Part 1

**Введение в асинхронность**

**Что такое асинхронность и её значение**

Асинхронность – это подход к программированию, при котором операции могут выполняться параллельно или независимо друг от друга, не блокируя основной поток программы. Основное преимущество асинхронного подхода заключается в возможности эффективно использовать время выполнения программы, не дожидаясь завершения одной операции для начала другой. Это важно когда мы работаем с сетевыми запросы, файлами или с базами данных.

Пример 1.

Представь, что ты используешь сайт для покупки билетов на концерт. Ты нажимаешь кнопку "купить", и если сайт работает синхронно (без асинхронности), весь сайт просто «замрёт», пока система проверяет твою карту, обрабатывает платёж и подтверждает покупку. За это время ты не можешь ни прокручивать страницу, ни нажимать на другие кнопки — всё заблокировано.

**Асинхронность** в этом случае позволяет сайту продолжать работать: пока твой платёж обрабатывается в фоновом режиме, ты можешь продолжать что-то делать на сайте, например, смотреть другие концерты. Это делает сайт быстрым и отзывчивым.

Если умными словами то значение асинхронности это:

* **Улучшение производительности.** Позволяют системе обрабатывать другие задачи, пока ожидается завершение текущих, что снижает время простоя процессора.
* **Меньшая блокировка потоков**. Предотвращает блокировку главного потока программы, что улучшает отзывчивость приложений, особенно тех, что работают с пользователем.
* **Эффективность использования ресурсов**: В некоторых случаях позволяет уменьшить потребление ресурсов по сравнению с многопоточностью и многопроцессорностью, так как не создает избыточных потоков или процессов.

**Разница между многопоточностью, многопроцессорностью и асинхронностью**

Чтобы лучше понять асинхронность, важно рассмотреть различия между этими концепциями:

**Многопоточность или Threading**: Это модель, при которой программа может запускать несколько потоков одновременно. Каждый поток представляет собой независимую последовательность выполнения внутри программы. Потоки могут выполняться параллельно, если процессор поддерживает многозадачность, но они разделяют память и ресурсы, что может привести к необходимости синхронизации между ними, чтобы избежать конфликтов доступа к данным.

Пример кода

import threading, time

def task():

print("Начало задачи")

time.sleep(2) # Симуляция долгой задачи

print("Задача завершена")

# Создаем два потока

thread1 = threading.Thread(target=task)

thread2 = threading.Thread(target=task)

# Запускаем потоки

thread1.start()

thread2.start()

# Ждем завершения потоков

thread1.join()

thread2.join()

print("Все задачи завершены")

**Многопроцессорность**: Многопроцессорность предполагает использование нескольких процессоров (или ядер процессора) для параллельного выполнения программ. Каждый процесс выполняется независимо от других и имеет свою область памяти, что снижает необходимость синхронизации данных между процессами, но при этом требует больше ресурсов, таких как память и процессорное время.

Пример

import multiprocessing

import time

def task():

print("Начало задачи")

time.sleep(2) # Симуляция долгой задачи

print("Задача завершена")

# Создаем два процесса

process1 = multiprocessing.Process(target=task)

process2 = multiprocessing.Process(target=task)

# Запускаем процессы

process1.start()

process2.start()

# Ждем завершения процессов

process1.join()

process2.join()

print("Все задачи завершены")

**Асинхронность**: Асинхронность не предполагает создание дополнительных потоков или процессов. Вместо этого используются специальные механизмы, которые позволяют одной части программы продолжать выполняться, в то время как другая часть ожидает завершения долгой операции (например, сетевой запрос). Основной поток программы не блокируется, что делает приложения более отзывчивыми без создания лишней нагрузки на систему.

import asyncio

async def task():

print("Начало задачи")

await asyncio.sleep(2) # Симуляция долгой задачи

print("Задача завершена")

async def main():

# Запускаем две задачи асинхронно

await asyncio.gather(task(), task())

# Запуск асинхронного события

asyncio.run(main())

**Сравнение:**

* **Многопоточность**: Несколько потоков внутри одного процесса, требующих синхронизации.
* **Многопроцессорность**: Разделение на независимые процессы, каждый из которых выполняется на отдельном процессоре.
* **Асинхронность**: Оптимизация работы одного потока путем выполнения задач в фоновом режиме без его блокировки.

**Это пособие направлено на** asyncio, но как говорит мой друг Лучше знать чем не знать)

Part 2

**2. Коруторы и Awaitables**

**Что такое корутины в Python (оооочень простыми словами)**

Корутину можно представить как **функцию с суперсилой паузы**. Обычная функция делает что-то, затем возвращает результат и всё. А вот корутина — это такая штука, которая может остановиться в любой момент (как будто говорит: "Погоди-ка, мне надо передохнуть") и потом продолжить с того же места, когда будет нужно.

**Представь себе повара**

Допустим, ты шеф-повар. Ты готовишь суп, но понимаешь, что тебе нужно дождаться, пока он закипит. А суп варится долго! Что ты будешь делать? Сидеть и смотреть на кастрюлю? Конечно нет! Ты начнёшь резать овощи для салата или делать что-то ещё.

Так вот, **корутина** — это такой шеф-повар в программировании. Пока она "ждёт", например, загрузки данных с интернета или ответа от базы данных, она может делать другие полезные вещи.

Теперь давай углубимся в эту тему и посмотрим на примеры.

- Объекты awaitable и их роль в асинхронном коде.

- Примеры создания корутин и использования `await`.

**Основные понятия корутин**

1. **Корутина — это функция, которая может "приостановиться" и "продолжить".**
2. **Ключевые слова**:
   * async: Когда ты пишешь async перед функцией, она становится корутиной. Это как дать ей суперсилу паузы.
   * await: Когда внутри корутины ты пишешь await, ты говоришь программе: "Здесь нужно подождать другую корутину, но не останавливай весь процесс, делай другие дела".

**Пример: "Корутиновый чайник"**

import asyncio

async def boil\_water():

print("Включаем чайник...")

await asyncio.sleep(3) # Ждем, пока вода закипит (но не блокируем всю программу)

print("Вода закипела!")

async def prepare\_cup():

print("Берём чашку, кладём пакетик чая...")

await asyncio.sleep(1) # Представим, что мы ищем ложку

print("Чашка готова!")

async def main():

# Мы запускаем оба процесса "параллельно"

await asyncio.gather(boil\_water(), prepare\_cup())

asyncio.run(main())

***Что происходит****:*

*Пока чайник закипает, программа не сидит сложа руки. Она занимается чашкой! Это и есть сила корутин — они не тормозят выполнение других задач.*

**Как корутины экономят время?**

Корутину можно сравнить с другом, который умеет работать одновременно над разными вещами. Например, если ты ожидаешь ответа от сервера или базы данных, корутина не будет просто стоять в углу и ждать. Она пойдёт делать что-то ещё! Это как если бы ты готовил суп и одновременно пылесосил квартиру.

**Почему await — это как пауза?**

Когда ты видишь слово await, представляй, что корутина говорит: "Мне нужно остановиться здесь и подождать, пока что-то произойдёт". Например, если ты ожидаешь загрузки файла с интернета, ты не будешь просто стоять и ждать (это было бы скучно и неэффективно!). Вместо этого, ты можешь делать другие задачи.

**Пример 2: "Мастер йоги"**

Представь, что ты мастер йоги. Ты умеешь выполнять упражнения, но некоторые из них требуют времени (например, ты должен держать позу в течение 10 секунд). Вот так можно сделать корутину, которая будет ждать, пока мастер йоги завершит упражнение, но не остановит всё остальное в мире.

import asyncio

async def hold\_pose(pose, seconds):

print(f"Держим позу: {pose}")

await asyncio.sleep(seconds) # Ждём завершения упражнения

print(f"Закончено упражнение: {pose}")

async def main():

await asyncio.gather(

hold\_pose("Поза дерева", 5),

hold\_pose("Поза лотоса", 3)

)

asyncio.run(main())

***Что происходит****:*

*Мы одновременно выполняем два упражнения: позу дерева и лотоса.*

*Ожидание завершения одной позы не мешает выполнять другую задачу!*

**Когда тебе нужно использовать корутины?**

**Когда ты работаешь с длительными операциями**: Например, ты ждёшь ответа от API (серверного запроса) или копируешь большие файлы.

**Когда не хочешь блокировать основную программу**: Приложения вроде чатов или игр не могут позволить себе "зависать" из-за одной долгой задачи. Если каждое нажатие на кнопку ждет ответа от сервера и приложение зависает — пользователи сбегут.

**Пример с ожиданием сетевого запроса:**

Представь, что ты пишешь программу, которая запрашивает данные с веб-сайта:

import asyncio

import aiohttp

async def fetch\_data(url):

async with aiohttp.ClientSession() as session:

async with session.get(url) as response:

print(f"Получены данные с {url}")

return await response.text()

async def main():

urls = ["http://example.com", "http://example.org", "http://example.net"]

tasks = [fetch\_data(url) for url in urls]

await asyncio.gather(\*tasks)

asyncio.run(main())

Здесь fetch\_data — корутина, которая делает HTTP-запрос. Мы используем aiohttp для асинхронных запросов. В этом примере программа делает несколько запросов одновременно и ждет их завершения, не блокируя выполнение.

**Что важно запомнить:**

1. **asyncio.run()**: Чтобы запустить асинхронную программу, используем asyncio.run(). Это как "точка входа" в асинхронную программу.
2. **asyncio.gather()**: Используется для того, чтобы запустить несколько корутин одновременно и ждать их завершения.