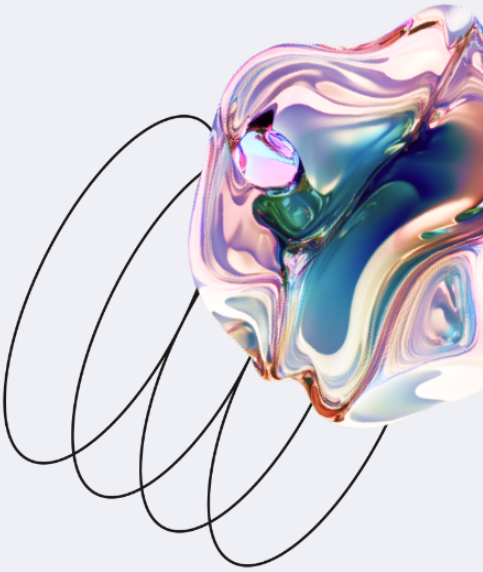
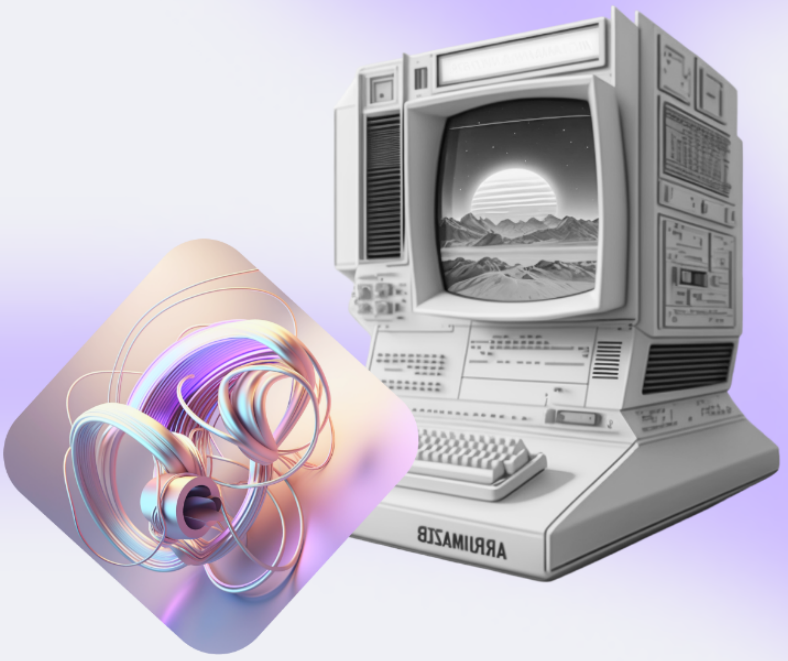
GeekBrains



Лекция 4.

Введение в многозадачность



Оглавление

На этой лекции мы

1. Вспомним про синхронный подход к решению задач
2. Узнаем о многопоточном подходе в программировании
3. Разберёмся с многопроцессорным подходом в Python
4. Изучим асинхронный подход в решении задач

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Узнали про работу с базами данных посредством Flask-SQLAlchemy
2. Разобрались с созданием форм средствами Flask-WTForm

Подробный текст лекции

Введение в многозадачность в Python

В современном программировании часто возникает необходимость выполнять несколько задач одновременно. Для этого используется многозадачность — способность компьютера или программы обрабатывать несколько задач одновременно.

В Python существует несколько подходов к реализации многозадачности, каждый из которых имеет свои преимущества и недостатки. Прежде чем начать использовать многозадачность в своих программах, необходимо понимать основные понятия и подходы.

Поток выполнения — это независимая последовательность инструкций, которая может выполняться параллельно с другими потоками. Каждый поток имеет свой стек вызовов и свой контекст выполнения.

Процесс — это экземпляр программы, который запущен на компьютере. Каждый процесс имеет свой собственный адресное пространство, в котором хранятся данные и код программы. Процессы могут выполняться параллельно на разных ядрах процессора.

Далее рассмотрим различные подходы к многозадачности в Python и их применение в различных ситуациях.

Синхронный подход

Синхронный код — это код, который выполняется последовательно, одна операция за другой. Когда программа выполняет какую-то операцию, она блокируется до тех пор, пока операция не будет завершена. Таким образом, если в программе есть

долгие операции, они могут занимать много времени и приводить к задержкам в работе программы.
Примеры синхронных операций в Python:

- чтение данных из файла
- отправка запроса на сервер и получение ответа
- выполнение сложных математических операций
- ожидание пользовательского ввода

Ограничения синхронного кода:

- задержки в работе программы из-за долгих операций
- невозможность выполнения нескольких задач одновременно
- ограниченность производительности

Для решения этих проблем можно использовать многопоточный или многопроцессорный подход. Однако, при использовании многопоточности и многопроцессорности возникают свои проблемы, такие как конкуренция за ресурсы и возможные блокировки.

Примеры программа на Python

Пример 1:

```
import time

def count_down(n):
    for i in range(n, 0, -1):
        print(i)
        time.sleep(1)

count_down(5)
```

Эта программа выводит обратный отсчет от заданного числа до 1 с интервалом в 1 секунду. В данном случае это синхронный код, так как каждое выполнение цикла for блокирует выполнение программы на 1 секунду.

Пример 2:

```
import time

def slow_function():
    print("Начало функции")
    time.sleep(5)
    print("Конец функции")

print("Начало программы")
slow_function()
print("Конец программы")
```

Эта программа вызывает функцию slow_function(), которая занимает 5 секунд на выполнение. Весь код работает синхронно, то есть выполнение программы блокируется на время выполнения функции.

Пример 3:

```
import random
import time

def long_running_task():
    for i in range(5):
        print(f"Выполнение задачи {i}")
        time.sleep(random.randint(1, 3))

def main():
    print("Начало программы")
    long_running_task()
    print("Конец программы")

main()
```

Эта программа запускает длительную задачу long_running_task(), которая выполняется в течение случайного времени от 1 до 3 секунд. Весь код работает синхронно, поэтому выполнение программы блокируется на время выполнения задачи.

В целом, синхронный код может быть полезен для простых задач, но он может стать проблемой при работе с длительными операциями или задачами, требующими большого количества вычислений. В таких случаях лучше использовать многопоточный или асинхронный код.

Многопоточный подход

Многопоточный код — это подход к многозадачности, при котором программа может выполнять несколько задач одновременно в разных потоках выполнения. Каждый поток выполняет свою задачу независимо от других потоков, что позволяет улучшить производительность программы.

Примеры многопоточных операций в Python:

- загрузка данных из нескольких файлов одновременно
- параллельная обработка большого объема данных
- одновременное выполнение нескольких запросов к базе данных
- многопоточный веб-сервер, обрабатывающий несколько запросов одновременно

Преимущества многопоточного кода:

- увеличение производительности программы за счет параллельного выполнения задач
- возможность выполнения нескольких задач одновременно без блокировки

Недостатки многопоточного кода:

- возможность возникновения конкуренции за ресурсы
- сложность отладки и тестирования многопоточных программ
- возможность блокировки потоков выполнения

Для решения проблем, связанных с конкуренцией за ресурсы и блокировками потоков, можно использовать механизмы синхронизации, такие как блокировки и семафоры. Однако, неправильное использование этих механизмов может привести к дедлокам (deadlock) и другим проблемам.
При разработке многопоточных программ необходимо учитывать особенности языка Python, такие как GIL (Global Interpreter Lock), который ограничивает параллелизм в исполнении Python-кода. Это означает, что в Python нельзя использовать несколько ядер процессора для выполнения одной программы.
В целом, многопоточный подход позволяет улучшить производительность программы и выполнить несколько задач одновременно без блокировки. Однако, при разработке многопоточных программ необходимо учитывать особенности

языка Python и правильно использовать механизмы синхронизации для избежания проблем.

Примеры программа на Python

Пример 1:

```
import threading
import time

def worker(num):
    print(f"Начало работы потока {num}")
    time.sleep(3)
    print(f"Конец работы потока {num}")

threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i, ))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("Все потоки завершили работу")
```

Эта программа создает 5 потоков и запускает функцию worker() в каждом из них. Функция worker() занимает 3 секунды на выполнение. Весь код работает многопоточно, то есть каждый поток работает независимо от других, и выполнение программы не блокируется на время выполнения функции.

Пример 2:

```
import threading
import time

def worker(num):
    print(f"Начало работы потока {num}")
    time.sleep(3)
    print(f"Конец работы потока {num}")

threads = []
```

```
for i in range(5):
    t = threading.Thread(target=worker, args=(i, ))
    threads.append(t)

for t in threads:
    t.start()
    t.join()

print("Все потоки завершили работу")
```

Эта программа создает 5 потоков и запускает функцию worker() в каждом из них. Функция worker() занимает 3 секунды на выполнение. Весь код работает многопоточно, но в отличие от предыдущего примера, потоки запускаются и завершаются последовательно, блокируя выполнение программы на время выполнения каждого потока.

Пример 3:

```
import threading

counter = 0

def increment():
    global counter
    for _ in range(1_000_000):
        counter += 1
    print(f"Значение счетчика: {counter:_}")

threads = []
for i in range(5):
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Значение счетчика в финале: {counter:_}")
```

Эта программа создает 5 потоков и запускает функцию increment() в каждом из них. Функция increment() увеличивает значение глобальной переменной counter на 1 миллион раз. Весь код работает многопоточно, но из-за того, что несколько потоков

работают с одной переменной, может возникнуть проблема гонки данных (race condition), когда результат выполнения программы может быть непредсказуемым.

Многопоточный код позволяет выполнять несколько задач параллельно, что может значительно ускорить выполнение программы. Однако при работе с общими ресурсами (например, глобальными переменными) может возникнуть проблема гонки данных, которую необходимо учитывать при написании многопоточного кода.

Многопроцессорный подход

Многопроцессорный код — это подход к многозадачности, при котором программа может выполнять несколько задач одновременно в разных процессах. Каждый процесс выполняет свою задачу независимо от других процессов, что позволяет улучшить производительность программы. При этом процессы могут быть запущены на разных процессорах многопроцессорного сервера.

Примеры многопроцессорных операций в Python:

- параллельная обработка большого объема данных
- одновременное выполнение нескольких запросов к базе данных
- многопроцессорный веб-сервер, обрабатывающий несколько запросов одновременно

Преимущества многопроцессорного кода:

- возможность использования нескольких ядер процессора для выполнения программы
- увеличение производительности программы за счет параллельного выполнения задач
- возможность выполнения нескольких задач одновременно без блокировки

Недостатки многопроцессорного кода:

- возможность возникновения конкуренции за ресурсы
- сложность управления и координации процессов
- возможность блокировки процессов выполнения

Для решения проблем, связанных с конкуренцией за ресурсы и блокировками процессов, можно использовать механизмы синхронизации, такие как блокировки и семафоры. Однако, неправильное использование этих механизмов может привести к дедлокам (deadlock) и другим проблемам.
При разработке многопроцессорных программ необходимо учитывать особенности языка Python, такие как использование модуля multiprocessing для создания и управления процессами. Также следует учитывать потребление ресурсов процессами и оптимизировать их работу.

В целом, многопроцессорный подход позволяет использовать несколько ядер процессора для выполнения программы и улучшить ее производительность. Однако, при разработке многопроцессорных программ необходимо учитывать особенности языка Python и правильно использовать механизмы синхронизации для избежания проблем.

Примеры программа на Python

Пример 1:

```
import multiprocessing
import time


def worker(num):
    print(f"Запущен процесс {num}")
    time.sleep(3)
    print(f"Завершён процесс {num}")

if __name__ == '__main__':
    processes = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print("Все процессы завершили работу")
```

Эта программа создает 5 процессов и запускает функцию worker() в каждом из них. Функция worker() просто выводит сообщение о запуске процесса, ждёт 3 секунды и сообщает о завершении. Весь код работает многопроцессорно, то есть каждый процесс работает независимо от других, и выполнение программы не блокируется на время выполнения функции.

Пример 2:

```
import multiprocessing
import time


def worker(num):
    print(f"Запущен процесс {num}")
    time.sleep(3)
    print(f"Завершён процесс {num}")

if __name__ == '__main__':
    processes = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        processes.append(p)

    for p in processes:
        p.start()
        p.join()

    print("Все процессы завершили работу")
```

Эта программа создает 5 процессов и запускает функцию worker() в каждом из них. Функция worker() просто выводит сообщение о запуске процессаа, ждёт 3 секунды и сообщает о завершении. Весь код работает многопроцессорно, но в отличие от предыдущего примера, процессы запускаются и завершаются последовательно, блокируя выполнение программы на время выполнения каждого процесса.

Пример 3:

```
import multiprocessing

counter = 0
```

```
def increment():
    global counter
    for _ in range(10_000):
        counter += 1
    print(f"Значение счетчика: {counter:_}")


if __name__ == '__main__':
    processes = []
    for i in range(5):
        p = multiprocessing.Process(target=increment)
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print(f"Значение счетчика: {counter:_}")
```

Эта программа создает 5 процессов и запускает функцию increment() в каждом из них. Функция increment() увеличивает значение глобальной переменной counter на 10 тысяч раз. Весь код работает многопроцессорно, но из-за того, что несколько процессов работают с одной переменной, может возникнуть проблема гонки данных (race condition), когда результат выполнения программы может быть непредсказуемым.
В нашем случае каждый из процессов работает со своей переменной counter. 5 процессов — 5 переменных со значением 10000 в финале.

Чтобы избежать этой проблемы, используется объект multiprocessing.Value, который обеспечивает безопасный доступ к общей переменной через механизм блокировки (lock). Каждый процесс получает доступ к переменной только после получения блокировки, что гарантирует правильность ее изменения. Для этого код необходимо изменить следующим образом.

```
import multiprocessing

counter = multiprocessing.Value('i', 0)

def increment(cnt):
    for _ in range(10_000):
```

```
        with cnt.get_lock():
            cnt.value += 1
    print(f"Значение счетчика: {cnt.value:_}")


if __name__ == '__main__':
    processes = []
    for i in range(5):
                p  =  multiprocessing.Process(target=increment,
args=(counter, ))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print(f"Значение счетчика в финале: {counter.value:_}")
```

Теперь 5 процессов используя доступ к одному объекту увеличивают его значение до 50 тысяч — 5 процессов по 10к каждый.

Асинхронный подход

Асинхронный код — это подход к многозадачности, при котором программа может выполнять несколько задач одновременно без создания отдельных процессов или потоков. Вместо этого задачи выполняются в рамках одного потока выполнения, но с использованием механизмов событий и обратных вызовов.

Примеры асинхронных операций в Python:

- обработка сетевых запросов
- чтение и запись в файлы
- обработка пользовательских событий в графическом интерфейсе

Преимущества асинхронного кода:

- более эффективное использование ресурсов процессора и памяти
- возможность обрабатывать большое количество задач одновременно без создания отдельных процессов или потоков
- упрощение кода и улучшение его читаемости

Недостатки асинхронного кода:

- сложность отладки и тестирования
- возможность возникновения ошибок из-за неправильного использования механизмов событий и обратных вызовов
- ограниченная поддержка сторонними библиотеками

Для разработки асинхронного кода в Python используется модуль asyncio, который предоставляет механизмы для организации асинхронного выполнения задач. Ключевыми понятиями в asyncio являются корутины, события и цикл событий.
Корутины — это функции, которые могут приостанавливать свое выполнение, чтобы дать возможность другим корутинам выполниться.
События используются для уведомления корутин о том, что какое-то событие произошло (например, завершение сетевого запроса).
Цикл событий — это основной механизм, который управляет выполнением корутин и обработкой событий.
При разработке асинхронного кода необходимо учитывать особенности работы с корутинами и правильно использовать механизмы событий для избежания проблем. Также следует учитывать поддержку сторонними библиотеками и оптимизировать работу программы.

В целом, асинхронный подход позволяет эффективно использовать ресурсы процессора и памяти, обрабатывать большое количество задач одновременно и упрощать код. Однако, при разработке асинхронного кода необходимо учитывать особенности работы с корутинами и правильно использовать механизмы событий для избежания проблем.

Примеры программа на Python

Пример 1:

```
import asyncio

async def print_numbers():
    for i in range(10):
        print(i)
        await asyncio.sleep(1)

async def print_letters():
    for letter in ['a', 'b', 'c', 'd', 'e', 'f']:
```

```
        print(letter)
        await asyncio.sleep(0.5)

async def main():
    task1 = asyncio.create_task(print_numbers())
    task2 = asyncio.create_task(print_letters())
    await task1
    await task2

asyncio.run(main())
```

Эта программа демонстрирует асинхронный код с использованием библиотеки asyncio. В функциях print_numbers() и print_letters() с помощью ключевого слова await осуществляется ожидание выполнения операции asyncio.sleep(), что позволяет переключаться между задачами без блокировки выполнения программы. В функции main() создаются две задачи, которые выполняются параллельно, а затем ожидается их завершение.

Пример 2:

```
import asyncio

async def count():
    print("Начало выполнения")
    await asyncio.sleep(1)
    print("Прошла 1 секунда")
    await asyncio.sleep(2)
    print("Прошло еще 2 секунды")
    return "Готово"

async def main():
    result = await asyncio.gather(count(), count(), count())
    print(result)

asyncio.run(main())
```

Эта программа демонстрирует асинхронный код с использованием функции asyncio.gather(), которая позволяет запускать несколько задач параллельно и ожидать их завершения. В функции count() с помощью ключевого слова await осуществляется ожидание выполнения операции asyncio.sleep(). В функции main() создаются три задачи с помощью функции count(), которые запускаются

параллельно с помощью функции asyncio.gather(). Результаты выполнения задач выводятся на экран после их завершения.

Пример 3:

```
import asyncio
from pathlib import Path

async def process_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        contents = f.read()
        # do some processing with the file contents
        print(f'{f.name} содержит {contents[:7]}...')

async def main():
    dir_path = Path('/path/to/directory')
    dir_path = Path('.')
     file_paths = [file_path for file_path in dir_path.iterdir()
if file_path.is_file()]
      tasks = [asyncio.create_task(process_file(file_path)) for
file_path in file_paths]
    await asyncio.gather(*tasks)

if __name__ == '__main__':
    asyncio.run(main())
```

Программа для асинхронной обработки большого количества файлов из примера 2 работает следующим образом:

1. Определяется путь к директории, в которой находятся файлы, которые нужно обработать.
2. Используется метод iterdir() для получения списка файлов в каталоге, и метод is_file() для проверки, что это файлы, а не каталоги или другие объекты.
3. Создается список задач для обработки каждого файла с помощью функции asyncio.create_task(), где каждая задача вызывает функцию process_file() для обработки соответствующего файла.
4. Запускается выполнение всех задач с помощью функции asyncio.gather().
5. Когда все задачи завершены, программа завершается.

Функция process_file() открывает файл для чтения и считывает его содержимое с помощью метода read(). Затем происходит обработка содержимого файла (которая может быть любой, в зависимости от требований конкретной задачи). После этого файл закрывается.

Таким образом, программа асинхронно обрабатывает каждый файл в директории, что позволяет ускорить выполнение задачи при большом количестве файлов.

Сравнение разных подходов на примере парсинга сайтов

Перед нами задача по скачиванию информации с главных страниц пяти сайтов. Рассмотрим решение задачи с использованием синхронного, многопоточного, многопроцессорного и асинхронного подходов.

Обычная синхронная загрузка:

```
import requests
import time

urls = ['https://www.google.ru/',
        'https://gb.ru/',
        'https://ya.ru/',
        'https://www.python.org/',
        'https://habr.com/ru/all/',
        ]

start_time = time.time()

for url in urls:
    response = requests.get(url)
    filename = 'sync_' + url.replace('https://', '').replace('.',
'_').replace('/', '') + '.html'
    with open(filename, "w", encoding='utf-8') as f:
        f.write(response.text)
      print(f"Downloaded {url} in {time.time() - start_time:.2f}
seconds")
```

В данном примере мы используем библиотеку requests для получения html-страницы каждого сайта из списка urls. Затем мы записываем полученный текст в файл с именем, соответствующим названию сайта.

🔥 **Важно!** Используйте pip install requests, если не устанавливали библиотеку ранее.

Загрузка в 5 потоков с использованием модуля threading:

```
import requests
import threading
import time

urls = ['https://www.google.ru/',
        'https://gb.ru/',
        'https://ya.ru/',
        'https://www.python.org/',
        'https://habr.com/ru/all/',
        ]


def download(url):
    response = requests.get(url)
          filename   =   'threading_'   +   url.replace('https://',
'').replace('.', '_').replace('/', '') + '.html'
    with open(filename, "w", encoding='utf-8') as f:
        f.write(response.text)
       print(f"Downloaded  {url}  in  {time.time()-start_time:.2f}
seconds")

threads = []
start_time = time.time()

for url in urls:
    thread = threading.Thread(target=download, args=[url])
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

Здесь мы создаем функцию download, которая загружает html-страницу и сохраняет ее в файл. Затем мы создаем по одному потоку для каждого сайта из списка urls, передавая функцию download в качестве целевой функции для каждого потока. Мы запускаем каждый поток и добавляем его в список threads. В конце мы ждем завершения всех потоков с помощью метода join.

Загрузка в 5 процессов с использованием модуля multiprocessing:

```
import requests
from multiprocessing import Process, Pool
import time

urls = ['https://www.google.ru/',
        'https://gb.ru/',
        'https://ya.ru/',
        'https://www.python.org/',
        'https://habr.com/ru/all/',
        ]


def download(url):
    response = requests.get(url)
       filename = 'multiprocessing_' + url.replace('https://',
'').replace('.', '_').replace('/', '') + '.html'
    with open(filename, "w", encoding='utf-8') as f:
        f.write(response.text)
     print(f"Downloaded {url} in {time.time() - start_time:.2f}
seconds")


processes = []
start_time = time.time()

if __name__ == '__main__':
    for url in urls:
        process = Process(target=download, args=(url,))
        processes.append(process)
        process.start()

    for process in processes:
        process.join()
```

Здесь мы используем модуль multiprocessing для создания процессов вместо потоков. Остальной код аналогичен предыдущему примеру.

Асинхронная загрузка с использованием модуля asyncio:

```
import asyncio
import aiohttp
import time

urls = ['https://www.google.ru/',
        'https://gb.ru/',
        'https://ya.ru/',
        'https://www.python.org/',
        'https://habr.com/ru/all/',
        ]


async def download(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            text = await response.text()
                filename = 'asyncio_' + url.replace('https://',
'').replace('.', '_').replace('/', '') + '.html'
            with open(filename, "w", encoding='utf-8') as f:
                f.write(text)
                    print(f"Downloaded {url} in {time.time() -
start_time:.2f} seconds")


async def main():
    tasks = []
    for url in urls:
        task = asyncio.ensure_future(download(url))
        tasks.append(task)
    await asyncio.gather(*tasks)

start_time = time.time()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Здесь мы используем модуль asyncio для асинхронной загрузки страниц. Мы создаем функцию download, которая использует aiohttp для получения

html-страницы и сохранения ее в файл. Затем мы создаем асинхронную функцию main, которая запускает функцию download для каждого сайта из списка urls и ожидает их завершения с помощью метода gather. Мы запускаем функцию main с помощью цикла событий asyncio.

Заключение

В этой лекции мы рассмотрели различные подходы к многозадачности в Python: синхронный, многопоточный, многопроцессорный и асинхронный подход. Каждый из этих подходов имеет свои преимущества и недостатки, и выбор подхода зависит от требований к производительности и конкретных задач.

Синхронный код прост в написании и понимании, но имеет ограничения в производительности. Многопоточный код позволяет эффективно использовать ресурсы процессора и памяти, но может привести к проблемам синхронизации. Многопроцессорный код также позволяет эффективно использовать ресурсы, но имеет дополнительные накладные расходы на коммуникацию между процессами. Асинхронный код позволяет обрабатывать большое количество задач одновременно без создания отдельных процессов или потоков, но требует особого внимания к правильному использованию механизмов событий и обратных вызовов. При выборе подхода к многозадачности необходимо учитывать требования к производительности и особенности конкретной задачи. Важно также учитывать возможность поддержки сторонними библиотеками и оптимизировать работу программы.

В целом, многозадачность — это важный инструмент в программировании, который позволяет эффективно использовать ресурсы компьютера и обрабатывать большое количество задач одновременно. Однако, при выборе подхода к многозадачности необходимо учитывать его преимущества и недостатки, а также правильно использовать механизмы синхронизации и событий для избежания проблем.

Вывод

На этой лекции мы:

1. Вспомнили про синхронный подход к решению задач
2. Узнали о многопоточном подходе в программировании
3. Разобрались с многопроцессорным подходом в Python
4. Изучили асинхронный подход в решении задач

Домашнее задание

Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.