Приглашаем преподавателей поучаствовать в интервью с нашей командой исследований. Если вы готовы участвовать, пожалуйста, заполните форму. Приглашение на интервью.

Урок Основы

Основы асинхронного программирования с использованием библиотеки asyncio

1. Введение

Как мы с вами уже выяснили на прошлом уроке, асинхронное программирование позволяет выполнять другие задачи, пока основная задача ждет завершения операции ввода-вывода (I/O). Это особенно полезно при работе с сетевыми запросами, файлами и другими операциями, которые могут занять значительное время, но не даст никакого выигрыша при вычислительных задачах. К счастью, пласт задач, где нам приходится долго ждать по не зависящим от нас причинам, настолько большой и часто встречающийся, что использовать не синхронный и не параллельный код имеет смысл.

Теперь давайте разберемся во всех этих async/await и прочей терминологии, которая добавляется с нашей асинхронностью и библиотекой asyncio. Для начала просто накинем непонятных слов, в которые мы будем сегодня погружаться:

- Корутина (Coroutine)
- Задача (Task)
- Цикл событий (event loop)
- Футура (Future)

2. Обычная функция и корутина

Все мы с вами знаем, как делать обычную синхронную функцию и что она из себя представляет.

```
def sync_func():
    return "Это обычная функция"


print(callable(sync_func), sync_func)
res = sync_func()
print(res)
```

После запуска такого кода мы увидем ожидаемое:

```
True <function sync_func at 0x000002A667313E20>
Это обычная функция
```

У нас есть объект-функция, которую можно вызвать, и сразу после вызова мы дождемся выполнения и получим результат.

Давайте посмотрим, что надо сделать, чтобы получить асинхронную функцию. На самом деле, не очень многое — достаточно перед ключевым словом `def` добавить еще одно ключевое `async` (сокращение от "асинхронный").

```
async def async_func():
    return "Это корутина"

print(callable(async_func), async_func)
res = async_func()
print(res)
```

Запустим и вот что получим:

```
True <function async_func at 0x00000204A32DEDD0>
<coroutine object async_func at 0x00000204A53C0430>
sys:1: RuntimeWarning: coroutine 'async_func' was never
```

Сначала мы получим на первый взгляд такой же объект функции, как и в первом примере, но отличия появятся уже в момент запуска. В переменную `res` у нас запишется не результат вызова, а объект корутины, а потом еще интерпретатор заботливо напишет предупреждение, что вообще-то результат выполнения корутины `async_func` никто не дождался. (Это предупреждение не уронит наше приложение, но в общем случае намекнет, что мы что-то делаем не так.)

Давайте сделаем первые выводы. Когда интерпретатор Python встречает ключевое слово `async` перед определением функции (`async def`), он создает специальный объект корутины. Это делает функцию асинхронной, и она больше не возвращает результат напрямую. Вместо этого она возвращает объект корутины, который может быть использован для управления её выполнением. (Это не единственное место, где может быть использовано ключевое слово `async`, на следующих уроках мы посмотрим и на асинхронный контекстный менеджер и на асинхронный итератор.)

Иными словами, **корутина** — это специальная функция, которая не возвращает значение напрямую, а кроме того может приостанавливать своё выполнение, позволяя другим функциям выполняться, и потом возобновлять своё выполнение.

Однако, если мы попробуем воспользоваться ключевым словом `await` для получения результата, как нам на то намекало предупреждение интерпретатора, из корня нашего модуля, то мы получим следующее:

```
res = await async_func()
```

```
res = await async_func()
        ^^^^^^^^^^^^^^^^^
SyntaxError: 'await' outside function
```

Что ж, мы с вами понимаем намеки, и с легкостью засунем код из корня модуля внутрь функции:

```
def main():
    print(callable(async_func), async_func)
    res = await async_func()
    print(res)


if __name__ == '__main__':
    main()
```

Но и тут провал:

```
SyntaxError: 'await' outside async function
```

Но погодите, если нам можно вызывать асинхронные функции только внутри асинхронных функций, то как нам разорвать этот круг, чтобы вызвать наш самый первый асинхронный код?

Задав себе этот правильный вопрос, мы стали понимать достаточно, чтобы поговорить про цикл событий.

3. Цикл событий (Event loop)

Event Loop (цикл событий) — это центральный компонент асинхронного программирования, который управляет выполнением асинхронных задач, корутин и других асинхронных операций. Он постоянно проверяет, есть ли готовые к выполнению задачи, и запускает их, приостанавливает выполнение задач, которые ожидают завершения операций ввода-вывода, и возобновляет выполнение задач, когда операции ввода-вывода завершены.

Таким образом, основные задачи Event Loop:

- Управление корутинами и задачами: Event loop отвечает за запуск, приостановку и возобновление корутин и задач
- Ожидание ввода-вывода (I/O): Он следит за операциями ввода-вывода и возобновляет выполнение корутин, когда данные готовы
- Обработка событий: Event loop может обрабатывать таймеры, сигналы и другие события

Самый простой способ запустить нашу первую асинхронную функцию внутри синхронного кода, это написать:

```
import asyncio

async def async_func():
    return "Это корутина"

async def main():
    print(callable(async_func), async_func)
    res = await async_func()
```

```
    print(res)


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

Функция `asyncio.run()` является удобным и безопасным способом запуска асинхронных функций в `Python`. Она создаёт новый event loop, запускает корутину, ждёт её завершения и затем закрывает event loop, освобождая ресурсы. В целом, мы могли бы запустить и сразу нашу функцию `async_func`, но хорошей практикой считается создание главной асинхронной точки входа, которая уже потом будет управлять остальными асинхронными вызовами.

К счастью, для запуска синхронного кода из асинхронного никаких дополнительных действий выполнять не надо, все работает и так. Однако, если вам по какой-то неизвестной причине надо сделать асинхронную функцию, которая вызывает синхронную, которая в свою очередь вызывает асинхронную, то придется написать более сложную конструкцию. Но не пугайтесь, такая задача не будет возникать у вас часто (если вообще когда-нибудь возникнет).

Пример на сложную цепочку вызовов

Но мы все же давайте посмотрим, что надо сделать, чтобы сделать такую "матрешку" вызовов, на примере.

```
import asyncio

def sync_func():
    """Синхронная функция снова вызывает асинхронную"""
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    res = loop.run_until_complete(async_func2())
    loop.close()
    return res


async def async_func1():
    """Асинхронная функция вызывает синхронную"""
    loop = asyncio.get_running_loop()
    res = await loop.run_in_executor(None, sync_func)
    return "Это корутина" + " " + res

async def async_func2():
    """Другая асинхронная функция"""
```

```
    return "Это другая корутина"

async def main():
    res = await async_func1()
    print(res)


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

В этом нетривиальном случае нам даже синхронный код запускать по особому: надо получить текущий цикл событий и запустить синхронный код именно в нем, а внутри синхронной функции сделать новый цикл событий, запустить в нем другую функцию, дождаться завершения её работы и закрыть цикл. В принципе `asyncio.run` под капотом делает примерно тоже самое, но любезно от нас это скрывает.

Метод asyncio.gather

Помимо простого запуска корутин, у цикла событий есть еще и ряд методов для управления корутинами, например `asyncio.gather` позволяет запускать несколько корутин "параллельно" и ждать их завершения.

```
import asyncio

async def cor1():
    await asyncio.sleep(1)
    return "Задача 1 завершена"

async def cor2():
    await asyncio.sleep(2)
    return "Задача 2 завершена"

async def main():
    results = await asyncio.gather(cor1(), cor2())
    print(results)

asyncio.run(main())
```

А метод `asyncio.as_completed` позволяет обрабатывать задачи по мере их завершения.

```
import asyncio
```

```
async def cor1():
    await asyncio.sleep(3)
    return "Задача 1 завершена"

async def cor2():
    await asyncio.sleep(2)
    return "Задача 2 завершена"

async def main():
    cors = [cor1(), cor2()]
    for finished_cor in asyncio.as_completed(cors):
        res = await finished_cor
        print(res)

asyncio.run(main())
```

4. Задачи

Как вы уже наверное догадались, корутина — самый базовый объект асинхронного программирования. Конечно, для написания асинхронного кода корутин достаточно самих по себе, но программисты не были бы программистами, если бы не написали поверх базового что-то более продвинутое. Поэтому встречайте — задачи.

Task (задача) — это обертка вокруг корутины, которая позволяет её запускать, отслеживать её состояние и получать результат. Когда вы создаете задачу, она автоматически добавляется в event loop для выполнения.

Чтобы создать задачу, необходимо вызвать функцию `asyncio.create_task`, в которую передать корутину.

```
import asyncio

async def coroutine():
    print("Корутина начинается")
    await asyncio.sleep(1)
    print("Корутина заканчивается")
    return 42

async def main():
    task = asyncio.create_task(coroutine())
```

```
    res = await task
    print(res)

asyncio.run(main())
```

Обратите внимание, что внутри корутины мы используем неблокирующий `asyncio.sleep` — это как раз тот момент, когда цикл событий может переключиться на выполнение другой функции (если бы она у нас была), при использовании обычного `time.sleep` такой магии бы не случилось и потенциальный выигрыш от использования асинхроного кода превратился бы в тыкву.

Давайте проговорим разницу между корутиной и задачей:

- Корутина — это функция, которая может приостановить выполнение и возобновить его позже
- Task (задача) — это объект, который управляет выполнением корутины. Он позволяет запустить корутину и отслеживать её выполнение

Когда вы вызываете корутину напрямую, вы получаете объект корутины, который нужно передать в event loop. Задача, с другой стороны, автоматически добавляется в event loop и начинает выполняться. Это в свою очередь можно прекрасно использовать для группировки задач для их одновременного запуска на выполнение.

Пример с несколькими задачами:

```
import asyncio

async def cor1():
    await asyncio.sleep(1)
    print("Задача 1 завершена")

async def cor2():
    await asyncio.sleep(2)
    print("Задача 2 завершена")

async def main():
    tasks = [
        asyncio.create_task(cor1()),
        asyncio.create_task(cor2())
    ]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

В этом примере две задачи создаются и запускаются параллельно. Функция `asyncio.gather` умеет работать не только с корутинами, но и с задачами (и даже рекомендуется работать с задачами) и позволяет дождаться завершения всех задач. При этом мы можем присвоить результат `gather` в переменную и там будет все то, что нам возвратили наши задачи в том порядке, в котором мы их добавили в список задач.

Но это еще не все, задачи можно отменять, если они вдруг выполняются дольше, чем нам бы хотелось:

```
import asyncio

async def long_cor():
    print("Начало долгосрочной задачи")
    try:
        await asyncio.sleep(10)
        print("Завершение долгосрочной задачи")
    except asyncio.CancelledError:
        print("Долгосрочная задача была отменена")

async def main():
    task = asyncio.create_task(long_cor())
    await asyncio.sleep(1)  # Ждем 1 секунду
    task.cancel()  # Отменяем задачу
    await task

asyncio.run(main())
```

И это снова не все. Бывают случаи, когда результат асинхронной операции нам может понадобится не прямо сейчас, а когда-нибудь в будущем. И тут на помощь нам приходят футуры (Future).

5. Футуры (Future)

Future (футура) — это объект, представляющий результат асинхронной операции, который может быть доступен в будущем. Футуры могут быть использованы для передачи результата между различными частями асинхронного кода. Они могут быть в одном из следующих состояний:

- Pending (ожидание): операция еще не завершена

- Done (завершена): операция завершена, и результат доступен
- Cancelled (отменена): операция была отменена

Футуры в `asyncio` используются для низкоуровневого управления результатами асинхронных операций.

Футуры могут быть созданы через цикл событий или с помощью метода `asyncio.Future()`. Пример создания футура с использованием цикла событий:

```
import asyncio

async def set_result(fut):
    await asyncio.sleep(1)
    fut.set_result("Результат установлен")

async def main():
    loop = asyncio.get_running_loop()
    fut = loop.create_future()
    asyncio.create_task(set_result(fut))
    res = await fut
    print(res)

asyncio.run(main())
```

Пример создания футура с использованием `asyncio.Future()`:

```
import asyncio

async def set_result(fut):
    await asyncio.sleep(1)
    fut.set_result("Результат установлен")

async def main():
    fut = asyncio.Future()
    asyncio.create_task(set_result(fut))
    res = await fut
    print(res)

asyncio.run(main())
```

Футуры в `asyncio` имеют несколько полезных методов и атрибутов:

- `set_result(result)`: Устанавливает результат футура
- `result()`: Возвращает результат футура, если он завершен
- `set_exception(exception)`: Устанавливает исключение в футур
- `exception()`: Возвращает исключение, если футур завершился с исключением
- `done()`: Проверяет, завершился ли футур
- `cancel()`: Отменяет футур
- `cancelled()`: Проверяет, был ли футур отменен

Еще несколько небольших примеров

Обработка исключения

```
import asyncio

async def set_exception(fut):
    await asyncio.sleep(1)
    fut.set_exception(ValueError("Произошла ошибка"))

async def main():
    fut = asyncio.Future()
    asyncio.create_task(set_exception(fut))
    try:
        res = await fut
        print(res)
    except ValueError as e:
        print(f"Исключение: {e}")

asyncio.run(main())
```

Отмена футура

```
import asyncio

async def canseling_cor(fut):
    await asyncio.sleep(1)
    if not fut.cancelled():
        fut.set_result("Результат установлен")

async def main():
    fut = asyncio.Future()
```

```
    task = asyncio.create_task(canseling_cor(fut))
    await asyncio.sleep(0.5)
    fut.cancel()
    try:
        res = await fut
        print(res)
    except asyncio.CancelledError:
        print("Футур был отменен")

asyncio.run(main())
```

Наиболее часто футуры используются для координации между различными частями асинхронного кода. Например, футуры могут быть использованы для ожидания завершения асинхронных операций и передачи результатов между ними.

```
import asyncio

async def cor1(fut):
    await asyncio.sleep(2)
    fut.set_result("Задача 1 завершена")

async def cor2(fut):
    await fut  # Ждем, пока задача1 установит результат
    print(fut.result())
    print("Задача 2 завершена")

async def main():
    fut = asyncio.Future()
    asyncio.create_task(cor1(fut))
    await cor2(fut)

asyncio.run(main())
```

6. Заключение

Итак, сегодня мы узнали, что:

- Корутина — асинхронная функция, приостанавливающая и возобновляющая выполнение

- Task (задача) — обертка вокруг корутины, которая управляет её выполнением
- Future (футура) — низкоуровневый примитив для представления будущего результата асинхронной операции
- Цикл событий разруливает все это великолепие

Использование корутин, задач и футур позволяет эффективно организовывать асинхронное выполнение кода, управлять состояниями задач и передавать результаты между различными частями асинхронных программ. Понятно, что большая часть кода пока выглядит "учебно", а польза от многих вещей на текущем этапе скорее теоретическая, но уже на следующем уроке мы начнем знакомиться с большими библиотеками, которые построены на фундаменте asyncio для решения реальных прикладных задач.