Асинхронность в python: aiohttp

Что может быть лучше котов? Только очень много котов. Напишем скрипт, который будет асинхронно загружать картинки сгенерированных нейронной сетью котов с сайта Cataas.

В библиотеке asyncio не содержатся методы для работы с сетью или HTTP, поэтому необходимо установить стороннюю библиотеку. Самая распространённая из них — AIOHTTP. В ней находятся:

- клинский код — тот, при помощи которого делают HTTP-запросы;
- серверный код — тот, которым обрабатывают HTTP-запросы.

Для написания асинхронных веб-приложений рекомендуется использовать библиотеку FastAPI. Её мы разберём в следующем уроке.

Установим библиотеку:

```
pip install aiohttp==3.8.0
```

Затем создадим каркас приложения:

```
import asyncio

import aiohttp

URL = 'https://cataas.com/cat'
CATS_WE_WANT = 10

async def get_cat():
    ...


async def get_all_cats():
    ...

def main():
    res = asyncio.run(get_all_cats())w
```

Импортируем нужные библиотеки и определим константы.
Необходимо написать несколько корутин:

- основную, которая станет точкой входа в асинхронную часть приложения;
- дополнительную, в которой будут создаваться HTTP-запросы.

Пока без аннотаций типов, так как писать будем в несколько итераций.

А дальше находится точка входа в приложения, где мы вызовем asyncio.run. Реализуем корутину get_all_cats:

```
async def get_all_cats():

   async with
aiohttp.ClientSession(timeout=aiohttp.ClientTimeout(15)) as
client:
       tasks = [get_cat(client) for _ in range(CATS_WE_WANT)]
       return await asyncio.gather(*tasks)
```

Внутри неё нужно инициализировать клиент, которым будем делать HTTP-запросы. Он похож на объект Session из библиотеки Requests, то есть внутри хранится пул соединений, который используется повторно между запросами. Обратите внимание на новый синтаксис: асинхронной может быть не только дефиниция функции, но и инициализация контекстного менеджера.

Чтобы реализовать асинхронный контекстный менеджер, нужно создать два метода:

- aeter,
- aexit.

Эти методы должны быть корутинами. Смысл их работы совпадает с идеей обычного контекстного менеджера. Асинхронный контекстный менеджер можно реализовать с помощью библиотеки contextlib. Подробнее смотрите в статье Utilities for with-statement contexts.

```
class AsyncContext:
   def __init_(self, something):
       self.something = something
   async def __aenter__():
       self.thing = await get_thing(self.something)
       return self.thing
   async def __aexit__(self, exc_type, exc, tb):
       await self.thing.close()
```

Далее внутри контекстного менеджера aiohttp.ClientSession инициализируем корутины, передаём в неё объект клиента. Частая ошибка на этом этапе — вызов await.

```
tasks = [await get_cat(client) for _ in range(CATS_WE_WANT)]
```

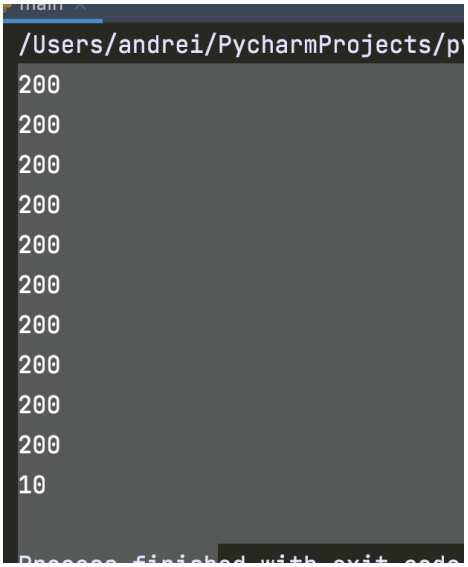
Сделаем список корутин, который выполним чуть позже. Чтобы передать их в ивент-пул, необходимо использовать метод asyncio.gather. Он принимает на вход список awaitable объектов и запускает их в многозадачном режиме. Если в работе не было ошибок, появляется список того, что вернули объекты.

Реализуем функцию, ответственную за HTTP-запрос. На вход она принимает объект клиента, Session.

```
async def get_cat(client: aiohttp.ClientSession):
    async with client.get(URL) as response:
        print(response.status)
        result = await response.read()
        return result
```

Это асинхронный контекстный менеджер с методами, которые соответствуют разным HTTP-глаголам. Нам необходим метод GET. Внутри контекстного менеджера работаем с объектом response. После обработки запроса зайдём в блок кода внутри контекстного менеджера, так как async with эквивалентен ключевому слову await: возвращаем контроль выполнения планировщику и засыпаем.

Далее обратимся к атрибуту status, который становится доступным после объекта ответа. Однако открыт не весь ответ, а лишь часть с заголовками. Обязательная часть HTTP-ответа не занимает много места, а массивное тело может быть ещё в пути. На следующей строчке метод read вернёт тело ответа. Дальше остаётся лишь вернуть его. Запустим этот код и проверим, что всё работает.



Статус коды появляются практически мгновенно. Теперь необходимо записать данные на диск. Вызов функции open будет блокирующим и существенно замедлит код. Заменить эту функцию для IO операций с диском можно при помощи библиотеки aiofiles.

Установим библиотеку aiofiles:

```
pip install aiofiles==0.7.0
```

Опишем отдельную корутину, которая будет вести запись на файл:

```
async def write_to_disk(content: bytes, id: int):
    file_path = "{}/{}.png".format(OUT_PATH, id)
    async with aiofiles.open(file_path, mode='wb') as f:
        await f.write(content)
```

На вход она будет принимать байты, которые вернул сервис и ID картинки. Мы пробросим ID в момент инициализации корутины в списковом выражении.

```
tasks = [get_cat(client, i) for i in range(CATS_WE_WANT)]
```

Также нужно обновить сигнатуру корутины get_cats:

```
async def get_cat(client: aiohttp.ClientSession, id: int):
```

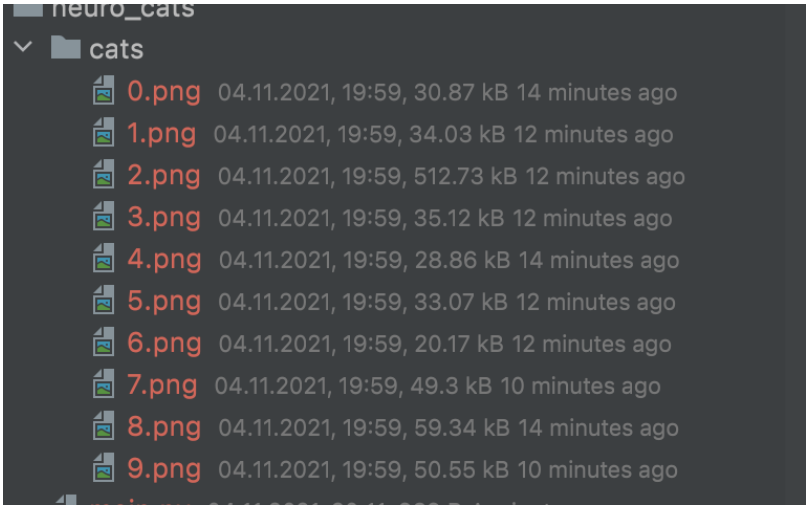
Заведём переменную для хранения пути, по которому необходимо сохранять картинки:

```
OUT_PATH = (Path(__file__).parent / 'cats').absolute()
```

Передадим байты в новую корутину для записи:

```
result = await response.read()
await write_to_disk(result, id)
```

Запустим этот скрипт. Десять картинок успешно загружаются.



В этом модуле вы познакомились с основами асинхронного программирования. Познакомились с подводными камнями генераторов и способами работы с ними. Теперь вы понимаете, как они связаны с асинхронностью в Python. Знаете об эволюции написания корутин: через генератор или через нативный способ с помощью ключевого слова await. Можете асинхронно получить десять картинок котов, используя библиотеки AIOHTTP.

До встречи в следующем модуле.