

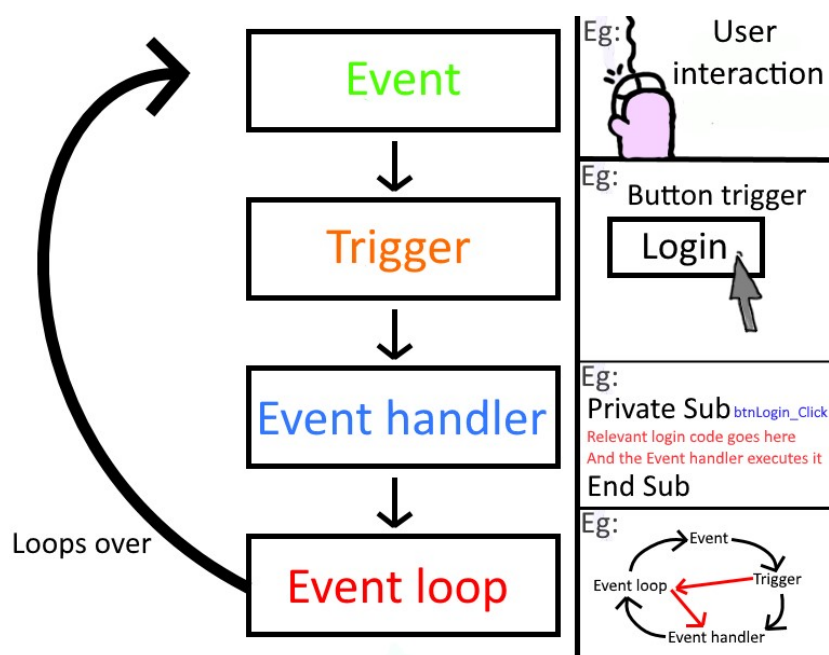
Асинхронность в Python: event loop и корутины

В Python асинхронность достигается за счёт использования событийно-ориентированного программирования (event-driven programming в английском).

В уже известных нам программах, будь то треды или процессы, поток выполнения был вполне детерминирован. Сначала выполни А, потом выполни Б и так далее. Хотя они и могли быть недетерминированы в плане затраченного времени или памяти, но в целом мы понимали от и до, что должно быть выполнено. В случае event-driven programming это не совсем так.

Как правило, у нас есть event loop, который постоянно слушает и ждёт входящих событий. Как только он будет запущен, он начнёт бесконечно обрабатывать события, поступающие в систему, будет решать, как и в каком порядке это сделать.

Пример — ввод с клавиатуры. Мы никак не можем предсказать, какая клавиша будет нажата следующей. Но у нас будет создана специальная функция-обработчик, которая будет запускать некую сущность при нажатии на определённую клавишу — то есть при получении события <клавиша-нажата> мы будем понимать, какая клавиша или комбинация клавиш нажата, и что-то с этим сделаем. Например, так работает комбинация Ctrl + C, когда мы хотим выключить работающий слишком долго Python-скрипт.



Этот концепт не специфичен для Python, а широко используется во многих языках и программах. В I/O-задачах, а это, в частности, HTTP-запросы, мы тратим много времени на простое ожидание ответа. Так почему бы нам не использовать это время на что-нибудь другое? А ожидающую ответа операцию сделать неблокирующей. Этого мы можем достичь за счёт тредов или асинхронного программирования.

Треды — довольно долго и дорого, ведь мы произведём огромное множество системных вызовов и к тому же будем терять время на смене контекста. Здесь уместно вспомнить, почему сервер Nginx такой быстрый, гораздо быстрее сервера Apache. Именно потому, что внутри него крутится такой event loop, а внутри Apache создаются треды для обработки запроса.

Как вы уже наверняка догадались, асинхронность в Python работает именно за счёт использования такого event loop. Находится он в библиотеке `asyncio`, и, скорее всего, нам редко придётся использовать более чем один-два метода у него, если только вы не будете разрабатывать асинхронные библиотеки, для которых нужно низкоуровневое API.

```
43 import asyncio

loop = asyncio.get_event_loop()
print(loop)

<_UnixSelectorEventLoop running=True closed=False debug=False>
```

Отлично, теперь что бы нам запустить в event loop? Конечно же, корутины. Или сопрограммы, если говорить на русском. Корутины — это тоже не специфичное для Python явление, а термин, относящийся к кооперативной многозадачности. Помните, мы его разбирали? Освежим примером: когда программа приостанавливает своё выполнение где-то посередине, полностью сохраняя своё состояние, стек вызова, переменные и так далее, она возвращает контроль в другое место, а конкретно — планировщику.

Планировщик даёт запускаться другим сопрограммам, проходит некоторое время, и очередь доходит до нашей изначальной корутины. Далее, будучи запущенной, она продолжает исполнение с того места, где остановилась. Обычная же функция, в данном случае это будет подпрограмма, имеет ровно одну точку входа — мы вызываем её с определёнными аргументами и никак иначе. Корутина же имеет множество точек входа — как вы уже успели догадаться, используя генераторы, мы писали именно их в прошлом уроке.

Итак, первый способ написать асинхронный код на Python — использовать классические генераторы и написать свой собственный шедюлер. До Python 3.4, когда появился модуль `asyncio`, асинхронные приложения именно так и писали. Далее в Python 3.4 появились так называемые `generator-based` корутины. То есть корутины на основе генераторов. И вместе с ними `event loop`, который позволял эти корутины обрабатывать.

Как это работало? Модуль `asyncio` предоставлял специальный декоратор, обернув в который генератор или функцию, мы получали корутину, с которой уже можно было работать в асинхронном коде.

Стандартная корутина выглядела так:

```
@asyncio.coroutine
def generator_based_coroutine():
    print("Hello World")
    # yield from asyncio.sleep(1)
```

Прежде всего заметим, что, начиная с Python 3.8, этот декоратор — `deprecated` (как видим, PyCharm перечеркнул его, // тут не смог сделать) — это значит, что в скором времени поддержка этой функции будет прекращена. А с приходом Python версии 3.10 они и вовсе будут удалены.

Нужно использовать встроенный способ создания корутин, о котором мы поговорим чуть позже.

Что мы тут видим?

- Декоратор `coroutine`. Обёрнутая в него функция начинает вести себя чуть-чуть по-другому. Если мы обернём обычную функцию, то внутри Python превратит её в генератор — корутину, на которой вызовет `yield from`. Если же раскомментировать строки со `sleep()`, то он вернёт ту же самую функцию, ведь это уже генератор. Эта подготовка необходима, чтобы в дальнейшем функция стала совместима с теми объектами, которые используются внутри `event loop` для организации кооперативной многозадачности.
- Далее мы вызываем некоторый произвольный код. В данном случае вызов `print` не является каноничным, всё же эта функция блокирующая.
- И наконец, если мы раскомментируем строку со `sleep()`, то так мы можем вернуть некоторое значение из другой корутины.

Что это нам даёт? Описав несколько корутин таким образом и передав их в event loop, мы напишем своё приложение с кооперативной многозадачностью. У нас будет ровно один тред, внутри которого будет работать бесконечный цикл, то есть event loop. Он будет менеджить все запущенные корутины: запускать их, ждать, пока они или завершат свою работу полностью (простой print), или уйдут в ожидание ответа от другой корутины (yield from). В последнем случае основная корутина уйдёт в состояние suspend, а asyncio.sleep() начнёт исполнение.

Кстати, как вы думаете, почему мы не пользуемся стандартным time.sleep()? Попробуйте дать себе ответ на этот вопрос: чем чревато выполнение такого кода внутри корутины?

Вызов любого синхронного кода приведёт к блокированию. Корутина, вместо того чтобы прервать своё исполнение и уступить очередь чему-то ещё, займёт всё эфирное время. В итоге профита от асинхронности не будет вообще, скорее всего, будет даже медленнее, чем в синхронном коде. И что самое неприятное, никаких ошибок на этот счёт не будет! Именно поэтому мы используем специальную функцию asyncio.sleep(), чтобы «усыпить» корутину на определённое время.

Теперь самое интересное: как же нам это дело запустить? Нужен event loop.

```
@asyncio.coroutine
def print_two_powers(name: str, limit: int, frequency: Union[int,
float] = 1):
    for i in range(limit):
        print(name, 2 ** i)
        yield from asyncio.sleep(frequency)
    else:
        print("Done")

@asyncio.coroutine
def coroutine_1():
    yield from print_two_powers('Worker_1', 10, 0.1)

@asyncio.coroutine
def coroutine_2():
    yield from print_two_powers('Worker_2', 10, 0.5)
```

Напишем три корутины. Первая — основная «логика». Печатаем степени двойки до некоторого числа, указываем, как часто это делать, и говорим, кто именно их печатает.

Далее две корутины, которые запускают эту функцию с разными параметрами. Одна чуть чаще, другая чуть реже. В теории они должны закончить работать в разное время.

Теперь нам нужно их запустить. Для этого напишем следующее:

```
@asyncio.coroutine
def main():
    task1 = asyncio.create_task(coroutine_1())
    task2 = asyncio.create_task(coroutine_2())
    yield from task1
    yield from task2

if __name__ == '__main__':
    start = time.time()
    asyncio.run(main())
    print(time.time() - start)
```

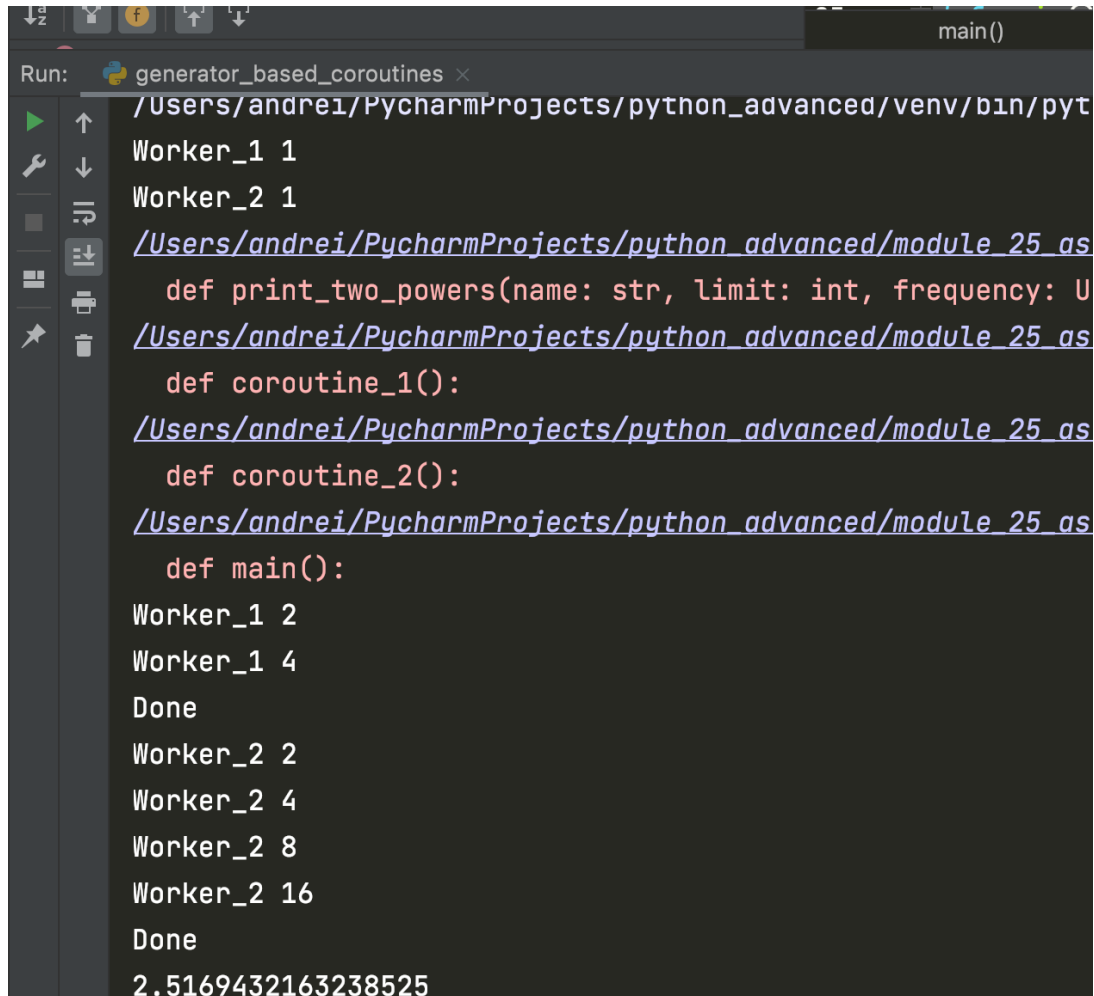
Напишем ещё одну корутину, которая будет точкой входа в наше приложение. Заметили, что мы вызываем `yield from` не напрямую у наших корутин, а у чего-то, что нам вернула функция `create_task`? Эта функция делает то, что и есть в названии — создаёт объект типа `Task`. Это специальный объект, который «шедулит» запуск корутины в будущем.

Давайте подумаем, как наш код работает в стандартном запуске через ключевое слово вызова `yield from`: сначала мы асинхронно дождёмся результата работы первой корутины, потом асинхронно дождёмся результата работы второй. Даже звучит бессмысленно. Так что мы делаем что-то вроде того, что делали с тредами — оборачиваем нужный нам код в служебный объект, который запустится где-то в «бэкграунде». Делаем такой запуск неблокирующим. Ну а дальше просто ждём конца работы.

И в самом низу вызовем высокоуровневую функцию `run`. Она появилась в Python версии 3.7. Внутри создаётся `event loop`, который запускает нашу корутину и работает до тех пор, пока она не выполнится. Данный способ считается стандартным и наиболее используемым для запуска асинхронных приложений.

Кстати, весь модуль `asyncio` написан на Python, так что вы легко можете самостоятельно изучить его внутренности.

Давайте запустим код и посмотрим на результат.



```
Run: generator_based_coroutines x
/Users/andrei/PycharmProjects/python_advanced/venv/bin/pyt
Worker_1 1
Worker_2 1
/Users/andrei/PycharmProjects/python_advanced/module_25_as
def print_two_powers(name: str, limit: int, frequency: U
/Users/andrei/PycharmProjects/python_advanced/module_25_as
def coroutine_1():
/Users/andrei/PycharmProjects/python_advanced/module_25_as
def coroutine_2():
/Users/andrei/PycharmProjects/python_advanced/module_25_as
def main():
Worker_1 2
Worker_1 4
Done
Worker_2 2
Worker_2 4
Worker_2 8
Worker_2 16
Done
2.5169432163238525
```

Видим пару предупреждений о том, что мы используем deprecated-код. Да, мы знаем и больше так не будем. И далее результат, очень похожий на тот, которого мы могли бы добиться, используя треды — две задачи работали асинхронно, независимо друг от друга и закончили в разное время. Что и требовалось. И не забываем, что всё это происходит в одном и том же треде. То есть профит по ресурсам и скорости должен быть существенным.

Ещё один важный комментарий, хоть он и очевидный, но его стоит проговорить: бессмысленно использовать асинхронность на CPU-bound задачах, ибо они будут блокирующими. Неспроста модуль называется `asyncIO`.

Основная мотивация добавления новых ключевых слов — конструкций `async/await` — была в том, чтобы визуально разделить синхронный и асинхронный код. Python предложил нативную поддержку корутин.

Давайте перепишем наш код, используя новый синтаксис:

```
async def print_two_powers(name: str, limit: int, frequency:
Union[int, float] = 1):
    for i in range(limit):
        print(name, 2 ** i)
        await asyncio.sleep(frequency)
    else:
        print("Done")

async def coroutine_1():
    return await print_two_powers('Worker_1', 3, 0.1)

async def coroutine_2():
    return await print_two_powers('Worker_2', 5, 0.5)
```

Удаляем все декораторы и добавляем в дефиницию функций ключевое слово `async`. Таким синтаксисом мы объявляем корутины, эквивалентные тому, что получали из декоратора.

Далее нам нужно заменить `yield from` на `await` — то есть дождаться результата из.

Каждый раз, когда в коде встречается слово `await`, функция отдаёт контроль обратно event loop'у до тех пор, пока то, чего мы ожидаем, не вернёт что-то. А в это время мы можем сделать другие полезные дела. `Await` можно вызывать на объектах, которые несут название `awaitable`. Как правило, это либо корутина, либо объект, который реализует дандер-метод `__await__`, который должен вернуть итератор. Скорее всего, весь наш код будет крутиться вокруг первого кейса, за редкими исключениями, когда нам вдруг нужно написать что-то низкоуровневое.

Теперь код стал выглядеть немного чище, результат тот же. И, что самое важное, внутри работает так же, как и на генераторах.

И чуть-чуть перепишем то, как мы эти корутины исполняем:

```
async def await_main():
    task1 = asyncio.create_task(coroutine_1())
    task2 = asyncio.create_task(coroutine_2())
```

```
await task1
await task2

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

В общем, всё то же самое, только заменяем `yield from` на `await`.

Итак, теперь мы понимаем, как написать асинхронный код на Python, а также как писать корутины, используя новый синтаксис. Знаем, что под маской модного `async/await`, скрываются старые добрые генераторы. В следующем уроке мы продолжим изучать возможности модуля `asyncio` на практике: напишем асинхронное приложение по скраппингу картинок из интернета. А пока, как всегда, немного практики.