

Путь к карьере Python Fullstack разработчика

Модуль 1. PYTHON CORE

Уровень 14. Асинхронность в Python. Часть 2.



Класс Future

Основные особенности:

- **Создание:** Обычно создается с помощью `loop.create_future()`.
- **Установка результата:** `set_result(result)` устанавливает результат, когда задача завершилась успешно.
- **Установка исключения:** `set_exception(exception)` устанавливает исключение, если при выполнении задачи произошла ошибка.
- **Получение результата:** `result()` возвращает результат, если он доступен, иначе поднимает исключение.
- **Проверка завершения:** `done()` возвращает `True`, если задача завершилась.
- **Обратные вызовы:** `add_done_callback(callback)` позволяет добавить функцию, которая будет вызвана при завершении задачи.

Обработка исключений

```
import asyncio

async def set_future_exception(fut, delay):
    await asyncio.sleep(delay)
    fut.set_exception(ValueError("An error occurred"))

async def main():
    loop = asyncio.get_running_loop()
    fut = loop.create_future()
    asyncio.create_task(set_future_exception(fut, 2))
    try:
        result = await fut
    except ValueError as e:
        print(f"Caught an exception: {e}")

asyncio.run(main())
```

Взаимодействие с задачами

Часто объекты **Future** используются в связке с задачами (**Tasks**). Когда задача создаётся с помощью **asyncio.create_task()**, она автоматически создаёт объект **Future**, который можно использовать для отслеживания и управления состоянием задачи.

```
import asyncio

async def example_coroutine():
    await asyncio.sleep(1)
    return "Task result"

async def main():
    task = asyncio.create_task(example_coroutine())
    print(await task)

asyncio.run(main())
```

Асинхронные контекстные менеджеры

Это специальные классы, которые позволяют нам автоматически выполнять какие-то действия перед началом работы с ресурсом (например, открыть файл) и после ее завершения (например, закрыть файл). Все это происходит внутри блока `async with`.

Как это работает?

- `__aenter__()`: Вызывается при входе в блок `async with`. Здесь мы можем выполнить какие-то подготовительные действия, например, открыть соединение с базой данных.
- `__aexit__()`: Вызывается при выходе из блока `async with`. Здесь мы можем выполнить очистку, например, закрыть соединение.

```
import asyncio

class AsyncContextManager:
    async def __aenter__(self):
        print("Enter context")
        return self

    async def __aexit__(self, exc_type, exc, tb):
        print("Exit context")

async def main():
    async with AsyncContextManager():
        print("Inside context")

asyncio.run(main())
```

Управление соединениями с помощью aiohttp

aiohttp — это популярная библиотека для выполнения асинхронных HTTP-запросов.

```
import aiohttp
import asyncio

async def fetch_page(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    html = await fetch_page('https://api.github.com/users/defunkt')
    print(html)

asyncio.run(main())
```

Методы `__aenter__` и `__aexit__` позволяют выполнять асинхронные операции при входе и выходе из контекста, обеспечивая параллельное выполнение задач. Использование асинхронных контекстных менеджеров помогает избежать утечек ресурсов и гарантирует, что все ресурсы будут корректно освобождены.

Асинхронные итераторы

Представьте, что вы хотите прочитать большой файл по частям, не загружая его весь сразу в память. Или, например, получать данные с сервера по мере их поступления. Для таких задач идеально подходят **асинхронные итераторы**.

Асинхронный итератор — это специальный объект, который позволяет поочередно получать элементы из последовательности, но делает это асинхронно.

Как это работает?

- `__aiter__()`: Этот метод возвращает сам итератор. Вызывается в начале итерации.
- `__anext__()`: Этот метод возвращает следующий элемент последовательности. Если элементов больше нет, он вызывает исключение `StopAsyncIteration`.

AsyncIterator

Что здесь происходит:

1. Мы создаем класс `AsyncIterator`, реализующий интерфейс асинхронного итератора.
2. Метод `__anext__()` имитирует асинхронную операцию (например, запрос к базе данных) и возвращает следующий элемент.
3. Цикл `async for` позволяет удобно работать с асинхронными итераторами.

```
import asyncio
```

```
class AsyncIterator:
```

```
    def __init__(self, start, end):
```

```
        self.current = start
```

```
        self.end = end
```

```
    def __aiter__(self):
```

```
        return self
```

```
    async def __anext__(self)(self):
```

```
        if self.current >= self.end:
```

```
            raise StopAsyncIteration
```

```
        await asyncio.sleep(1) # Имитация асинхронной задержки
```

```
        self.current += 1
```

```
        return self.current
```

```
async def main():
```

```
    async for number in AsyncIterator(1, 5):
```

```
        print(number)
```

```
asyncio.run(main())
```

Асинхронные генераторы

Представьте, что у вас есть очень большой файл, и вы хотите обрабатывать его построчно, но не загружать весь файл сразу в память. Или, например, вы хотите получать данные с сервера по мере их поступления. Для таких задач идеально подходят **асинхронные генераторы**.

Асинхронный генератор — это функция, которая может приостанавливать свое выполнение и возобновлять его позже, при этом генерируя значения по мере необходимости.

```
import asyncio

async def async_generator():
    for i in range(5):
        await asyncio.sleep(1)
        yield i

async def main():
    async for value in async_generator():
        print(value)

asyncio.run(main())
```

Как это работает?

- **async def**: Определяет асинхронную функцию.
- **yield**: Выдает значение и приостанавливает выполнение функции до следующего вызова.
- **await**: Ожидает завершения асинхронной операции.

Почему Python "боится" многопоточности

Global Interpreter Lock (GIL) — это механизм, который ограничивает одновременное выполнение нескольких потоков Python. Это значит, что даже если у вас многоядерный процессор, Python будет выполнять код только в одном потоке за раз.

Почему так?

- **Упрощение реализации:** GIL упрощает управление памятью и сборку мусора в Python.
- **Потокобезопасность:** GIL предотвращает возникновение ошибок, связанных с одновременным доступом к одним и тем же данным.

Но есть и обратная сторона:

- **Ограничение производительности:** В многопоточных приложениях, особенно вычислительно интенсивных, GIL может значительно снизить производительность, так как только один поток может выполняться в данный момент.
- **Неполное использование многоядерных процессоров:** Если у вас многоядерный процессор, Python не сможет эффективно использовать все ядра.

Как обойти это ограничение?

Многопроцессорность (multiprocessing)

- Создает отдельные процессы Python, каждый со своим собственным GIL.
- Подходит для задач, которые могут быть разбиты на независимые части.

Асинхронное программирование (asyncio)

- Позволяет выполнять I/O-bound задачи параллельно, не блокируя основной поток.
- Идеально подходит для задач, связанных с вводом-выводом (например, сетевые запросы).

Использование библиотек с собственным управлением потоками

- Библиотеки, написанные на C/C++ (например, NumPy, SciPy), могут обходить GIL для числовых вычислений.
- Это позволяет эффективно использовать многоядерные процессоры для научных вычислений.

Выполнение вычислений вне интерпретатора Python

- Использование расширений на C/C++ или других языках для выполнения ресурсоемких вычислений.

End



Домашнее задание

Модуль 1. PYTHON CORE

Уровень 14. Асинхронность в Python.

