументы и возвращать значения, как и синхронная функция.  # Определяем корутину с аргументами и возвращаем результат выполнение функции  async def summ\_coroutine(arg1, arg2, arg3):  # ...  return arg1 + arg2 + arg3 |
| Вызов определенной корутины  создаст объект корутины .  # Создание объекта корутины  coro = custom\_coroutine()  Корутину  можно выполнить, используя ее как \*точку входа, через функцию asyncio.run(). Функция asyncio.run() из библиотеки asyncio принимает в качестве аргумента корутину (точку входа) и запускает ее в новом событийном цикле. |
| Объект корутины  может быть запланирован для выполнения внутри другой корутины, приостанавливая выполнение вызывающей корутины  до тех пор, пока вызванная корутина  не будет выполнена. Этого можно добиться с помощью ключевого слова await.  ...  coro = custom\_coroutine() # Создание корутины  await coro # Планирование корутины  и ожидание ее выполнения |
| Ключевое слово await используется для вызова асинхронных функций или корутин. await приостанавливает выполнение текущей корутины до тех пор, пока асинхронная операция не завершится, и возвращает результат выполнения этой операции:  import asyncio  async def fetch\_data():  await asyncio.sleep(3)  return "Результат выполнения корутины fetch\_data"  async def main(): # Первая запускаемая корутина, из неё можно запускать остальные и возвращать результат для удобства  print("Получение данных...")  data = await fetch\_data()  print("Данные:", data)  asyncio.run(main())# Точка входа |
| # Чтобы запустить корутину в asyncio, необходимо использовать функцию asyncio.run().  # Эта функция запускает цикл событий и выполняет все задачи, добавленные в цикл.  # Вот пример того, как запустить корутину my\_coroutine() из предыдущего примера:  import asyncio  async def my\_coroutine(duration):   print("Начинаем my\_coroutine")   await asyncio.sleep(duration)     print("Завершаем my\_coroutine")   return "Результат"    asyncio.run(my\_coroutine(1)) |
| асинхронный код, который выполняет несколько задач конкурентно  import asyncio  async def first\_coroutine(duration):   print("Начинаем first\_coroutine")   await asyncio.sleep(duration)     print("Завершаем first\_coroutine")   return "Результат 1"   async def second\_coroutine(duration):   print("Начинаем second\_coroutine")   await asyncio.sleep(duration)   print("Завершаем second\_coroutine")   return "Результат 2"   async def main():  result1, result2 = await asyncio.gather(first\_coroutine(5),   second\_coroutine(3))  print("Результат 1:", result1)  print("Результат 2:", result2)   asyncio.run(main()) |
| ways to run async code: import asyncio import asyncio  async def coro\_1():  await asyncio.sleep(1)  print('coro\_1 says, hello coro\_2!')   async def coro\_2():  await asyncio.sleep(2)  print('coro\_2 says, hello coro\_1!')   async def main():  # task1 = asyncio.create\_task(coro\_1())  # task2 = asyncio.create\_task(coro\_2())  # or  # await asyncio.gather(task1, task2) # run 2 tasks  # or  # await task1 # run task 1  # await task2 # run task 2   # or  # await coro\_1() # run coroutine 1  # await coro\_2() # run coroutine 2  asyncio.run(main()) |
| TASK  обертки для корутин и используются для их конкурентного выполнения. Когда корутина оборачивается в задачу с помощью таких функций, как asyncio.**create\_task()**, она автоматически запускается в ближайшее время. |
| import asyncio  async def my\_coroutine():   await asyncio.sleep(1)   print("Задача выполнена")   async def main():   task = asyncio.create\_task(my\_coroutine())   await task   asyncio.run(main()) |
| Формат TASK asyncio.create\_task(**coro**, \*, **name**=None, **context**=None)  - принимает следующие аргументы:   * coro - корутина, которую нужно обернуть в задачу и запустить в цикле событий. * name - необязательный параметр, который позволяет установить имя для задачи. Это может быть полезно для отладки и мониторинга программы. * context - необязательный параметр, который позволяет задать контекст (context) для задачи. Контекст – это словарь с дополнительными данными, которые могут быть использованы внутри корутины. |
| В каких случаях применяются задачи? Задачи применяются в ситуациях, когда вам нужно конкурентно запустить несколько корутин и, возможно, дождаться их завершения. Это особенно полезно в сценариях, где корутины выполняют**I**/**O**-операции, такие как чтение/запись файлов, отправка/получение данных по сети и т.д., которые могут быть эффективно выполнены конкурентно. |
| Важно! Сохраняйте ссылку на результат этой функции, чтобы задача не исчезла в процессе выполнения. Цикл событий сохраняет только слабые ссылки на задачи. Задача, на которую больше нигде нет ссылок, может быть удалена сборщиком мусора в любое время, даже до того, как она будет выполнена. Для надежных фоновых задач типа "запустил и забыл" нужно собрать их в коллекцию(множество, список или кортеж). |
| Future Future -  awaitable объект, который позволяет выполнять задачи в фоновом режиме и получать результаты позже. Они могут быть использованы для запуска длительных операций, таких как загрузка данных из сети, в отдельном потоке или процессе, чтобы не блокировать основной поток приложения. |
| Пример import asyncio  # Функция "производителя" async def producer():  future = asyncio.Future() # Создаем экземпляр Future  await asyncio.sleep(2) # Имитация долгой операции  future.set\_result('Future is done!') # Устанавливаем результат Future  return future  # Функция "потребителя" async def consumer():  future = await producer() # Ждем Future от производителя  print(await future) # Получаем результат Future и выводим его   asyncio.run(consumer())  output:  Future is done! |
| Пример import asyncio  async def do\_some\_work(x):   print(f"Выполнение некоторой работы: {x}")   await asyncio.sleep(x)   return x \* 2   async def main():   future = asyncio.ensure\_future(do\_some\_work(2))   result = await future   print(f"Результат: {result}")   asyncio.run(main())    result:  Выполнение некоторой работы: 2  Результат: 4 |
| Example3 # Пример с управлением зависимостями между корутинами и последовательным их выполнением:  import asyncio   async def do\_some\_work\_1(x):   print(f"Doing some work 1: {x}")   await asyncio.sleep(x)   return x \* 2    async def do\_some\_work\_2(x):   print(f"Doing some work 2: {x}")   await asyncio.sleep(x)   return x + 2    async def main():   future\_1 = asyncio.ensure\_future(do\_some\_work\_1(2))   result\_1 = await future\_1   future\_2 = asyncio.ensure\_future(do\_some\_work\_2(result\_1))   result\_2 = await future\_2   print(f"Result future\_1: {result\_1}")   print(f"Result future\_2: {result\_2}")    asyncio.run(main())  # Вывод: # # Doing some work 1: 2 # Doing some work 2: 4 # Result future\_1: 4 # Result future\_2: 6 |
| Объект Future  * Объект Future - это объект, который представляет результат асинхронной операции, которая еще не завершена, но уже ожидается. * При создании объекта Future он находится в состоянии ожидания (**pending**). * Когда операция завершится, объект Future перейдёт в состояние завершения (**completed**) и будет содержать результат операции. * Объект Future также может находиться в состоянии ошибки (**exception**), если операция завершилась с ошибкой.   Объекты Future широко используются в asyncio для управления асинхронными операциями. Также он является **основным строительным блоком** для асинхронного программирования.  Вот некоторые характеристики объектов Future:   1. **Состояние**: объект Future может находиться в одном из трёх состояний:    * **Pending**(ожидание) -  в этом состоянии операция, связанная с объектом Future, еще не выполнена. Это начальное состояние для всех объектов Future. В этом состоянии вы не можете получить результат операции, поскольку он еще не доступен.    * **Completed**(завершён) - когда операция завершена, объект Future переходит в состояние "завершено". В этом состоянии вы можете получить результат операции, вызвав метод result() объекта Future. Если операция завершилась с ошибкой, вызов result() вызовет исключение.    * **Cancelled** (отменён) - если операция была отменена, объект Future переходит в состояние "отменено". Операция может быть отменена вызовом метода cancel() объекта Future. Если операция была отменена, вызов result() вызовет исключение CancelledError. |
| Результат объект Future хранит результат асинхронной операции, который может быть успешным значением или исключением. Методы result() и exception() используются для получения результата или исключения соответственно.  Проверить, был ли выполнен Future, можно с помощью метода **fut.done()**  Метод возвращает True, если Future выполнен. Future считается выполненным, если он был отменен или если для него был установлен результат или исключение с помощью вызовов **set\_result()** или **set\_exception()** |
| Проверка состояния **Проверка состояния без блокировки:**done() позволяет проверить, завершился ли Future  done() мгновенно возвращает состояние Future, не вызывая ожидания его завершения  Метод result() может вызывать исключения, если Future был отменен  Метод exception() могут вызывать исключения, если Future  завершился с ошибкой.  Использование done() позволяет сначала проверить, завершился ли Future, и затем безопасно обработать результат или исключение. |
| Example4 future, callback import asyncio  # Это наша функция обратного вызова def callback(future):  print(f"Обратный вызов: результат асинхронной операции равен {future.result()}")  async def do\_some\_work(x):  print(f"Выполнение некоторой работы: {x}")  await asyncio.sleep(x)  return x \* 2  async def main():  task = asyncio.ensure\_future(do\_some\_work(2)) # Создаем задачу, используя ensure\_future  task.add\_done\_callback(callback) # Добавляем обратный вызов к задаче  await task # Дожидаемся завершения работы функции  asyncio.run(main()) |
| Future отмена Отмена: объекты Future могут быть отменены, если асинхронная операция еще не началась или не завершилась. Метод cancel() используется для отмены объекта Future.  Если отмена прошла успешно, cancelled() вернет True. |
| Good To Know |
| Long polling **Long polling** (также известный как Comet-схема или длинный опрос) - это веб-технология, которая используется для улучшения веб-приложений. Она работает так: клиент инициирует запрос, сервер устанавливает соединение и ждет новых данных, прежде чем ответить на запрос клиента. Как только новые данные появятся, сервер вернет их клиенту и закроет соединение. После получения данных или истечения тайм-аута клиент немедленно инициирует новый запрос, чтобы установить новое соединение, продолжая таким образом процесс получения обновлений в режиме реального времени. |
| **Примеры использования Long polling:**  * **Чаты в реальном времени**: клиенты могут получать уведомления, например, о новых электронных письмах или обновлениях социальных сетей, без необходимости постоянно обновлять страницу. * **Игры**: игроки могут получать обновления о состоянии игры, например, о передвижении других игроков, без необходимости постоянно запрашивать обновления. * **Напоминания**: пользователи могут получать напоминания в реальном времени, например, о предстоящем событии или дедлайне, без необходимости постоянно проверять |
| **Недостатки Long polling:**  * **Ограниченная поддержка браузеров: некоторые старые версии браузеров могут не поддерживать Long polling, что может привести к проблемам с \*компатибильностью.** * **Ограничения по времени: некоторые прокси-серверы или фаерволы могут ограничивать время жизни запроса, что может привести к проблемам с Long polling.** * **Увеличение нагрузки на сервер: в сравнении с другими технологиями, такими как WebSockets, Long polling может создавать дополнительную нагрузку на сервер, так как соединения удерживаются открытыми в течение длительного времени. Это может быть неэффективно в средах с большим количеством одновременных пользователей. В зависимости от реализации серверной архитектуры, возможна ситуация, когда для каждого соединения создается свой процесс и каждый процесс занимает значительный объем памяти. Чем больше соединений, тем больше расход памяти.** * **Задержки в передаче данных: в сравнении с WebSockets, Long polling может привести к большим задержкам в передаче данных, так как он использует отдельный запрос для каждой передачи данных.** |
| **Example** # Пример кода, демонстрирующий Long polling:  import asyncio import aiohttp  async def fetch\_data(url):  async with aiohttp.ClientSession() as session:   async with session.get(url) as resp:   data = await resp.json()   return data   async def process\_data(data):   print(f"Received data: {data}")     # Может включать в себя любую другую логику, которую вы хотите выполнить с полученными данными,  # например, сохранение в базу данных, отправку уведомлений или обновление интерфейса пользователя.  async def long\_polling(url):   while True:   await asyncio.sleep(5)   data = await fetch\_data(url)   if data:   await process\_data(data)   async def main():   task = asyncio.create\_task(long\_polling("https://jsonplaceholder.typicode.com/posts"))  await task   asyncio.run(main()) |
| Polling **Polling**(от англ. "опрос") - это метод мониторинга состояния какого-либо устройства, системы или процесса. Он основывается на периодическом опрашивании устройства для получения информации о его состоянии.  В программировании поллинг может использоваться для отслеживания состояния ввода, например, состояния клавиатуры или мыши, или для определения состояния сетевых устройств, таких как принтеры или серверы.  Однако поллинг может быть медленным и ресурсоемким, поскольку процесс мониторинга выполняется постоянно. Для улучшения эффективности используются другие методы, такие как прерывания (Interrupts).  **Примеры использования polling:**   * **Отслеживание состояния клавиатуры**: Программа постоянно опрашивает состояние клавиатуры и обрабатывает нажатые клавиши. * **Отслеживание состояния сетевого устройства**: Программа постоянно опрашивает состояние сетевого устройства, такого как принтер или сервер, чтобы определить, доступно оно или занято. * **Отслеживание состояния потока данных**: Программа постоянно опрашивает состояние потока данных и обрабатывеат приходящие данные.   **Недостатки polling:**   * **Ресурсоемкость**: Поллинг требует постоянного мониторинга, что может привести к высокой нагрузке на процессор и память. * **Медленный отклик**: Поллинг может быть медленным, поскольку он должен постоянно опрашивать устройство или систему, чтобы получить информацию о ее состоянии. * **Неэффективность**: Поллинг может быть неэффективным, поскольку он требует постоянного мониторинга, даже если устройство или система не имеют никаких изменений.   Из-за этих недостатков в некоторых случаях лучше использовать прерывания (\*Interrupts) вместо поллинга, чтобы улучшить эффективность и уменьшить нагрузку на процессор |
| \***Interrupts**(прерывания) - это механизм, который позволяет процессору прерывать текущую задачу и переключаться на выполнение другой задачи, когда происходит определенное событие, такое как нажатие клавиши или получение данных от устройства. Прерывания позволяют улучшить эффективность системы, так как процессор не нуждается в постоянном мониторинге устройств или событий. Вместо этого он может продолжать выполнение текущей задачи, а когда происходит событие, он прерывается и выполняет соответствующую задачу. |
| Polling Example # Пример кода, демонстрирующий polling:  import asyncio import random   async def process\_data(data):  print(f"Полученные данные: {data}")    # Может включать в себя любую другую логику, которую вы хотите выполнить с полученными данными,  # например, сохранение в базу данных, отправку уведомлений или обновление интерфейса пользователя.  async def polling\_network\_device():   while True:   data = random.randint(0, 1000000)   await process\_data(data)       await asyncio.sleep(5)    async def main():   task = asyncio.create\_task(polling\_network\_device())   await task    asyncio.run(main()) |
| Interruption example Пример кода с прерыванием  import asyncio  async def print\_message():  while True:  print("Имитация работы функции")   await asyncio.sleep(1)   async def interrupt\_handler(interrupt\_flag):  while True:  await asyncio.sleep(0.5)   if interrupt\_flag.is\_set():   print("Произошло прерывание!, "  "в этом месте может быть установлен любой обработчик")  interrupt\_flag.clear()   break   async def main():  interrupt\_flag = asyncio.Event()   task1 = asyncio.create\_task(print\_message())   task2 = asyncio.create\_task(interrupt\_handler(interrupt\_flag))   while True:  await asyncio.sleep(3)   interrupt\_flag.set()   await task2   task2 = asyncio.create\_task(interrupt\_handler(interrupt\_flag))   asyncio.run(main())  Вывод:  Имитация работы функции  Имитация работы функции  Имитация работы функции  Имитация работы функции  Произошло прерывание!, в этом месте может быть установлен любой обработчик  Имитация работы функции  Имитация работы функции  Имитация работы функции  Произошло прерывание!, в этом месте может быть установлен любой обработчик  ...  #Код будет выполнятся до бесконечности |
| Mutex Мьютекс (от англ. "[mutual exclusion](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81)"), сокращение от "взаимное исключение" - это механизм синхронизации, который используется для предотвращения одновременного доступа к общему ресурсу. Он представляет собой объект, который блокирует доступ к критической секции кода, пока другая задача не освободит этот ресурс. |
| Мьютекс может находиться в двух состояниях: заблокированном (**locked**) и разблокированном (**unlocked**).  Некоторые ситуации, в которых может потребоваться использовать мьютекс:   1. **Общие структуры данных**: Если несколько задач читают и изменяют общую структуру данных (например, список, словарь или пользовательский объект), мьютекс может быть использован для обеспечения согласованности данных и предотвращения состояния гонки. 2. **Доступ к файлам**: Если несколько задач записывают данные в один и тот же файл, мьютекс может быть использован для синхронизации доступа к файлу и обеспечения корректной записи данных. 3. **Общие ресурсы системы**: Если задачи используют общие ресурсы системы, такие как сетевые соединения или устройства, мьютекс может использоваться для синхронизации доступа и предотвращения конфликтов. 4. **Ограничение одновременного выполнения**: В некоторых случаях требуется ограничить число задач, которые могут выполнять определенную операцию одновременно. Мьютекс может использоваться для этой цели, ограничивая число потоков, которые имеют доступ к ресурсам или выполняют задачи. 5. **Кэширование**: Мьютекс также может использоваться для синхронизации доступа к кэшированным данным между задачами и предотвращения ситуации, когда несколько задач пытаются одновременно обновить кэш. |
| Lock В библиотеке asyncio нет классического мьютекса, но для синхронизации доступа к ресурсам можно использовать объекты типа Lock. Они функционируют аналогично мьютексам и предоставляют методы для получения и освобождения блокировки.  import asyncio  lock = asyncio.Lock()  Основные методы для работы с объектом блокировки – это acquire() и release().  Метод acquire() позволяет получить блокировку, а  метод release() – освободить ее. Метод acquire()являются асинхронными и должен быть вызван с использованием ключевого слова await, в то время как release() вызывается напрямую без await.  await lock.acquire()  # Критическая секция кода  lock.release() |
| Example Lock Acquire Release # Пример кода с использованием acquire() и release  import asyncio  async def access\_resource(i, resource, lock):  await lock.acquire()   try:  print(f"{i} Доступ получен {resource}")   await asyncio.sleep(1)   print(f"{i} Доступ завершён {resource}")   finally:  lock.release()   async def main():  lock = asyncio.Lock()   resource = "--Защищаемый ресурс--"   tasks = [access\_resource(i, resource, lock) for i in range(5)]  await asyncio.gather(\*tasks)   asyncio.run(main())  Output  0 Доступ получен --Защищаемый ресурс--  0 Доступ завершён --Защищаемый ресурс--  1 Доступ получен --Защищаемый ресурс--  1 Доступ завершён --Защищаемый ресурс--  2 Доступ получен --Защищаемый ресурс--  2 Доступ завершён --Защищаемый ресурс--  3 Доступ получен --Защищаемый ресурс--  3 Доступ завершён --Защищаемый ресурс--  4 Доступ получен --Защищаемый ресурс--  4 Доступ завершён --Защищаемый ресурс--  Process finished with exit code 0 |
| Мьютексы с помощью менеджера контекста with: async with lock:  # Критическая секция кода  # Пример кода с использованием async with lock:  import asyncio  async def access\_resource(resource, lock):  async with lock:   print(f"Доступ получен {resource}")   await asyncio.sleep(1)   print(f"Доступ завершён {resource}")   async def main():  lock = asyncio.Lock()   resource = "--Защищаемый ресурс--"   tasks = [access\_resource(resource, lock) for \_ in range(5)]  await asyncio.gather(\*tasks)   asyncio.run(main())  Вывод:  Доступ к ресурсу получен --Защищаемый ресурс--  Доступ к ресуру завершён --Защищаемый ресурс--  Доступ к ресурсу получен --Защищаемый ресурс--  Доступ к ресуру завершён --Защищаемый ресурс--  ... |
| Атомарность операций Атомарность операций означает, что операция выполняется непрерывно и полностью или не выполняется вообще, без возможности частичного выполнения или перекрытия с другими операциями.  Для достижения атомарности в Python можно использовать **механизмы синхронизации**, такие как **блокировки Lock**, **условные переменные Condition, семафоры Semaphore** |
| Atomic bad example import asyncio  # Общий ресурс, который будет обновляться shared\_resource = 0  # Асинхронная функция для обновления общего ресурса async def update\_resource():  # Используем глобальную переменную shared\_resource  global shared\_resource  print('Начинаем обновление shared\_resource')  # Сохраняем текущее значение shared\_resource во временную переменную  temp = shared\_resource  await asyncio.sleep(1) # Имитация операции ввода-вывода  # Увеличиваем значение shared\_resource на 1  shared\_resource = temp + 1  print('Обновление shared\_resource завершено')   async def main():  # Запускаем две асинхронные задачи для обновления shared\_resource  await asyncio.gather(update\_resource(), update\_resource())  print(f'shared\_resource: {shared\_resource}')  asyncio.run(main())  output:  Начинаем обновление shared\_resource  Начинаем обновление shared\_resource  Обновление shared\_resource завершено  Обновление shared\_resource завершено  shared\_resource: 1 |
| Atomic Good Example # Для обеспечения безопасности/атомарности при обновлении общего ресурса используем асинхронный замок asyncio.Lock(). # # Замок lock используется для обеспечения того, что только одна корутина может обновлять shared\_resource в любой момент времени, предотвращая таким образом условия гонки.  import asyncio  # Общий ресурс, который будет обновляться shared\_resource = 0 # Создаем асинхронный замок для обеспечения безопасности при обновлении shared\_resource lock = asyncio.Lock()   async def update\_resource():  # Используем глобальную переменную shared\_resource  global shared\_resource  print('Начинаем обновление shared\_resource')  # Используем асинхронный замок для обеспечения безопасности при обновлении shared\_resource  async with lock:  # Сохраняем текущее значение shared\_resource во временную переменную  temp = shared\_resource  await asyncio.sleep(1) # Имитация операции ввода-вывода  # Увеличиваем значение shared\_resource на 1  shared\_resource = temp + 1  print('Обновление shared\_resource завершено')  async def main():  # Запускаем две асинхронные задачи для обновления shared\_resource  await asyncio.gather(update\_resource(), update\_resource())  print(f'shared\_resource: {shared\_resource}')  asyncio.run(main())  Output:  Начинаем обновление shared\_resource  Начинаем обновление shared\_resource  Обновление shared\_resource завершено  Обновление shared\_resource завершено  shared\_resource: 2 |
| Atomic Good Example based on class import asyncio  class Update:  lock = asyncio.Lock()  shared\_resource=0   async def update\_resource(self):  print(f'Начинаем обновление shared\_resource')  await \_\_class\_\_.lock.acquire()  temp = \_\_class\_\_.shared\_resource  await asyncio.sleep(1)  \_\_class\_\_.shared\_resource = temp + 1  \_\_class\_\_.lock.release()  print(f'Обновление shared\_resource завершено-{\_\_class\_\_.shared\_resource}')  async def main():  await asyncio.gather(\*[Update().update\_resource() for \_ in range(2)])  print(f'shared\_resource: {Update.shared\_resource}')  asyncio.run(main()) |
| DEADLOCK блокировка  * [**Deadlock**](https://en.wikipedia.org/wiki/Deadlock) (тупик) – это ситуация, когда два или более процесса ожидают друг друга, чтобы завершить свою работу, но ни один из них не может продолжить выполнение. Deadlock является одной из наиболее распространенных проблем, возникающих в многопоточных, многопроцессорных и асинхронных приложениях. |
| Взаимная блокировка может быть предотвращена с помощью различных стратегий:   * Иерархия захвата ресурсов; * Обратный таймаут; * Обнаружение и восстановление. |
| **Общие примеры причин возникновения тупиков корутин включают:**   * Корутина, которая ожидает себя (например, пытается получить ту же мьютекс-блокировку дважды). * Корутины, которые ожидают друг друга (например, **A** ждет **B**, **B** ждет **A**). * Корутина, которая не освобождает ресурс (например, мьютекс-блокировка, семафор, барьер, условие, событие и т.д.). * Корутины, которые получают мьютекс-блокировки в разных порядках (например, не выполняют упорядочение блокировок). * Тупики могут быть легко созданы, но в действительности их сложно обнаружить в приложении только на основании чтения кода. |
| Пример Deadlock, ожидает саму себя # Пример Deadlock, вызванного тем, что корутина ожидает саму себя: import asyncio  # Корутина, которая дважды получает замок async def task(lock):  print('Задача пытается захватить замок...') # Здесь мы информируем о начале процесса захвата замка   # Захватываем замок  async with lock: # Мы используем контекстный менеджер для безопасного захвата и освобождения замка  print('Задача снова пытается захватить замок...') # Информируем о повторном попытке захвата замка   # Снова захватываем замок  async with lock: # Повторный захват замка приведет к взаимной блокировке, так как замок уже занят той же задачей  # Сюда мы никогда не дойдем  pass   async def main():  # Создаем общий замок  lock = asyncio.Lock() # Создаем экземпляр замка для использования в корутине  # Запускаем и ожидаем выполнения корутины  await task(lock) # Запускаем корутину с передачей ей замка   asyncio.run(main()) # Запускаем асинхронную функцию main, которая стартует нашу задачу  # Вывод: # Задача пытается захватить замок... # Задача снова пытается захватить замок... # ... # ... # # deadlock / бесконечное ожидание корутины / зависание приложения |
| Deadlock: Корутины ждут друг друга. # пример: Новая корутина: Ожидает основную корутину.  # Основная корутина: Ожидает новую корутину.  import asyncio   # принимает задачу и ждет ее завершения async def task(other):   print(f'ожидание задачи: {other.get\_name()}')     await other    async def main():   task1 = asyncio.current\_task()   task2 = asyncio.create\_task(task(task1))       await task(task2)   asyncio.run(main())  # Вывод: # # ожидание задачи: Task-2 # ожидание задачи: Task-1 # ... # ... # ... # deadlock / бесконечное ожидание корутины / зависание приложения |
| Deadlock: Корутины получают блокировки в неправильном порядке: Обычной причиной такого тупика является то, что две корутины получают блокировки в разном порядке одновременно.   **Coroutine1**: Удерживает **lock1**, ожидает **lock2**.   **Coroutine2**: Удерживает **lock2**, ожидает **lock1**.  import asyncio   # корутина, выполняющая задачу с использованием блокировок async def task(number, lock1, lock2):   print(f'Задача {number} приобретает блокировку {lock1}')   async with lock1:   await asyncio.sleep(1)   print(f'Задача {number} приобретает блокировку {lock2}')   async with lock2:   # сюда никогда не дойдем..  pass    async def main():   lock1 = asyncio.Lock()   lock2 = asyncio.Lock()   coro1 = task(1, lock1, lock2)   coro2 = task(2, lock2, lock1)     await asyncio.gather(coro1, coro2)    asyncio.run(main())  output:  Задача 1 приобретает блокировку <asyncio.locks.Lock object at 0x0000019FB9EB1000 [unlocked]>  Задача 2 приобретает блокировку <asyncio.locks.Lock object at 0x0000019FB9EB1060 [unlocked]>  Задача 1 приобретает блокировку <asyncio.locks.Lock object at 0x0000019FB9EB1060 [locked]>  Задача 2 приобретает блокировку <asyncio.locks.Lock object at 0x0000019FB9EB1000 [locked]>  Решением является гарантирование того, что блокировки всегда получаются в одном и том же порядке во всей программе. |
| Deadlock: Ошибка при освобождении блокировки Еще одной общей причиной тупика является невыполнение корутиной обязанности освободить ресурс.  Обычно это вызвано тем, что корутина вызывает ошибку или исключение в критическом разделе, что мешает корутине освободить ресурс.  Некоторые примеры ошибки освобождения ресурса:   * Неудача при освобождении блокировки (lock\*). * Неудача при освобождении семафора (semaphore\*). * Неудача при достижении барьера (barrier\*). * Неудача при оповещении корутин об условии (condition\*). * Неудача при установке события (event\*). * И так далее.   Корутина может получить блокировку для выполнения критического раздела, затем вызвать исключение, которое мешает корутине освободить блокировку.  Другая корутина может затем попытаться получить ту же блокировку и должна ждать вечно, так как блокировка никогда не будет освобождена, что приводит к тупику. |
| # Ошибка при освобождении блокировки.  import asyncio   # корутина, которая приобретает блокировку async def task(lock):   print('Задача приобретает блокировку...')   await lock.acquire()   raise Exception('Произошло что-то плохое')      print('Задача освобождает блокировку...')     lock.release()      async def main():  lock = asyncio.Lock()   asyncio.create\_task(task(lock))     await asyncio.sleep(1)     print('Main приобретает блокировку...')   await lock.acquire()  # освобождаем блокировку (сюда никогда не дойдем)  lock.release()       asyncio.run(main())  Задача приобретает блокировку...  Exception: Произошло что-то плохое  Main приобретает блокировку... |
| Советы по предотвращению Deadlock некоторые из этих важных передовых практик для избегания взаимных блокировок.   1. Используйте менеджеры контекста при захвате и освобождении блокировок. 2. Используйте таймауты при ожидании. 3. Всегда захватывайте блокировки в одном и том же порядке. |
| Традиционно рекомендуется всегда приобретать и освобождать блокировку в структуре try/finally. Или менеджер контекста # 1. рекомендуется всегда приобретать  # приобретаем блокировку lock.acquire() # Мето  try:  # критический участок кода...  # Здесь выполняется код, который  finally:  # всегда освобождаем блокировку  lock.release() # Мет  # 2. Менеджер контекста # Устанавливаем блокировку async with lock: # Конструкция with   # критический участок кода...  # Здесь выполняется код, который |
| Используйте таймауты при ожидании. В отличие от потоков и процессов, блокирующие вызовы на примитивах параллелизма, asyncio не принимают аргумент "timeout".  Мы должны использовать функцию asyncio.wait\_for()  3. timeout  try:  # пытаемся приобрести блокировку, ожидая не более 10 секунд  await asyncio.wait\_for(lock.acquire(), timeout=10) # Метод wait\_for() asyncio  except asyncio.TimeoutError:  # здесь обрабатываем ситуацию, когда время ожидания истекло  # ... |
| Race Condition Race Condition (гонка потоков) - это состояние, когда два или более потока (или процесса) пытаются получить доступ к одному и тому же ресурсу одновременно. |
| Race condition example – Bank account import asyncio  bank\_account = 1000   async def withdraw\_money(amount):   global bank\_account   if bank\_account >= amount:   print(f"Снятие {amount}р успешно")   await asyncio.sleep(1)   bank\_account -= amount   async def main():   task1 = asyncio.create\_task(withdraw\_money(900))     task2 = asyncio.create\_task(withdraw\_money(900))    await asyncio.gather(task1, task2)      print(f'Остаток средств {bank\_account}р')   asyncio.run(main())  # Вывод: # # Снятие 900 успешно # Снятие 900 успешно # Остаток средств -800 |
|  |

|  |
| --- |
| Основы работы с Asyncio |
| Основные методы asyncio **Asyncio** предоставляет функциональность для управления событийным циклом (event loop). Т.к. центральной концепцией в asyncio является цикл событий, все методы, реализованные в модуле asyncio, так или иначе влияют на него. Например, метод asyncio.run(coro) запустит указанную корутину в цикле событий |
| цикл событий – это такая конструкция, которая не только ожидает и отправляет события или задачи на выполнение, но и следит за ходом выполнения каждой корутины и их статуcом. Каждая корутина, запущенная в цикле событий, общается "сообщениями" под капотом, передавая информацию о своём статусе для эффективного управления ею. |
| короткое описание основных методов:  * asyncio.run(**coro**, \*, debug=False)  - создаёт цикл событий и запускает указанный coroutine в нём. После завершения выполнения **coro** автоматически закрывает цикл событий. Этот метод является основным методом для запуска и управления asyncio приложениями. * asyncio.sleep(**delay**, result=None, \*, loop=None) - приостанавливает выполнение текущей корутины на delay секунд. * asyncio.create\_task(**coro**, \*, name=None) - оборачивает coro в обёртку задачи, т.е. она будет запланирована для выполнения в цикле событий. * asyncio.wait\_for(**coro**, timeout, \*, loop=None) - ожидает завершения coro, и, если выполнение не завершено в течение **timeout** секунд, вызывается asyncio.TimeoutError. * asyncio.gather(\***aws**, loop=None, return\_exceptions=False) - конкурентно запускает awaitable объекты, переданные в функцию как последовательность \*aws, и собирает результаты их работы. * asyncio.wait(**fs**, \*, loop=None, timeout=None, return\_when=ALL\_COMPLETED) - возвращает кортеж из двух множеств Task/Future в виде (done, pending). Это позволяет вам узнать, какие задачи были выполнены и какие еще ожидают выполнения. * asyncio.ensure\_future(**obj**, \*, loop=None) -  Аргумент **obj**возвращается как есть, если obj является объектом Future, Task или Future-подобным объектом. * Если obj является корутиной, будет создан и возвращен новый объект Task, который оборачивает эту корутину; в этом случае корутина будет запланирована для выполнения с помощью ensure\_future().       Если obj является объектом допускающим ожидание, будет возвращен объект Task, который будет ожидать выполнения obj.  Если obj не является ни одним из вышеупомянутых, будет вызвано исключение TypeError.   * asyncio.**get\_event\_loop**() - метод возвращает текущий цикл событий для текущего контекста. * asyncio.**new**\_event\_loop() - метод создает новый цикл событий. * asyncio.**set**\_event\_loop(loop) - метод устанавливает цикл событий для текущего контекста. * asyncio.**get\_running**\_**loop**() - метод возвращает текущий запущенный цикл событий в текущем потоке ОС. Вызывает исключение **RuntimeError**, если нет запущенного цикла событий. * asyncio.shield() - создает объект shield, который может использоваться для защиты кода от воздействия внешних циклов событий или от отмены с помощью  Task.cancel() * asyncio.run\_until\_complete(**future**) - метод запускает цикл событий, пока **future** не выполнится. * asyncio.run\_coroutine\_threadsafe(coro, loop) - метод является потокобезопасным и предназначен для запуска корутины в указанном цикле событий. |
| Отличие asyncio.get\_event\_loop() от asyncio.get\_running\_loop() Методы asyncio.**get\_event\_loop**() и asyncio.**get\_running\_loop**() используются для получения текущего цикла событий, но они работают немного по-разному.  asyncio.**get\_event\_loop**() возвращает текущий цикл событий для текущего контекста выполнения. Если текущий цикл событий еще не установлен, метод **get\_event\_loop**() будет пытаться установить новый цикл событий и вернуть его. Если цикл событий не может быть получен или установлен, **get\_event\_loop**() вызовет ошибку.  С другой стороны, asyncio.**get\_running\_loop**() возвращает текущий цикл событий только если он активно выполняется. Если цикл событий не запущен, **get\_running\_loop**() вызовет ошибку RuntimeError. Этот метод полезен, когда вы хотите быть уверены, что ваш код выполняется в контексте запущенного цикла событий.  В общем, asyncio.**get\_running\_loop**() более строгий, и его использование может помочь избежать некоторых ошибок, связанных с попытками использовать цикл событий, который еще не запущен. |
| asyncio.run() asyncio.**run(**coro, \*, debug=None**)** - этот метод выполняет переданную корутину (coro), заботясь об управлении циклом событий asyncio.  asyncio.**run()** всегда создает новый цикл событий и закрывает его после завершения работы всех корутин. Его следует использовать в качестве основной точки входа для программ в asyncio, и в идеале он должен вызываться только один раз.  **Поддержка только корутин**: asyncio.run() может быть использован только с корутинами. Если вы попытаетесь передать обычную функцию вместо корутины, вы получите ошибку.  **Порядок выполнения**: asyncio.run(main()) не гарантирует порядок выполнения асинхронных операций внутри main(). Операции будут выполняться в порядке их завершения, а не в порядке их начала, но есть способ это исправить, но об этом в следующих уроках.  Эту функцию нельзя вызывать, когда другой цикл событий работает в том же потоке. Если debug равно True, цикл событий будет работать в режиме отладки. |
| Обработка исключений Вы можете столкнутся с подобной проблемой в  следующих случаях:   * **Сетевые запросы**: при выполнении асинхронных сетевых запросов может произойти исключение (например, если сервер недоступен). При этом важно, чтобы все остальные асинхронные запросы были завершены корректно. * **Асинхронные операции ввода/вывода**: при работе с асинхронными операциями ввода/вывода (например, чтение/запись файлов) также может произойти исключение (например, если файл не найден). В этом случае необходимо гарантировать, что все другие операции ввода/вывода будут завершены корректно, чтобы не потерять данные и не заблокировать ресурсы. * **Параллельные вычисления**: если вы выполняете набор долгих вычислительных задач асинхронно и одна из них вызывает исключение (например, из-за ошибки в данных), вы можете хотеть, чтобы все остальные задачи все равно были выполнены до конца, чтобы не терять время на их повторное выполнение. * **Обработка ошибок и исключений**: в некоторых случаях вы можете хотеть позволить вашему коду продолжить выполнение после возникновения исключения, чтобы обработать ошибку и, возможно, восстановить работу программы. Благодаря этому методу вы можете быть уверены, что даже в случае исключения все ваши задачи будут корректно завершены.     Для того, чтобы обработать исключение, необходимо добавить блок try/except на потенциально опасном участке кода. |
| Task\_Name async def task\_func(duration):  name = asyncio.current\_task().get\_name()  print(f'Задача {name} запущена, будет выполнена за {duration} секунд.')  await asyncio.sleep(duration)  print(f'Задача {name} завершена.')  async def main():  task1 = asyncio.create\_task(task\_func(3), name='first')  task2 = asyncio.create\_task(task\_func(1), name='second')  await asyncio.gather(task1, task2)   try:  asyncio.run(main()) |
| asyncio.sleep() asyncio.sleep(delay, result=None, \*, loop=None) –  это функция, которая приостанавливает выполнение текущей корутины на указанное количество секунд, которое задается параметром delay. Как только время ожидания истекает, выполнение корутины возобновляется.  delay - время в секундах, на которое нужно приостановить выполнение корутины. Значение этого параметра может быть как целочисленным, так и дробным числом. Если мы передаем целое число, то функция будет приостанавливать выполнение на это количество секунд. Если мы передаем дробное число, то функция будет приостанавливать выполнение на это количество секунд, округленное до ближайшей миллисекунды.  result - необязательный параметр, который задает значение, которое будет возвращено при возобновлении выполнения корутины. Если этот параметр не задан, то asyncio.sleep() возвращает None.  import asyncio  async def main():  print(await asyncio.sleep(1, result='Сообщение из result'))  asyncio.run(main())  >>> Сообщение из result  Параметр result в функции asyncio.sleep() может быть полезен, когда вы хотите вернуть некоторый результат или значение после определенной задержки, например, для имитации задержки выполнения асинхронной операции или возвращения фиктивного значения в тестах.  loop - необязательный параметр, который задает экземпляр asyncio.AbstractEventLoop, на котором будет выполнена корутина. Если этот параметр не задан, то asyncio.sleep() использует текущий event loop. |
| пример использования asyncio.sleep(): import asyncio  async def my\_coroutine():  print("Запуск my\_coroutine")  await asyncio.sleep(1.0)  print("Запуск my\_coroutine после sleep")  asyncio.run(my\_coroutine())  Вывод:  Запуск my\_coroutine  Запуск my\_coroutine после sleep |
| пример использования asyncio.sleep(): import asyncio  async def task(name, delay):  print(f"Запуск задачи {name}")  await asyncio.sleep(delay)  print(f"Завершение задачи {name}")  async def main():  tasks = [  asyncio.create\_task(task("-Задача 1-", 1)),  asyncio.create\_task(task("-Задача 2-", 0.5)),  asyncio.create\_task(task("-Задача 3-", 2))  ]   await asyncio.gather(\*tasks)  asyncio.run(main())  Запуск задачи -Задача 1-  Запуск задачи -Задача 2-  Запуск задачи -Задача 3-  Завершение задачи -Задача 2-  Завершение задачи -Задача 1-  Завершение задачи -Задача 3- |
| asyncio.sleep() может быть полезен в следующих ситуациях:   * **Тестирование асинхронного кода**: asyncio.sleep() может быть использована для имитации асинхронных операций, которые занимают некоторое время для выполнения. Это может быть полезно при тестировании вашего асинхронного кода, чтобы убедиться, что он правильно обрабатывает ожидание. * **Ограничение скорости запросов**: если вы делаете много асинхронных запросов к внешнему API или другому ресурсу, который имеет ограничения на скорость запросов, вы можете использовать asyncio.sleep() для введения задержек между запросами, чтобы не превысить эти ограничения. * **Избежание загруженности CPU**: в некоторых случаях, когда вы выполняете много асинхронных операций, CPU может быть перегружен. Введение небольших задержек с помощью asyncio.sleep() может помочь снизить нагрузку на CPU. * **Симуляция задержек в реальном времени**: если вы создаете систему, которая должна имитировать реальные задержки (например, в игре или в системе моделирования), asyncio.sleep() может быть использована для введения этих задержек. * **Планирование задач**: если вам нужно запустить определенную задачу через определенный промежуток времени, вы можете использовать asyncio.sleep() для введения этой задержки. * **Ожидание внешних ресурсов**: если ваш код зависит от внешнего ресурса, который может быть недоступен или занят (например, файл или сетевое соединение), вы можете использовать asyncio.sleep() для введения задержки перед повторной попыткой доступа к ресурсу.    asyncio.sleep() не блокирует поток выполнения программы, а только приостанавливает выполнение текущей корутины. Это означает, что другие корутины могут быть запущены и выполнены в то время, когда текущая корутина находится в состоянии ожидания. |
| asyncio.create\_task() asyncio.**create\_task**(coro, \*, name=None, context=None)  - это метод, который позволяет создавать задачи (**tasks**) из корутин (coroutines). Задачи являются обертками для корутин и позволяют управлять жизненным циклом корутины |
| asyncio.create\_task() была добавлена в Python 3.7.  Для более ранних версий Python (3.5 и ниже) вы можете использовать  asyncio.ensure\_future(), который предлагает похожую функциональность. |
| При создании задачи в цикле событий с помощью метода asyncio.create\_task(my\_coroutine())  корутина my\_coroutine() будет выполнена немедленно, код ниже наглядно это демонстрирует.  import asyncio  async def my\_coroutine():  print('Корутина my\_coroutine запустилась сразу после создания задачи')  async def main():  asyncio.create\_task(my\_coroutine())  asyncio.run(main())  >>> Корутина my\_coroutine запустилась сразу после создания задачи |
| Одна из потенциальных проблем, с которой вы можете столкнуться – это зависание задачи. Если одна из задач зависает, то все остальные задачи также могут зависнуть. Чтобы избежать этого, вы можете установить тайм-аут для задачи с помощью метода asyncio.wait\_for(), о котором мы будем говорить в следующих уроках. |
| несколько важных моментов, при работе с asyncio.create\_task().  Использование с asyncio.run(): asyncio.create\_task() должна использоваться в контексте асинхронной функции, которая затем запускается с помощью asyncio.run(). Это связано с тем, что asyncio.run() создает новый цикл событий и закрывает его по завершении. Если вы создадите задачу вне этого контекста, она может быть отменена, когда цикл событий закроется.  Ожидание завершения задачи: важно помнить, что asyncio.create\_task() немедленно возвращает управление, не дожидаясь завершения задачи. Это означает, что если вы не дождетесь завершения задачи с помощью await, задача может быть отменена, когда цикл событий закроется.  Обработка ошибок: если в корутине, которую вы запускаете как задачу, происходит исключение, оно не будет немедленно выброшено. Вместо этого оно будет сохранено и выброшено, когда вы попытаетесь получить результат задачи с помощью await. Это означает, что вы должны быть готовы обрабатывать исключения при ожидании завершения задачи.  Отмена задач: задачи, созданные с помощью asyncio.create\_task(), могут быть отменены. Это можно сделать с помощью метода cancel() объекта задачи. Однако отмена задачи не гарантирует немедленного прекращения ее выполнения, она лишь запускает процесс отмены.  Планирование задач: asyncio.create\_task() планирует выполнение корутины в цикле событий. Это означает, что если у вас есть несколько задач, они будут выполняться в том порядке, в котором они были запланированы, но их фактическое выполнение может переключаться в зависимости от того, какие задачи в данный момент ожидают IO или другие операции. |
| Multiple tasks  Множественное создание task в asyncio может быть достигнуто с помощью цикла. Например, если вы хотите создать 5 задач, вы можете использовать цикл for  import asyncio  async def my\_task(num\_task):  print(f"Task {num\_task} started ")  await asyncio.sleep(3)  print(f"Task {num\_task} finished")  async def main():  tasks = []  for i in range(5):  task = asyncio.create\_task(my\_task(i))  tasks.append(task)  await asyncio.gather(\*tasks)  asyncio.run(main()) |
| Отладки или мониторинг Пример использования create\_task() может помочь при мониторинге состояния задачи. Например, вы можете использовать метод done() класса Task, чтобы определить, завершилась задача или нет. Вы можете использовать эту информацию для мониторинга или обработки ошибок. |
| Предотвращаем "зависание" корутин asyncio.wait\_for() asyncio.wait\_for(**coro**, **timeout**) - это функция, которая позволяет вам устанавливать ограничения по времени на выполнение асинхронных операций. Это может быть особенно полезно в ситуациях, когда вы не хотите, чтобы ваша программа "зависла", ожидая завершения операции, которая может занять слишком много времени.  Вы можете установить максимальное время ожидания для выполнения конкретной задачи и обработать ситуации, когда она не завершается в установленное время.   * coro - корутина, выполнение которой необходимо ожидать. * timeout - это время в секундах, в течение которого wait\_for будет ожидать выполнения coro. Если coro не будет завершена в указанный период времени, то выбрасывается исключение TimeoutError.   **Возвращаемое значение**: asyncio.wait\_for() возвращает то же значение, что и завершенная задача.  **Использование с корутинами и Future**: asyncio.wait\_for() может быть использован как с корутинами, так и с объектами Future. Это делает его универсальным инструментом для работы с асинхронными операциями в Python.  **Не блокирует цикл событий**: Поскольку asyncio.wait\_for() является асинхронной функцией, она не блокирует цикл событий. Это означает, что другие задачи могут продолжать выполняться, пока вы ждете завершения вашей задачи. |
| В каких случаях это применяется? В каких случаях это применяется?  asyncio.wait\_for() часто используется в ситуациях, когда вы работаете с операциями, которые могут потенциально занимать много времени, такими как сетевые запросы, чтение или запись в базу данных. Он также может быть полезен, когда вы хотите ограничить время выполнения определенной задачи в вашем приложении. |
| Пример использования: В этом примере корутина my\_coro() выполняется 5 секунд, но wait\_for ждет только 3 секунды, поэтому выбрасывается исключение TimeoutError  import asyncio  async def my\_coro():  await asyncio.sleep(5)  return 'Привет мир!'  async def main():  await asyncio.wait\_for(my\_coro(), timeout=3)  asyncio.run(main())  # output: # raise exceptions.TimeoutError() from exc # asyncio.exceptions.TimeoutError  **asyncio.wait\_for() полезен в следующих случаях:**   * Ограничение времени ожидания выполнения корутины. * Избежание зацикливания или бесконечных ожиданий. * Управление тайм-аутом при работе с API или другими удаленными ресурсами. * Управление временем ожидания выполнения длительных операций, таких как загрузка данных или обработка больших объемов данных. * Управление тайм-аутом при работе с устройствами ввода/вывода. * Управление временем ожидания выполнения длительных операций с базами данных или файлами. * Управление временем ожидания выполнения длительных задач, таких как обработка изображений или видео. |
| Поведение asyncio.wait\_for(coro): **Запуск задачи:** когда вы вызываете asyncio.wait\_for(), она начинает выполнение переданной coro или Future.  **Ожидание завершения: asyncio.wait\_for()** ожидает завершения задачи в течение указанного таймаута. В это время другие асинхронные задачи могут продолжать выполняться, поскольку asyncio.wait\_for() не блокирует цикл событий.  **Проверка таймаута:** Если задача завершается до истечения таймаута, asyncio.wait\_for() возвращает результат задачи. Если задача не завершается в течение таймаута, asyncio.wait\_for() автоматически отменяет задачу и генерирует исключение TimeoutError.  **Обработка исключений:** Если во время выполнения задачи возникает исключение, asyncio.wait\_for() пробросит его дальше. Это означает, что вы должны быть готовы обрабатывать исключения, которые могут возникнуть во время выполнения вашей задачи. |
| Example import asyncio  async def my\_coro():  print("start 5-sec task")  await asyncio.sleep(5)  print("end 5-sec task")  async def main():  task = asyncio.create\_task(my\_coro())  try:  await asyncio.wait\_for(task, timeout=3)  except asyncio.TimeoutError:  print("Задача не была завершена в установленное время")  asyncio.run(main()) |
| Если вы хотите предотвратить отмену задачи после истечения таймаута, вы можете использовать asyncio.shield(). import asyncio  async def long\_coro():  print("starting long coro")  await asyncio.sleep(7)  print("ending long coro")  return "Long-run coroutine finished work"  async def main():  task1 = asyncio.create\_task(long\_coro())  try:  await asyncio.wait\_for(asyncio.shield(task1), timeout=5)  except asyncio.TimeoutError:  print('timeout exceed')   result = await task1  print(f'{result=}')  asyncio.run(main())  printout:  starting long coro timeout exceed ending long coro result='Long-run coroutine finished work' |
| Самая распространённая ошибка при использовании asyncio.wait\_for(): Если корутина, возвращаемая функцией asyncio.wait\_for(), никогда не ожидается, т.е. не используется ключевое слово await, возникает ошибка времени выполнения. Это является распространенной ошибкой при использовании функции wait\_for().  async def main():  task = task\_coro(1)  **asyncio.wait\_for(task, timeout=1)** # Здесь происходит ошибка, потому что мы не используем await  asyncio.run(main())  Вывод с ошибкой:  ...  ...  RuntimeWarning: Enable tracemalloc to get the object allocation traceback  Правильно:  **await** asyncio.wait\_for(task, timeout=1) |
| Example : waiting result before and after timeout import asyncio  async def long\_coro():  print("starting long coro")  await asyncio.sleep(7)  print("ending long coro")  return "Long-run coroutine finished work"  async def main():  task1 = asyncio.create\_task(long\_coro())  try:  result = await asyncio.wait\_for(asyncio.shield(task1), timeout=10)  print(f'{result}') # result before  except asyncio.TimeoutError:  print('timeout exceed')  result = None  if not result:  result = await task1 ) # result after  print(f'{result=}')  asyncio.run(main())  # printout: # # starting long coro # timeout exceed # ending long coro # result='Long-run coroutine finished work' # # Process finished with exit code 0 |
| asyncio.gather() asyncio.**gather**(\*aws, loop=None, return\_exceptions=False) - позволяет группировать несколько awaitable-объектов, таких как корутины и задачи, для их одновременного выполнения.  Функция asyncio.gather() возвращает результаты ожидаемых объектов в виде списка в том же порядке, в котором вы передаете ожидаемые объекты в функцию.  Параметры:   * aws - это последовательность awaitable-объектов. Если любой объект в aws является корутиной, функция asyncio.gather() автоматически запланирует его как задачу. * loop - устарел начиная с Python 3.8 и удален в Python 3.10. Этот аргумент использовался для указания конкретного цикла событий, который должен быть использован для выполнения задач. С Python 3.7 и выше рекомендуется использовать asyncio.run(), который автоматически создает и управляет циклом событий. * return\_exceptions=**False** - (дефолтное значение) Если любая из асинхронных операций завершается с исключением, asyncio.gather немедленно прекращает выполнение и поднимает первое пойманное исключение. Остальные асинхронные операции продолжают свое выполнение, но их результаты (или исключения) игнорируются.  Такое поведение предпочтительно, когда важно, чтобы все операции были успешно выполнены и любое исключение является критическим сбоем, который требует немедленного внимания.   Когда return\_exceptions установлен в **True**: asyncio.gather продолжает выполнение, даже если одна или несколько асинхронных операций завершаются с исключением. Вместо поднятия исключений они включаются в результирующий список вместе с успешно завершенными операциями. |
| Example import asyncio  async def call\_api(message : str, result : int, delay : int =2 ):  print(f'{message} started')  await asyncio.sleep(delay)  print(f'{message} ended')  return result   async def main():  a, b = await asyncio.gather(call\_api('coro1 runs', 5, delay = 3 ),  call\_api('coro2 runs', 3, delay = 2,)  )  print(f'{a=} \n {b=}')   asyncio.run(main())  output:  coro1 runs started  coro2 runs started  coro2 runs ended  coro1 runs ended  a=5  b=3  Process finished with exit code 0 |
| Особенности asyncio.gather()  * **Одновременное выполнение**: asyncio.gather() позволяет выполнять несколько корутин или футур одновременно, что ускоряет выполнение программы и повышает ее производительность. * **Управление исключениями**: asyncio.gather() позволяет управлять исключениями, возникающими в корутинах. С помощью параметра return\_exceptions=**True** мы можем выбрать, нужно ли возвращать исключения в результате работы функции. * **Простота использования**: asyncio.gather() очень проста в использовании и позволяет выполнять несколько корутин или футур одновременно с помощью одной функции. * **Блокировка цикла**: asyncio.gather() блокирует цикл asyncio, пока все корутины или футуры не будут выполнены. Это означает, что цикл не будет выполнять другие задачи, пока все корутины или футуры, переданные в функцию asyncio.gather(), не будут выполнены. **НО** эта "блокировка" не означает традиционную блокировку, как в синхронном подходе. Вместо этого это означает, что цикл событий будет продолжать переключаться между доступными задачами, пока все корутины или футуры в gather() не будут завершены. * **Ограничение на количество корутин**: asyncio.gather() имеет ограничение на количество корутин или футур, которые можно выполнять одновременно. Это ограничение зависит от реализации и платформы, поэтому следует учитывать это при проектировании программы. В некоторых случаях может быть ограничено максимальное количество одновременных корутин, а в других случаях может быть ограничено максимальное количество ресурсов, используемых для выполнения корутин. В случае, если ограничение на количество корутин слишком низкое, может потребоваться использование другого метода, такого как asyncio.wait(), который позволяет выполнять корутины по очереди, или разбиение корутин на группы и выполнение их по очереди с помощью нескольких вызовов asyncio.gather(). |
| Example asyncio.gather(\*list) import asyncio  async def my\_coro(num):   print(f"Корутина {num} началась")   await asyncio.sleep(num)   print(f"Корутина {num} закончилась")   return num   async def main():   coros = [my\_coro(i) for i in range(1, 6)]  results = await asyncio.gather(\*coros)   print(results)   asyncio.run(main()) |
| Ожидание группы корутин asyncio.wait() asyncio.wait(aws, loop=None, timeout=None, return\_when=asyncio.ALL\_COMPLETED) - этот метод используется для одновременного запуска awaitable-объектов (Future и Task) из переданного множества (aws не должно быть пустым) и блокировки выполнения программы до выполнения условия, указанного в аргументе return\_when=.  asyncio.wait() возвращает два  множества (set) Task/Future: (done, pending).   1. **Выполненные (done):** Этот набор содержит все задачи или фьючерсы, которые были завершены на момент достижения условия, указанного в return\_when. В данном случае, при использовании asyncio.ALL\_COMPLETED, этот набор будет содержать задачи или фьючерсы, которые завершили свое выполнение, независимо от того, успешно они завершились или завершились с ошибкой. 2. **Ожидающие (pending):** Этот набор включает задачи или фьючерсы, которые еще не завершились на момент достижения условия в return\_when. При использовании asyncio.ALL\_COMPLETED этот набор, как правило, будет пустым, поскольку asyncio.wait возвращает управление, когда все переданные операции завершены. |
| константы в return\_when: Вот основные константы, которые можно использовать в return\_when:  **asyncio.FIRST\_COMPLETED** первая из переданных задач или фьючерсов завершается или отменяется.  **asyncio.FIRST\_EXCEPTION**  если любая из переданных задач или фьючерсов завершается с исключением.  **asyncio.ALL\_COMPLETED** когда все переданные задачи и фьючерсы завершены, независимо от того, были ли они успешно выполнены, завершились с исключением или были отменены. |
| Examples done, pending = await asyncio.wait(tasks,return\_when=asyncio.FIRST\_COMPLETED)  done, pending = await asyncio.wait(tasks, return\_when=asyncio.ALL\_COMPLETED)  done, pending = await asyncio.wait(tasks,return\_when=asyncio.FIRST\_EXCEPTION)  loop.run\_until\_complete(asyncio.wait(tasks)) loop.close() |
| Особенности работы с asyncio.wait()  1. **Работа с асинхронными функциями**: asyncio.wait() позволяет вам управлять несколькими асинхронными функциями и ждать их завершения. 2. **Возврат результатов**: asyncio.wait() возвращает два набора: множество завершенных задач и множество незавершенных задач, что позволяет вам получить детальную информацию о состоянии их выполнения. 3. **Контроль порядка выполнения**: с помощью asyncio.wait() вы можете указать, когда вы хотите получить результаты (return\_when=), после завершения первой задачи или после возникновения первого исключения. 4. **Обработка исключений**: если в одной из задач возникло исключение, то вы можете получить его с помощью метода task.exception(), о котором мы будем говорить далее. 5. **Оптимизация производительности**: asyncio.wait() позволяет оптимизировать производительность вашего кода, так как вы можете ждать завершения нескольких задач одновременно, а не по одной. 6. **Легкость использования**: asyncio.wait() имеет простой интерфейс, который легко использовать, что делает его идеальным для управления асинхронными задачами. |
| В каких случаях может понадобиться asyncio.wait()?  1. **Параллельные запросы к API**: Если ваше приложение должно отправить несколько запросов к API и обработать их результаты, вы можете использовать asyncio.wait() для одновременного запуска этих запросов и обработки результатов по мере их получения. 2. **Веб-скрапинг**: Если вы пишете веб-скрапер, который должен загрузить и обработать данные с нескольких веб-страниц, вы можете использовать asyncio.wait() для одновременной загрузки этих страниц и обработки данных по мере их получения. 3. **Асинхронные операции ввода-вывода**: Если ваше приложение выполняет множество операций ввода-вывода (например, чтение и запись файлов или общение с базой данных), которые могут быть выполнены асинхронно, вы можете использовать asyncio.wait() для управления этими операциями и улучшения производительности вашего приложения. 4. **Операции с таймаутом**: Если вам нужно ограничить время ожидания выполнения задач, вы можете использовать параметр timeout в asyncio.wait(). Это может быть полезно, например, при отправке запросов к API, которые могут занимать много времени. 5. **Обработка исключений**: Если ваши асинхронные задачи могут вызвать исключения, вы можете использовать asyncio.wait() для управления этими исключениями и решения, что делать с задачами, которые вызвали исключения. |
| Запуск и работа с футурами asyncio.ensure\_future() asyncio.ensure\_future(coro\_or\_future, \*, loop=None) -  это функция, которая используется для запуска корутины в асинхронном режиме. Она принимает один обязательный аргумент - объект корутины, которую нужно запустить. Возвращает объект Task, который может быть использован для отслеживания состояния выполнения задачи. Вы также можете ожидать завершения задачи с помощью await в любой момент. Кроме того, функция принимает несколько необязательных аргументов:   * coro\_or\_future - асинхронная корутина или объект типа Future или Task. * *loop* - *в Python 3.10, была признана устаревшей. Пользоваться ей не стоит.* |
| Если есть корутина и необходимо запланировать ее выполнение, правильный API для использования - asyncio.create\_task(). Единственный раз, когда вы должны вызывать [asyncio.ensure\_future()](https://docs-python.ru/standart-library/modul-asyncio-python/funktsii-obekt-future-modulja-asyncio/#asyncio.ensure_future) - это когда разработчик предоставляет пользователю API, который принимает либо корутину, либо Future, с которым нужно что-то сделать.  **Важно**: предпочтительным способом создания новых задач является функция asyncio.create\_task(). |
| import asyncio  async def my\_coroutine(i):  # Начало корутины с номером i  print(f"Начинается корутина {i}")  # Ожидание 1 секунды  await asyncio.sleep(1)  # Завершение корутины с номером i  print(f"Завершается корутина {i}")  async def main():  tasks = []  # Создание 5 корутин  for i in range(5):  # Запуск корутины и добавление в список задач  task = asyncio.ensure\_future(my\_coroutine(i))  tasks.append(task)  # Ожидание завершения всех корутин  await asyncio.gather(\*tasks)  # Запуск главной функции asyncio.run(main())  # Вывод: # # Начинается корутина 0 # Начинается корутина 1 # Начинается корутина 2 # Начинается корутина 3 # Начинается корутина 4 # Завершается корутина 0 # Завершается корутина 2 # Завершается корутина 4 # Завершается корутина 1 # Завершается корутина 3 |
| контроль над выполнением корутин и их синхронизацией example2 async def main():  # Создание и запуск корутины 1  task1 = asyncio.ensure\_future(coroutine1())  # Создание и запуск корутины 2  task2 = asyncio.ensure\_future(coroutine2())   # Ожидание завершения обеих корутин  await asyncio.gather(task1, task2)  # Запуск главной функции asyncio.run(main())  # output: # Начинается корутина 1 # Начинается корутина 2 # Завершается корутина 2 # Завершается корутина 1 |
| Example3 import asyncio  async def download\_file(url):  print(f"Начинается загрузка {url}")  await asyncio.sleep(3) # Имитация загрузки файла  print(f"Загрузка завершена {url}")  async def process\_file(filename):  print(f"Начинается обработка {filename}")  await asyncio.sleep(2) # Имитация обработки файла  print(f"Обработка завершена {filename}")  async def download\_and\_process(url, filename):  task1 = asyncio.ensure\_future(download\_file(url))  task2 = asyncio.ensure\_future(process\_file(filename))  await asyncio.gather(task1, task2)  async def main():  # Списки URL и имен файлов  urls = ["http://example.com/file1", "http://example.com/file2", "http://example.com/file3"]  filenames = ["file1.txt", "file2.txt", "file3.txt"]   tasks = [asyncio.ensure\_future(download\_and\_process(url, filename)) for url, filename in zip(urls, filenames)]   await asyncio.gather(\*tasks)  # Запуск главной функции asyncio.run(main()) |
| # output: # Начинается загрузка http://example.com/file1 # Начинается обработка file1.txt # Начинается загрузка http://example.com/file2 # Начинается обработка file2.txt # Начинается загрузка http://example.com/file3 # Начинается обработка file3.txt # Обработка завершена file1.txt # Обработка завершена file3.txt # Обработка завершена file2.txt # Загрузка завершена http://example.com/file2 # Загрузка завершена http://example.com/file1 # Загрузка завершена http://example.com/file3 |
| Получаем ссылку на текущий цикл событий asyncio.get\_event\_loop() asyncio.get\_event\_loop() - метод, который позволяет получить ссылку на текущий цикл событий. Это может быть полезно, если вы хотите запланировать новую асинхронную задачу в текущем цикле, либо если вы хотите получить статус текущего цикла. |
| Используя asyncio.get\_event\_loop(), мы можем получить доступ к текущему циклу событий, чтобы управлять его работой, например, запустить задачу в цикле с помощью \*run\_until\_complete() или остановить цикл с помощью stop().  *.run\_until\_complete()* устарел, его использование не рекомендуется.  рекомендуется создавать цикл событий один раз в начале программы и использовать его во всех остальных частях программы. |
| Цикл событий работает следующим образом: он ожидает входящие события, после чего выполняет задачи, связанные с этими событиями. Это позволяет асинхронным программам эффективно использовать ресурсы системы, поскольку они могут выполнять несколько задач одновременно вместо того, чтобы ждать завершения каждой задачи перед началом следующей.  Для того чтобы получить доступ к текущему циклу событий, необходимо вызвать метод asyncio.get\_event\_loop(). В большинстве случаев этот метод возвращает текущий цикл событий, если он уже был создан, или создает новый цикл событий, если это первый вызов. |
| новый цикл событий asyncio.new\_event\_loop() asyncio.new\_event\_loop() - метод используется для создания нового цикла событий. Он возвращает объект типа asyncio.BaseEventLoop, который может быть использован для запуска и управления асинхронными задачами.  Одним из преимуществ использования asyncio.new\_event\_loop() является то, что вы можете создать несколько разных циклов событий в рамках одного приложения и использовать их для управления разными асинхронными задачами. Это может быть полезно, если у вас есть несколько независимых потоков выполнения, которые необходимо контролировать отдельно. |
| import asyncio  async def main():  print("Hello, World!")   loop = asyncio.new\_event\_loop() asyncio.set\_event\_loop(loop)  loop.run\_until\_complete(main()) loop.close()   # Вывод: # # Hello, World! |
| import asyncio import threading  async def task1(id, loop):  print(f"Task {id} starting, loop = {loop}")  print(f"Task {id} starting, loop = {asyncio.get\_event\_loop()}")  await asyncio.sleep(2)  print(f"Task {id} completed")  async def task2(id, loop):  print(f"Task {id} starting, loop = {loop}")  print(f"Task {id} starting, loop = {asyncio.get\_event\_loop()}")  await asyncio.sleep(3)  print(f"Task {id} completed")  def start\_loop(loop, coroutine):  asyncio.set\_event\_loop(loop)  loop.run\_until\_complete(coroutine)  async def main():  loop1 = asyncio.new\_event\_loop()  loop2 = asyncio.new\_event\_loop()   coroutine1 = task1(1, loop1)  coroutine2 = task2(2, loop2)   thread1 = threading.Thread(target=start\_loop, args=(loop1, coroutine1,))  thread2 = threading.Thread(target=start\_loop, args=(loop2, coroutine2,))   thread1.start()  thread2.start()   thread1.join()  thread2.join()  asyncio.run(main()) |
| Устанавливаем цикл событий по умолчанию asyncio.set\_event\_loop(loop) .set\_event\_loop(loop) - позволяет установить экземпляр цикла событий loop в качестве текущего цикла событий для текущего потока, а в противном случае asyncio использует цикл событий, созданный по умолчанию. Это полезно в случаях, когда вам нужно использовать разные циклы событий в разных потоках.  **Предостережения и рекомендации**   * Нельзя запустить два цикла событий в одном потоке одновременно. * Использование нескольких циклов событий в разных потоках может усложнить код. * Необходимо тщательно рассмотреть, когда стоит использовать один или несколько циклов событий, в зависимости от специфики задачи. * Критически важно применять разные циклы событий для различных потоков или процессов. * Помимо технических соображений, следует учитывать возможные трудности и риски. Принимая решение о применении множественных циклов событий, важно провести глубокий анализ и убедиться, что это действительно оправдано для вашего конкретного случая. |
| # bad example : several event\_loops in single thread :  import asyncio   async def my\_task(idx):  print(f"Задача {idx} выполняется")  print(f'Идентификатор цикла задачи: {id(asyncio.get\_running\_loop())}')  await asyncio.sleep(1)  print(f"Задача {idx} завершена")   async def main():  print(f'Исходный идентификатор цикла: {id(asyncio.get\_running\_loop())}')   # Создаем новый цикл выполнения задач  loop = asyncio.new\_event\_loop()   # Выводим идентификатор нового цикла выполнения  print(f'Идентификатор нового цикла: {id(loop)}')   # Устанавливаем новый цикл выполнения как текущий  asyncio.set\_event\_loop(loop)   # Выводим идентификатор текущего цикла выполнения после установки нового  print(f'Текущий идентификатор цикла: {id(asyncio.get\_running\_loop())}')   # Создаем список задач, каждая из которых представляет функцию my\_task с разными индексами  tasks = [asyncio.ensure\_future(my\_task(i)) for i in range(5)]   # Ожидаем завершения выполнения всех задач  await asyncio.gather(\*tasks)   loop.close()   asyncio.run(main()) |
| output: ( in single thread – only one event\_loop)  Исходный идентификатор цикла: 1791636799440  Идентификатор нового цикла: 1791667343696  Текущий идентификатор цикла: 1791636799440  Задача 0 выполняется  Идентификатор цикла задачи: 1791636799440  Задача 1 выполняется  Идентификатор цикла задачи: 1791636799440  Задача 2 выполняется  Идентификатор цикла задачи: 1791636799440  Задача 3 выполняется  Идентификатор цикла задачи: 1791636799440  Задача 4 выполняется  Идентификатор цикла задачи: 1791636799440  Задача 0 завершена  Задача 1 завершена  Задача 2 завершена  Задача 3 завершена  Задача 4 завершена |
| Example 2 import asyncio import threading   async def task\_func(id1: int) -> None:  print(f'task {id1} starting at loop {id(asyncio.get\_running\_loop())}')  await asyncio.sleep(id1)  print(f'task {id1} ended at loop {id(asyncio.get\_running\_loop())}')  def thr\_task(loop, coro):  asyncio.set\_event\_loop(loop)  loop.run\_until\_complete(coro)  async def main():  print(f'current loop : {id(asyncio.get\_running\_loop())}')  loop1 = asyncio.new\_event\_loop()  print(f'loop 1 creaated with id : {id(loop1)}')  loop2 = asyncio.new\_event\_loop()  print(f'loop 2 creaated with id : {id(loop2)}')   task1 = task\_func(1)  task2 = task\_func(5)   thr1 = threading.Thread(target=thr\_task, args=(loop1, task1))  thr2 = threading.Thread(target=thr\_task, args=(loop2, task2))   thr1.start()  thr2.start()  thr1.join()  thr2.join()   asyncio.run(main()) |
| Output ( + 2 threads, 3 event\_loop ) Идентификатор оригинального цикла: 2386607079184  Идентификатор первого нового цикла: 2386637576640  Идентификатор второго нового цикла: 2386637577024  Задача 1 начинается, цикл = <ProactorEventLoop running=True closed=False debug=False>  Идентификатор цикла в задаче 1: 2386637576640  Задача 2 начинается, цикл = <ProactorEventLoop running=True closed=False debug=False>  Идентификатор цикла в задаче 2: 2386637577024  Задача 1 завершена  Задача 2 завершена |
| доступ к текущему циклу событий get\_running\_loop() Эта функция возвращает экземпляр класса asyncio.BaseEventLoop, который представляет текущий цикл событий. Это позволяет вам добавлять задачи, изменять параметры цикла событий и многое другое. Это особенно полезно, когда вы хотите выполнить асинхронные задачи **внутри других асинхронных функций**. |
| asyncio.get\_running\_loop()  работает только внутри цикла событий. Если вы вызываете эту функцию вне цикла событий, она вызовет исключение RuntimeError. |
| <https://stackoverflow.com/questions/65206110/when-to-use-asyncio-get-running-loop-vs-asyncio-get-event-loop>  get\_event\_loop has more complex behaviour, thus get\_running\_loop can be used widely in the applications  asyncio.get\_event\_loop() deprecated since version 3.12:  <https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.get_event_loop> |
| Example  import asyncio  async def my\_task():  print("Running my task")  async def main():  loop = asyncio.get\_running\_loop()  loop.create\_task(my\_task())  print(f"{id(loop)}")  await asyncio.sleep(1)  asyncio.run(main())  output:  1999724150736  Running my task |
| to\_thread() Запуск блокирующих функций в отдельном потоке асинхронный запуск синхронной функции в отдельном потоке.  **asyncio.to\_thread(func, /, \*args, \*\*kwargs)**—метод возвращает корутину, которую можно ожидать, чтобы получить конечный результат работы func. Все \*args и \*\*kwargs, переданные для этой функции, напрямую передаются в func. Также передается текущий contextvars.Context, что позволяет обращаться к переменным контекста из потока цикла событий в отдельном потоке.  Этот метод в первую очередь предназначен для выполнения функций/методов, связанных с вводом-выводом, которые в противном случае блокировали бы цикл событий, если бы они выполнялись в основном потоке. Это полезная функция, которую можно использовать, когда у нас есть асинхронная программа, которой необходимо выполнять как неблокирующий ввод-вывод (например, с сокетами), так и блокирующий ввод-вывод (например, с файлами). |
| Проблема с asyncio заключается в том, что если в цикле событий будет выполнен блокирующий вызов, то это приведет к приостановке приложения на время выполнения такого вызова. |
| Пример работы блокирующей функции в основном потоке import asyncio import time   # Блокирующая функция def blocking\_fn():  print(f"Старт blocking\_fn(): {time.strftime('%X')}")  # Обратите внимание, что time.sleep() может быть заменен любой блокирующей  # IO-bound операцией, например операцией с файлами.  time.sleep(1) # Имитация выполнения длительной операции.  print(f"Завершение blocking\_fn() {time.strftime('%X')}")   # Вызов блокирующей функции async def fn():  return blocking\_fn()   # Функция асинхронного sleep() async def sleep\_fn():  print(f"Старт sleep\_fn(): {time.strftime('%X')}")  await asyncio.sleep(1)  print(f"Завершение sleep\_fn() {time.strftime('%X')}")   async def main():  print(f"Старт main в {time.strftime('%X')}")  await asyncio.gather(fn(), sleep\_fn(), sleep\_fn())  print(f"Завершение main в {time.strftime('%X')}")   start = time.time() asyncio.run(main()) print(f'Время выполнения программы: {(time.time() - start)}') |
| Пример работы блокирующей функции в отдельном потоке import asyncio import threading import time   # Блокирующая функция def blocking\_fn():  # Печать сообщения о старте и номере используемого потока.  print(f"Старт blocking\_fn() в потоке c id {threading.current\_thread().ident} в {time.strftime('%X')}")  # Обратите внимание, что time.sleep() может быть заменен любой блокирующей  # IO-bound операцией, например операцией с файлами.  time.sleep(1) # Имитация выполнения длительной операции.  print(f"Завершение blocking\_fn() в {time.strftime('%X')}")    # Функция асинхронного sleep() async def sleep\_fn():  # Печать сообщения о старте и номере используемого потока.  print(f"Старт sleep\_fn() в потоке c id {threading.current\_thread().ident} в {time.strftime('%X')}")  await asyncio.sleep(1)  print(f"Завершение sleep\_fn() в {time.strftime('%X')}")   async def main():  # Печать сообщения о старте и номере используемого потока  print(f"Старт main в потоке c id {threading.current\_thread().ident} в {time.strftime('%X')}")  # Создание корутины, для запуска в независимом потоке  coro = asyncio.to\_thread(blocking\_fn)  # Проверка типа объекта для coro  print(f'Тип объекта coro: {type(coro)}')  # Асинхронный запуск задач.  await asyncio.gather(coro, sleep\_fn(), sleep\_fn())  print(f"Завершение main в {time.strftime('%X')}")   start = time.time() asyncio.run(main()) print(f'Время выполнения программы: {(time.time() - start)}')  # output: # Старт main в потоке c id 13792 в 12:18:05 # Тип объекта coro: <class 'coroutine'> # Старт blocking\_fn() в потоке c id 12040 в 12:18:05 # Старт sleep\_fn() в потоке c id 13792 в 12:18:05 # Старт sleep\_fn() в потоке c id 13792 в 12:18:05 # Завершение blocking\_fn() в 12:18:06 # Завершение sleep\_fn() в 12:18:06 # Завершение sleep\_fn() в 12:18:06 # Завершение main в 12:18:06 # Время выполнения программы: 1.0116322040557861 |
| Asyncio.to\_thread with args import asyncio import threading import time   # Блокирующая функция def blocking\_fn(arg1, arg2):  print(f"Старт blocking\_fn() в потоке c id {threading.current\_thread().ident} в {time.strftime('%X')}")  time.sleep(arg1)  print(f"Завершение blocking\_fn() в {time.strftime('%X')}")  return f'В blocking\_fn() были переданы два аргмента arg1: {arg1} и arg2: {arg2}'   # Функция асинхронного sleep() async def sleep\_fn():  print(f"Старт sleep\_fn() в потоке c id {threading.current\_thread().ident} в {time.strftime('%X')}")  await asyncio.sleep(1)  print(f"Завершение sleep\_fn() в {time.strftime('%X')}")   async def main():  print(f"Старт main в потоке c id {threading.current\_thread().ident} в {time.strftime('%X')}")  coro = asyncio.to\_thread(blocking\_fn, 1, 'Привет')  print(f'Тип объекта coro: {type(coro)}')   result = await asyncio.gather(coro, sleep\_fn(), sleep\_fn())  print(result[0])  print(f"Завершение main в {time.strftime('%X')}")   start = time.time() asyncio.run(main()) print(f'Время выполнения программы: {(time.time() - start)}') |
| Что произойдет при попытке передачи корутины в asyncio.to\_thread()? Так как этот инструмент предназначен для работы с синхронными блокирующими функциями, то передача в неё корутины или любого другого асинхронного объекта, требующего ожидания, не соответствует её предназначению и приведет к возникновению ошибки.  import asyncio  async def соrо():  print("Вы ошиблись! Я работаю!")  await asyncio.sleep(1)  print("Моя работа завершилась!")  async def main():  # Попытка передать корутину для выполнения в отдельном потоке => error  await asyncio.to\_thread(соrо)  asyncio.run(main())  # output: Error: sending async function to to\_threading # RuntimeWarning: Enable tracemalloc to get the object allocation traceback |
| Работа с задачами TASK |
| Введение **Задача/Task** — это объект, который планирует, управляет и независимо от других задач запускает сопрограмму в цикле событий**.**Задача создается из сопрограммы. Задача оборачивает сопрограмму, планирует ее выполнение и предоставляет способы взаимодействия с ней. |
| Когда сопрограмма "обёрнута" в задачу функцией asyncio.create\_task(), сопрограмма автоматически запускается в ближайшее время.  Поскольку задача является ожидаемой/awaitable, это означает, что сопрограмма может ожидать выполнения задачи с помощью выражения ожидания await. |
| Задачи предоставляют \*дескриптор независимо от запланированных и запущенных сопрограмм в цикле событий и позволяют запрашивать, отменять задачу, а также получать результаты и исключения.  *\*Handle (*[*дескриптор*](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%80)*) - это уникальный идентификатор, который используется для доступа к ресурсу или объекту. Он может быть использован для идентификации файлов, сетевых соединений, процессов и т.д.*  Асинхронная задача - это задача, которая выполняется параллельно с другими задачами и не блокирует основной поток выполнения программы. |
| Жизненный цикл задачи: Жизненный цикл асинхронной задачи - это последовательность состояний, через которые проходит задача от момента ее создания до завершения. Рассмотрим каждый этап подробнее:   1. **Создание**: Задача создается из сопрограммы. Сопрограмма - это специальный тип функции, который может быть приостановлен и возобновлен, что позволяет выполнять асинхронные операции. 2. **Планирование**: Задача планируется для выполнения. Это означает, что она добавляется в очередь задач, которые должны быть выполнены. В этом состоянии задача еще не начала выполняться. 3. **Выполнение**: Задача начинает выполняться. Во время выполнения задачи она может быть приостановлена, например, если ей надо подождать завершения другой сопрограммы. 4. **Завершение**: Задача может завершиться двумя способами: нормально или с ошибкой. Если задача завершилась нормально, она возвращает результат. Если задача завершилась с ошибкой, она генерирует исключение. 5. **Отмена**: Задача может быть отменена в любой момент. Если задача отменена, она не может быть возобновлена. 6. **Готово**: После завершения или отмены задача переходит в состояние "Готово". В этом состоянии задача не может быть выполнена снова. |
|  |
| Основные методы управления задачами TASK task- это объект, представляющий обёртку для асинхронной операции. Она может быть создана и запущена с помощью функции asyncio.**create\_task()** и выполнена в фоновом режиме, пока другие операции выполняются. Задача может быть остановлена, приостановлена или возобновлена в любой момент времени |
| Доступные методы класса Task  [task.**done()**](https://stepik.org/lesson/933726/step/1?unit=939625) - возвращает True, если задача выполнена или отменена. Этот метод удобно использовать, чтобы проверить, закончила ли задача свое выполнение, чтобы начать следующую задачу;   [asyncio.**current\_task()**](https://stepik.org/lesson/933728/step/1?unit=939627) - возвращает объект задачи, выполняющейся в текущем потоке. Этот метод может быть полезен, если вы хотите узнать, какая задача выполняется в данный момент;   [task.**result()**](https://stepik.org/lesson/933727/step/1?unit=939626) - возвращает результат выполнения задачи, если она выполнена. Если результат задачи еще не доступен, то метод вызывает исключение InvalidStateError . Этот метод может быть полезен, если вы хотите получить результат выполнения задачи;   [asyncio.**all\_tasks()**](https://stepik.org/lesson/933729/step/1?unit=939628) - возвращает множество (set) объектов Task, которые еще не завершены и выполняются циклом событий. Этот метод может быть полезен, если вы хотите получить список всех задач, которые были созданы, но не были завершены в момент применения метода.   [task.**get\_name()**](https://stepik.org/lesson/933730/step/1?unit=939629) - возвращает имя задачи. Этот метод может быть полезен, если вы хотите узнать имя задачи;   [task.**set\_name(name)**](https://stepik.org/lesson/933730/step/1?unit=939629) - устанавливает имя задачи. Этот метод может быть полезен, если вы хотите установить имя задачи;   [task.**exception()**](https://stepik.org/lesson/933731/step/1?unit=939630) - возвращает исключение, которое было выброшено при выполнении задачи. Если задача еще не выполнена или не выбрасывала исключение, то метод возвращает None. Этот метод может быть полезен, если вы хотите получить информацию об исключении, выброшенном при выполнении задачи;   [task.**cancel()**](https://stepik.org/lesson/933732/step/1?unit=939631) - отменяет выполнение задачи. Этот метод может быть полезен, если вы хотите отменить выполнение задачи;   [task.**cancelled()**](https://stepik.org/lesson/933732/step/1?unit=939631) - возвращает True, если задача была отменена до ее выполнения. Этот метод может быть полезен, если вы хотите узнать, была ли задача отменена;   [task.**cancelling()**](https://stepik.org/lesson/933732/step/1?unit=939631) - возвращает True, если задача находится в процессе отмены. Этот метод может быть полезен, если вы хотите узнать, находится ли задача в процессе отмены;   [task.**uncancel()**](https://stepik.org/lesson/933732/step/1?unit=939631) - отмена отмены задачи. Этот метод может быть полезен, если вы хотите отменить отмену выполнения задачи;   [task.**get\_coro()**](https://stepik.org/lesson/933733/step/1?unit=939632) - возвращает корутину, связанную с задачей. Этот метод может быть полезен, если вы хотите получить объект корутины, связанный с задачей;   [task.**get\_loop()**](https://stepik.org/lesson/933733/step/1?unit=939632) - возвращает цикл событий, связанный с задачей. Этот метод может быть полезен, если вы хотите получить объект цикла событий, связанный с задачей;   [task.**get\_stack()**](https://stepik.org/lesson/933734/step/1?unit=939633) - возвращает стек выполнения задачи. Этот метод может быть полезен, если вы хотите получить информацию о стеке выполнения задачи;   [task.**print\_stack(file=None)**](https://stepik.org/lesson/933734/step/1?unit=939633) - выводит стек выполнения задачи в файл или на экран. Этот метод может быть полезен, если вы хотите вывести информацию о стеке выполнения задачи;   [task.**add\_done\_callback(fn)**](https://stepik.org/lesson/933735/step/1?unit=939634) - добавляет функцию обратного вызова, которая будет вызвана после завершения выполнения задачи. Этот метод может быть полезен, если вы хотите выполнить какой-то код после того, как задача завершится;   [task.**remove\_done\_callback(fn)**](https://stepik.org/lesson/933735/step/1?unit=939634) - удаляет функцию обратного вызова, связанную с задачей. Этот метод может быть полезен, если вы хотите удалить функцию обратного вызова, которую ранее добавили в задачу; |
| Проверка статуса задачи task.done() task.done() возвращает True, если задача завершена, и False, если нет. Она является надстройкой над функцией task.result(), которая возвращает результат задачи. Используя эти функции, мы можем проверять, завершена ли задача, и получать ее результат. |
| import asyncio  async def coro():  print('coro started')  await asyncio.sleep(2)  print('coro completed')  return f'result of coro'  async def main():  task = asyncio.create\_task(coro())  print(f'task status done = {task.done()}')  print(f'task created and set to event loop')  await task  print(f'task status done = {task.done()}')  result = task.result()  print(f'{result=}')  asyncio.run(main()) |
| Output:  task status = False  task created and set to event loop  coro started  coro completed  task status = True  result='result of coro' |
|  |
| Получение предварительного результата задачи task.result task.result() - Возвращает результат выполнения задачи.  Если задача выполнена, то возвращается результат обернутой корутины  если корутина вызвала исключение, то это исключение будет вызвано повторно).  Если задача была отменена, то метод вызывает исключение CancelledError.  Если результат задачи еще не доступен, то метод вызывает исключение InvalidStateError. |
| Example1 import asyncio  async def my\_coro() -> str :  print(f'my\_coro started')  await asyncio.sleep(4)  print(f'my\_coro completed')  return f'this is result of "my\_coro" execution'   async def main() -> None:  task = asyncio.create\_task(my\_coro())  print('task started')  await asyncio.sleep(2)  print('main still alive')  await task  print(f'result of {task.get\_name()} : {task.result()}')  asyncio.run(main()) |
| Example 2 import asyncio   async def my\_coro():  print("Корутина запустилась")  await asyncio.sleep(5)  print("Корутина завершилась")  return 42   async def main() -> None:  print('main started')  task = asyncio.create\_task(my\_coro())  await asyncio.sleep(1)  print('main continues work')  try:  await asyncio.wait\_for(task, timeout=1)  result = task.result()  except asyncio.TimeoutError :  print(' the task wasn"t completed in time')  # print(f'{task.result()=}')  asyncio.run(main()) |
| Example 3 import asyncio import random   async def my\_coro(num: int) -> str:  time\_to\_run = random.randint(1,10)  print(f' coro-{num} started, need: {time\_to\_run} sec')  await asyncio.sleep(time\_to\_run)  print(f' coro-{num} completed')  return f'coro {num} result'  async def main():  tasks = [asyncio.create\_task(my\_coro(i)) for i in range(5)]  timeout\_list = [random.randint(1,4) for \_ in range(5)]  for i, task in enumerate(tasks):  try:  await asyncio.wait\_for(task, timeout=timeout\_list[i])  result = task.result()  print(f'result of {task.get\_name()} is: {task.result()}')  except asyncio.TimeoutError:  print(f"Задача {i} не была завершена в указанное время (таймаут: {timeout\_list[i]} сек.).")  asyncio.run(main()) |
| Работа с текущей задачей asyncio.current\_task() asyncio.current\_task() - является методом, который возвращает текущий запущенный экземпляр Task  или None, если ни одна задача не запущена.  Это полезно для получения информации о текущем статусе задачи, ее приоритете, доступа к другим атрибутам или методам Task, таким как cancel(), done() или result() |
| Example import asyncio  async def my\_task(number):  current\_task = asyncio.current\_task()  print(  f"Задача {number} стартовала. Текущий объект задачи: {current\_task}")  await asyncio.sleep(1)  print(f"Задача {number} выполнена")  async def main():  task1 = asyncio.create\_task(my\_task(1))  task2 = asyncio.create\_task(my\_task(2))  print('ha-ha')  await asyncio.gather(task1, task2)   asyncio.run(main()) |
| Output: output: ha-ha Задача 1 стартовала. Текущий объект задачи: <Task pending name='Task-2' coro=<my\_task() running at D:\GB\pythonProject\asyncio\Chapter\_6 \_Task\6\_5\_current\_task\main1.py:6> cb=[gather.<locals>.\_done\_callback() at C:\Users\VMAL\AppData\Local\Programs\Python\Python310\lib\asyncio\tasks.py:720]> Задача 2 стартовала. Текущий объект задачи: <Task pending name='Task-3' coro=<my\_task() running at D:\GB\pythonProject\asyncio\Chapter\_6 \_Task\6\_5\_current\_task\main1.py:6> cb=[gather.<locals>.\_done\_callback() at C:\Users\VMAL\AppData\Local\Programs\Python\Python310\lib\asyncio\tasks.py:720]> Задача 1 выполнена Задача 2 выполнена |
| Получение всех задач в цикле событий asyncio.all\_tasks() asyncio.all\_tasks(loop=None) - возвращает **множество(<class 'set'>)** еще не завершенных объектов Task, выполняемых циклом.  Параметры:   * loop - объект цикла событий. По умолчанию используется текущий цикл событий. |
| Examples несколько примеров, когда функция asyncio.all\_tasks() может оказаться полезной:   * **Мониторинг и отладка:** В процессе разработки и отладки асинхронного кода иногда требуется получить список текущих задач, чтобы увидеть, какие из них выполняются и в каком состоянии они находятся. * **Управление задачами:** В некоторых случаях вам может потребоваться отменить все текущие задачи, например, при остановке сервера или приложения. С помощью asyncio.all\_tasks() вы можете получить список всех задач и отменить нужные, вызывая метод Task.cancel() для каждой из них. * **Ожидание завершения всех задач:** Если вам нужно дождаться завершения всех текущих задач перед тем, как продолжить выполнение кода, вы можете использовать asyncio.all\_tasks() в сочетании с asyncio.gather() для ожидания их завершения. Так как выполняемая объемлющая задача тоже будет присутствовать в списке, то необходимы специальные механизмы контроля (использование таймаута или вызов другого exception - как вариант), иначе в чистом виде await asyncio.gather(\*asyncio.all\_tasks()) никогда сам не сможет завершиться, порождая классический Deadlock. * **Ограничение количества одновременно выполняющихся задач:** В случае, если вы хотите контролировать количество одновременно выполняющихся задач, asyncio.all\_tasks() может помочь вам определить текущее количество активных задач и, соответственно, управлять их запуском. Для этой цели можно использовать асинхронные очереди, о которых речь пойдет далее в главе 8 "Работа с очередями в Asyncio". * **Реализация собственных стратегий обработки ошибок:** Иногда может быть полезно реализовать собственную логику обработки ошибок для асинхронных задач. Получив список всех задач с помощью asyncio.all\_tasks(), вы можете применить свою стратегию обработки ошибок, например, перезапускать задачи в случае ошибок или изменять приоритеты задач в зависимости от их состояния. |
| Example import asyncio  async def my\_coro(num: int) -> str:  print(f'task-{num} is running')  await asyncio.sleep(2)  return f'task-{num} result'   async def main():  tasks = [asyncio.create\_task(my\_coro(i)) for i in range(1,6)]  tasks\_list = asyncio.all\_tasks()  for el in tasks\_list:  print(el)  print(f'total tasks : {len(tasks\_list)}')  await asyncio.gather(\*tasks)   for el in tasks\_list:  print(el)  if el.done():  print(el.result())   asyncio.run(main()) |
| Установка и получение имени задачи get\_name() и set\_name() Метод set\_name(value) Устанавливает имя задачи. Аргументом value может быть любой объект, который затем преобразуется в строку. Синтаксис метода выглядит следующим образом:  asyncio.task.set\_name(value)  Метод get\_name() возвращает имя задачи. Метод не принимает аргументов и имеет следующий синтаксис:  asyncio.task.get\_name() |
| Example2 import asyncio  async def my\_coroutine():  print(f"Имя задачи: {asyncio.current\_task().get\_name()}")  async def main():  task = asyncio.create\_task(my\_coroutine(), name="my\_task")  await task  asyncio.run(main()) |
| Example 2 import asyncio  async def my\_coroutine():  print(f"Имя задачи до изменения: {asyncio.current\_task().get\_name()}")  asyncio.current\_task().set\_name("new\_name")  print(f"Имя задачи после изменения: {asyncio.current\_task().get\_name()}")  async def main():  task = asyncio.create\_task(my\_coroutine(), name="my\_task")  await task  asyncio.run(main())  # output: # Имя задачи до изменения: my\_task # Имя задачи после изменения: new\_name |
| Работа с исключениями exception() Корутины могут возбуждать исключения при выполнении операций. Для обработки исключений в корутинах в asyncio предоставляется метод exception() .  task.exception() используется для получения исключения, которое возникло в корутине. Если в корутине не было возбуждено исключение, то метод вернет None. В противном случае он вернет последнее возбужденное исключение. Метод exception() не очищает исключение из корутины, поэтому при последующих вызовах он вернет тот же объект исключения. |
| Example import asyncio   async def raise\_exception():  raise RuntimeError("--Установленное исключение--")    async def main():   task = asyncio.create\_task(raise\_exception())   await asyncio.sleep(0.1)   try:   await task   except Exception as e:   print(f"Пойманное исключение: {e}")   exception = task.exception()   if exception:   print(f"Тут можно обработать возникшее исключение: {exception}")  asyncio.run(main()) |
| Отмена задач cancel(), cancelled(), cancelling() и uncancel() Методы task.**cancel**(), task.**cancelled**(), task.**cancelling**() и task.**uncancel**() применяются в случае необходимости управления выполнением задач. Например, они могут использоваться для отмены задач, которые уже не нужны, или для управления выполнением группы задач. |
| Порядок применения методов Для управлением задачами в другой корутине можно использовать синтаксис  asyncio.current\_task(). Таким образом, не нужно передавать объект task в другую корутину для обработки. Это будет работать, т.к. все объекты задач передаются в цикл событий и обрабатываются внутри него, а обращение к задаче через цикл событий осуществляется через asyncio.  Порядок применения методов:   1. Создайте асинхронную задачу, используя asyncio.create\_task() или asyncio.ensure\_future(). 2. Выполняйте задачу. 3. В случае необходимости отмены задачи вызовите метод task.cancel(). 4. Дождитесь завершения задачи, используя, например, await task или asyncio.gather(). 5. Проверьте состояние задачи после её завершения, вызвав метод task.cancelled() или другие по необходимости. |
| обращать внимание На что обращать внимание:   * Обрабатывайте исключение asyncio.CancelledError внутри корутины для корректного завершения задачи при отмене. * Учтите, что отмена задачи не происходит мгновенно, и задача может продолжить выполняться некоторое время после вызова метода cancel(). * В случае отмены задачи убедитесь, что все ресурсы, используемые задачей, корректно освобождаются. * Это можно сделать с помощью обработки исключения asyncio.CancelledError и использования блоков finally или контекстных менеджеров async with для закрытия файлов, сетевых соединений и т. д. * В некоторых случаях отмена задачи может привести к нежелательному поведению, например, когда корутина находится в критической секции кода. В таких случаях стоит рассмотреть использование синхронизации (например, с помощью asyncio.Lock) или обработку отмены внутри корутины, чтобы избежать проблем. * Учитывайте, что вызов метода cancel() не гарантирует отмену задачи, если корутина не обрабатывает исключение asyncio.CancelledError или игнорирует его. В этом случае задача может продолжать выполняться и после вызова метода cancel(). |
| Методы cancel(), cancelled(), cancelling() и uncancel()task.cancel() Метод task.cancel() используется для отмены задачи. Он отправляет запрос на отмену, но не гарантирует, что задача будет немедленно остановлена. Задача будет остановлена только в том случае, если она имеет возможность обработать исключение asyncio.CancelledError, которое будет вызвано при запросе на отмену. Когда задача отменяется, она переходит в состояние CANCELLED. Если задача уже была выполнена, то отменить ее уже нельзя. task.cancelled() Метод task.cancelled() используется для проверки, была ли задача отменена. Он возвращает True, если задача отменена, и False в противном случае. task.cancelling() Метод task.cancelling() - используется для проверки, находится ли задача в процессе отмены. Он возвращает True, если задача в процессе отмены, и False в противном случае. task.uncancel() Метод task.uncancel() - используется для отмены запроса на отмену задачи. Он предотвращает отмену задачи, если она еще не была остановлена,  то есть, если задача была отменена, но мы вызвали метод task.uncancel(), то задача перейдет из состояния CANCELLED в состояние RUNNING. |
| *Важно отметить* *Важно отметить, что исключение CancelledError может быть возбуждено только внутри предложения await. То есть, если вызвать метод cancel, когда задача исполняет Python-код, этот код будет продолжать работать, пока не встретится следующее предложение await (если встретится), и только тогда будет возбуждено исключение CancelledError. Вызов cancel не прерывает задачу, делающую свое дело; он снимает ее, только если она уже находится в точке ожидания или когда дойдет до следующей такой точки.*  *При этом в обернутую корутину на следующем цикле цикла событий будет брошено исключение CancelledError.*  async def my\_coroutine(seconds):  print(f'Корутина {asyncio.current\_task().get\_name()} начала работу и будет выполняться {seconds} секунд')  try:  await asyncio.sleep(seconds)  except asyncio.CancelledError:  print(f'Корутина {asyncio.current\_task().get\_name()} была отменена')  raise  *отменяем снвружи, exception –* внутри *coro*  async def main():  task\_1 = asyncio.create\_task(my\_coroutine(5))  await asyncio.sleep(2)  task\_1.cancel()  await asyncio.gather(task\_1, task\_2, return\_exceptions=True)  if task\_1.cancelled():  print(f'Задача {task\_1.get\_name()} была отменена, cancelled:   {task\_1.cancelled()}') |
| Example cancel import asyncio   async def main\_task():  print("Корутина main\_task запустилась")   await asyncio.sleep(5)   print("Корутина main\_task завершилась")    async def main():  task = asyncio.create\_task(main\_task())   await asyncio.sleep(1)   task.cancel()    try:  await task   except asyncio.CancelledError:  print("Задача отменена")  if task.cancelled():  print(f"Задача отменена - {task.cancelled()}")  asyncio.run(main()) |
| Example cancelled import asyncio   async def main\_task():   print("Корутина main\_task запустилась")   await asyncio.sleep(5)   print("Корутина main\_task завершилась")    async def main():   task = asyncio.create\_task(main\_task())   await asyncio.sleep(1)   await asyncio.sleep(2)   if task.cancelled():   print(f"Задача отменена - {task.cancelled()}")   asyncio.run(main()) |
| Example cancelling import asyncio  async def my\_coroutine(seconds):  # Определяем асинхронную функцию-корутину, которая будет выполняться указанное количество секунд  print(f'Корутина {asyncio.current\_task().get\_name()} начала работу и будет выполняться {seconds} секунд')  try:  # Ожидаем указанное количество секунд  await asyncio.sleep(seconds)  except asyncio.CancelledError:  # Обрабатываем исключение, если корутина была отменена  print(f'Корутина {asyncio.current\_task().get\_name()} была отменена')  raise  # Проверяем, отменяется ли сейчас задача, и выводим соответствующее сообщение  if asyncio.current\_task().cancelling():  print(f'Корутина {asyncio.current\_task().get\_name()} сейчас отменяется')  async def main():  # Создаем две асинхронные задачи, используя нашу корутину с разными аргументами  task\_1 = asyncio.create\_task(my\_coroutine(5))  task\_2 = asyncio.create\_task(my\_coroutine(10))  # Ожидаем 2 секунды  await asyncio.sleep(2)  # Отменяем первую задачу  task\_1.cancel()  # Дожидаемся завершения обеих задач, игнорируя исключения (return\_exceptions=True)  await asyncio.gather(task\_1, task\_2, return\_exceptions=True)  # Проверяем, была ли отменена первая задача, и выводим соответствующее сообщение  if task\_1.cancelled():  print(f'Задача {task\_1.get\_name()} была отменена, её флаг отмены {task\_1.cancelled()}')  # Проверяем, была ли отменена вторая задача, и выводим соответствующее сообщение  if task\_2.cancelled():  print(f'Задача {task\_2.get\_name()} была отменена, её флаг отмены {task\_2.cancelled()}')  # Запускаем главную функцию asyncio.run(main()) |
| состоянии цикла событий и корутин get\_coro() и get\_loop() Оба метода - task.get\_coro() и task.get\_loop() - являются полезными инструментами в асинхронном программировании и позволяют получать информацию о текущем состоянии корутин и цикла событий в asyncio. |
| Where use Вот несколько ситуаций, в которых использование task.get\_coro() может быть полезным:   1. Отладка и логирование:   for task in asyncio.all\_tasks():  coro = task.get\_coro()  print(f"Корутина: {coro}")   1. Тестирование: При написании тестов для асинхронного кода иногда вам может потребоваться получить корутину из задачи.   import unittest  class TestMyCoroutine(unittest.TestCase):  async def test\_my\_coroutine(self):  task = asyncio.create\_task(my\_coroutine())  coro = task.get\_coro()  self.assertIsInstance(coro,asyncio.coroutines.CoroWrapper) |
| После получения объекта coro с помощью метода task.get\_coro() к нему можно применить  методы **.send(), .throw() и .close()**   * task.get\_coro()**.send(**value**) -**Этот метод используется для отправки значения в корутину и продолжения ее выполнения с места, где она была приостановлена с использованием выражения yield.  Однако в асинхронном коде с использованием asyncio вы должны использовать асинхронные механизмы, такие как asyncio.create\_task() и await, чтобы дождаться выполнения корутины. * task.get\_coro()**.throw(**exc\_type[, exc\_value[, traceback]]**) -**Этот метод используется для возбуждения исключения в корутине. Он возбуждает указанное исключение в месте, где корутина была приостановлена, и продолжает ее выполнение с этого места. Если корутина обрабатывает исключение, она может продолжить выполнение с помощью выражения yield. Если корутина не обрабатывает исключение, выполнение корутины завершается с ошибкой, и исключение передается обратно вызывающему коду. * task.get\_coro()**.close() -**Этот метод используется для принудительного завершения корутины. Он возбуждает специальное исключение GeneratorExit в корутине, которое должно быть обработано корутиной или вызывающим кодом. Если корутина обрабатывает исключение GeneratorExit, она должна либо завершиться, либо возбудить исключение GeneratorExit снова. Если корутина не обрабатывает исключение GeneratorExit, выполнение корутины завершается без ошибок. |
| Task.get\_coro example import asyncio  async def foo():  await asyncio.sleep(1)  return "Завершено"  async def main():  task = asyncio.create\_task(foo())  print("Корутина:", task.get\_coro())  asyncio.run(main())  Корутина: <coroutine object foo at 0x00000191CC332E00> |
| Метод get\_loop(): Метод get\_loop() возвращает ссылку на событийный цикл и используется для получения текущего цикла событий (event loop).  Вернёт информацию о цикле событий, с которым связан текущий объект Future или Task. В то же время он возвращает экземпляр класса AbstractEventLoop, который представляет собой интерфейс для работы с событийным циклом.  task.get\_loop() получает его из контекста объекта Task,   из контекста конкретной задачи asyncio.get\_event\_loop() получает его из текущего контекста исполнения, т.е. из основного модуля asyncio. |
| Example task.get\_loop() import asyncio  async def print\_hello():  current\_task = asyncio.current\_task()  current\_loop = current\_task.get\_loop()  print("Событийный цикл:", current\_loop)  async def main():  await asyncio.create\_task(print\_hello())  asyncio.run(main())  Событийный цикл: <ProactorEventLoop running=True closed=False debug=False>   * \*ProactorEventLoop использует механизм "проактивного ввода-вывода" (proactor I/O)   + Объект **ProactorEventLoop** предоставляет следующую информацию:   + **running**: Булево значение, указывающее, работает ли событийный цикл в данный момент.   + **closed**: Булево значение, указывающее, закрыт ли событийный цикл. Закрытый цикл не может быть использован для выполнения новых задач.   + **debug**: Булево значение, указывающее, активирован ли режим отладки для событийного цикла. |
| 1. **Реактивный способ (реактор)**: Компьютер ждет, пока что-то произойдет (например, данные придут), и только тогда реагирует на это. Это как если бы вы ждали звонка и только тогда брали трубку. 2. **Проактивный способ (proactor)**: Компьютер сам инициирует действие и заранее готовится к ответу. Это как если бы вы заранее настроили телефон так, чтобы он автоматически отвечал на звонки. |
| Работа с асинхронным стеком вызова get\_stack() и print\_stack() task.**get\_stack**() - это метод, который возвращает список \*стеков вызовов для задачи. |
| task.**print\_stack**(*\**,limit = None, file = None) позволяет вывести стек вызовов на текущий момент времени.   * limit - Этот аргумент указывает максимальное количество записей стека вызовов, которые будут выведены.   + Если limit равен None, выводятся все записи стека вызовов.   + Если limit задан числом, будут выведены только указанное количество последних записей стека.   + Если limit меньше или равен 0, вывод стека вызовов будет отключен. * file - Этот аргумент указывает файл, в который будет записан вывод стека вызовов.   + Если file равен None, вывод будет отправлен в стандартный поток вывода (sys.stdout). В противном случае file должен быть файловым объектом с поддержкой метода write(). |
| Example import asyncio  async def foo():  await asyncio.sleep(1)  for stack in asyncio.current\_task().get\_stack():  print(stack)  async def main():  await asyncio.gather(foo(), foo())  asyncio.run(main()) |
| Функции callback add\_done\_callback(fn) и remove\_done\_callback(fn) **Callback**-функция  - это функция, которая передается в качестве аргумента в другую функцию и вызывается после завершения определенных действий. В asyncio, **callback**-функции могут быть привязаны к объектам **future/task**с использованием метода add\_done\_callback(). Когда операция завершена, все привязанные к ней **callback**-функции будут вызваны с данными результата в качестве аргумента. |
| * Метод add\_done\_callback() используется для добавления функции обратного вызова, которая будет вызвана после завершения задачи. Данный метод может быть использован для выполнения дополнительных действий после завершения задачи, например, для записи результатов в файл или базу данных. * Метод remove\_done\_callback() используется для удаления обратных вызовов, которые были ранее зарегистрированы с помощью метода add\_done\_callback. |
| Аргументы принимаемые методами **Аргументы принимаемые методами task.add\_done\_callback() и task.remove\_done\_callback()**  task.add\_done\_callback(callback, \*, context=None)   * callback - Это функция (или другой вызываемый объект), который будет вызван, когда объект Future завершится. Колбек должен принимать один аргумент - сам объект Future.  Функция callback, передаваемая в add\_done\_callback, должна быть синхронной. Если вы попытаетесь передать асинхронную функцию как коллбек, то она не будет вызвана как корутина и вы можете получить непредвиденное поведение. * context (опциональный) Этот параметр используется для передачи \*контекста выполнения для колбека. Это может быть полезно для расширенного управления контекстом выполнения, особенно в приложениях, использующих асинхронное программирование. Если этот параметр не указан, колбек будет выполнен в текущем контексте выполнения. |
| Контекст исполнения может быть объектом, производным от класса concurrent.futures.\_base.ExecutionContext. Он содержит информацию о контексте выполнения, которая может быть использована callback-функцией. Например, это может быть использовано для передачи настроек логирования, значений переменных окружения или конфигурации. |
| Метод remove\_done\_callback() Метод remove\_done\_callback() принимает только один аргумент, callback-функцию.  task.remove\_done\_callback(callback)  callback: функция-обратный вызов, которую нужно удалить. |
| Callback example def greetings(txt):  return f'Hello {txt}'  def farewell(txt):  return f'Bye, {txt}'   def call\_back(func, param):  return func(param)  name = "Student"  result = call\_back(greetings, name) print(result)  result = call\_back(farewell, name) print(result) |
| В асинхронном программировании и обработке событий, **callback**-функции обычно вызываются после завершения определенных асинхронных операций или в ответ на возникновение определенных событий, что позволяет выполнять пользовательский код без блокирования основного потока программы. |
| Использование callback-функций в программировании:  1. **Асинхронное программирование**: callback-функции позволяют выполнять код после завершения асинхронной операции, такой как чтение из файла, выполнение HTTP-запроса или ожидание ответа от базы данных. Вместо блокирования выполнения программы, вы можете использовать **callback**, чтобы продолжить работу, когда асинхронная операция завершится. 2. **Обработка событий**: callback-функции широко используются для обработки событий в графических интерфейсах и событийных системах. Вы можете назначить **callback**-функцию на определенное событие (например, клик по кнопке), и эта функция будет вызвана при наступлении события. 3. **Планирование задач**: callback-функции могут использоваться для запуска задач по расписанию. Вы можете создать планировщик, который вызывает определенные функции через определенные промежутки времени, используя **callback**-функции. 4. **Реализация паттерна "наблюдатель"**: В некоторых случаях, вам может потребоваться оповещать несколько объектов об изменении состояния другого объекта. В этом случае, вы можете использовать **callback**-функции для реализации паттерна "наблюдатель" (**observer pattern**). Объекты "наблюдатели" предоставляют свои **callback**-функции, которые будут вызваны при изменении состояния "субъекта". 5. **Обработка результатов выполнения других функций**: callback-функции могут использоваться для обработки результатов выполнения других функций, особенно если эти функции могут быть вызваны в разных частях программы с разными обработчиками результатов.   Важно отметить, что в современных версиях Python, особенно при работе с асинхронным кодом, **callback**-функции постепенно заменяются более выразительными и удобными средствами, такими как асинхронные функции и async/await. Однако, **callback**-функции всё ещё имеют своё место и применение в определенных ситуациях. |
| Example asyncio callback: import asyncio  async def async\_operation():   print("Асинхронная операция началась...")  await asyncio.sleep(2)  print("Асинхронная операция завершена.")  return "Результат асинхронной операции"  def on\_completion(task):   result = task.result()   print(f"Callback функция вызвана. Получен результат: {result}")  async def main():   task = asyncio.create\_task(async\_operation())   task.add\_done\_callback(on\_completion)   await task    asyncio.run(main())  output: Асинхронная операция началась... Асинхронная операция завершена. Callback функция вызвана. Получен результат: Результат асинхронной операции |
| Вывод: #  # # При передаче флага False в asyncio.run(main(False))  # Тренировка отменена.  #  # # При передаче флага True в asyncio.run(main(True))  # Тренировка начинается, идите в спортзал. |
| Example Pizza import asyncio import random   async def prepare\_pizza(): # Асинхронная функция для имитации приготовления пиццы  print("Готовим пиццу...")  await asyncio.sleep(5) # Имитация времени на приготовление пиццы  return "Пицца готова!"   def notify\_delivery(task): # Callback-функция для уведомления о доставке пиццы  print(f"{task.result()} \n Курьер: Ваш заказ доставлен!")   def cancel\_notification(task): # Callback-функция для отмены уведомления о доставке  print(f"{task.result()} \n Курьер: Уведомление отменено, заберите пиццу самостоятельно.")   async def main(): # Основная асинхронная функция  task = asyncio.create\_task(prepare\_pizza()) # Создание задачи из асинхронной функции prepare\_pizza  task.add\_done\_callback(notify\_delivery) # Регистрация callback-функций  task.add\_done\_callback(cancel\_notification)  if random.choice([True, False]): # Вероятность отмены уведомления (50%)  print('Доставка подтверждена, везём пиццу')  task.remove\_done\_callback(cancel\_notification) # Отмена уведомления об отмене доставки  else:  print('Доставка отменена, самовывоз')  task.remove\_done\_callback(notify\_delivery) # Отмена уведомления о доставке   await task   asyncio.run(main()) |
| Доставка подтверждена, везём пиццу # Готовим пиццу... # Пицца готова!  # Курьер: Ваш заказ доставлен! |
| К одной задаче (или "future") можно назначить неограниченное количество callback-ов. Каждый callback будет вызван после завершения задачи, с объектом Future или Task в качестве аргумента. Вот пример того, как можно назначить несколько callback-ов одной и той же задаче: |
| Example few\_callbacks  import asyncio   def callback1(task):  print('Callback 1: Task completed!')  print('Result:', task.result())   def callback2(task):  print('Callback 2: Task completed!')  print('Result:', task.result())   async def my\_coro():  return 42   async def main():  # Создание задачи  task = asyncio.create\_task(my\_coro())   # Добавление callback-ов к задаче  task.add\_done\_callback(callback1)  task.add\_done\_callback(callback2)   # Ожидание завершения задачи  await task  # Запуск основного асинхронного корутина asyncio.run(main())  # Callback 1: Task completed! # Result: 42 # Callback 2: Task completed! # Result: 42 |
| Защита корутины от отмены asyncio.shield() asyncio.shield() - это корутина, предоставляемая модулем asyncio для защиты  awaitable объекта от отмены (cancellation) во время выполнения. |
| awaitable asyncio.shield(aw) — Защищает ожидаемый объект (aw - awaitable object) от отмены.  Если aw является корутиной, то он автоматически планируется как Task. |
| Example import asyncio  # Корутина имитирует выполнение критической операции async def my\_coroutine():  print("Агент приступил к выполнению своего задания")  await asyncio.sleep(1)  print("Злодей побежден! Миссия успешно завершена!")  # Корутина попытается прервать выполнение защищенной корутины async def cancel\_coroutine(task):  await asyncio.sleep(0.5)  task.cancel()  print("Банг!!! Злодей стреляет в агента!")   async def main():  shielded\_coroutine = asyncio.shield(my\_coroutine())  print("Бронежилет надет на агента")  cancel\_task = asyncio.create\_task(cancel\_coroutine(shielded\_coroutine))  print("Пистолет злодея заряжен")  try:  await asyncio.gather(shielded\_coroutine, cancel\_task)  except asyncio.CancelledError as e:  print(f'Внимание! Бронежилет разрушен! {e}')  await asyncio.sleep(1)   asyncio.run(main()) |
| Однако, следует учитывать, что использование  asyncio.shield() может привести к некоторым некоторым нежелательным последствиям, если использовать ее неосторожно:   1. **Утечка ресурсов**: Защищённые awaitable объекты могут продолжать использовать ресурсы, даже если остальная программа завершилась. 2. **Зависание программы**: Программа может не завершиться корректно, если остались незавершённые защищённые awaitable объекты. 3. **Сложность отладки**: Защита awaitable объектов может затруднить отладку, делая неочевидными причины появления ошибок в работе программы. 4. **Невозможность отмены**: Защищённые awaitable объекты нельзя просто отменить, что может быть проблематично в некоторых случаях. |
| Example2 import asyncio   async def background\_timer(n: int):  print(f'Timer {n} started')  i = 1  while i <= n:  print(f'Прошел {i} тик')  await asyncio.sleep(1)  i += 1  print(f'Timer {n} stopped')  return 1   async def main():  task = asyncio.create\_task(background\_timer(5))  shielded = asyncio.shield(task)  await asyncio.sleep(2)  shielded.cancel()  print('trying to stop shielded task')  try:  await task  except asyncio.CancelledError:  print("Отмена задачи")  else:  print("Задача успешно выполнена")   print(f'Обьект задачи {task}')  print(f'Результат задачи {task.result()}')  print(f'Обьект щита {shielded}')   asyncio.run(main())  # Timer 5 started # Прошел 1 тик # Прошел 2 тик # trying to stop shielded task # Прошел 3 тик # Прошел 4 тик # Прошел 5 тик # Timer 5 stopped # Задача успешно выполнена # Обьект задачи <Task finished name='Task-2' coro=<background\_timer() done, defined at D:\GB\pythonProject\asyncio\Chapter\_6 \_Task\6\_13\_asyncio\_shield\main3\_cancel\_shield.py:4> result=1> # Результат задачи 1 # Обьект щита <Future cancelled> |
| Example3 import asyncio   async def background\_timer(n: int):  print(f'Timer {n} started')  i = 1  while i <= n:  print(f'Прошел {i} тик')  await asyncio.sleep(1)  i += 1  print(f'Timer {n} stopped')  return 1   async def main():  task = asyncio.create\_task(background\_timer(5))  shielded = asyncio.shield(task)  await asyncio.sleep(2)  task.cancel()  print('trying to stop task')  try:  await task  except asyncio.CancelledError:  print("Отмена задачи")  else:  print("Задача успешно выполнена")   print(f'Обьект задачи {task}')  # print(f'Результат задачи {task.result()}')  print(f'Обьект щита {shielded}')   asyncio.run(main())  # Timer 5 started # Прошел 1 тик # Прошел 2 тик # trying to stop task # Отмена задачи # Обьект задачи <Task cancelled name='Task-2' coro=<background\_timer() done, defined at D:\GB\pythonProject\asyncio\Chapter\_6 \_Task\6\_13\_asyncio\_shield\main4\_cancel\_task.py:4>> # Обьект щита <Future cancelled> |
| TaskGroup() !!! Python 3.11**Краткая справка** В Python 3.11 представлен новый подход к управлению группой корутин, называемый asyncio.**TaskGroup()**. Теперь этот способ работы с группами задач является предпочтительным. **TaskGroup** призван объединить функциональность asyncio.**createTask()** и asyncio.**gather()** в одном инструменте. Это означает что **asyncio.TaskGroup()** сочетает в себе API создания задач с удобным и надежным способом ожидания завершения всех задач в группе.  Обработка группы корутин позволяет объединить в себе функции:   1. Ожидание завершения всех задач. 2. Отмена всех задач, если хотя бы одна задача завершится с ошибкой. 3. Обработка исключения, возникшего в любой задаче. |
| Объект asyncio.TaskGroup реализует интерфейс асинхронного контекстного менеджера, и это предпочтительное использование класса. Это означает, что экземпляр класса создается и используется через выражение  async with .  async with asyncio.TaskGroup() as tg:  асинхронный контекстный менеджер реализует методы \_\_aenter\_\_() и \_\_aexit\_\_(), эти объекты  являются **awaitable** и их можно ожидать .  В случае asyncio.**TaskGroup()** метод \_\_aexit\_\_(), который вызывается автоматически при выходе из блока контекстного менеджера, будет ожидать выполнения всех задач, созданных внутри **asyncio.TaskGroup**.  После завершения последней задачи и выхода из блока async with добавление новых задач в группу становится невозможным. **Создание задачи внутри группы задач.** tg.create\_task(coro, \*, name=None, context=None) |
| Example task in group async def some\_coro(num):  # Имитируем выполнение длительной I/O зависимой операции.  await asyncio.sleep(num)  # Возвращаем результат работы.  return num / 2   async def main():  # Создаем группу задач.  async with asyncio.TaskGroup() as tg:  # Создаем в группе две задачи.  task1 = tg.create\_task(some\_coro(1))  task2 = tg.create\_task(some\_coro(2))  print(f"Oбе задачи выполнены с результатом: {task1.result()} и {task2.result()} соответственно.")  asyncio.run(main()) |
| Example2 coro in group   import asyncio  # Корутина для создания задач async def coro():  name = asyncio.current\_task().get\_name()  print(f'{name} начала свою работу!')  await asyncio.sleep(1)  print(f'{name} завершена!')   # Базовая корутина async def main():  async with asyncio.TaskGroup() as group:  [group.create\_task(coro(), name=f'Задача\_0{i}') for i in range(1, 6)]   print('Все задачи были выполнены!')   asyncio.run(main()) |
| Example 3 args and result import asyncio   async def coro(value):  await asyncio.sleep(1)  return value \* 100   async def main():  async with asyncio.TaskGroup() as group:  tasks = [group.create\_task(coro(i)) for i in range(1, 11)]  for el in tasks: print(el)   for task in tasks:  print(task)  print(task.result())  asyncio.run(main()) |
| Task\_group Exceptions CancelledError**Как отменить задачу в группе задач?** Для отмены запущенной задачи используйте метод cancel(). Его вызов приведет к тому, что задача выбросит исключение CancelledError в обернутую корутину. Полученное исключение asyncio.CancelledError прервет выполнение await, но не будет передано на уровень выше за пределы оператора async with.  Таким образом, исключение, вызванное в контексте TaskGroup, не приведет к прекращению выполнения main() за пределами блока async with.  ***Важно!!!***Обращаю внимание, что все вышесказанное относится только к ***CancelledError***,  Рассмотрим поведение **asyncio.TaskGroup()** при отмене задач с помощью метода **cancel()**объекта **asyncio.Task**.  **Ожидаемое поведение**:   * Запрос на отмену задачи будет обработан. * Отмененная задача прекратит свое выполнение. * Остальные задачи продолжат свое выполнение в обычном режиме. |
| **Как отменить все задачи, если одна задача завершилась с ошибкой, используя**asyncio.TaskGroup()**?** Еще одной особенностью **TaskGroup()**является то, что при первом сбое любой из задач, входящих в группу, с исключением, отличным от asyncio.CancelledError, остальные задачи в группе автоматически отменяются. После этого никакие другие задачи не могут быть добавлены в группу. В этот момент, если тело оператора async with еще активно (т.е. \_\_aexit\_\_() еще не вызвана), задача, непосредственно содержащая оператор async with, также отменяется. |
| Taskgrpup exception |
| Обработка ошибок |
| Обработка ошибок asyncio.gather import asyncio   async def file\_reader(filename: str) -> str:  """Корутина для чтения файла"""  with open(filename) as f:  data: str = f.read()  return data   async def get\_data(data: int) -> dict:  """Корутина, для возврата переданного числа в виде словаря вида {'data': data}"""  if data == 0:  raise Exception('Нет данных...')  return {'data': data}   # Базовая корутина async def main():  tasks = asyncio.gather(  get\_data(1),  get\_data(2),  # Передаем имя несуществующего файла, чтобы вызвать ошибку  file\_reader('fake.png'),  # Этот вызов тоже должен вызвать ошибку  get\_data(0),  return\_exceptions=True  )  result = await tasks  print('Готово!!!', result)  asyncio.run(main())  Без return\_exceptions=True - доходит до первой ошибки и вываливается  С return\_exceptions=True - выполняет все task, результат:  Готово!!!  [ {'data': 1},  {'data': 2},  FileNotFoundError(2, 'No such file or directory'),  Exception('Нет данных...')  ] |
| Обработка ошибок asyncio. taskGroup Example1  # Базовая корутина async def main():  try:  async with asyncio.TaskGroup() as tg:  task1 = tg.create\_task(get\_data(1))  task2 = tg.create\_task(get\_data(2))  task3 = tg.create\_task(get\_data(0))  task4 = tg.create\_task(file\_reader('fake.png'))  task5 = tg.create\_task(file\_reader('new\_fake.png'))  task6 = tg.create\_task(get\_data(0))  # Результат мы все равно не увидим, так как спровоцируем вызов исключений.  result = [task1.result(), task2.result(), task3.result(), task4.result(), task5.result(), task6.result()]  print('Готово!!!', result)  # Добавляем обработчики, которые будут группировать ошибки одного типа.  except\* FileNotFoundError as e:  # print(e.exceptions)  for error in e.exceptions:  print(error)  except\* Exception as e:  print(e.exceptions)  for error in e.exceptions:  print(error)   asyncio.run(main()) |
| Example2  async with asyncio.TaskGroup() as tg:  task1 = tg.create\_task(my\_coro(1))  task2 = tg.create\_task(my\_coro(0))  task3 = tg.create\_task(my\_coro(2))  task4 = tg.create\_task(my\_coro(0))  task5 = tg.create\_task(my\_coro(3))  task6 = tg.create\_task(my\_coro(3))  tasks = [task1, task2, task3, task4, task5, task6]  print('all tasks ready', \*tasks, sep='\n')  await tg  except\* ValueError as e:  for el in e.exceptions:  print(el)  except\* FileExistsError as e:  for el in e.exceptions:  print(el)  except\* Exception as e:  for el in e.exceptions:  print(el)  print('all tasks ready', \*tasks, sep='\n')  for el in tasks:  if el.\_result:  print(el.\_result)  if el.\_exception:  print(el.\_exception)  asyncio.run(main()) |

|  |  |
| --- | --- |
| Асинхронные конструкции | |
| Основы async with | |
| async with - это специальное выражение в Python, которое используется для асинхронного управления ресурсами. В контексте асинхронного программирования async with позволяет выполнять код внутри блока без блокирования и автоматически освобождать ресурсы после завершения выполнения блока. | |
| Вот несколько типичных случаев, когда использование async with может быть полезно в программировании: | |
| **Асинхронное взаимодействие с файлами:** import aiofiles import asyncio  async def read\_file(file\_name):  async with aiofiles.open(file\_name, 'r') as file:   contents = await file.read()   print(contents)   asyncio.run(read\_file('example.txt')) | |
| Асинхронные сетевые соединения: import aiohttp import asyncio   async def fetch\_url(url):  async with aiohttp.ClientSession() as session:   async with session.get(url) as response:   content = await response.text()   print(content)    asyncio.run(fetch\_url('https://example.com')) | |
| Асинхронные блокировки и семафоры import asyncio  async def protected\_section(lock, task\_id):  async with lock:   print(f'Task {task\_id} has entered the protected section')  await asyncio.sleep(1)   print(f'Task {task\_id} has left the protected section')   async def main():   lock = asyncio.Lock()   tasks = [protected\_section(lock, i) for i in range(3)]   await asyncio.gather(\*tasks)    asyncio.run(main()) | |
| Асинхронные контекстные операции: class AsyncTransaction:  async def \_\_aenter\_\_(self):   print("Starting transaction")  await asyncio.sleep(0.5)   async def \_\_aexit\_\_(self, exc\_type, exc\_val, exc\_tb):   print("Ending transaction")  await asyncio.sleep(0.5)   async def perform\_transaction():  async with AsyncTransaction():   print("Performing transaction operations")   await asyncio.sleep(1)    asyncio.run(perform\_transaction()) | |
| некоторые потенциальные проблемы Однако при использовании async with также могут возникнуть некоторые потенциальные проблемы:   1. **Неправильное использование контекстных менеджеров**: если вы используете обычный контекстный менеджер (который определяет \_\_enter\_\_ и \_\_exit\_\_) вместо асинхронного контекстного менеджера (который определяет \_\_**a**enter\_\_ и \_\_**a**exit\_\_), это может привести к неправильному управлению ресурсами. 2. **Неправильная обработка исключений**: если исключения не обрабатываются правильно внутри метода \_\_aexit\_\_, это может привести к непредсказуемому поведению вашего асинхронного кода. Убедитесь, что вы обрабатываете исключения корректно и возвращайте True или False из метода \_\_aexit\_\_ в зависимости от того, нужно ли подавить исключение. 3. **Неправильная организация кода**: если ваш код не разделен на асинхронные функции и вы используете async with внутри обычных функций, это вызовет синтаксическую ошибку. Убедитесь, что вы используете async with только внутри асинхронных функций. 4. **Зависимость от сторонних библиотек**: в некоторых случаях вам может потребоваться использовать сторонние библиотеки, которые предоставляют асинхронные контекстные менеджеры, такие как aiofiles или aiohttp. Обратите внимание на совместимость и поддержку этих библиотек, а также на то, как они могут повлиять на производительность вашего кода. | |
| несколько рекомендаций несколько рекомендаций, которые помогут вам избежать этих проблем:   1. **Используйте правильный контекстный менеджер**: убедитесь, что вы используете асинхронные контекстные менеджеры (\_\_aenter\_\_ и \_\_aexit\_\_) для async with, а не обычные контекстные менеджеры (\_\_enter\_\_ и \_\_exit\_\_). Это поможет избежать блокировок и неправильного управления ресурсами. 2. **Обрабатывайте исключения правильно**: внутри метода \_\_aexit\_\_ обрабатывайте исключения корректно и возвращайте True или False, чтобы указать, нужно ли подавить исключение. Это поможет избежать непредсказуемого поведения вашего асинхронного кода. 3. **Разделите код на асинхронные функции**: убедитесь, что вы используете async with только внутри асинхронных функций, и ваш код разделен на асинхронные и обычные функции соответственно. Это предотвратит синтаксические ошибки и сделает ваш код более читаемым и понятным. 4. **Оцените сторонние библиотеки**: при использовании сторонних библиотек для асинхронных контекстных менеджеров обратите внимание на их совместимость, поддержку и производительность. Используйте только проверенные и надежные библиотеки, чтобы минимизировать возможные проблемы. | |
| async for async for появился в Python 3.5 и представляет собой асинхронный итератор, который позволяет выполнять асинхронные операции в цикле. В отличие от обычного цикла for, async for предназначен для работы с асинхронными генераторами, которые возвращают объекты, поддерживающие асинхронное итерирование, не блокируя выполнение других задач.   В отличие от обычного for, который блокирует выполнение программы до тех пор, пока не завершится итерация, async for позволяет выполнять другие задачи во время итерации. | |
| основные сценарии, в которых используется async for Вот основные сценарии, в которых используется async for:   1. **Обработка асинхронных потоков данных**: Когда вам нужно обрабатывать большие объемы данных, которые поступают по мере готовности, например, чтение из файла или потока данных через сеть. 2. **Работа с асинхронными API**: Когда используется асинхронное API для работы с внешними сервисами, такими как RESTful API или облачные сервисы, которые возвращают данные по частям. 3. **Websockets и асинхронное общение**: Когда работаете с веб-сокетами или асинхронными системами обмена сообщениями, где требуется обрабатывать входящие сообщения последовательно и без блокировки. 4. **Параллельная обработка задач**: Когда вам нужно выполнять множество асинхронных задач параллельно и обрабатывать результаты по мере их выполнения. | |
| async for number in async\_gen(): import asyncio   async def async\_gen():   for i in range(5):   await asyncio.sleep(.5)   yield i    async def main():   async for number in async\_gen():   print(number)    asyncio.run(main())  Чтобы объект мог быть использован в асинхронном цикле async for, он должен поддерживать протокол асинхронного итератора. Объект должен иметь метод \_\_aiter\_\_ и \_\_anext\_\_. Для проверки объекта необходимо использовать метод \_\_dir\_\_(), который покажет все магические методы объекта. | |
| **Важно**: Обратите внимание, что наличие метода \_\_aiter\_\_ еще не гарантирует, что объект является асинхронным итератором. Вы также должны проверить наличие метода \_\_anext\_\_ и убедиться, что эти методы являются асинхронными (определены с помощью ключевого слова async def).  async for можно использовать в любом случае, когда требуется итерироваться асинхронно через коллекции. Однако, не следует злоупотреблять использованием async for в случаях, когда итерация не занимает много времени и не блокирует выполнение других задач. | |
| import aiohttp import asyncio   async def async\_url\_generator(urls):   for url in urls:   yield url    async def fetch(session, url):   async with session.get(url) as response:   return await response.text()    async def main():     urls = [  'https://example.com',  'https://example.org',  'https://example.net',  ]  async with aiohttp.ClientSession() as session:   async for url in async\_url\_generator(urls):   content = await fetch(session, url)   print(f'Fetched content from {url}: {content[:100]}')    asyncio.run(main()) | |
| import aiofiles import asyncio   async def read\_file\_line\_by\_line(file\_path):   async with aiofiles.open(file\_path, mode='r') as file:   async for line in file:   print(line.strip())    async def main():   file\_path = 'example.txt'   await read\_file\_line\_by\_line(file\_path)    asyncio.run(main()) | |
| потенциальные проблемы и ограничения потенциальные проблемы и ограничения, которые следует учитывать при работе с асинхронными итераторами:   1. **Ограниченная поддержка стандартной библиотеки**: Многие стандартные библиотеки Python, такие как open() для работы с файлами, не поддерживают асинхронные операции "из коробки". Вам может потребоваться использовать сторонние библиотеки, такие как aiofiles или aiohttp, для работы с асинхронными итераторами. 2. **Производительность**: В некоторых случаях использование асинхронного кода может быть менее эффективным, чем синхронный код, особенно если задачи не имеют значительных задержек ввода-вывода(как в последнем коде). Также создание и управление асинхронными итераторами может добавить дополнительные накладные расходы. 3. **Управление ошибками**: Обработка ошибок в асинхронных итераторах может быть сложнее, так как исключения могут возникать в разных контекстах выполнения. Вам может потребоваться использовать блоки try/except внутри асинхронных итераторов или корутин для корректной обработки ошибок. 4. **Вложенность и сложность кода**: Использование async for и других асинхронных конструкций может увеличить вложенность и сложность кода. Это может затруднить чтение и отладку кода, особенно для тех, кто мало знаком с асинхронным программированием. 5. **Отсутствие гарантий порядка выполнения**: Так как асинхронные операции выполняются параллельно, порядок выполнения может быть непредсказуемым. Это может привести к сложностям при отладке или неожиданным результатам, если порядок выполнения имеет значение. 6. **Ограничение на использование синхронных функций**: Вы не можете напрямую вызывать синхронные функции из асинхронных итераторов или корутин, так как это может заблокировать основной поток выполнения. Вам нужно будет преобразовать синхронные функции в асинхронные или использовать специальные механизмы, такие как run\_in\_executor, для выполнения синхронных функций в асинхронном контексте. | |
| Объекты которые поддерживают асинхронные итерации: Объекты которые поддерживают асинхронные итерации: Асинхронные генераторы: Функции-генераторы, определенные с ключевым словом async def и содержащие выражение yield. Они могут использоваться для создания пользовательских асинхронных итераторов.  async def async\_generator():  for i in range(5):  await asyncio.sleep(1)  yield i | |
| Асинхронные объекты файлов **Асинхронные объекты файлов**: Объекты файлов, открытые с помощью асинхронной библиотеки aiofiles.  import aiofiles    async with aiofiles.open('example.txt', mode='r') as file:  async for line in file:  print(line.strip()) | |
| Асинхронные объекты потоков **Асинхронные объекты потоков**: Объекты потоков, использующие асинхронные операции ввода-вывода, например, потоки чтения и записи для сетевых соединений или веб-сокетов.  import asyncio  reader, writer = await asyncio.open\_connection('example.com', 80)  async for line in reader:  print(line.decode().strip()) | |
| Асинхронные объекты курсоров баз данных **Асинхронные объекты курсоров баз данных**: Курсоры баз данных, поддерживающие асинхронное выполнение запросов и извлечение данных. Например, курсоры асинхронных баз данных, таких как aiomysql или aiopg.  import aiomysql  async with aiomysql.connect(...) as conn:  async with conn.cursor() as cur:  await cur.execute("SELECT \* FROM example\_table")  async for row in cur:  print(row) | |
| Пользовательские асинхронные итераторы **Пользовательские асинхронные итераторы**: Объекты, реализующие асинхронные методы \_\_aiter\_\_() и \_\_anext\_\_(). Эти объекты могут быть созданы для решения специфических задач или для интеграции с асинхронными API.  class AsyncCounter:  def \_\_init\_\_(self, start, end):  self.start = start  self.end = end  def \_\_aiter\_\_(self):  return self  async def \_\_anext\_\_(self):  if self.start >= self.end:  raise StopAsyncIteration  self.start += 1  await asyncio.sleep(1)  return self.start – 1 | |
| Асинхронные итераторы для библиотеки aiohttp: **Асинхронные итераторы для библиотеки** aiohttp: Используя библиотеку aiohttp для асинхронных HTTP-запросов, вы можете использовать асинхронные итераторы для чтения тела ответа по частям или обработки многостраничных API.  import aiohttp  async with aiohttp.ClientSession() as session:  async with session.get('https://example.com/large\_file') as response:  async for chunk in response.content.iter\_chunked(1024):  process\_chunk(chunk) | |
| Асинхронные итераторы для библиотеки aioredis: **Асинхронные итераторы для библиотеки** aioredis: Библиотека aioredis предоставляет асинхронные итераторы для работы с ключами и значениями в хранилище данных Redis.  import aioredis  async def main():  redis = aioredis.from\_url("redis://localhost")  async for key in redis.scan\_iter("example:\*"):  value = await redis.get(key)  print(key, value)  asyncio.run(main()) | |
| Для асинхронного async for можно использовать короткий синтаксис, который подробно описан в списках-выражениях. Действует только в теле асинхронной функции (сопрограммы) async def.  # асинхронное выражение-генератор:  (i \*\* 2 async for i in agen()).  # асинхронное список-выражение:  [i async for i in agen()]  # асинхронное словарь-выражение:  {i: i \*\* 2 async for i in agen()} ;  # асинхронное выражение-множество:   {i async for i in agen()} | |
| Разрешено использование выражений await как в асинхронном, так и в синхронном понимании коротких выражений генераторов.  # на примере списка-выражения  result = [await fun() for fun in funcs]  result = [await fun() for fun in funcs if await smth]  result = [await fun() async for fun in funcs]  result = [await fun() async for fun in funcs if await smth] | |
| Давайте создадим аналогичные генераторы для словарей.  # Асинхронное выражение-генератор для словаря:  {k: v async for k, v in adict.items()}  # Асинхронное словарь-выражение:  {k: await vfun() async for k, vfun in funcs\_dict.items()}  # Асинхронное словарь-выражение с условием:  {k: await vfun() async for k, vfun in funcs\_dict.items() if await conditionfun(k)} | |
| Работа с очередями в Asyncio | |
| 8.1 Введение в очереди **Очередь** - это абстрактный тип данных, который поддерживает две основные операции: добавление элемента в очередь (enqueue) и извлечение элемента из очереди (dequeue).   asyncio.Queue() **FIFO** (**First-In-First-Out**) - Простая реализация очереди для использования в асинхронном коде, типа "первым пришел - первым ушел" .   asyncio.LifoQueue()  **LIFO** (**Last In, First Out**) - Реализация очереди в коде, типа "последним пришел, первым ушел" для использования в асинхронном коде.   asyncio.PriorityQueue() - Реализация очереди с приоритетами для использования в асинхронном коде. | |
| Алгоритм BFS: **BFS** (Breadth-First Search/ Поиск в ширину) - это алгоритм поиска в ширину, который используется для поиска всех узлов в графе или дереве в порядке, соответствующем уровням узлов.  Алгоритм BFS работает следующим образом:   1. Выбирается начальный узел и добавляется в очередь. 2. Извлекается первый узел из очереди и просматриваются его соседи. 3. Если соседний узел еще не посещен, то он добавляется в очередь. 4. Процесс повторяется, пока очередь не пуста.   BFS может быть использован для решения различных задач, таких как поиск кратчайшего пути в графе, поиск компонент связности в графе и т.д. BFS может быть реализован с помощью очереди, так как он требует посещения узлов в порядке уровней. Каждый уровень узлов может быть представлен как одна итерация в очереди, где все узлы, добавленные в очередь в предыдущей итерации, будут посещены в текущей итерации. | |
| Алгоритм DFS: DFS (Depth-First Search/ Поиск в глубину) - это алгоритм поиска в глубину, который используется для поиска всех узлов в графе или дереве в порядке, соответствующем глубине узлов.  Алгоритм DFS работает следующим образом:   1. Выбирается начальный узел. 2. Узел помечается как посещенный и просматриваются его непосещенные соседи. 3. Если есть непосещенный сосед, то процесс повторяется с него как с новым начальным узлом. 4. Если все соседи посещены, то возвращаемся к предыдущему узлу и продолжаем поиск с другого соседа. 5. Процесс повторяется, пока не будут посещены все узлы.   DFS может быть использован для решения различных задач, таких как поиск компонент связности в графе, топологическая сортировка в графе и т.д.  DFS может быть реализован с помощью стека или рекурсии, так как он требует посещения узлов в порядке глубины. При каждой итерации DFS переходит к следующему узлу, пока не достигнет конечной точки, затем возвращается к предыдущему узлу и продолжает поиск с другого соседа. | |
|  | |
| Очередь FIFO Простая очередь asyncio.Queue() asyncio.Queue() (***FIFO /*First-In-First-Out,**"первым пришел - первым ушел") - представляет собой простую реализацию очереди для использования в асинхронном коде. Он предоставляет несколько методов, которые позволяют добавлять и извлекать элементы из очереди, а также управлять ее состоянием. Зачем это нужно? asyncio.Queue() позволяет вам создавать очередь из задач, которые могут быть выполнены в асинхронном режиме. Это особенно полезно, когда у вас есть "производитель" данных, который работает в одном темпе, и "потребитель" данных, который работает в другом. Очередь позволяет этим двум сторонам работать вместе более эффективно и безопасно.  Для использования очереди необходимо создать экземпляр класса asyncio.Queue(), после этого станут доступны все методы. asyncio.Queue()принимает один аргумент конструктора «maxsize», по умолчанию установленный на ноль (без ограничений), но её нельзя использовать вне программы asyncio.  queue = asyncio.Queue(maxsize=0) | |
| **Очередь** — это структура данных, в которую элементы могут быть добавлены вызовом put() и из которых элементы могут быть извлечены вызовом get().  Мы можем установить ограничение на размер очереди. Эффект ограничения размера означает, что, когда очередь заполнена и корутины пытаются добавить объект, они блокируются до тех пор, пока не освободится место, или терпят неудачу, если используется неблокирующий метод. Мы можем узнать размер очереди при помощи print(queue.maxsize).  queue = asyncio.Queue(maxsize=100) | |
| Члены класса:  * **maxsize** - возвращает количество элементов, допустимых в очереди. * **empty()** - возвращает True, если очередь пуста, False в противном случае. * **full()** - возвращает True, если очередь достигла своего максимального размера. Если очередь была инициализирована с maxsize=0 (по умолчанию), то full() никогда не вернет True. * coroutine **get()** -  извлечение и возврат элемента из очереди. Если очередь пуста, ожидание, пока элемент не будет доступен. * **get\_nowait()** - возвращает элемент, если он немедленно доступен, иначе выдает QueueEmpty. * coroutine **join()** - блокирует выполнение программы до тех пор, пока все элементы в очереди не будут получены и обработаны. Счетчик незавершенных задач увеличивается при добавлении элемента в очередь. Счетчик уменьшается всякий раз, когда сопрограмма-потребитель вызывает task\_done(), указывая на то, что элемент получен и работа с ним завершена. Когда счетчик незавершенных задач опускается до нуля, join() разблокирует ход выполнения программы. * coroutine **put(item)**- помещает элемент в очередь. Если очередь переполнена, то перед добавлением элемента необходимо дождаться появления свободного места. * **put\_nowait(item)** - помещает элемент в очередь без блокировки. Если свободное место в очереди не освободилось, выдается сообщение QueueFull. * **qsize()** - возвращает количество элементов в очереди. * **task\_done()** - указывает, что ранее поставленная в очередь задача завершена. Используется потребителями очереди. Для каждого вызова get(), используемого для получения задачи, последующий вызов task\_done() сообщает очереди, что обработка задачи завершена. Если в данный момент блокируется join(), то он возобновится, когда все элементы будут обработаны (то есть вызов task\_done() был получен для каждого элемента, который был помещен в очередь). Вызывает ошибку ValueError, если вызывается большее количество раз, чем было помещено элементов в очередь. | |
| Добавить элементы в очередь Asyncio: # добавить объект в очередь  await queue.put(item)  Элемент также можно добавить в очередь без блокировки с помощью метода put\_nowait(). Если в очереди нет места, то метод  put\_nowait() вызывает исключение asyncio.QueueFull.  Это исключение можно обработать следующим образом.  try:  # попытка добавить элемент  queue.put\_nowait(item)  except asyncio.QueueFull:  # ... | |
| Получить элементы из очереди Asyncio: Элементы можно получить из очереди, вызвав метод get().  На самом деле это корутина, которую нужно дождаться. В данный момент в очереди может не быть элементов для извлечения, и вызывающей корутине может потребоваться блокировка до тех пор, пока элемент не станет доступным.  # получить элемент из очереди  item = await queue.get()  Извлеченный элемент будет самым старым добавленным элементом.  Элемент можно извлечь из очереди без блокировки с помощью метода get\_nowait().  Этот метод не является корутиной и немедленно вернет элемент, если он доступен, в противном случае произойдет сбой с исключением asyncio.QueueEmpty().  ...  try:  # попытка получить элемент  item = queue.get\_nowait()  except asyncio.QueueEmpty:  # ... | |
| Проверяем размер очереди: Узнать, какое максимальное количество элементов может принять очередь, можно, используя maxsize, который и возвращает количество элементов, допустимых в очереди.  ...  # сообщить размер очереди  print(queue.maxsize)  Мы можем проверить, пуста ли очередь, с помощью метода empty(), который возвращает True, если очередь не содержит элементов, или False в противном случае.  ...  # сообщить пуста ли очередь  print(queue.empty())  Также можем проверить, заполнена ли очередь полностью, с помощью метода full(), который возвращает True, если очередь содержит максимально возможное количество элементов. Если очередь была инициализирована с maxsize=0 (по умолчанию), то full() никогда не вернет True, так как, если  maxsize меньше или равен нулю, то размер очереди бесконечен.  ...  # проверить заполнена ли очередь полностью  if queue.full():  # ... | |
| Контроль выполнения задач в очереди: Метод task\_done() вызывается после обработки элемента очереди. Он указывает, что обработка элемента очереди завершена. При вызове этого метода счетчик, который был инкрементирован методом put(), декрементируется. Когда счетчик достигнет нуля (т.е. все элементы очереди будут обработаны), метод join() разблокирует выполнение программы. Обычно этот метод вызывается внутри цикла while, который обрабатывает элементы очереди. Пример использования:  while True:  # получить элемент из очереди  item = await queue.get()  process\_item(item)  # оповестить очередь, что обработка задачи завершена  queue.task\_done()  Другие корутины могут быть заинтересованы в том, чтобы знать, когда все элементы, добавленные в очередь, были извлечены и помечены как выполненные, по этому task\_done() может быть весьма полезен.  # ждем, пока все элементы в очереди будут отмечены как выполненные  await queue.join()  Если очередь пуста или все элементы уже отмечены как выполненные, то корутина join() вернется немедленно. | |
| FIFO Example 1 import asyncio   async def producer(queue):   for i in range(5):  item = f'Элемент {i}'   await queue.put(item)   print(f'producer добавил "в" очередеь элемент: {item}')    async def consumer(queue):   while True:  item = await queue.get()   print(f'consumer получил "из" очереди элемент: {item}')  if item is None:   break   # вывод сообщения о получении элемента из очереди   async def main():   queue = asyncio.Queue()    # создание задачи для функции producer и consumer  prod\_task = asyncio.create\_task(producer(queue))  cons\_task = asyncio.create\_task(consumer(queue))  await prod\_task   await queue.put(None)   await cons\_task    asyncio.run(main()) | |
| FIFO Example 2 import asyncio  async def read\_queue(queue):   while True:   item = await queue.get()   print("Получен элемент из очереди:", item)    async def main():   queue = asyncio.Queue()   asyncio.create\_task(read\_queue(queue))   await queue.put("Первый элемент")   await queue.put("Второй элемент")    asyncio.run(main()) | |
| FIFO Example 3 import asyncio   async def producer(queue):   for i in range(5):  item = f'Элемент {i}'   await queue.put(item)   print(f'producer добавил "в" очередеь элемент: {item}')    async def consumer(queue):   while True:  item = await queue.get()   print(f'consumer получил "из" очереди элемент: {item}')  if item is None:   break   # вывод сообщения о получении элемента из очереди   async def main():   queue = asyncio.Queue()    # создание задачи для функции producer и consumer  prod\_task = asyncio.create\_task(producer(queue))  cons\_task = asyncio.create\_task(consumer(queue))  await prod\_task   await queue.put(None)   await cons\_task    asyncio.run(main()) | |
| Очередь LIFO asyncio.LifoQueue asyncio.LifoQueue **LIFO** (**Last In, First Out, "**последним пришел, первым ушел") - является реализацией очереди в асинхронном коде. Это означает, что последний элемент, добавленный в очередь, будет первым, который будет из этой очереди извлечен. Это класс, который представляет собой структуру данных типа "стек", которая используется для организации очереди задач в асинхронной программе. | |
| Для использование очереди необходимо создать экземпляр класса asyncio.LifoQueue(), после этого станут доступны все методы, LifoQueue() принимает один аргумент конструктора «maxsize», по умолчанию установленный на ноль (без ограничений).  lifo\_queue = asyncio.LifoQueue() | |
| Методы:  * lifo\_queue**.put(item)** - добавляет элемент item в очередь. Если очередь заполнена, блокирует поток до тех пор, пока место не освободится. * lifo\_queue**.get()** - удаляет и возвращает последний добавленный элемент из очереди. Если очередь пуста, блокирует поток до тех пор, пока элемент не станет доступным. * lifo\_queue**.qsize()** - возвращает количество элементов в очереди. * lifo\_queue**.empty()** - возвращает True, если очередь пуста, и False в противном случае. * lifo\_queue**.full()** - возвращает True, если очередь заполнена, и False в противном случае. * lifo\_queue**.task\_done()** - используется в паре с методом .get(), чтобы указать, что задача была завершена. * lifo\_queue**.join()** - блокирует поток, пока не будут завершены все задачи в очереди. * lifo\_queue**.get\_nowait()** - удаляет и возвращает последний добавленный элемент из очереди. Если очередь пуста, этот метод не блокирует поток в ожидании момента, когда в очередь будет помещен новый элемент, а вызывает исключение asyncio.QueueEmpty. Не является корутиной, при использовании этого метода использовать ключевое слово await не нужно. * lifo\_queue**.put\_nowait(item)** - добавляет элемент item в конец стека. Если стек заполнен, то вызывается исключение QueueFull. Этот метод не блокирует поток, то есть если стек заполнен, то он не будет ждать, пока элементы освободятся. Не является корутиной, при использовании этого метода использовать ключевое слово await не нужно.   Также, стоит отметить, что методы put() и get() являются корутинами, т.е. они могут быть вызваны только в асинхронной функции с помощью ключевого слова await. | |
| Когда использовать очередь LIFO? Пять распространенных примеров включают в себя:   * **Возврат назад** - сохранение списка вариантов в порядке их появления или доступности, на случай, если текущий вариант не сработает. * **Отмена изменений** - сохранение изменений объекта, чтобы их можно было отменить или вернуть в порядке их применения. * **Поиск в глубину** - сохранение списка узлов при обходе дерева или графа в порядке поиска в глубину. * **Парсеры** - сохранение списка выражений в порядке их выполнения. * **Свежесть** - сохранение списка подключений или данных в обратном порядке их использования, чтобы можно было получить самые последние, когда это нужно. | |
| Как использовать LifoQueue Можно добавлять в очередь с помощью метода put(). На самом деле put() это корутина, которую нужно дождаться. Проблема может быть в том, что вызывающая корутина может заблокироваться, если очередь заполнена.  # добавляем объект в очередь  await lifo\_queue.put(item) | |
| Элемент также можно добавить в очередь без блокировки с помощью метода put\_nowait(). Этот метод не является корутиной и либо добавит элемент, либо немедленно вернет результат, либо завершится ошибкой с исключением asyncio.QueueFull, если очередь заполнена и элемент не может быть добавлен.  ...  try:  # попытка добавить элемент  lifo\_queue.put\_nowait(item)  except asyncio.QueueFull:  # ... | |
| Получение элемента из LifoQueue(): Элементы можно получить из очереди, вызвав метод get(). Этот метод является корутиной, которую нужно ждать. Причина в том, что в данный момент в очереди может не быть элементов для извлечения, и вызывающей корутине может потребоваться блокировка до тех пор, пока элемент не станет доступным.  # получаем элемент из очереди  item = await lifo\_queue.get()  Это вернет последний добавленный элемент, а не первые добавленные элементы, например. Помним про принцип работы **LIFO**. Так же, элемент можно извлечь из очереди без блокировки с помощью метода get\_nowait(). Т.к. этот метод не является корутиной и немедленно вернет элемент, если он доступен, в противном случае произойдет сбой с исключением asyncio.QueueEmpty.  ...  try:  # попытка получить элемент  item = lifo\_queue.get\_nowait()  except asyncio.QueueEmpty:  # ... | |
| Запрос размера очереди LifoQueue(): Мы можем получить фиксированный размер очереди через свойство «maxsize».  # сообщить размер очереди  print(lifo\_queue.maxsize) | |
| пуста ли очередь Мы можем проверить, пуста ли очередь с помощью empty(), которая возвращает True, если очередь не содержит элементов, или False в противном случае.  ...  # проверить, пуста ли очередь  if lifo\_queue.empty():  # ... | |
| заполнена ли очередь Мы также можем проверить, заполнена ли очередь с помощью метода full(), который возвращает True, если очередь заполнена, или False в противном случае.  ...  # проверить заполнена ли очередь  if lifo\_queue.full():  # ... | |
| Добавление задач в очередь LifoQueue и их выполнение: Элементы в очереди можно рассматривать как задачи, которые могут быть помечены как процессы потребительскими корутинами.  Это может быть сделано с помощью потребительских корутин, извлекающих элементы из очереди с помощью get() или get\_nowait() и после обработки помечающих их с помощью метода task\_done(). | |
| ...  # получить элемент из очереди  item = await lifo\_queue.get()  # обработать элемент  # ...  # пометить элемент как заверёшнный процесс/задача  lifo\_queue.task\_done()  Другие корутины могут быть заинтересованы в том, чтобы знать, когда все элементы, добавленные в очередь, были извлечены и помечены как выполненные.  Это может быть сделано с помощью ожидающей в очереди корутины join().  Метод join() не разблокирует поток, пока все элементы, добавленные в очередь до вызова, не будут отмечены как выполненные.  ...  # подождать, пока все элементы в очереди будут отмечены как выполненные  await lifo\_queue.join()  Если очередь пуста или все элементы уже отмечены как выполненные, присоединенная корутина вернется немедленно. | |
| Примеры кода с использованием LifoQueue():Пример1 import asyncio  async def lifoqueue\_example():  lifo\_queue = asyncio.LifoQueue(maxsize=3)    # добавление элементов в очередь  await lifo\_queue.put("Первый элемент")  await lifo\_queue.put("Второй элемент")  await lifo\_queue.put("Третий элемент")   last\_item = await lifo\_queue.get()   print(last\_item)   second\_last\_item = await lifo\_queue.get()   print(second\_last\_item)   first\_item = await lifo\_queue.get()   print(first\_item)   asyncio.run(lifoqueue\_example()) | |
| Пример2 import asyncio   async def worker(name, lifo\_queue):  while True:  task = await lifo\_queue.get() #  print(f"Рабочий\_{name}. получил задачу: {task}")  await asyncio.sleep(0.5) #  print(lifo\_queue.qsize())  # if name != "1":  lifo\_queue.task\_done() #   async def main():  lifo\_queue = asyncio.LifoQueue()   for i in range(10):  await lifo\_queue.put(i)    workers = [asyncio.create\_task(worker(f"{i}", lifo\_queue)) for i in range(3)]  await lifo\_queue.join()   for w in workers:   w.cancel()   if \_\_name\_\_ == "\_\_main\_\_":  asyncio.run(main()) | |
| Example3 import asyncio  # При выводе в консоль можно использовать ANSI коды RED = '\033[31m' YELLOW = '\033[33m' GREEN = '\033[32m' END = '\033[0m' # Возврат к настройкам   async def customer(queue):  # Цикл работает до опустошения очереди  while not queue.empty():  await asyncio.sleep(.5)  elem = queue.get\_nowait()  print(f'Из очереди получен элемент\_{elem}')   print(f'{YELLOW}Очередь опустошена{END}')  # Применяем метод get\_nowait() для вызова исключения asyncio.QueueEmpty  try:  elem = queue.get\_nowait()  print(f'Из очереди получен элемент\_{elem}') # Принт не сработает  except asyncio.QueueEmpty:  print(f'{RED}Попытка получить элемент из пустой очереди методом get\_nowait() вызвала asyncio.QueueEmpty{END}')   async def producer(queue):  for i in range(5):  await asyncio.sleep(.5)  queue.put\_nowait(i)  print(f'В очередь поставлен элемент\_{i}')  if queue.full(): # Если очередь заполнена  print(f'{YELLOW}Очередь заполнена{END}')  try:  # Применяем метод put\_nowait() для вызова исключения asyncio.QueueFull  queue.put\_nowait(i + 1)  print(f'В очередь поставлен элемент\_{i}') # Принт не сработает  except asyncio.QueueFull:  print(  f'{RED}Попытка поместить элемент в заполненную очередь методом put\_nowait() вызвала asyncio.QueueFull{END}')  print(f'{GREEN}Запускаем процесс получения элементов из очереди{END}')  # Приостанавливаем producer() и ожидаем выполнения customer()  await customer(queue)   async def main():  # Создаем очередь с maxsize == 5  queue = asyncio.LifoQueue(5)  print(f'{GREEN}Емкость созданной очереди: {queue.maxsize} элементов {END}')  prod = asyncio.create\_task(producer(queue))  # Приостанавливаем текущую задачу и ожидаем выполнения prod  await prod  print('End')   asyncio.run(main()) | |
| Используйте **put()** и **get()**:  * Когда вы работаете в асинхронном контексте и хотите позволить корутине ожидать добавления или извлечения элемента. * Когда у вас есть асинхронные операции, которые могут работать параллельно с ожиданием добавления или извлечения элемента. * Когда вы хотите избежать блокировки других корутин во время ожидания. | |
| Используйте put\_nowait() и get\_nowait():  * Когда вы работаете в синхронном контексте и не можете (или не хотите) использовать асинхронное программирование. * Когда вы хотите немедленно добавить или извлечь элемент, не дожидаясь возможности это сделать. * Когда вы готовы обрабатывать исключения, возникающие при попытке добавить элемент в полную очередь или извлечь элемент из пустой очереди.   Следует помнить, что методы put\_nowait() и get\_nowait() не являются корутинами, awaitдля их вызова применять не нужно! | |
| Приоритетная очередь asyncio.PriorityQueue() asyncio.PriorityQueue() - реализует приоритетную очередь. Эта разновидность Queue извлекает записи в порядке приоритета (наименьшая - первая).  Записи обычно представляют собой кортежи вида (номер\_приоритета, данные).  asyncio.PriorityQueue() упорядочивает элементы в очереди на основе их приоритета. Приоритет каждого элемента определяется числовым значением. Когда элемент добавляется в очередь, он помещается в правильную позицию на основе его приоритета.  Для использование очереди необходимо создать экземпляр класса asyncio.PriorityQueue(), после этого станут доступны все методы, класс принимает один аргумент конструктора «maxsize», по умолчанию установленный на ноль (без ограничений). Но её нельзя использовать вне программы asyncio.  priority\_queue = asyncio.PriorityQueue(maxsize=10) | |
| asyncio.PriorityQueue() методы:  * priority\_queue.put(item) - используется для добавления элемента в очередь с приоритетами. item обычно представлен в форме кортежа (priority, value), где priority — это число, определяющее приоритет элемента, а value — это значение элемента (элементы с более низким значением priority обладают более высоким приоритетом). Когда очередь не полная, put(item) добавляет элемент в очередь сразу же. Если очередь полная (т.е., достигнут максимальный размер очереди, если он установлен), корутина put(item) будет ждать (асинхронно блокироваться), пока в очереди не появится свободное место, прежде чем добавить элемент. * priority\_queue.get() - используется для извлечения элементов из очереди с приоритетами. Он извлекает элемент с наивысшим приоритетом. Если очередь пуста, метод блокирует выполнение до появления новых элементов. * priority\_queue.empty() - используется для проверки, пуста ли очередь с приоритетами. * priority\_queue.full() - используется для проверки, заполнена ли очередь с приоритетами до максимального размера. * priority\_queue.put\_nowait(priority\_number, item) - используется для добавления элемента в очередь с приоритетами без блокировки выполнения программы. Если очередь заполнена до максимального размера, метод вызовет исключение asyncio.Queue**Full**. * priority\_queue.get\_nowait() - используется для извлечения элементов из очереди с приоритетами без блокировки выполнения программы. Если очередь пуста, метод вызовет исключение asyncio.Queue**Empty**. * priority\_queue.join() - используется для блокировки выполнения программы до тех пор, пока все элементы очереди не будут извлечены и обработаны. * priority\_queue.task\_done() - используется для уведомления очереди о том, что элемент был обработан. Каждый вызов метода get() должен быть сопровожден вызовом метода task\_done(), чтобы join() мог корректно завершить выполнение. * priority\_queue.qsize() - используется для получения количества элементов в очереди с приоритетами. | |
| Добавление элементов в очередь: Для добавления элемента в очередь с приоритетами используется метод put(). Он принимает два аргумента: его приоритет и элемент. Приоритет должен быть числом.  await priority\_queue.put((1, "Первый элемент"))  await priority\_queue.put((2, "Второй элемент"))  await priority\_queue.put((3, "Третий элемент")) | |
| Извлечение элементов из очереди: Для извлечения элементов из очереди с приоритетами используется метод get(). Он извлекает элемент с наивысшим приоритетом. Если очередь пуста, метод блокирует выполнение до появления новых элементов.  first\_element = await priority\_queue.get()  second\_element = await priority\_queue.get()  third\_element = await priority\_queue.get() | |
| Возможные проблемы: При использовании asyncio.PriorityQueue() можно столкнуться с проблемами, связанными с блокировкой выполнения программы. Если очередь с приоритетами заполнена до максимального размера и свободных мест в ней нет, метод put() блокирует выполнение программы до освобождения места в очереди. Аналогично, если очередь пуста и мы вызываем метод get(), выполнение программы будет заблокировано до появления новых элементов в очереди. | |
| Example import asyncio   async def task(name, priority, queue):   await queue.put((priority, name))   print(f"{name}, добавлен в очередь, с приоритетом {priority}")    async def main():   queue = asyncio.PriorityQueue(maxsize=10)   await task("Задача 1", 1, queue)   await task("Задача 2", 3, queue)   await task("Задача 3", 2, queue)   await task("Задача 4", 2, queue)   while not queue.empty():   priority, name = await queue.get()   print(f"{name}, с приоритетом {priority} выполняется")   asyncio.run(main()) | |
| Базовые примитивы синхронизации asyncio | |
| Введение в примитивы Во время изучения и работы в asyncio вы столкнетесь с таким понятием, как "примитивы синхронизации". Это ключевые инструменты, которые помогают организовать корректное взаимодействие между асинхронными задачами. В данном степе мы кратко рассмотрим эти примитивы и дадим примеры их использования, а более подробно разберём дальше в курсе.  **Базовые примитивы синхронизации**помогают решать различные задачи синхронизации, такие как контроль доступа к общим ресурсам, синхронизация задач и другие.   1. asyncio.**Lock()** - механизм, обеспечивающий монопольный доступ к общему ресурсу. Используется для предотвращения одновременного доступа к общему ресурсу из нескольких корутин.   import asyncio  # Создаем экземпляр Lock lock = asyncio.Lock()  async def my\_coroutine(id):  print(f'Корутина {id} хочет получить блокировку')  async with lock:  # Код внутри этого блока будет выполняться только одной корутиной в один момент времени  print(f'Корутина {id} получила блокировку')  await asyncio.sleep(1)  print(f'Корутина {id} отпустила блокировку')  # Запускаем несколько корутин async def main():  await asyncio.gather(my\_coroutine(1), my\_coroutine(2), my\_coroutine(3))  asyncio.run(main())   # Корутина 1 хочет получить блокировку # Корутина 1 получила блокировку # Корутина 2 хочет получить блокировку # Корутина 3 хочет получить блокировку # Корутина 1 отпустила блокировку # Корутина 2 получила блокировку # Корутина 2 отпустила блокировку # Корутина 3 получила блокировку # Корутина 3 отпустила блокировку | |
| 1. asyncio**.Event()** - объект, позволяющий организовать доступ к ресурсу на основе заданных событий. Он представляет собой сигнальный флаг, который может быть установлен или сброшен, и задачи могут ожидать этого события и продолжить выполнение после его возникновения.   import asyncio  # Создаем экземпляр Event event = asyncio.Event()  async def waiter():  print('waiter: ожидаем события')  await event.wait()  print('waiter: событие произошло')  async def setter():  print('setter: засыпаем')  await asyncio.sleep(1)  print('setter: просыпаемся и устанавливаем событие')  event.set()  async def main():  # Запускаем waiter и setter параллельно  await asyncio.gather(waiter(), setter())  asyncio.run(main()) | |
| 1. asyncio**.Condition()** - комбинирует функциональность Lock() и Event(). Это объект условия, который позволяет задачам ожидать определенного состояния или сигнала, чтобы продолжить выполнение. Условие может быть связано с определенным Lock(), и задачи могут ожидать его установки и освобождения.   import asyncio  # Создаем экземпляр Condition condition = asyncio.Condition()  async def consumer(condition):  async with condition:  print('consumer: ждем условия')  await condition.wait()  print('consumer: условие выполнено')  async def producer(condition):  print('producer: засыпаем')  await asyncio.sleep(1)  print('producer: просыпаемся и устанавливаем условие')  async with condition:  condition.notify\_all()  async def main():  # Запускаем consumer и producer параллельно  await asyncio.gather(consumer(condition), producer(condition))  asyncio.run(main())   # consumer: ждем условия # producer: засыпаем # producer: просыпаемся и устанавливаем условие # consumer: условие выполнено | |
| 1. asyncio**.Semaphore()** - семафорный объект, который управляет внутренним счетчиком. Он позволяет ограничить доступ к общему ресурсу заданному количеству задач. Задачи могут запрашивать доступ к ресурсу, и если счетчик не превышает заданное значение, доступ будет предоставлен и счетчик будет увеличен.   import asyncio  # Создаем экземпляр Semaphore с максимум двумя разрешениями import random  semaphore = asyncio.Semaphore(2)  async def my\_coroutine(id):  print(f'Корутина {id} хочет получить семафор')  async with semaphore:  # Код внутри этого блока будет выполняться только двумя корутинами в один момент времени  print(f'Корутина {id} получила семафор')  await asyncio.sleep(random.randint(1,5))  print(f'Корутина {id} отпустила семафор')  # Запускаем несколько корутин async def main():  await asyncio.gather(my\_coroutine(1), my\_coroutine(2), my\_coroutine(3))  asyncio.run(main())  # Корутина 1 хочет получить семафор # Корутина 1 получила семафор # Корутина 2 хочет получить семафор # Корутина 2 получила семафор # Корутина 3 хочет получить семафор # Корутина 2 отпустила семафор # Корутина 3 получила семафор # Корутина 1 отпустила семафор # Корутина 3 отпустила семафор | |
| 1. asyncio**.BoundedSemaphore()** - ограниченная версия семафора. Он ограничивает внутренний счетчик, чтобы не превышал заданное значение. Вся остальная функциональность аналогична Semaphore().   работает так же, как и обычный Semaphore, но еще он не позволяет увеличить счетчик вызовами release() выше исходного значения.  import asyncio  # Создаем экземпляр BoundedSemaphore с максимум двумя разрешениями semaphore = asyncio.BoundedSemaphore(2)  async def my\_coroutine(id):  print(f'Корутина {id} хочет получить семафор')  async with semaphore:  # Код внутри этого блока будет выполняться только двумя корутинами в один момент времени  print(f'Корутина {id} получила семафор')  await asyncio.sleep(1)  print(f'Корутина {id} отпустила семафор')  # Запускаем несколько корутин async def main():  await asyncio.gather(my\_coroutine(1), my\_coroutine(2), my\_coroutine(3))  asyncio.run(main())  # Корутина 1 хочет получить семафор # Корутина 1 получила семафор # Корутина 2 хочет получить семафор # Корутина 2 получила семафор # Корутина 3 хочет получить семафор # Корутина 1 отпустила семафор # Корутина 2 отпустила семафор # Корутина 3 получила семафор # Корутина 3 отпустила семафор | |
| 1. asyncio**.Barrier()** - это простой примитив синхронизации, позволяющий блокировать выполнение до тех пор, пока на нем не будет ожидать определенное количество задач. Задачи могут ожидать метод wait() и будут блокироваться до тех пор, пока указанное количество задач не закончит ожидать wait(). В этот момент все ожидающие задачи одновременно разблокируются. В качестве альтернативы ожиданию в wait() может использоваться async with.   import asyncio import random   async def worker(barrier, worker\_num):  print(f'Работник {worker\_num} начал свою работу')  await asyncio.sleep(random.randint(1,4)) # Имитируем выполнение работы  print(f'Работник {worker\_num} дошел до барьера и ожидает остальных')  await barrier.wait() # Ожидаем, пока все работники не завершат свою работу  print(f'Все работники дошли до барьера. Работник {worker\_num} продолжает выполнение.')  async def main():  barrier = asyncio.Barrier(3) # Создаем барьер для 3х работников  await asyncio.gather(  worker(barrier, 1),  worker(barrier, 2),  worker(barrier, 3),  )  asyncio.run(main())  # Работник 1 начал свою работу # Работник 2 начал свою работу # Работник 3 начал свою работу # Работник 2 дошел до барьера и ожидает остальных # Работник 1 дошел до барьера и ожидает остальных # Работник 3 дошел до барьера и ожидает остальных # Все работники дошли до барьера. Работник 3 продолжает выполнение. # Все работники дошли до барьера. Работник 2 продолжает выполнение. # Все работники дошли до барьера. Работник 1 продолжает выполнение. | |
| 1. asyncio**.Queue()** - объект очереди, который обеспечивает безопасный и упорядоченный обмен данными между асинхронными задачами. Задачи могут помещать элементы в очередь и извлекать их в порядке, в котором они были помещены. Очередь также может быть ограничена по размеру, чтобы ограничить потребление ресурсов.   import asyncio  # Корутина производителя async def producer(queue):  for i in range(5):  await queue.put(i)  print(f'Производитель: Положил в очередь элемент {i}') #   await asyncio.sleep(1) #   # Корутина потребителя async def consumer(queue):  while True:  item = await queue.get()  print(f'Потребитель: Получил элемент из очереди {item}')   await asyncio.sleep(0.5)   queue.task\_done()   # Основная корутина async def main():  queue = asyncio.Queue()   task\_producer = asyncio.create\_task(producer(queue))   task\_consumer = asyncio.create\_task(consumer(queue))   await asyncio.gather(task\_producer, task\_consumer)   # Запускаем асинхронную программу asyncio.run(main())    # Производитель: Положил в очередь элемент 0 # Потребитель: Получил элемент из очереди 0 # Производитель: Положил в очередь элемент 1 # Потребитель: Получил элемент из очереди 1 # Производитель: Положил в очередь элемент 2 # Потребитель: Получил элемент из очереди 2 # Производитель: Положил в очередь элемент 3 # Потребитель: Получил элемент из очереди 3 # Производитель: Положил в очередь элемент 4 # Потребитель: Получил элемент из очереди 4 | |
| Защита ресурса от одновременного доступа asyncio.Lock() asyncio.Lock() - является классом, предназначенным для управления синхронизацией. Он используется для защиты ресурсов от одновременного доступа и использования несколькими потоками или корутинами.  asyncio.Lock() - реализует концепцию блокировки, которая гарантирует, что в любой момент времени только один поток или корутина может использовать защищаемый ресурс. Он может быть использован для синхронизации доступа к базам данных, файлам или любым другим общим ресурсам.  Для работы с asyncio.Lock() необходимо создать экземпляр класса:  lock = asyncio.Lock()  Затем можно использовать методы lock.acquire() , lock.release() и lock**.**locked()для управления блокировкой.   * Метод lock**.acquire()** асинхронно запрашивает блокировку. Если блокировка уже удерживается другой корутиной, текущая корутина будет приостановлена (а не заблокирована в традиционном смысле) до тех пор, пока блокировка не станет доступной. Как только блокировка освобождается, она предоставляется одной из ожидающих корутин, которая тем самым получает возможность безопасно использовать защищаемый ресурс. * Метод lock**.release()** освобождает блокировку, чтобы другой поток или корутина могли запросить ее. * Метод lock**.locked()** возвращает значение True, если блокировка в текущий момент захвачена, и False в противном случае. | |
| Особенности использования: При использовании asyncio.Lock() необходимо убедиться, что блокировка всегда освобождается, иначе это может привести к блокировке всего приложения. Для этого можно использовать конструкцию try/finally | |
| Example1 async def some\_coroutine():  await lock.acquire()  try:  # код, который требует эксклюзивного доступа к общим ресурсам  finally:  lock.release() | |
| Example2 await lock.acquire() import asyncio  # Создание объекта Lock lock = asyncio.Lock()  # Определение корутины "my\_task" с аргументом "task\_id" async def my\_task(task\_id):   # Вывод сообщения о том, что задача ждет блокировки  print(f"Задача {task\_id} ожидает блокировки с помощью Lock")   # Ожидание получения блокировки  await lock.acquire()   try:  # Вывод сообщения о том, что задача получила блокировку  print(f"Задача {task\_id} получила блокировку")   # Остановка выполнения корутины на 2 секунды  await asyncio.sleep(2)   finally:  # Вывод сообщения о том, что задача освобождает блокировку  print(f"Задача {task\_id} блокировка снята")   # Освобождение блокировки  lock.release()  # Определение главной корутины "main" async def main():   # Создание списка задач из 5 вызовов "my\_task"  tasks = [asyncio.create\_task(my\_task(i)) for i in range(5)]   # Ожидание выполнения всех задач  await asyncio.gather(\*tasks)  # Запуск главной корутины asyncio.run(main())  # Задача 0 ожидает блокировки с помощью Lock # Задача 0 получила блокировку # Задача 1 ожидает блокировки с помощью Lock # Задача 2 ожидает блокировки с помощью Lock # Задача 3 ожидает блокировки с помощью Lock # Задача 4 ожидает блокировки с помощью Lock # Задача 0 блокировка снята # Задача 1 получила блокировку # Задача 1 блокировка снята # Задача 2 получила блокировку # Задача 2 блокировка снята # Задача 3 получила блокировку # Задача 3 блокировка снята # Задача 4 получила блокировку # Задача 4 блокировка снята | |
| Example 3 async with lock: import asyncio  # Создание объекта Lock lock = asyncio.Lock()  # Определение корутины "my\_task" с аргументом "task\_id" async def my\_task(task\_id):   # Вывод сообщения о том, что задача ждет блокировки  print(f"Задача {task\_id} ожидает блокировки с помощью Lock")   # Запускаем менеджер контекста with с Lock  async with lock:  # Вывод сообщения о том, что задача получила блокировку  print(f"Задача {task\_id} получила блокировку")   # Остановка выполнения корутины на 2 секунды  await asyncio.sleep(2)   # Вывод сообщения о том, что задача освобождает блокировку  print(f"Задача {task\_id} блокировка снята")   # Определение главной корутины "main" async def main():   # Создание списка задач из 5 вызовов "my\_task"  tasks = [asyncio.create\_task(my\_task(i)) for i in range(5)]   # Ожидание выполнения всех задач  await asyncio.gather(\*tasks)  # Запуск главной корутины asyncio.run(main())  # Задача 0 ожидает блокировки с помощью Lock # Задача 0 получила блокировку # Задача 1 ожидает блокировки с помощью Lock # Задача 2 ожидает блокировки с помощью Lock # Задача 3 ожидает блокировки с помощью Lock # Задача 4 ожидает блокировки с помощью Lock # Задача 0 блокировка снята # Задача 1 получила блокировку # Задача 1 блокировка снята # Задача 2 получила блокировку # Задача 2 блокировка снята # Задача 3 получила блокировку # Задача 3 блокировка снята # Задача 4 получила блокировку # Задача 4 блокировка снята | |
| Пример 4 import asyncio  # Объявляем глобальную переменную counter со значением 0 counter = 0  # Создаем объект lock для управления доступом к counter lock = asyncio.Lock()   async def worker\_1():   # Объявляем использование глобальной переменной counter  global counter   # Захватываем lock, чтобы исключить конкурентный доступ к counter  async with lock:   # В цикле инкрементируем counter и выводим сообщение  for i in range(10):  counter += 1  print(f"Переменная увеличена на 1 из корутины worker\_1, counter = {counter}")   # Останавливаем выполнение на 1 секунду  await asyncio.sleep(1)   async def worker\_2():   global counter  async with lock:  for i in range(10):  counter += 1  print(f"Переменная увеличена на 1 из корутины worker\_2, counter = {counter}")  await asyncio.sleep(1)   async def main():   task1 = asyncio.create\_task(worker\_1())   task2 = asyncio.create\_task(worker\_2())   await task1   await task2  # Запускаем основную функцию main с помощью asyncio.run asyncio.run(main())   # Переменная увеличена на 1 из корутины worker\_1, counter = 1 # Переменная увеличена на 1 из корутины worker\_1, counter = 2 # ... # Переменная увеличена на 1 из корутины worker\_1, counter = 9 # Переменная увеличена на 1 из корутины worker\_1, counter = 10 # Переменная увеличена на 1 из корутины worker\_2, counter = 11 # Переменная увеличена на 1 из корутины worker\_2, counter = 12 # .... # Переменная увеличена на 1 из корутины worker\_2, counter = 19 # Переменная увеличена на 1 из корутины worker\_2, counter = 20 | |
| Пример 4 lock not used import asyncio  counter = 0 lock = asyncio.Lock()  async def worker\_1():   # Объявляем использование глобальной переменной counter  global counter   # Захватываем lock, чтобы исключить конкурентный доступ к counter  # async with lock:   # В цикле инкрементируем counter и выводим сообщение  for i in range(10):  counter += 1  print(f"Переменная увеличена на 1 из корутины worker\_1, counter = {counter}")   # Останавливаем выполнение на 1 секунду  await asyncio.sleep(1)   async def worker\_2():   global counter  # async with lock:  for i in range(10):  counter += 1  print(f"Переменная увеличена на 1 из корутины worker\_2, counter = {counter}")  await asyncio.sleep(1)   async def main():   task1 = asyncio.create\_task(worker\_1())  task2 = asyncio.create\_task(worker\_2())  await task1  await task2   asyncio.run(main())   # Переменная увеличена на 1 из корутины worker\_1, counter = 1 # Переменная увеличена на 1 из корутины worker\_2, counter = 2 # Переменная увеличена на 1 из корутины worker\_1, counter = 3 # Переменная увеличена на 1 из корутины worker\_2, counter = 4 # Переменная увеличена на 1 из корутины worker\_1, counter = 5 # ... | |
| Пример5 Bank account import asyncio  # Устанавливаем начальный баланс balance = 100  # Создаем объект lock для синхронизации lock = asyncio.Lock()  # Определяем асинхронную функцию пополнения счета async def deposit(amount):   # Объявляем переменную balance как глобальную  global balance   # Используем lock, чтобы защитить доступ к переменной balance  async with lock:   # Выводим сообщение о пополнении счета  print(f"Баланс пополнен на {amount} у.е.")   # Пополняем баланс  balance += amount   # Выводим текущий баланс  print(f"Текущий баланс {balance}")  # Определяем асинхронную функцию снятия средств async def withdraw(amount):   # Объявляем переменную balance как глобальную  global balance   # Используем lock, чтобы защитить доступ к переменной balance  async with lock:   # Проверяем, хватает ли средств для снятия  if balance >= amount:   # Выводим сообщение о снятии средств  print(f"Снятие {amount} у.е.")   # Снимаем средства  balance -= amount   #Выводим текущий баланс  print(f"Текущий баланс {balance}")  else:  # Выводим сообщение о недостаточности средств  print(f"Попытка снять {amount}, недостаточно средств, текущий баланс {balance} у.е.")  # Определяем основную асинхронную функцию async def main():   # Создаем задачу для функции deposit и пополняем баланс на 50 у.е.  task1 = asyncio.create\_task(deposit(50))   # Создаем задачу для функции withdraw и пытаемся списать с баланса 200 у.е.  task2 = asyncio.create\_task(withdraw(200))   # Ожидаем завершения задачи task1  await task1   # Ожидаем завершения задачи task2  await task2  # Запускаем основную асинхронную функцию asyncio.run(main())  # Баланс пополнен на 50 у.е. # Текущий баланс 150 # Попытка снять 200, недостаточно средств, текущий баланс 150 у.е. | |
| Ожидание событий для синхронизации доступа asyncio.Event() asyncio.Event() используется для синхронизации между задачами, чтобы управлять последовательностью выполнения задач в асинхронном коде. Например, если одна задача должна выполнить какую-то работу, а другая задача должна ждать ее завершения, то вы можете использовать asyncio.Event() для уведомления второй задачи о том, что первая задача завершила работу. Объект Event управляет внутренним флагом, который может быть установлен в True с помощью метода set() и сброшен в False с помощью метода clear(). Метод wait() блокирует работу до тех пор, пока флаг не будет установлен в True. Изначально флаг устанавливается в False.  Чтобы использовать asyncio.Event(), вы должны сначала создать его экземпляр:  event = asyncio.Event()  Затем можно использовать методы event.set() , event.clear(), event.wait()и event.is\_set() для управления событием Event.   * event**.set()** - устанавливает флаг события в True. Это означает, что все задачи, которые вызывают метод wait(), прекратят блокирование и продолжат выполнение. * event**.clear()** - устанавливает флаг события в False. Это означает, что все задачи, которые вызывают метод wait(), будут заблокированы, пока флаг не будет установлен в True. * event**.wait()** - блокирует выполнение задачи, пока флаг события не будет установлен в True. Как только флаг будет установлен в True, метод wait() вернет управление и задача продолжит выполнение. В противном случае блокируется до тех пор, пока другая задача не вызовет set().  wait() может привести к проблемам с производительностью, если событие которое ожидает wait() долго не наступает, поэтому использовать его необходимо с осторожностью. * event**.is\_set()** - возвращает True, если флаг события установлен в True, и False в противном случае. Это полезно для проверки состояния флага, не блокируя выполнение задачи. | |
| Example 1 import asyncio  # Создаем экземпляр события event = asyncio.Event()   # Определяем корутину для ожидания события async def wait\_for\_event():  await asyncio.sleep(1)  # Выводим сообщение о начале ожидания события  print('Ждём события')  # Ожидаем событие  await event.wait()   # Выводим сообщение о получении события  print('Событие получено')   # Определяем корутину для установки события async def set\_event():  await asyncio.sleep(2)  # Выводим сообщение о начале установки события  print('Установка события')   # Устанавливаем событие  event.set()   # Определяем главную корутину async def main():  # Создаем задачи для корутин wait\_for\_event и set\_event  task1 = asyncio.create\_task(wait\_for\_event())  task2 = asyncio.create\_task(set\_event())   # Ожидаем завершения обеих задач  await asyncio.gather(task1, task2)   # Запускаем главную корутину asyncio.run(main())  # Ждём события # Установка события # Событие получено | |
| Example 2 import asyncio  # Определение функции worker, которая принимает на вход список чисел и событие async def worker(numbers, event):   # Цикл по числам в списке  for number in numbers:   # Если число равно 15, устанавливаем событие  if number == 15:  event.set()   # Вывод на экран информации о поиске события  print(f"Поиск события: Обрабатываемое число {number}")   # Остановка на 0.3 секунды для эмуляции задержки  await asyncio.sleep(0.3)  # Определение функции manager, которая принимает событие на вход async def manager(event):   # Вывод на экран сообщения о ожидании события  print("Ожидаем событие")   # Ожидание события  await event.wait()   # Вывод на экран сообщения о свершении события  print("Событие свершилось")  # Определение функции main async def main():   # Создание списка чисел от 1 до 30  numbers = range(1, 30)   # Создание события  event = asyncio.Event()   # Запуск обоих корутин (worker и manager) и ожидание их завершения  await asyncio.gather(worker(numbers, event), manager(event))   # Запуск функции main с помощью asyncio.run asyncio.run(main()) | |
| # Поиск события: Обрабатываемое число 1 # Ожидаем событие # Поиск события: Обрабатываемое число 2 # Поиск события: Обрабатываемое число 3 # ... # Поиск события: Обрабатываемое число 14 # Поиск события: Обрабатываемое число 15 # Событие свершилось # Поиск события: Обрабатываемое число 16 # Поиск события: Обрабатываемое число 17 # ... # Поиск события: Обрабатываемое число 27 # Поиск события: Обрабатываемое число 28 # Поиск события: Обрабатываемое число 29 | |
| Test 1 import asyncio  event = asyncio.Event()  async def sensor(n, ip):  await asyncio.sleep(n/10)  print(f'Датчик {n} IP-адрес {ip} настроен и ожидает срабатывания')  await event.wait()  print(f'Датчик {n} IP-адрес {ip} активирован, "Wee-wee-wee-wee"')  async def run\_event():  await asyncio.sleep(5)  # # наступило событие asyncio.Event()  print(f'Датчики зафиксировали движение')  event.set()  async def main():  ip : list = ["192.168.0.3", "192.168.0.1", "192.168.0.2", "192.168.0.4", "192.168.0.5"]  tasks: list = []  for i, i\_ip in enumerate(ip):  tasks.append(asyncio.create\_task(sensor(i, i\_ip)))   tasks.append(asyncio.create\_task(run\_event()))  await asyncio.gather(\*tasks)   # await asyncio.create\_task(run\_event())   asyncio.run(main())  #  # Датчик 0 IP-адрес 192.168.0.3 настроен и ожидает срабатывания # Датчик 1 IP-адрес 192.168.0.1 настроен и ожидает срабатывания # Датчик 2 IP-адрес 192.168.0.2 настроен и ожидает срабатывания # Датчик 3 IP-адрес 192.168.0.4 настроен и ожидает срабатывания # Датчик 4 IP-адрес 192.168.0.5 настроен и ожидает срабатывания # Датчики зафиксировали движение # Датчик 0 IP-адрес 192.168.0.3 активирован, "Wee-wee-wee-wee" # Датчик 1 IP-адрес 192.168.0.1 активирован, "Wee-wee-wee-wee" # Датчик 2 IP-адрес 192.168.0.2 активирован, "Wee-wee-wee-wee" # Датчик 3 IP-адрес 192.168.0.4 активирован, "Wee-wee-wee-wee" # Датчик 4 IP-адрес 192.168.0.5 активирован, "Wee-wee-wee-wee" | |
| Test2 import asyncio import random  error = None count = 0 sek = 0  async def monitor\_rocket\_launches(interrupt\_flag):  global count  global error  global sek   error\_case = [False, True]  try:  while True:  error = random.choices(error\_case, cum\_weights=(75, 25), k=1)  print(f"Мониторинг ракетных запусков... (Запуск номер {count} прошёл успешно)")  count += 1  sek += 1  await asyncio.sleep(1)  finally:  # Поместите сообщение о завершении мониторинга  "Завершение мониторинга ракетных запусков"   async def main():  global error  global count  global sek   interrupt\_flag = asyncio.Event()  # Создайте Task задачу  monitoring = asyncio.create\_task(monitor\_rocket\_launches(interrupt\_flag))  # Допишите сюда цикл  while True:  await asyncio.sleep(5)  count +=1  if count == 50:  break  if error:  print(f"Ошибка при запуске произошла на {sek} секунде =(")  print(f"Отмена мониторинга ракетных запусков...")  break   # Запустите созданную корутину в пункте 2 через await  if \_\_name\_\_ == "\_\_main\_\_":  asyncio.run(main()) | |
| Синхронизация задач asyncio.Condition() asyncio.Condition() это метод синхронизации задач, который позволяет одной задаче ожидать условия, которое должна установить другая задача. Может использоваться задачей для ожидания какого-либо события и затем получения эксклюзивного доступа к общему ресурсу.    Объект **Condition** сочетает в себе функциональность **Event**и **Lock**. Можно иметь несколько объектов **Condition**, которые делят один **Lock**, что позволяет координировать эксклюзивный доступ к общему ресурсу между различными задачами, заинтересованными в определенных состояниях этого общего ресурса.  asyncio.Condition() предлагает методы acquire() и release() и другие, которые позволяют захватывать и освобождать условие.  Чтобы использовать asyncio.Condition(), вы должны сначала создать его экземпляр:  condition = asyncio.Condition() | |
| * condition**.wait()** - этот метод блокирует текущую корутину, дожидаясь уведомления от другой корутины с помощью метода notify() или notify\_all(). Когда условие будет уведомлено, корутина, которая вызывает wait(), разблокируется и продолжит выполнение. * condition**.locked()** - этот метод возвращает True, если условие сейчас захвачено, и False в противном случае. * condition**.notify(***n=1***)** - этот метод уведомляет не более *n* задач (по умолчанию *n=1*), ожидающих выполнения этого условия. * condition**.acquire()** - этот метод захватывает условие. Это значит, что другие корутины, которые вызывают wait, будут блокированы, пока условие не будет освобождено с помощью метода release. * condition**.release()** - этот метод освобождает условие, которое было захвачено с помощью acquire. Это позволяет другим корутинам, которые ожидают условия с помощью wait, продолжить свое выполнение. * condition**.notify\_all()** - этот метод уведомляет все корутины, которые ожидают условия с помощью wait. Все корутины, которые были заблокированы, будут разблокированы и продолжат свое выполнение. * condition.wait\_for() - этот метод позволяет корутине ожидать условия, пока предикатная функция predicate не вернет True. Когда predicate вернет True, корутина, которая вызывает wait\_for, разблокируется и продолжит свое выполнение.   Важно отметить, что вы должны использовать эти методы внутри контекстного менеджера async with condition, чтобы гарантировать правильную синхронизацию. | |
| Получение и освобождение блокировки acquire() и release(). Для того, чтобы корутина могла использовать условие, она должна получать его и освободить, как блокировку mutex.  Использование блокировки, может быть установлено вручную с помощью методов acquire() и release(), или более удобный вариант через контекстный менеджер async with condition.  Получение условия через acquire() возвращает объект корутины, который требует, чтобы вызов использовал выражение await. | |
| ...  # получить условие  await condition.acquire()  # сделай что-нибудь  # ...  # снять условие  condition.release()  Альтернативой вызова методов acquire() и release() это использование асинхронного контекстного менеджера, который автоматически выполнит acquire/release самостоятельно:  ...  # получить условие  async with condition:  # сделай что-нибудь  # ... | |
| Ожидание уведомления .wait(): Как только условие получено, мы можем ждать его.  Это приостановит вызываемую корутину, пока другая корутина не оповестит цикл событий с при помощи функции notify() .  Это можно сделать с помощью метода wait(), который вернет корутину и должен быть ожидаемым,  await.  ...  # получить условие  async with condition:  # ждем уведомления  await condition.wait() | |
| Уведомить все ожидающие сопрограммы .notify\_all(): Мы можем уведомить все корутины, ожидающие выполнения условия, с помощью метода notify\_all().  ...  # получить условие  async with condition:  # уведомить все корутины, ожидающие выполнения условия  condition.notify\_all()  Корутина должна получить условие, прежде чем уведомить  другие корутины. | |
| Example import asyncio  # Объявление корутину worker async def worker(condition, msg):   # Захватываем блокировку условия  async with condition:   # Выводим сообщение о том, что корутина worker получила блокировку  print(f"worker() получил блокировку, сообщение {msg}")   # Останавливаем выполнение корутины до тех пор, пока не получит сигнал о разблокировке  await condition.wait()   # Выводим сообщение о том, что корутина worker разблокирована  print('В worker() сработал await condition.wait() и она продолжает выполнять любую логику')  print(f"worker() разблокирована, сообщение {msg}")  # Объявление корутины main async def main():   # Создаем условие  condition = asyncio.Condition()   # Создаем задачу, которая выполняет корутину worker с аргументами condition и 'task1'  task1 = asyncio.create\_task(worker(condition, 'task1'))   # Создаем задачу, которая выполняет корутину worker с аргументами condition и 'task2'  task2 = asyncio.create\_task(worker(condition, 'task2'))   # Останавливаем выполнение корутины main на 1 секунду  await asyncio.sleep(1)   # Захватываем блокировку условия  async with condition:   # Выводим сообщение о том, что корутина main получила блокировку  print("Корутина main получила блокировку")  print("Корутина main реализует любую логику приложения")   # Оповещаем все корутины, которые ждут разблокировки условия  condition.notify\_all()   # Выводим сообщение о том, что все корутины были разблокированы  print('main() оповещает все корутины с помощью - condition.notify\_all(), и передаёт управление в цикл событий')  print("Корутина main разблокирована")   # Ожидаем выполнения задач task1 и task2  await task1  await task2  asyncio.run(main()) | |
| # worker() получил блокировку, сообщение task1 # worker() получил блокировку, сообщение task2 # Корутина main получила блокировку # Корутина main реализует любую логику приложения # main() оповещает все корутины с помощью - condition.notify\_all(), и передаёт управление в цикл событий # Корутина main разблокирована # В worker() сработал await condition.wait() и она продолжает выполнять любую логику # worker() разблокирована, сообщение task1 # В worker() сработал await condition.wait() и она продолжает выполнять любую логику # worker() разблокирована, сообщение task2 | |
| Example 2 import asyncio   # Объявление корутины consumer async def consumer(condition, name):  # Блокировка условия  async with condition:  # Печатаем сообщение, что потребитель ждет  print(f'{name} - ждет')   # Ожидание сигнала  await condition.wait()   # Печатаем сообщение, что потребитель проснулся  print(f'{name} - проснулся')   # Объявление корутины producer async def producer(condition):  # Блокировка условия  async with condition:  # Печатаем сообщение, что производитель производит  print('Производитель производит')   # Уведомление всех ожидающих корутин  condition.notify\_all()   # Объявление корутины main async def main():  # Создание условия  condition = asyncio.Condition()   # Создание списка корутин-потребителей  consumers = [asyncio.create\_task(consumer(condition, f'Потребитель {i}')) for i in range(3)]   # Ожидание выполнения всех корутин (производителя и потребителей)  await asyncio.gather(producer(condition), \*consumers)   # Запуск корутины main asyncio.run(main()) | |
| # Потребитель 0 - ждет # Потребитель 1 - ждет # Потребитель 2 - ждет # Производитель производит # Потребитель 0 - проснулся # Потребитель 1 - проснулся # Потребитель 2 - проснулся | |
| Принцип простого семафора asyncio.Semaphore() | |
| Семафор ([Semaphore](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))) - это примитив синхронизации, который позволяет ограничить количество потоков, которые могут захватить блокировку. Semaphore работает как счетчик, который уменьшается каждый раз, когда задача начинает использовать общий ресурс и увеличивается, когда задача заканчивает использование ресурса. Если счетчик достигает нуля, это означает, что все доступные ресурсы используются, и новые задачи должны ждать, пока не освободится какой-либо ресурс.  **Как использовать Asyncio Semaphore.**  Чтобы использовать asyncio.Semaphore(), вы должны сначала создать его экземпляр:  # создаем семафор с лимитом 100  semaphore = asyncio.Semaphore(100)  где value - это начальное значение счетчика ресурсов.  Некоторые примеры использования семафора:   * Ограничение одновременных сокет-соединений к серверу. * Ограничение одновременных операций с файлами на жестком диске. * Ограничение одновременных вычислений. | |
| Доступные методы семафора:  * semaphore.acquire() - Этот метод запрашивает доступ к ресурсу. Если счетчик ресурсов больше нуля, то он уменьшается на 1, и метод возвращает True. Если счетчик равен нулю, то метод блокируется, пока другой поток не вызовет метод release(). * semaphore.release() - Этот метод освобождает ресурс, увеличивая счетчик на 1, в момент освобождения своего потока. Освобождение должно происходить после выполнения корутиной своей задачи. * semaphore.locked() - Этот метод возвращает True, если счетчик ресурсов равен нулю, и False в противном случае. * semaphore.\_value - Это свойство хранит текущее значение счетчика ресурсов. | |
| Семафор управляет внутренним счетчиком, который уменьшается на единицу при каждом вызове acquire() и увеличивается на единицу при каждом вызове release(). Счетчик никогда не может опускаться ниже нуля, когда функция acquire() обнаруживает, что счетчик равен нулю, она блокируется, ожидая, пока какая-либо задача не вызовет функцию release(). В случае когда счётчик равен нулю, это означает что все потоки заняты в данный момент, а когда одна из корутин вызывает функцию release(), она тем самым освобождает один поток семафора. | |
| функция release(), она увеличивает внутренний счетчик семафора независимо от начального значения. Это можно использовать для динамического увеличения пропускной способности семафора.  ...  # получить семафор  semaphore.acquire()  # сделай что-нибудь  # ...  # освободить семафор  semaphore.release()  # увеличить пропускную способность семафора  for i in range(5):  semaphore.release() | |
| Класс asyncio.Semaphore() поддерживает интерфейс менеджера контекста, который автоматически получает и освобождает семафор.  semaphore = asyncio.Semaphore(value)  # получить семафор  async with semaphore:  # ... | |
| Корутина может проверить, загружен ли семафор в данный момент, с помощью метода locked(). Он вернет True, если семафор не может быть получен во время вызова, или False в противном случае.  ...  # проверяем, есть ли место на семафоре  if not semaphore.locked():  # ... | |
| пример использования Semaphore для управления доступом к файлу: import asyncio   semaphore = asyncio.Semaphore(1)   async def write\_to\_file(text):  async with semaphore:   with open("file.txt", "a") as file:  file.write(text)   async def main():  # Создаем список задач  tasks = [write\_to\_file("строка 1\n"), write\_to\_file("строка 2\n"), write\_to\_file("строка 3\n")]  await asyncio.gather(\*tasks)    asyncio.run(main()) | |
| Example2 from random import random import asyncio   async def task(semaphore, number):  async with semaphore:   value = random()   await asyncio.sleep(value)   print(f'Задача {number} получила {value}')    async def main():  semaphore = asyncio.Semaphore(2)   tasks = [asyncio.create\_task(task(semaphore, i)) for i in range(10)]  await asyncio.wait(tasks)   # Запускаем asyncio программу asyncio.run(main()) | |
| Example3 import asyncio   sem = asyncio.Semaphore(3)   async def task(id):  async with sem:   print(f'Задача {id} начала выполнение')   await asyncio.sleep(1)   print(f'Задача {id} завершила выполнение') t   async def main():  tasks = [task(i) for i in range(50)]   await asyncio.gather(\*tasks)   # Запускаем все задачи asyncio.run(main()) | |
| сложный пример использования двойного Semaphore: import asyncio  # Создаем два семафора с максимальным количеством  sem1 = asyncio.Semaphore(2)  sem2 = asyncio.Semaphore(2)  async def task(id):  async with sem1, sem2:   print(f'Задача {id} начала выполнение')   await asyncio.sleep(1)   print(f'Задача {id} завершила выполнение')   async def main():  tasks = [task(i) for i in range(5)]   await asyncio.gather(\*tasks)   # Запускаем все задачи asyncio.run(main()) | |
| примеры сценариев, где это может быть полезно:   1. **Ограничение доступа к нескольким ресурсам**: Семафоры могут использоваться для ограничения доступа к нескольким ограниченным ресурсам, таким как подключения к базе данных или файлам. В этом случае, каждая задача может требовать доступа к обоим ресурсам, и оба ресурса могут поддерживать ограниченное количество одновременных задач. 2. **Управление параллелизмом**: В многозадачной среде, где задачи могут выполняться одновременно, иногда необходимо управлять числом задач, которые выполняются одновременно, чтобы предотвратить перегрузку системы. В данном случае, каждый семафор может представлять определенный уровень параллелизма в системе. 3. **Синхронизация задач**: В некоторых сценариях, может потребоваться, чтобы определенные задачи были синхронизованы таким образом, чтобы они начинались и завершались вместе. В этом случае, семафоры могут быть использованы для управления порядком и темпом выполнения задач. | |
| Стоит отметить, что использование двух семафоров может привести к ситуации взаимной блокировки (**Deadlock**), если не управлять ими правильно. Это происходит, когда две или более задач взаимно ожидают друг друга для освобождения ресурсов, которые они захватили, об этом мы говорили ранее. | |
| Example 4 Обработка ошибок при использовании Semaphore: import asyncio   sem = asyncio.Semaphore(2)   async def task(id):  try:  async with sem:   print(f'Задача {id} начала выполнение')   await asyncio.sleep(1)   if id == 2:   raise Exception('Ошибка в задаче')   print(f'Задача {id} завершила выполнение')   except Exception as e:  print(f'В задаче {id} произошла ошибка: {e}')    async def main():  tasks = [task(i) for i in range(5)]   await asyncio.gather(\*tasks)   # Запускаем все задачи asyncio.run(main())  # listing # Задача 0 начала выполнение # Задача 1 начала выполнение # Задача 0 завершила выполнение # Задача 1 завершила выполнение # Задача 2 начала выполнение # Задача 3 начала выполнение # В задаче 2 произошла ошибка: Ошибка в задаче # Задача 3 завершила выполнение # Задача 4 начала выполнение # Задача 4 завершила выполнение | |
| Test1 Semaphore import asyncio  total\_requests: int = 0 sem = asyncio.Semaphore(3)  async def server\_request(name):  global total\_requests   async with sem:  print(f'Пользователь {name} делает запрос')  total\_requests += 1  await asyncio.sleep(0.1)  print(f'Запрос от пользователя {name} завершен')  print(f'Всего запросов: {total\_requests}')  async def main():  users = ["sasha", "petya", "masha", "katya", "dima",   "olya", "igor", "sveta", "nikita", "lena",  "vova", "yana", "kolya", "anya", "roma",   "nastya", "artem", "vera", "misha", "liza"]   tasks =[server\_request(name) for name in users]  await asyncio.gather(\*tasks)   asyncio.run(main()) | |
| Test listing Пользователь sasha делает запрос  Пользователь petya делает запрос  Пользователь masha делает запрос  Запрос от пользователя sasha завершен  Всего запросов: 3  Запрос от пользователя petya завершен  Всего запросов: 3  Запрос от пользователя masha завершен  Всего запросов: 3  Пользователь katya делает запрос  Пользователь dima делает запрос  Пользователь olya делает запрос  Запрос от пользователя katya завершен  Всего запросов: 6  ......  Запрос от пользователя artem завершен  Всего запросов: 18  Запрос от пользователя vera завершен  Всего запросов: 18  Пользователь misha делает запрос  Пользователь liza делает запрос  Запрос от пользователя misha завершен  Всего запросов: 20  Запрос от пользователя liza завершен  Всего запросов: 20  Process finished with exit code 0 | |
|  | |
| Принцип ограниченного семафора asyncio.BoundedSemaphore() | |
| asyncio.BoundedSemaphore()(ограниченный семафор) - это объект семафора, который используется для управления доступом к ресурсу в Python. Он позволяет ограничить количество одновременных доступов к ресурсу.  .BoundedSemaphore() реализован как обертка над asyncio.Semaphore(). Он имеет дополнительный параметр "bound", который указывает максимальное количество одновременных доступов к ресурсу. Доступные методы ограниченного семафора Доступные методы ограниченного семафора выполняют точно такую же работу, как и в неограниченном семафоре:   * boundedsemaphore.acquire() - Этот метод запрашивает доступ к ресурсу. Если счетчик ресурсов больше нуля, то он уменьшается на 1, и метод возвращает True. Если счетчик равен нулю, то метод блокируется, пока другой поток не вызовет метод release(). * boundedsemaphore.release() - Этот метод освобождает ресурс, увеличивая счетчик на 1 в момент освобождения своего потока. Освобождение должно происходить после выполнения корутиной своей задачи. * boundedsemaphore.locked() - Этот метод возвращает True, если счетчик ресурсов равен нулю, и False в противном случае.  Semaphore или Bounded Semaphore ? Semaphore : семафор может увеличивать значение своего внутреннего счетчика выше начального значения.    Bounded Semaphore: ограниченный семафор не может увеличить значение своего внутреннего счетчика выше его начального значения. | |
| К примеру, вы работаете с сервером, который обрабатывает только 300 открытых соединений в секунду, а если будет получен 301 запрос на соединение, то сервер может отвергнуть его, и вам придётся либо обрабатывать исключение, либо посылать повторный запрос, таким образом производительность вашего приложения будет снижена, а количество ошибок в логах будет только увеличиваться. Bounded Semaphore гарантирует то, что будет открыто точное количество потоков или запущенных корутин, сколько было указано при его вызове.  Между Semaphore и Bounded Semaphore имеется только одно отличие — это то, что Bounded Semaphore не будет превышать установленного значения, в отличие от классического семафора. Весь синтаксис и принцип работы идентичны, кроме обозначенного различия. | |
| Пример для **Semaphore**и **Bounded Semaphore** с превышением первоначального значения.  import asyncio  # Создаем экземпляр семафора с лимитом одновременных доступов == 2. # Внимание!!! Ниже строки использующие различные типы семафора. # Поочередно выполните код с каждым из них, чтобы понять различие в их работе   semaphore = asyncio.Semaphore(2) # semaphore = asyncio.BoundedSemaphore(2)  print(f'Исходный лимит для {type(semaphore).\_\_name\_\_}: {semaphore.\_value}')   async def my\_coroutine(id):  print(f'Корутина {id} хочет получить семафор')  await semaphore.acquire()  print(f'Корутина {id} получила семафор')  await asyncio.sleep(1)  semaphore.release()  print(f'Корутина {id} отпустила семафор')  semaphore.release()  print(f'Корутина {id} отпустила семафор еще раз')   # Запускаем несколько корутин async def main():  try:  await asyncio.gather(my\_coroutine(1), my\_coroutine(2), my\_coroutine(3))  # Перехват исключения для ситуации, когда слишком много release()  except ValueError:  print(f'Попытка несанкционированного release().\n'  f'Количество попыток освободить семафор превышает количество его захватов')  finally:  print(f'Лимит {type(semaphore).\_\_name\_\_}: {semaphore.\_value} ', end='/')  print(f'(Лимит был увеличен)' if semaphore.\_value > 2 else f'(Увеличения лимита не произошло)')   asyncio.run(main()) | |
| Listing 1 semaphore = asyncio.Semaphore(2)  example1.py  Исходный лимит для Semaphore: 2  Корутина 1 хочет получить семафор  Корутина 1 получила семафор  Корутина 2 хочет получить семафор  Корутина 2 получила семафор  Корутина 3 хочет получить семафор  Корутина 1 отпустила семафор  Корутина 1 отпустила семафор еще раз  Корутина 2 отпустила семафор  Корутина 2 отпустила семафор еще раз  Корутина 3 получила семафор  Корутина 3 отпустила семафор  Корутина 3 отпустила семафор еще раз  Лимит Semaphore: 5 /(Лимит был увеличен)  Process finished with exit code 0 | Listing2 semaphore = asyncio.BoundedSemaphore(2)  example1.py  Исходный лимит для BoundedSemaphore: 2  Корутина 1 хочет получить семафор  Корутина 1 получила семафор  Корутина 2 хочет получить семафор  Корутина 2 получила семафор  Корутина 3 хочет получить семафор  Корутина 1 отпустила семафор  Корутина 1 отпустила семафор еще раз  Корутина 2 отпустила семафор  Корутина 3 получила семафор  Попытка несанкционированного release().  Количество попыток освободить семафор превышает количество его захватов  Лимит BoundedSemaphore: 2 /(Увеличения лимита не произошло)  Process finished with exit code 0 |
| Реализация барьера для защиты одновременного доступа Barrier class asyncio.Barrier(parties) - объект барьера. Не является потокобезопасным.  В асинхронном программировании на Python часто возникает необходимость синхронизировать выполнение нескольких задач, для этих целей в Python 3.11 был введен новый примитив синхронизации - asyncio.Barrier.  Этот объект позволяет блокировать выполнение задач до тех пор, пока их (задач) определенное количество не достигнет этого барьера. После чего все задачи, ожидающие на барьере, будут одновременно разблокированы. Барьер можно использовать многократно, что делает его удобным инструментом для организации поэтапного выполнения задач.  Для создания барьера необходимо указать количество задач (parties), которые должны достигнуть барьера для его преодоления.  ​​​​​​​barrier = asyncio.Barrier(3) Специальные методы и атрибуты wait(): Метод, который используется задачами для ожидания на барьере. Когда задача вызывает await barrier.wait(), она блокируется до тех пор, пока барьера не достигнет заданное количество задач (parties). Как только последняя необходимая задача вызовет wait(), все они одновременно разблокируются и смогут продолжить свое выполнение.  Этот метод может вызвать исключение asyncio.BrokenBarrierError, если барьер сломан или сброшен во время ожидания задачи. Метод также может вызвать asyncio.CancelledError, если задача будет отменена.  wait() можно заменить использованием контекстного менеджера async with.  ​​​​​​​​​​​​​​  reset(): Возвращает барьер в исходное состояние. Все задачи, ожидающие на барьере, получат исключение BrokenBarrierError.  При вызове reset() в момент, когда некоторые корутины уже ожидают у барьера, они также получат исключение BrokenBarrierError, так как состояние барьера неожиданно изменится. Однако в отличие от abort(), reset() позволяет барьеру быть повторно использованным после сброса.  reset() предполагает контролируемый "перезапуск" барьера и подразумевает, что все участники знают о перезапуске и готовы начать новый цикл синхронизации.  ​​​​​​​  abort(): Переводит барьер в состояние "сломан". Все текущие и будущие вызовы wait() завершатся с исключением BrokenBarrierError.  abort() предназначен для ситуаций, когда барьер должен быть немедленно сброшен из-за возникновения ошибки или иной непредвиденной ситуации, делающей дальнейшее ожидание барьера бессмысленным или невозможным.  После вызова Barrier.abort(), барьер переходит в "сломанное" состояние, и любые последующие попытки использовать его приведут к немедленному возникновению BrokenBarrierError.  Этот метод предназначен для прерывания работы с барьером и сигнализирует участникам о том, что совместная операция прервана и не может быть продолжена.  parties: Возвращает количество задач, необходимых для прохождения барьера.    n\_waiting: Возвращает количество задач, ожидающих в данный момент на барьере.    broken: Возвращает булево значение (True или False), указывающее на то, находится ли барьер в сломанном состоянии. Исключение BrokenBarrierError: exception asyncio.BrokenBarrierError:Исключение, являющееся подклассом RuntimeError, возникает, когда объект Barrier сбрасывается или ломается. | |
| Example1 import asyncio   # Корутина worker, принимающая объект asyncio.Barrier. async def worker(barrier: asyncio.Barrier, num):  await asyncio.sleep(num)  print(f"worker\_{num} ждет на барьере")   # В этой точке выполнение задачи приостанавливается  # до момента накопления на барьере заданного количества задач.  await barrier.wait()  # После преодоления барьера работа задачи возобновляется.   await asyncio.sleep(0.5)  # Вывод сообщения о прохождении барьера worker-ом.  print(f"worker\_{num} прошел барьер")   async def main():  # Создание объекта asyncio.Barrier (для разблокировки ожидаем 4 задачи).  barrier = asyncio.Barrier(4)  tasks = [asyncio.create\_task(worker(barrier, num)) for num in range(3)]   print(f'Состояние {barrier=}')  print("Ждем, пока worker's пройдут барьер")  await asyncio.sleep(0)  print(f'Состояние {barrier=}')  # Регистрируем на нашем барьере последнюю, 4-ю задачу для преодоления барьера.   await asyncio.sleep(6)  print("4-я задача достигла барьера")  await barrier.wait()   # Часть кода которая может быть выполнена только после преодоления барьера.  # Имитация выполнения длительной I/O операции.  await asyncio.sleep(1)  print("Все задачи успешно прошли барьер")  print(f'Состояние {barrier=}')   asyncio.run(main()) | |
| Example1 Listing # # output listing: # C:\Users\VMAL\AppData\Local\Programs\Python\Python311\python.exe "D:/GB/pythonProject/asyncio/Chapter\_9\_Primitives/Барьер для защиты доступа Barrier/example1.py" # Состояние barrier=<asyncio.locks.Barrier object at 0x00000282EDE929D0 [filling, waiters:0/4]> # Ждем, пока worker's пройдут барьер # Состояние barrier=<asyncio.locks.Barrier object at 0x00000282EDE929D0 [filling, waiters:0/4]> # worker\_0 ждет на барьере # worker\_1 ждет на барьере # worker\_2 ждет на барьере # 4-я задача достигла барьера # worker\_0 прошел барьер # worker\_1 прошел барьер # worker\_2 прошел барьер # Все задачи успешно прошли барьер # Состояние barrier=<asyncio.locks.Barrier object at 0x00000282EDE929D0 [filling, waiters:0/4]> # # Process finished with exit code 0 | |
| Example2 import asyncio   # Корутина для задачи прохождения барьера. async def task(name, num, barrier):  await asyncio.sleep(num / 10)  print(f'{name} начинает и ожидает у барьера.')   # Проверяем, сколько задач ожидают на барьере.  print(f' На барьере ожидает задач: {barrier.n\_waiting + 1}')   # Проверяем, сколько задач еще нужно для преодоления барьера.  print(f' Для прохождения нужно еще задач: {barrier.parties - (barrier.n\_waiting + 1)}')   # Для перехвата исключения используем try/except.  try:  async with barrier:  print(f'{name} прошла через барьер.')   except asyncio.BrokenBarrierError:  print(f'{name} обнаружила, что барьер {["сброшен", "сломан"][barrier.broken]}.')   async def aborting\_task(name, barrier):  await asyncio.sleep(1)  print(f'--{name} сбрасывает/ломает барьер.')   # Вариант 1. Прерываем работу барьера.  # await barrier.abort()  # Вариант 2. Сбрасываем барьер в исходное состояние.  await barrier.reset()   async def main():  # Создаем барьер на 2 задачи  barrier = asyncio.Barrier(2)  tasks1 = [asyncio.create\_task(task(f'Задача {i}', i, barrier)) for i in range(1, 4)  ] + [asyncio.create\_task(aborting\_task('Сбрасывающая задача', barrier))]   await asyncio.gather(\*tasks1)  # Создаем новый список задач.  tasks2 = [asyncio.create\_task(task(f'Задача {i}\_new', i, barrier)) for i in range(1, 7)]   # Проверяем состояние барьера  print(f'--Барьер разрушен: {(state := barrier.broken)}')  if not state:  # Если барьер цел запускаем на него вторую партию задач  print('--Барьер сброшен,продолжаем использовать барьер.')  else:  print('--Барьер сломан, все новые задачи получат BrokenBarrierError.')  # Это для эксперимента со сломанным барьером:  # await barrier.reset()  # У меня барьер успешно перезапускается, но в доках написано:  # Если барьер разрушен, возможно, лучше просто оставить его и создать новый.   await asyncio.gather(\*tasks2)   asyncio.run(main()) | |
| Example 2 listing await barrier.reset() Задача 1 начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 2 начинает и ожидает у барьера.  На барьере ожидает задач: 2  Для прохождения нужно еще задач: 0  Задача 2 прошла через барьер.  Задача 1 прошла через барьер.  Задача 3 начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  --Сбрасывающая задача сбрасывает/ломает барьер.  Задача 3 обнаружила, что барьер сброшен.  --Барьер разрушен: False  --Барьер сброшен,продолжаем использовать барьер.  Задача 1\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 2\_new начинает и ожидает у барьера.  На барьере ожидает задач: 2  Для прохождения нужно еще задач: 0  Задача 2\_new прошла через барьер.  Задача 1\_new прошла через барьер.  Задача 3\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 4\_new начинает и ожидает у барьера.  На барьере ожидает задач: 2  Для прохождения нужно еще задач: 0  Задача 4\_new прошла через барьер.  Задача 3\_new прошла через барьер.  Задача 5\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 6\_new начинает и ожидает у барьера.  На барьере ожидает задач: 2  Для прохождения нужно еще задач: 0  Задача 6\_new прошла через барьер.  Задача 5\_new прошла через барьер.  Process finished with exit code 0 | |
| Example 2 listing await barrier.abort() Barrier/example2.py"  Задача 1 начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 2 начинает и ожидает у барьера.  На барьере ожидает задач: 2  Для прохождения нужно еще задач: 0  Задача 2 прошла через барьер.  Задача 1 прошла через барьер.  Задача 3 начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  --Сбрасывающая задача сбрасывает/ломает барьер.  Задача 3 обнаружила, что барьер сломан.  --Барьер разрушен: True  --Барьер сломан, все новые задачи получат BrokenBarrierError.  Задача 1\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 1\_new обнаружила, что барьер сломан.  Задача 2\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 2\_new обнаружила, что барьер сломан.  Задача 3\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 3\_new обнаружила, что барьер сломан.  Задача 4\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 4\_new обнаружила, что барьер сломан.  Задача 5\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 5\_new обнаружила, что барьер сломан.  Задача 6\_new начинает и ожидает у барьера.  На барьере ожидает задач: 1  Для прохождения нужно еще задач: 1  Задача 6\_new обнаружила, что барьер сломан.  Process finished with exit code 0 | |

|  |
| --- |
| Асинхронное чтение файлов aiofiles |
| Introduction |
| Обработка файлов - это важный аспект многих программ, и она может быть требовательной к ресурсам и времени, особенно при работе с большими файлами. Если файловая обработка выполняется синхронно, это может привести к значительному замедлению выполнения программы, поскольку весь код должен ожидать завершения операций чтения или записи файла, прежде чем он сможет продолжить выполнение. |
| Асинхронное чтение файла позволяет другим асинхронным задачам продолжать выполнение, пока файл читается. Это достигается с помощью библиотеки aiofiles, которая использует asyncio для асинхронного чтения файлов.  Вот пример асинхронного чтения файла:  import asyncio import aiofiles  async def read\_file\_async(filename):  async with aiofiles.open(filename, mode='r') as file:  content = await file.read()  return content  content = asyncio.run(read\_file\_async('myfile.txt')) print(content) |
| Aiofiles начало Aiofiles - это библиотека Python, предназначенная для асинхронной работы с файлами. Она обеспечивает простой и удобный интерфейс для асинхронного IO, основываясь на асинхронной модели выполнения, предоставляемой модулем asyncio в Python. Библиотека aiofiles особенно полезна в ситуациях, когда необходимо обрабатывать большие файлы или выполнить множество операций IO с файлами без блокировки основного потока выполнения программы. |
| Преимущества Aiofiles  * Повышение производительности: благодаря асинхронному IO, приложение может продолжать выполнять другие задачи во время другой IO, вместо того чтобы ожидать ее завершения. * Совместимость с asyncio: aiofiles тесно интегрирован с asyncio. * Схожий синтаксис с синхронным IO: Один из больших плюсов aiofiles заключается в том, что его синтаксис очень похож на стандартный синхронный IO в Python. Это упрощает переход на асинхронное IO для тех, кто уже знаком с синхронными операциями IO в Python. * Улучшенная масштабируемость: Благодаря своей асинхронной природе, aiofiles может помочь улучшить масштабируемость приложений, особенно когда необходимо обрабатывать большое количество файлов одновременно. Это может быть особенно полезно для веб-серверов и других сетевых приложений, которые могут одновременно обслуживать множество запросов к файлам. * Поддержка различных форматов файлов: aiofiles поддерживает различные режимы чтения и записи, такие как текстовый, бинарный и даже поддерживает обработку файлов в формате CSV или JSON, о которых мы будем говорить в этом курсе. |
| Подводные камни асинхронного чтения файлов  * Хотя асинхронное IO может улучшить производительность вашего приложения, стоит помнить о некоторых ограничениях. * Ограничение ОС: Дисковый ввод-вывод все еще блокируется на уровне операционной системы, и на данный момент нет нативного способа сделать его полностью асинхронным. * Использование потоков: aiofiles предоставляет интерфейс для асинхронной работы с файлами, используя под капотом потоки из пула потоков. Это влечёт за собой некоторые накладные расходы по сравнению с нативными асинхронными операциями I/O, такими как те, что используются для сетевых запросов. * Высокий порог входа: Asyncio и aiofiles основаны на концепции корутин и async/await синтаксиса, которые могут быть сложными для понимания, если вы не знакомы с асинхронным программированием. Прежде чем применять асинхронное IO, убедитесь, что вы хорошо понимаете эти концепции. * Ошибки обработки: С асинхронным кодом часто сложнее работать, когда дело доходит до обработки ошибок. Исключения, возникающие в корутинах, не всегда передаются туда, где вы могли бы их ожидать. Вместо этого они обычно прокидываются назад в событийный цикл, где они могут быть неочевидными или затруднительными для отслеживания. * Сложности отладки: Плавно вытекает из предыдущего пункта. Асинхронные программы могут быть сложными для отладки, поскольку они часто включают в себя множество задач, работающих конкурентно. Это может затруднить отслеживание потока выполнения и точного места возникновения проблемы. * Контроль параллелизма: При асинхронном чтении или записи большого количества файлов важно управлять количеством одновременно выполняемых операций, чтобы избежать истощения системных ресурсов. Это может потребовать дополнительного кода и управления (например, asyncio.Semaphore или asyncio.Queue). |
| Основной метод aiofiles.open() **Использование aiofiles может быть полезным в двух сценариях:**   1. **Обработка большого количества файлов**: Если ваше приложение одновременно работает с большим количеством файлов, то aiofiles позволит асинхронно и эффективно обрабатывать эти IO. 2. **Обработка больших файлов**: Если ваше приложение обрабатывает большие файлы, aiofiles позволит асинхронно читать или записывать эти файлы по кусочкам. |
| Установка aiofiles очень проста:  pip install aiofiles |
| aiofiles.open(file, mode='r', buffering=-1, encoding='utf-8', errors=None, newline=None, closefd=True, opener=None, \*, loop=None, executor=None)  aiofiles.open() принимает следующие аргументы:   * file: путь к файлу. * mode: режим открытия файла.   + 'r': открыть файл для чтения (по умолчанию)   + 'w': открыть файл для записи, создав новый файл или перезаписывая существующий   + 'a': открыть файл для добавления информации в конец файла, создавая новый файл, если он не существует   + 'x': создать новый файл и открыть его для записи   + 'b': открыть файл в бинарном режиме   + 't': открыть файл в текстовом режиме (по умолчанию)   + '+': открыть файл для обновления (чтения и записи) * buffering: политика буферизации.   + Если buffering равно 1, строковый буфер активирован.   + Если buffering больше 1, буферизация блока активирована.   + Если buffering равно -1, используется системная политика по умолчанию. * encoding: имя кодировки, используемое для кодирования или декодирования файла. * errors: строка, указывающая, как кодековые ошибки обрабатываются. * newline: контролирует, как переводы строк обрабатываются при чтении или записи. * closefd: если равно **False** и новый файловый дескриптор передается в виде первого аргумента, файловый дескриптор будет оставлен открытым, когда файл будет закрыт. * opener: опциональный пользовательский опенер. * loop: опциональный аргумент, который позволяет передать собственный цикл событий. * executor: опциональный аргумент, который позволяет передать собственный executor. |
| менеджер контекста with не поддерживает асинхронность, поэтому мы используем его асинхронный аналог [async with](https://stepik.org/lesson/933820/step/1?unit=939717). Это гарантирует, что файл будет корректно закрыт, даже если при чтении или записи возникнет исключение. |
| mode='r' Параметр mode в функции aiofiles.open() определяет, в каком режиме открывается файл. Значение mode - это строка, состоящая из одного или нескольких символов. Вот основные режимы, которые можно использовать:   * 'r': открыть файл для чтения (по умолчанию) * 'w': открыть файл для записи, создав новый файл или перезаписывая существующий * 'a': открыть файл для добавления информации в конец файла, создавая новый файл, если он не существует * 'x': создать новый файл и открыть его для записи * 'b': открыть файл в бинарном режиме * 't': открыть файл в текстовом режиме (по умолчанию) * '+': добавляет дополнительную функциональность к основным режимам чтения 'r', записи 'w' и добавления 'a': -'r+': открывает файл для чтения и записи. Файл должен существовать; -'w+': открывает файл для записи и чтения. Если файл существует, его содержимое удаляется. Если файл не существует, создается новый файл для чтения и записи; -'a+': открывает файл для добавления и чтения. Данные будут добавлены в конец файла без удаления существующего содержимого. Если файл не существует, создается новый файл для чтения и добавления. |
| Общие практики открытия некоторых типов файлов:  | **Расширение файла** | **Режим чтения** | **Режим записи** | | --- | --- | --- | | .txt, .csv, .json, .md, .html, .xml, .py, .js, .css | 'r' или 'rt' | 'w', 'wt', 'a', 'at', 'x' или 'xt' | | .jpg, .jpeg, .png, .gif, .bmp, .ico, .tiff | 'rb' | 'wb', 'ab', 'xb' | | .mp3, .wav, .ogg | 'rb' | 'wb', 'ab', 'xb' | | .mp4, .avi, .mkv, .mov | 'rb' | 'wb', 'ab', 'xb' | | .bin, .dat | 'rb' | 'wb', 'ab', 'xb' | | .gzip, .zip, .tar, .rar | 'rb' | 'wb', 'ab', 'xb' | |
| Тонкости использования параметра mode:  * Выбор правильного режима открытия файла зависит от того, что вы собираетесь делать с файлом. Если вы хотите только прочитать файл, используйте 'r'. Если вы хотите записать в файл, используйте 'w', 'a' или 'x', в зависимости от того, хотите ли вы перезаписать существующий файл, добавить в конец файла или гарантировать создание нового файла. * Если вы открываете файл в текстовом режиме ('t'), вы можете также указать кодировку с помощью параметра encoding. Если вы открываете файл в бинарном режиме ('b'), кодировка игнорируется. * Режим 'x' может вызвать исключение FileExistsError, если файл уже существует. Это может быть полезно, если вы хотите гарантировать, что не перезапишете существующий файл. * Режим '+' полезен, когда вы хотите работать с файлом, выполняя и чтение, и запись, без необходимости открывать его несколько раз. Однако нужно быть осторожным при использовании этих режимов, чтобы избежать потери данных, так как операции чтения и записи могут влиять друг на друга в зависимости от текущей позиции указателя в файле и специфики работы с файлом в вашей программе. |
| encoding='utf-8' Параметр encoding используется для определения кодировки файла при чтении или записи. Кодировка — это набор правил, которые используются для преобразования символов строки в байты (при записи) и байтов в символы (при чтении).  Стандартная кодировка в Python — это 'utf-8'. UTF-8 является одной из наиболее популярных кодировок и может представлять большое количество символов из разных языков.  Однако в некоторых случаях возникает необходимоь открыть файл в другой кодировке. Например, вам в руки попадет файл, который был сохранен в кодировке 'windows-1251' (часто используется для русского языка в Windows). |
| Пример использования параметра encoding в aiofiles.open():  import aiofiles  async def read\_from\_file():  async with aiofiles.open('example.txt', mode='r', encoding='windows-1251') as f:  content = await f.read()  return content |
| Тонкости использования параметра encoding:  * Если вы откроете файл в неправильной кодировке, вы можете получить искаженный текст или ошибку **UnicodeDecodeError**. Важно знать, в какой кодировке был записан ваш файл. * Не все кодировки могут представлять все символы. Например, кодировка ASCII может представлять только базовые английские символы. Если вы попытаетесь записать символ, который не может быть представлен в вашей кодировке, вы получите ошибку **UnicodeEncodeError**. * UTF-8 является хорошим выбором кодировки по умолчанию, поскольку она может представлять большое количество символов и широко поддерживается различными системами и приложениями. |
| errors='ignore' Параметр errors в функции open() или aiofiles.open() указывает, как следует обрабатывать ошибки кодеков, которые могут возникнуть при кодировании (при записи файла) или декодировании (при чтении файла).  Вот несколько примеров значений, которые может принимать этот параметр:   * 'strict': Это значение используется по умолчанию. Если при кодировании или декодировании происходит ошибка, то будет вызвано исключение UnicodeError. * 'ignore': При возникновении ошибки она будет проигнорирована, и символ, вызвавший ошибку, будет пропущен. Это может привести к потере данных, так как все символы, которые не могут быть корректно закодированы или декодированы, будут удалены. * 'replace': Если символ не может быть закодирован или декодирован, то он заменяется на знак вопроса ? (при кодировании) или на специальный символ замены Юникода � (при декодировании). |
| Пример использования параметра errors в aiofiles.open():  import aiofiles  async def read\_from\_file():  async with aiofiles.open('example.txt', mode='r', errors='ignore') as f:  content = await f.read()  return content |
| Тонкости использования параметра errors:   * Ошибки кодеков обычно возникают, когда файл содержит символы, которые не могут быть корректно интерпретированы с использованием выбранной кодировки. Например, если файл был сохранён в кодировке UTF-8, но вы пытаетесь прочитать его, используя кодировку ASCII, то при попытке прочитать любой символ, который не входит в стандарт ASCII, возникнет ошибка. * Параметр errors позволяет вам определить, как следует обрабатывать такие ошибки. Однако важно помнить, что значения 'ignore' и 'replace' могут привести к потере данных. Лучшим решением обычно является корректное определение кодировки файла, чтобы избежать подобных ошибок. |
| **Н**есколько способов корректного определения кодировки файла:    1. **Использование инструментов ОС**: Некоторые операционные системы и файловые менеджеры могут показывать информацию о кодировке файла в свойствах файла. 2. **Программное определение**: Использование библиотек, таких как Python chardet, для анализа содержимого файла и определения кодировки. chardet делает это, анализируя вероятности того, какие символы могут соответствовать какой кодировке. 3. **Редакторы и IDE**: Большинство текстовых редакторов и интегрированных сред разработки (IDE) автоматически определяют кодировку при открытии файла или позволяют выбрать кодировку вручную. 4. **Команды Unix/Linux**: Использование команды file в Unix-подобных системах, которая может показать информацию о кодировке текстовых файлов. 5. **HTTP заголовки**: При загрузке файлов через HTTP, сервер может указывать кодировку в заголовке Content-Type.   Эти методы позволяют определить или предположить кодировку файла перед его обработкой. |
| closefd=True Параметр closefd используется только тогда, когда первый аргумент функции open() (или aiofiles.open()) является файловым дескриптором.  ⚠️ ***Файловый дескриптор*** - это целое число, которое используется в операционной системе для идентификации открытого файла или другого ресурса ввода-вывода (например, сокета).  Значение closefd указывает, следует ли автоматически закрыть файловый дескриптор, когда объект файла закрывается.  Если closefd установлен в True (что является значением по умолчанию), то файловый дескриптор будет автоматически закрыт, когда объект файла закрывается.  Если closefd установлен в False, то файловый дескриптор останется открытым даже после того, как объект файла будет закрыт. |
| Тонкости использования параметра closefd:  * В большинстве случаев, когда вы открываете файл по его имени, вам не нужно волноваться о параметре closefd, поскольку он имеет значение только тогда, когда вы открываете файл по файловому дескриптору. * Если вы оставите файловый дескриптор открытым (closefd=False), вы должны вручную закрыть его позже, используя функцию os.close(), чтобы освободить ресурсы системы. Если вы оставите файловый дескриптор открытым, и не закроете его, это может привести к утечке ресурсов в вашей программе. * Если вы передаете файловый дескриптор в aiofiles.open(), убедитесь, что вы установили правильный режим открытия (в данном случае 'r' для чтения) соответствующим образом с режимом открытия файлового дескриптора. |
| opener=custom\_opener Параметр opener функции open() или aiofiles.open() представляет собой вызываемый объект (функцию или другой объект, который можно вызвать), который принимает два аргумента (file, flags). Python вызывает opener вместо встроенной функции open() для открытия файла. Это может быть полезно, если вы хотите использовать альтернативный метод открытия файла. |
| Вот пример использования параметра opener:  import aiofiles import os  def custom\_opener(file, flags):  return os.open(file, flags | os.O\_NONBLOCK)  async def read\_from\_file():  async with aiofiles.open('example.txt', mode='r', opener=custom\_opener) as f:  content = await f.read()  return content |
| Тонкости использования параметра opener:   * Параметр opener может быть полезным, если вы хотите использовать альтернативный метод открытия файла, который отличается от стандартного поведения Python. * opener должен быть вызываемым объектом, который принимает два аргумента: имя файла и флаги. Он должен возвращать файловый дескриптор. * Необходимо быть осторожным при использовании opener, потому что он может изменить стандартное поведение операций ввода-вывода в вашей программе. Всегда проверяйте, что ваш opener корректно работает с теми файлами и операциями ввода-вывода, которые вы планируете использовать. |
| # Открытие файла в неблокирующем режиме:  import os  def non\_blocking\_opener(file, flags):  return os.open(file, flags | os.O\_NONBLOCK)  with open('example.txt', 'r', opener=non\_blocking\_opener) as f:  content = f.read() |
| # Открытие файла с эксклюзивным доступом (только для Unix-подобных систем):  import os  def exclusive\_opener(file, flags):  return os.open(file, flags | os.O\_EXCL)  with open('example.txt', 'w', opener=exclusive\_opener) as f:  f.write('Hello, world!') |
| # Открытие файла с использованием пользовательских прав доступа # (только для Unix-подобных систем):  import os  def custom\_mode\_opener(file, flags):  return os.open(file, flags, mode=0o644) # права доступа rw-r--r--  with open('example.txt', 'w', opener=custom\_mode\_opener) as f:  f.write('Hello, world!') |
| loop=loop Параметр loop используется для передачи собственного экземпляра цикла событий, который будет использоваться для асинхронного чтения и записи файлов. Цикл событий - это сердце каждой асинхронной программы, он отвечает за исполнение асинхронных операций и вызовы обратных вызовов. |
| Тонкости использования параметра loop:  * В большинстве случаев вам не нужно передавать собственный цикл событий в aiofiles.open(), так как по умолчанию используется текущий цикл событий. * Если вы все же решите передать собственный цикл событий, убедитесь, что он будет запущен и остановлен должным образом. * Начиная с Python 3.7, передача собственного цикла событий считается устаревшей практикой, и рекомендуется использовать функции asyncio.get\_event\_loop() или asyncio.get\_running\_loop(). |
| executor=executor Параметр executor в aiofiles.open() используется для указания пула потоков исполнения (executor), который будет использоваться для выполнения блокирующих операций ввода-вывода в отдельных потоках. Это позволяет асинхронному коду продолжать выполнение, пока блокирующие операции выполняются в фоновом режиме.  В Python пул потоков исполнения обычно создают с помощью функции concurrent.futures.ThreadPoolExecutor(). |
| Вот пример использования параметра executor:  # Вот пример использования параметра executor:  import asyncio import aiofiles from concurrent.futures import ThreadPoolExecutor  async def main():  with ThreadPoolExecutor(max\_workers=5) as executor:  async with aiofiles.open('example.txt', mode='r', executor=executor) as f:  content = await f.read()  print(content)  asyncio.run(main())  В этом примере мы создаем пул потоков исполнения с максимум 5 потоками и передаем его в aiofiles.open() через параметр executor. |
| Тонкости использования параметра executor:   * В большинстве случаев вам не нужно передавать свой собственный executor в aiofiles.open(), так как по умолчанию используется concurrent.futures.ThreadPoolExecutor, создаваемый asyncio. * Если вы передаете свой собственный executor, вы должны управлять его жизненным циклом (создание, запуск и остановка). * Количество потоков в executor определяет количество блокирующих операций, которые могут выполняться одновременно. Выбирая это число, следует учесть возможности вашей системы и характер вашей программы. |
| Buffering Буферизация — это процесс хранения данных в памяти перед их записью на диск или чтением с диска, что помогает увеличить производительность, так как операции с диском обычно занимают значительно больше времени, чем операции с памятью.  Параметр buffering в функции aiofiles.open() позволяет настроить политику буферизации при чтении и записи в файл. Это опциональный аргумент, и его значение по умолчанию -1, что означает использование системной политики по умолчанию.  Возможные значения параметра buffering:   * buffering=0 − Буферизация не используется (разрешено только в \*двоичном режиме). Немедленное чтение и запись данных в файл может увеличить количество системных вызовов. Этот режим подходит для интерактивных приложений или для работы с небольшими объемами данных. * buffering=1 − Используется буферизация строк. Каждая строка обрабатывается как отдельный буфер, так как данные считываются или записываются в файл фрагментами размером с одну строку. Этот режим целесообразен при работе с текстовыми файлами, в которых в качестве единицы обработки используются строки. * buffering>1 − Положительные целые числа больше 1 представляют собой размер буфера в байтах. Файл читается или записывается кусками, равными заданному размеру буфера. Этот режим подходит для управления огромными файлами или оптимизации использования памяти. * buffering=-1(default) − Oперационная система и базовая библиотека ввода/вывода выбирают размер буфера автоматически (обычно строковая буферизация для tty устройств и буферизация блока для остальных). |
| Вот пример использования параметра buffering:  # Вот пример использования параметра buffering:  import asyncio import aiofiles   async def main():  # Буферизация отключена, используется режим чтения бинарных данных  async with aiofiles.open('myfile.txt', mode='rb', buffering=0) as f:  data = await f.read()  print(data) # Увидим бинарные данные файла   # Буферизация отключена, но приведет к ошибке  try:  async with aiofiles.open('myfile.txt', mode='r', buffering=0) as f:  data = await f.read()  print(data)  except ValueError as eq:  print(f"Обнаружена ошибка: {eq}")   # Строковый буфер  async with aiofiles.open('myfile.txt', mode='r', buffering=1) as f:  print(await f.read())   # Буферизация блока с размером буфера 4096  async with aiofiles.open('myfile.txt', mode='r', buffering=4096) as f:  print(await f.read())   asyncio.run(main()) |
| # output /example11\_buffering.py # b'Hello' # Обнаружена ошибка: can't have unbuffered text I/O # Hello # Hello # # Process finished with exit code 0   * b'Hello' — бинарное представление содержимого файла. * Ошибка ValueError: can't have unbuffered text I/O указывает на то, что вы пытаетесь открыть текстовый файл без использования буферизации (buffering=0), что не допускается в Python для текстовых потоков. В контексте асинхронного ввода-вывода с использованием библиотеки aiofiles, параметр buffering управляет размером буфера: когда вы задаете buffering=0, это указывает на отсутствие буферизации. Однако, для текстового ввода-вывода (когда mode='r' или 'w', и не указан 'b' для бинарного режима) Python требует буферизацию, так как обработка текста включает в себя дополнительные операции, такие как кодирование/декодирование и обработка символов новой строки. |
| Тонкости использования параметра buffering:  * Отключение буферизации (установка buffering в 0) может привести к увеличению накладных расходов, так как каждая операция чтения или записи будет прямо взаимодействовать с операционной системой. Это может быть полезно в случаях, когда вам нужен максимальный контроль над временем и порядком операций ввода-вывода, но в большинстве случаев это приведет к снижению производительности. * Строковый буфер (buffering=1) может быть полезен, если вы часто читаете или пишете строки в файл, и хотите минимизировать количество системных вызовов. * Буферизация блока (buffering больше 1) может быть полезна, если вы часто читаете или пишете большие объемы данных в файл, и хотите минимизировать количество системных вызовов. Размер буфера можно настроить, чтобы оптимизировать производительность для конкретного размера чтения/записи.   Помните, что использование buffering — это вопрос баланса между производительностью и использованием памяти. Оптимальное значение buffering зависит от конкретных требований вашего приложения и его рабочей нагрузки. |
| \* Python различает двоичный (бинарный) и текстовый ввод-вывод. Файлы, открытые в двоичном режиме (в том числе *'b'*в аргументе режима), возвращают содержимое в виде [*bytes*](https://docs-python-org.translate.goog/3/library/stdtypes.html?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=sc#bytes)объектов без какого-либо декодирования. В текстовом режиме (по умолчанию или когда *'t'*включен в аргумент режима) содержимое файла возвращается как [*str*](https://docs-python-org.translate.goog/3/library/stdtypes.html?_x_tr_sl=en&_x_tr_tl=ru&_x_tr_hl=ru&_x_tr_pto=sc#str), причем байты сначала декодируются с использованием кодировки, зависящей от платформы, или с использованием указанной кодировки, если она задана. |
|  |
| Асинхронный файловый объект и доступные методы |
| **Асинхронный файловый объект** - это объект, который предоставляет методы для асинхронного чтения и записи данных в файл. Эти методы включают операции, такие как чтение, запись, поиск и закрытие файла. Использование асинхронного файлового объекта позволяет вашему коду продолжать выполнение других задач, пока операция с файлом выполняется в фоновом режиме. |
| Вот пример использования асинхронного файлового объекта с помощью библиотеки aiofiles: |
| В примере выше f является экземпляром асинхронного файлового объекта, возвращаемого функцией aiofiles.open(). Этот объект предоставляет следующие асинхронные методы, которые соответствуют методам стандартного файлового объекта Python:  **Обозначение None означает что метод не принимает никаких аргументов.** |
| 1. [f.write(string)](https://stepik.org/lesson/1029068/step/2?unit=1037338): Асинхронно записывает строку string в файл. Возвращает количество записанных символов. 2. [f.writelines(lines)](https://stepik.org/lesson/1029068/step/2?unit=1037338): Асинхронно записывает список строк lines в файл. 3. [f.writable(None)](https://stepik.org/lesson/1029068/step/3?unit=1037338): Возвращает True, если файл открыт для записи, и False в противном случае. 4. [f.read(n=-1)](https://stepik.org/lesson/1029068/step/4?unit=1037338): Асинхронно читает и возвращает до n символов из файла. Если n не указан или равен -1, читает и возвращает все символы до конца файла. 5. [f.readline(None)](https://stepik.org/lesson/1029068/step/5?unit=1037338): Асинхронно читает и возвращает одну строку из файла. Если достигнут конец файла, возвращает пустую строку. 6. [f.readlines(None)](https://stepik.org/lesson/1029068/step/5?unit=1037338): Асинхронно читает и возвращает все строки файла в виде списка строк. 7. [f.readinto(b)](https://stepik.org/lesson/1029068/step/6?unit=1037338): Асинхронно читает данные в предоставленный байтовый массив b, возвращает количество прочитанных байтов. 8. [f.close(None)](https://stepik.org/lesson/1029068/step/7?unit=1037338): Асинхронно закрывает файл. После закрытия файла больше нельзя проводить никаких операций с ним. 9. [f.flush(None)](https://stepik.org/lesson/1029068/step/8?unit=1037338): Асинхронно очищает буфер записи файла, если он поддерживает это. 10. [f.isatty(None)](https://stepik.org/lesson/1029068/step/9?unit=1037338): Возвращает True, если файл подключен к терминалу, и False в противном случае. 11. [f.read1(n=-1)](https://stepik.org/lesson/1029068/step/10?unit=1037338): Асинхронно читает и возвращает до n символов из файла, используя только один вызов системы ввода-вывода. 12. [f.seek(offset, whence=0)](https://stepik.org/lesson/1029068/step/11?unit=1037338): Асинхронно изменяет позицию файла на offset байт, относительно whence. whence может быть 0 (для отсчета от начала файла), 1 (для отсчета от текущей позиции) или 2 (для отсчета от конца файла). 13. [f.seekable(None)](https://stepik.org/lesson/1029068/step/12?unit=1037338): Возвращает True, если в файле можно изменять позицию, и False в противном случае. 14. [f.tell(None)](https://stepik.org/lesson/1029068/step/13?unit=1037338): Асинхронно возвращает текущую позицию указателя в байтах от начала файла. 15. [f.truncate(size=None)](https://stepik.org/lesson/1029068/step/14?unit=1037338): Асинхронно урезает файл до size байт. Если size не указан, урезает файл до текущей позиции. Возвращает новый размер файла. |
| f.write(string) и f.writelines(lines) Метод f.write(string) используется для асинхронной записи строки в файл. Этот метод принимает один аргумент - строку, которую нужно записать в файл, и возвращает количество записанных символов. |
| Вот пример использования f.write(string):  # Вот пример использования f.write(string):  import asyncio import aiofiles  async def main():  async with aiofiles.open('myfile.txt', mode='w') as f:  num\_chars = await f.write('Hello, world!')  print(f'Записано {num\_chars} символов в файл.')  asyncio.run(main())  # >>> Записано 13 символов в файл. |
| Несколько важных моментов, которые следует учесть при использовании f.write(string):   1. Если файл, в который вы пишете, уже существует и содержит данные, то запись в файл с помощью f.write(string) перезапишет существующие данные. Если вы хотите добавить данные в конец файла, вместо перезаписи, используйте режим 'a' (добавление) при открытии файла. 2. Метод f.write(string) записывает данные в буфер, и эти данные могут не сразу появиться в файле. Чтобы гарантировать, что все данные были записаны в файл, используйте метод f.flush() после записи. 3. Метод f.write(string) принимает только строки. Если вы хотите записать другие типы данных, вам нужно будет преобразовать их в строки перед записью. 4. Не забывайте закрыть файл после завершения работы с ним. Это можно сделать автоматически, используя оператор with, как показано в примере выше. |
| f.writelines(lines) Метод f.writelines(lines) используется для асинхронной записи списка строк в файл. Этот метод принимает один аргумент - список строк, которые нужно записать в файл. Он не возвращает никакого значения.  f.writelines(lines) записывает строки из списка в файл, но не добавляет символы новой строки между ними. Это означает, что если вы передадите список строк ['Hello, world!', 'How are you?', 'Goodbye!'], то в файле они будут записаны как одна непрерывная строка: 'Hello, world!How are you?Goodbye!'.  Если вы хотите, чтобы каждая строка была записана на новой строке, вам нужно будет включить символ новой строки ('\n') в конце каждой строки в списке |
| # Вот пример использования f.writelines(lines):  import asyncio import aiofiles  async def main():  lines = ['Hello, world!\n', 'How are you?\n', 'Goodbye!\n']  async with aiofiles.open('myfile.txt', mode='w') as f:  await f.writelines(lines)  asyncio.run(main()) |
| Несколько важных моментов, которые следует учесть при использовании f.writelines(lines):   1. Метод f.writelines(lines) записывает строки в файл без добавления символов новой строки между ними. Если вы хотите, чтобы каждая строка была записана на новой строке, вам нужно будет включить символ новой строки ('\n') в конце каждой строки в списке. 2. Если файл, в который вы пишете, уже существует и содержит данные, то запись в файл с помощью f.writelines(lines) перезапишет существующие данные. Если вы хотите добавить строки в конец файла, вместо перезаписи, используйте режим 'a' (добавление) при открытии файла. 3. Метод f.writelines(lines) записывает данные в буфер, и эти данные могут не сразу появиться в файле. Чтобы гарантировать, что все данные были записаны в файл, используйте метод f.flush() после записи. 4. Не забывайте закрыть файл после завершения работы с ним. Это можно сделать автоматически, используя оператор with, как показано в примере выше. |
| f.writable() Метод f.writable() используется для проверки, открыт ли файл для записи. Этот метод не принимает никаких аргументов и возвращает True, если файл открыт для записи, и False в противном случае. |
| Вот пример использования f.writable():  # Вот пример использования f.writable():  import asyncio import aiofiles  async def main():  async with aiofiles.open('myfile.txt', mode='w') as f:  print(f'File is writable: {await f.writable()}')  asyncio.run(main())  # >>> File is writable: True |
| Несколько важных моментов, которые следует учесть при использовании f.writable():   1. Если файл был открыт только для чтения (т.е., с режимом 'r'), f.writable() вернет False. 2. Если файл был открыт для записи или добавления (т.е., с режимами 'w', 'a', 'w+', 'a+', 'r+'), f.writable() вернет True. 3. Если файл был закрыт с помощью f.close(), вызов f.writable() вызовет исключение ValueError. Поэтому всегда убедитесь, что файл открыт, прежде чем вызывать f.writable(). 4. Метод f.writable() не гарантирует, что запись в файл будет успешной. Например, если диск заполнен или если произошла другая ошибка ввода-вывода, попытка записи в файл может все равно привести к ошибке, даже если f.writable() вернул True. |
| f.read() Метод f.read(n=-1) используется для асинхронного чтения данных из файла. Этот метод принимает один аргумент - количество символов, которые нужно прочитать из файла, и возвращает прочитанные символы в виде строки. Если n не указан или равен **-1**, метод читает и возвращает все символы до конца файла. |
| Вот пример использования f.read(n=-1):  Вот пример использования f.read(n=-1):  import asyncio import aiofiles  async def main():  async with aiofiles.open('myfile.txt', mode='r') as f:  contents = await f.read(5)  print(contents)  asyncio.run(main()) |
| Несколько важных моментов, которые следует учесть при использовании f.read():   1. Если файл был открыт в текстовом режиме (т.е., без 'b' в режиме), f.read() вернет строку. Если файл был открыт в двоичном режиме (т.е., с 'b' в режиме), f.read() вернет объект bytes. 2. Если n больше, чем количество символов, оставшихся в файле, f.read() прочитает и вернет все оставшиеся символы. 3. Если достигнут конец файла, f.read() вернет пустую строку или пустой объект bytes, в зависимости от режима файла. 4. Если файл был закрыт с помощью f.close(), вызов f.read() вызовет исключение ValueError. Поэтому всегда убедитесь, что файл открыт, прежде чем вызывать f.read(). |
| f.readline() и f.readlines() Метод f.readline() используется для асинхронного чтения **одной строки из файла**. Этот метод не принимает никаких аргументов и возвращает прочитанную строку. Если достигнут конец файла, метод возвращает пустую строку. |
| Вот пример использования f.readline():  # Вот пример использования f.readline():  import asyncio import aiofiles  async def main():  async with aiofiles.open('myfile.txt', mode='r') as f:  line = await f.readline()  print(line)  asyncio.run(main()) |
| Несколько важных моментов, которые следует учесть при использовании f.readline():   1. Строка, возвращаемая f.readline(), включает символ новой строки ('\n' или '\r\n'), если он присутствует в файле. Если вы не хотите этого символа в возвращаемой строке, вы можете использовать метод str.rstrip() для его удаления. 2. Если файл был открыт в текстовом режиме (т.е., без 'b' в режиме), f.readline() вернет строку. Если файл был открыт в двоичном режиме (т.е., с 'b' в режиме), f.readline() вернет объект bytes. 3. Если достигнут конец файла, f.readline() вернет пустую строку или пустой объект bytes, в зависимости от режима файла. 4. Если файл был закрыт с помощью f.close(), вызов f.readline() вызовет исключение ValueError. Поэтому всегда убедитесь, что файл открыт, прежде чем вызывать f.readline(). |
| f.readlines() Методы f.readline() и f.readline**s**() оба используются для чтения строк из файла, но они работают немного по-разному.   * f.readline(): Этот метод асинхронно читает и возвращает **одну** **строку из файла**. * f.readline**s**(): Этот метод асинхронно читает **все строки из файла** и возвращает их в виде списка строк.   В обоих случаях каждая строка включает символ новой строки ('\n' или '\r\n'), если он присутствует в файле. Если вы не хотите этих символов в возвращаемых строках, вы можете использовать метод str.rstrip() для их удаления.    Метод f.readline**s**() используется для асинхронного чтения **всех строк из файла**. Если достигнут конец файла, он возвращает пустой список. Этот метод полезен, когда вы хотите прочитать все строки файла сразу, например, когда вы работаете с небольшим файлом и хотите выполнить операции с каждой строкой. |
| Вот пример использования f.readline**s**():  # Вот пример использования f.readlines():  import asyncio import aiofiles  async def main():  async with aiofiles.open('myfile.txt', mode='r') as f:  lines = await f.readlines()  print(lines)  asyncio.run(main()) |
| Несколько важных моментов, которые следует учесть при использовании f.readline**s**():   1. Каждая строка в возвращаемом списке включает символ новой строки ('\n' или '\r\n'), если он присутствует в файле. Для удаления этих символов в возвращаемых строках, рекомендуется использовать метод str.rstrip(). 2. Если файл был открыт в текстовом режиме (т.е., без bв режиме), f.readlines() вернет список строк. Если файл был открыт в двоичном режиме (т.е., с b в режиме), f.readline**s**() вернет список объектов bytes. 3. Если достигнут конец файла, f.readline**s**() вернет пустой список. 4. Если файл был закрыт с помощью f.close(), вызов f.readline**s**() вызовет исключение ValueError. Поэтому всегда убедитесь, что файл открыт, прежде чем вызывать f.readline**s**(). |
| f.readinto(b) Метод f.readinto(b) используется для асинхронного чтения данных из файла и записи их в предоставленный байтовый массив b. Этот метод принимает один аргумент - байтовый массив, в который нужно записать данные, и возвращает количество прочитанных байтов.  Важно отметить, что f.readinto(b) доступен только для файлов, открытых в двоичном режиме (т.е., с указанием на бинарный режим: '**b**'). |
| Вот пример использования f.readinto(b):  # Вот пример использования f.readinto(b):  import asyncio import aiofiles  async def main():  ba = bytearray(10) # Создайте байтовый массив для хранения данных  async with aiofiles.open('myfile.bin', mode='rb') as f:  num\_bytes = await f.readinto(ba)  print(f'Считать {num\_bytes} байт в байтовый массив.')  asyncio.run(main()) |
| Несколько важных моментов, которые следует учесть при использовании f.readinto(b):   1. Если файл был открыт в текстовом режиме (т.е., без указателя 'b' в режиме), вызов f.readinto(ba) вызовет исключение UnsupportedOperation. 2. Если ba больше, чем количество байтов, оставшихся в файле, f.readinto(ba) считает их и вернет только эти считанные байты. 3. Если достигнут конец файла, f.readinto(ba) вернет 0. 4. Если файл был закрыт с помощью f.close(), вызов f.readinto(ba) вызовет исключение ValueError. Поэтому всегда убедитесь, что файл открыт, прежде чем вызывать f.readinto(ba). |
|  |
|  |
|  |
|  |
|  |