АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ НА PYTHON

КОНЦЕПЦИИ, ПРИНЦИПЫ, ПРИМЕРЫ С ИСПОЛЬЗОВАНИЕМ ASYNCIO

АВТОР: РОМАН СПИРИДОНОВ

THINKNETICA
онлайн-школа для разработчиков

РОМАН СПИРИДОНОВ

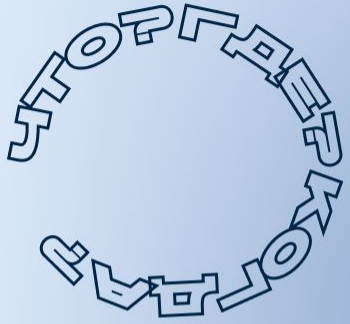
- Я - инженер программист с **6-летним опытом** коммерческой разработки на **Python** и опытом работы на **Golang**.
- Специализируюсь на разработке облачных сервисов и backend-сервисов для **высоконагруженных** систем.
- Прошел сертификацию **PCI DSS** и оценку безопасности от Visa и банков, что подчеркивает мою заботу о безопасности в разработке.
- Имею опыт в оптимизации серверной части под высокие нагрузки.
- Участвовал в проектировании и реализации различных систем, в том числе **MDM-систем** и систем управления **удаленными кассовыми устройствами**.
- Использую здоровый итеративный процесс разработки ПО, состоящий из современных практик декомпозиции, рефакторинга, дизайна ПО и разделения ответственности объекта.

Speccy-Rom

ArtFix13

КАК БУДЕМ ДВИГАТЬСЯ ПО ТЕМЕ?

- Введение в асинхронное программирование на Python
- Основные концепции и принципы asyncio
- Примеры использования asyncio в реальном мире
- Разбор вопросов и ответов



КОГДА И КУДА ЗАДАВАТЬ ВОПРОСЫ?

- В процессе рассказа можно задавать вопросы в чат и мы разберем их в конце мастер класса.

Асинхронное программирование на Python с помощью библиотеки asyncio - **востребованный подход для создания высокопроизводительных и масштабируемых приложений**. Однако, разработчики часто сталкиваются с проблемами при написании асинхронного кода:

- **сложность написания,**
- **возможные ошибки**
- **необходимость управления потоками выполнения и синхронизации данных.**

Для успешной разработки приложений, использующих асинхронное программирование, **необходимо обучение** разработчиков и предоставление решений для устранения возможных проблем.

ЦЕЛИ:

- Ознакомиться с концепциями асинхронного программирования на Python.
- Повторить основные принципы асинхронности в Python.
- Разобраться в примерах использования asyncio в Python.
- Обсудить принципы применения asyncio для создания асинхронных приложений на Python.

РАССМОТРИМ

- Что такое асинхронное программирование на Python.
- Какие принципы лежат в его основе.
- Что такое asyncio и как он работает.
- Какие особенности имеет асинхронное программирование на Python.

ПОВТОРИМ

- Как работают синхронные функции в Python.
- Как работают асинхронные функции в Python.
- Как использовать ключевые слова async и await для создания асинхронных функций.

РАЗБЕРЕМ НА ПРИМЕРАХ

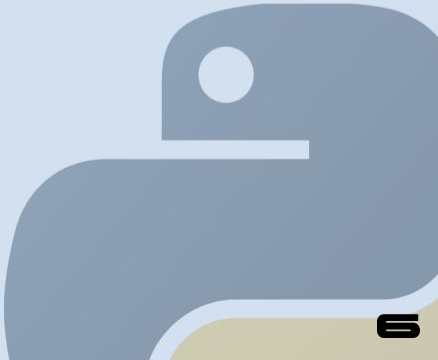
- Примеры использования asyncio для создания асинхронных приложений на Python.
- Примеры работы с асинхронными функциями и корутинами.
- Примеры использования asyncio для работы с сетевыми запросами.

ОБСУДИМ ПРИНЦИПЫ ПРИМЕНЕНИЯ

- Какие принципы следует учитывать при использовании asyncio для создания асинхронных приложений на Python.
- Какие особенности имеют асинхронные приложения и как их оптимизировать.

СОЗДАДИМ

- Простое асинхронное приложение на Python с использованием asyncio.
- Разберем основные этапы создания такого приложения и какие принципы при этом следует учитывать.

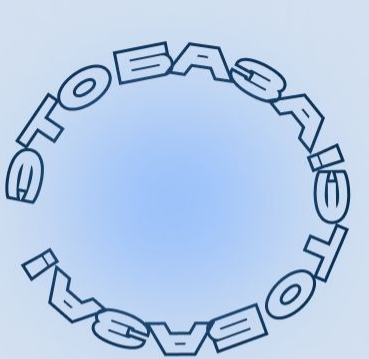


ГЛАВА 1

ВВЕДЕНИЕ

ОПРЕДЕЛЕНИЕ:

Асинхронное программирование – это подход к разработке программного кода, который позволяет эффективно использовать ресурсы компьютера и повысить производительность приложений. В отличие от синхронного программирования, где каждая операция выполняется последовательно, асинхронное программирование позволяет выполнять несколько операций параллельно и без блокировки потоков.



ПРЕИМУЩЕСТВА:

- Улучшение производительности приложений
- Экономия ресурсов компьютера
- Улучшение отзывчивости приложений
- Большая гибкость и возможность создания более сложных приложений

Пример 1. Синхронный код


```
import time

def long_operation():
    print(“Starting long operation”)
    time.sleep(5)
    print(“Long operation completed”)
```

Мы можем вызвать эту функцию и убедиться, что выполнение программы блокируется в течение 5 секунд:

```
print(“Start”)
long_operation
print(“End”)
```

Вывод:

```
Start
Starting long operation
Long operation completed
End
```

ПРИМЕР 2. АСИНХРОННЫЙ КОД


```python
import asyncio

async def long_operation():
    print("Starting long operation")
    await asyncio.sleeps(5)
    print("Long operation completed")

async def main():
    print("Start")
    task = asyncio.create_task(long_operation())
    await task
    print("End")

asyncio.run(main())
```

Вывод:

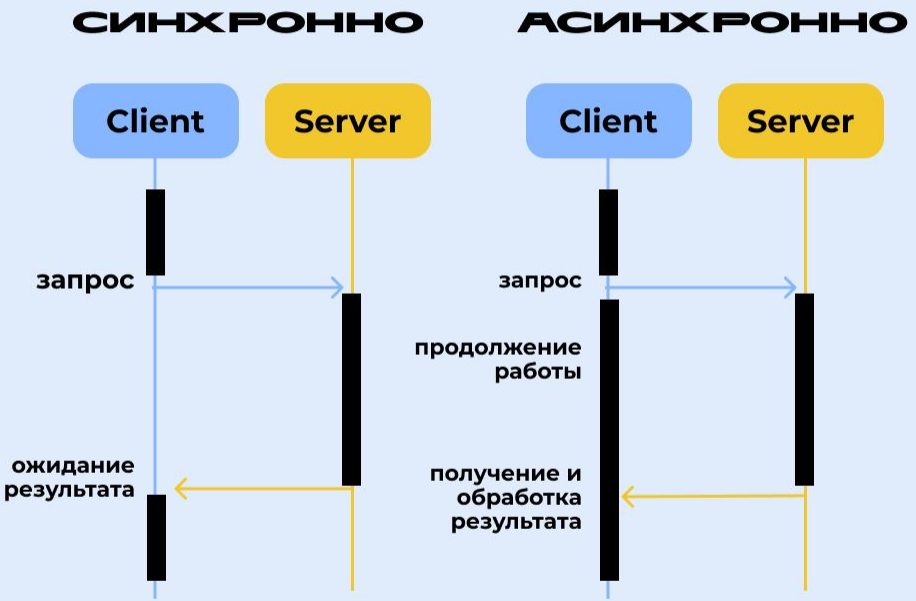
```
Start
Starting long operation
Long operation completed
End
```

ОБЗОР МОДУЛЯ ASYNCIO

Модуль **asyncio** является встроенным модулем в Python, который предоставляет инструменты для написания асинхронного кода. Используется для создания:

- **сетевых приложений**
- **обработки ввода-вывода, выполнения параллельных задач**
- **и многого другого.**

Он включает в себя реализацию событийного цикла, который обрабатывает асинхронные операции, а также механизмы для создания и управления сопрограммами.



ОСНОВНЫЕ МЕХАНИЗМЫ МОДУЛЯ

1. Корутины:
asyncio использует корутины вместо потоков для выполнения асинхронного кода. Корутины позволяют выполнять множество задач без создания новых потоков.

2. Событийный цикл:
asyncio использует событийный цикл для управления асинхронным кодом. Событийный цикл управляет выполнением корутин и обрабатывает события ввода-вывода.

3. Протоколы:
asyncio предоставляет реализацию различных протоколов для работы с сетевыми приложениями, такими как TCP, UDP, HTTP и другие.

4. Функции ожидания:
asyncio предоставляет функции ожидания, которые позволяют корутинам ожидать выполнения определенных событий, таких как получение данных из сети, завершение задачи и другие.

5. Контекстные менеджеры:
asyncio предоставляет контекстные менеджеры для управления ресурсами, такими как сокеты и файлы.

6. Интеграция с другими библиотеками:
asyncio может быть интегрирован с другими библиотеками, такими как aiohttp, для создания полноценных асинхронных приложений.

ГЛАВА 2

ПРИНЦИПЫ АСИНХРОННОГО ПРОГРАММИРОВАНИЯ

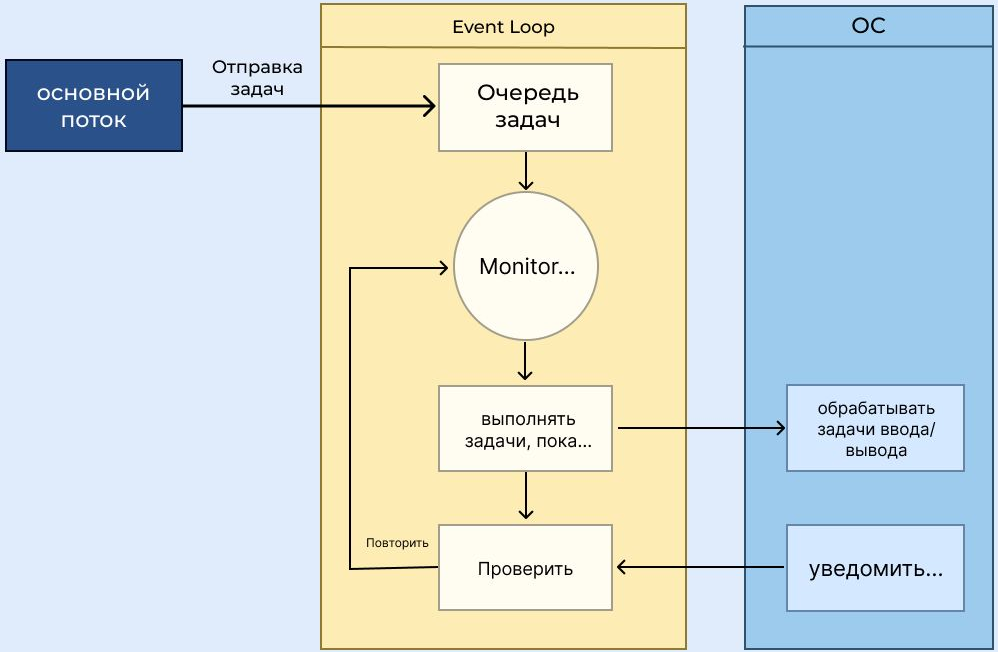
В асинхронном программировании
используются **три основных понятия:**

1. Корутины
это функции, которые могут приостанавливаться и возобновляться в процессе выполнения для выполнения других задач. Корутины в Python обозначаются ключевым словом "async def".

2. Событийный цикл (event loop)
это механизм, который позволяет выполнить несколько задач одновременно, используя корутины. Событийный цикл работает по принципу "ожидания" событий и вызова соответствующих обработчиков. Он запускается при старте программы и ожидает событий, таких как запросы на ввод-вывод, и вызывает обработчики, связанные с этими событиями.

3. **Сопрограммы**
это способ написания асинхронного кода, используя корутины и событийный цикл. Сопрограммы позволяют эффективно использовать ресурсы компьютера, позволяя выполнять несколько задач одновременно без использования отдельных потоков.

Событийный цикл (event loop) - это цикл обработки событий, который управляет выполнением корутин в асинхронной программе. Он ожидает поступления событий (например, запросов на ввод-вывод или сигналов) и вызывает соответствующие обработчики для обработки этих событий. Event loop запускается при старте программы и продолжает работу до завершения программы.



```python
import asyncio

async def print_numbers():
    for i in range(1, 6):
        print(i)
        await asyncio.sleep(1)

async def print_letters():
    for letter in 'abcde':
        print(letter)
        await asyncio.sleep(0.5)

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

    await task1
    await task2

asyncio.run(main())
```

```
import time

def read_file(filename):
    with open(filename, 'r') as f:
        time.sleeps(5) # имитация долгой операции чтения файла
        return f.read()
```

Простой пример блокировки операции ввода-вывода можно показать на примере чтения файла.

```
import asyncio

async def main():
    print ('Начало')
    result = await asyncio.gather(read_file('example.txt'))
    print('Конец:', result)

asyncio.run(main())
```

Пример с использованием неблокирующих функций, таких как asyncio.aopen

asyncio.aopen возвращает объект asyncio.StreamReader, который можно использовать для чтения содержимого файла в асинхронном режиме:

```
import time

async def read_file_async(filename):
    async with asyncio.aopen(filename, 'r') as f:
        content = await f.read()
        return content
```

Теперь, если мы используем эту функцию внутри сопрограммы, она не будет блокировать выполнение других задач:

```
import asyncio

async def main():
    print ('Начало')
    result = await asyncio.gather(read_file('example.txt'))
    print('Конец:', result)

asyncio.run(main())
```

Веб-скрапинг

```
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSessions() as session:
        html = await fetch(session, 'https://www.example.com')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Боты для социальных сетей

```
import tweepy
import asyncio

class MyStreamListener(tweepy.StreamListener):
    async def on_status(self, status):
        print(status.text)

async def main():
    auth = tweepy.0AuthHandler('consumer_key', 'consumer_secret')
    auth.set_access_token('access_token', 'access_token_secret')

    api = tweepy.API(auth)
    myStreamListener = MyStreamListener()
    myStream = tweepy.Stream(auth=api.auth, listener=myStreamListener)

    mySrteam.filter(track=['python'], is_async=True)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Вычисления на GPU

```python
import torch
import asyncio

# Функция для обучения модели на заданном устройстве (GPU)
async def train_on_device(device, x, y):
    # Создаем нейронную сеть
    model = torch.nn.Sequential(
        torch.nn.Linear(1, 10),
        torch.nn.ReLU(),
        torch.nn.Linear(10, 1)
    )
    # Определяем функцию потерь и оптимизатор
    loss_fn = torch.nn.MSELoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

    # Перемещаем модель и данные на указанное устройство
    model.to(device)
    x = x.to(device)
    y = y.to(device)

    # Обучаем модель
    for i in range(1000):
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```python

    return model

# Основная функция
async def main():
    # Генерируем случайные данные
    x = torch.randn(1000, 1)
    y = x * 2 + torch.randn(1000, 1) * 0.1

    # Обучаем модель на нескольких GPU
    devices = ['cuda:0', 'cuda:1']
    tasks = [asyncio.create_task(train_on_device(device, x, y)) for device in devices]
    models = await asyncio.gather(*tasks)

    # Объединяем модели, обученные на разных устройствах, в одну модель
    combined_model = torch.nn.Sequential(
        torch.nn.Linear(1, 10),
        torch.nn.ReLU(),
        torch.nn.Linear(10, 1)
    )
    for model in models:
        combined_model.load_state_dict(model.state_dict())
    print(combined_model)

# Запускаем основную функцию с помощью asyncio.run
asyncio.run(main())
```

ГЛАВА 3

ИСПОЛЬЗОВАНИЕ ASYNCIO

Asyncio - это стандартная библиотека Python для асинхронного программирования.

"Future" - это объект, представляющий результат выполнения асинхронной операции. Как правило, объект "Future" создается для операции, которая выполняется в отдельном потоке или процессе, и должен быть получен или преобразован в другой формат в асинхронном коде.

"Task" - это подкласс "Future", который используется для запуска асинхронной операции. "Task" обычно создается внутри асинхронной функции, и его выполнение может быть отложено, пока не будут завершены другие операции.

Для создания объекта "Future" в asyncio можно использовать функцию "asyncio.Future()". Для создания объекта "Task" можно использовать функцию "asyncio.create_task()".

Пример использования объектов "Future" и "Task":

```
import asyncio

async def coro():
    await asyncio.sleep(1)
    return “Result”

async def main():
    # Создаем объект Future
    fut = asyncio.Future()

    # Создаем объект Task
    task = asyncio.create_task(coro())

    # Получаем результат из объекта Future
    result = awair fut

    # Получаем результат из объекта Task
    result = await task

asyncio.run(main())
```

У объектов "Future" и "Task" в Python есть следующие методы:

1. **. add_done_callback(callback) -** добавляет обратный вызов, который будет вызван после завершения операции.
2. **cancel()** - пытается отменить операцию.
3. **done()** - возвращает True, если операция завершена.
4. **exception()** - возвращает исключение, если операция завершилась с ошибкой.
5. **result()** - возвращает результат выполнения операции.
6. **set_result(result)** - устанавливает результат выполнения операции.
7. **set_exception(exception)** - устанавливает исключение, если операция завершена с ошибкой.

```python
import asyncio

async def some_task():
    await asyncio.sleep(1)
    return “Task completed”

async def main():
    task = asyncio.create_task(some_task())
    task.add_done_callback(callback)
    print(task.done()) # False
    task.camcel()
    print(tasr.done()) # True
    try:
        await task
    except asyncio.CancelledError:
        print(“Task was cancelled”)
    else:
        print(task.result()) # This line will not be reached
    print(task.exception()) # None
    task.set_result(“New result”)
    print(task.result()) # “New result”
    task.set_exception(ValueError(“New error”))
    print(task.exception()) # ValueError(“New error”)

def callback(future):
    print(“Callback called”)

asyncio.run(main())
```

Модуль AIOHTTP. Пример работы с сетевыми запросами и обработкой ответов:

```python
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'https://www.google.com')
        print(html)

asyncio.run(main())
```

Asyncio.gather(). Рассмотрим пример использования asyncio.gather() для параллельного выполнения нескольких запросов к API:

```python
import asyncio
import aiohttp

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        urls = [
            'https://jsonplaceholder.typicode.com/posts/1',
            'https://jsonplaceholder.typicode.com/posts/2',
            'https://jsonplaceholder.typicode.com/posts/3'
        ]
        tasks = [asyncio.create_task(fetch(session, url)) for url in urls]
        results = await asyncio.gather(*tasks)
        print(results)

asyncio.run(main())
```

Пример создания асинхронной функции и ее вызов внутри сопрограммы:

```
import asyncio

async def my_async_function():
    print("Start")
    await asyncio.sleep(1)
    print("End")

async def my_coroutine():
    print("Before")
    await my_async_function()
    print("After")

loop = asyncio.get_event_loop()
loop.run_until_complete(my_coroutine())
```

3. Обработка ошибок в асинхронном коде

В асинхронном коде для обработки ошибок используется блок try-except, как и в синхронном коде. Однако, в асинхронном коде возможны и другие варианты обработки ошибок, которые связаны с работой объектов "Future" и "Task".

```
async def func_with_error():
    print("Function with error started")
    await asyncio.sleep(1)
    raise ValueError("Error in function with error")
```

Затем, изменим сопрограмму, чтобы вызывать эту функцию:

```
async def main():
    print("Main started")
    tasks = [asyncio.create_task(func()), asyncio.create_task(func_with_error
    await asyncio.gather(*tasks)
    print("Main ended")
```

Теперь, при запуске программы, мы увидим следующий вывод:

```
Main started
Function started
Function with error started
Function started
Unhandled exception in asyncio.run() at ...:
...
ValueError: Error in function with error
```

Обработка ошибок с помощью объектов "Future" и "Task"

метод add_done_callback(), который позволяет добавить функцию обработки ошибки:

```
async def main():
    print("Main started")
    task1 = asyncio.create_task(func())
    task2 = asyncio.create_task(func_with_error())
    task3 = asyncio.create_task(func())
    for task in asyncio.all_tasks():
        task.add_done_callback(handle_error)
    await asyncio.gather(task1, task2, task3)
    print("Main ended")

def handle_error(task):
    if task.exception():
        print(f"Error occurred in task {task}: {task.exception()}")
```

Теперь, при запуске программы, мы увидим следующий вывод:

```
Main started
Function started
Function with error started
Function started
Error occurred in task <Task finished coro=<func_with_error() done, defined
Main ended
```

Использование контекстного менеджера asyncio.shield для игнорирования ошибок:

asyncio.shield() - этот метод используется для защиты coroutine от отмены, что позволяет обрабатывать ошибки в coroutine.

```python
import asyncio

async def task():
    await asyncio.sleep(1)
    raise Exception("Something went wrong")

async def main():
    try:
        task_future = asyncio.ensure_future(task())
        await asyncio.sleep(0.5)
        asyncio.shield(task_future)
    except Exception as e:
        print("Error occurred:", e)

asyncio.run(main())
```

Использование asyncio.wait() для обработки ошибок:

asyncio.wait() - это функция в модуле asyncio, которая позволяет выполнять несколько coroutine одновременно и ожидать их завершения.

```
import asyncio

async def coro1():
    print("Starting coro1")
    await asyncio.sleep(2)
    raise ValueError("Error in coro1")

async def coro2():
    print("Starting coro2")
    await asyncio.sleep(3)
    print("Ending coro2")

async def main():
    tasks = [asyncio.ensure_future(coro1()), asyncio.ensure_future(coro2())]
    done, pending = await asyncio.wait(tasks, return_when=asyncio.ALL_COMPLI

    for task in done:
        try:
            result = await task
        except Exception as e:
            print(f"Error in {task}: {e}")
        else:
            print(f"{task} completed: {result}")

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

В результате выполнения программы, вывод будет следующим:

```
Starting coro1
Starting coro2
Error in <Task pending name='Task-1' coro=<coro1() running at example.py:4>>: Error in coro1
<Task finished coro=<coro2() done, defined at example.py:8> result=None> completed: None
```

ГЛАВА 4

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ASYNCIO

1. Создание асинхронного сервера на базе asyncio:


```python
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print(f"Received {message!r} from {addr!r}")

    writer.write(data)
    await writer.drain()

    print(f"Sent {message!r} to {addr!r}")
    writer.close()

async def main():
    server = await asyncio.start_server(handle_echo, '127.0.0.1', 8888)
    addr = server.sockets[0].getsockname()
    print(f"Serving on {addr}")
    async with server:
        await server.serve_forever()

asyncio.run(main())
```

```python
import asyncio
import aiohttp

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://example.com')
        print(html)

asyncio.run(main())
```

3. Использование asyncio для параллельной обработки больших объемов данных:

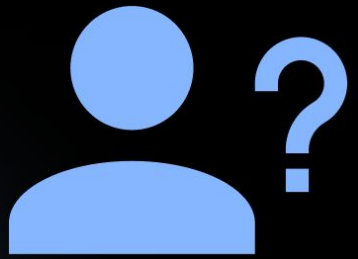

```python
import asyncio

async def process_item(item):
    # обработка отдельного элемента данных
    ...

async def process_items(items):
    tasks = []
    for item in items:
        task = asyncio.create_task(process_item(item))
        tasks.append(task)
    await asyncio.gather(*tasks)

async def main():
    # загрузка данных из базы данных или другого источника
    items = await load_items()
    await process_items(items)

asyncio.run(main())
```

- **Мы рассмотрели основные концепции и принципы работы с asyncio**
- **посмотрели на примеры использования этой библиотеки для решения различных задач.**

Дополнительные ресурсы, которые помогут вам углубить свои знания в этой области:

1. Официальная документация asyncio: https://docs.python.org/3/library/asyncio.html - здесь вы найдете полную документацию по библиотеке asyncio, включая описание классов и функций, примеры кода и руководства.
2. Официальный блог Python: https://blog.python.org/ - здесь вы найдете статьи и заметки от разработчиков Python, в том числе и по теме асинхронного программирования на Python.
3. "Asyncio in Python 3.7+" от Real Python: https://realpython.com/async-io-python/ - это статья от Real Python, в которой подробно объясняются основные концепции асинхронного программирования на Python с использованием asyncio.
4. Python 3.11+" от Real Python: https://realpython.com/python311-exception-groups/#asynchronous-task-groups-in-python-311 - это статья от Real Python, в которой рассматриваются новые возможности библиотеки asyncio, появившиеся в Python 3.11.
5. “Реализация асинхронности в Python с модулем asyncio” - руководство по модулю asyncio в Python https://pythonru.com/osnovy/python-asyncio
6. «Asyncio и конкурентное программирование на Python» Автор: Мэтью Фаулер https://dmkpress.com/catalog/computer/programming/python/978-5-93700-166-5/

Актуальная, развернутая шпаргалка по использованию asyncio



ВОПРОСЫ?

СПАСИБО ЗА ВНИМАНИЕ!

ЖДЕМ ВАШИ ОТЗЫВЫ