Event Loop

В этом уроке мы напишем свой Event Loop. Готовый пример можно посмотреть в *examples/coroutines.*

Генераторы

Event Loop в Python возможен благодаря *генераторам.* Давайте вспомним, что это такое.
Основной признак генератора в Python — инструкция *yield*. Ее наличие показывает, что функция является генератором.

Пример генератора:

```
def gen():
    print('генератор: я начал выполняться!')
    a = 1
    yield a
    a += 1
    yield a



# генератор сохраняет свое текущее состояние. То есть «замораживается» после очередного yield
g = gen()
# g.gi_frame.f_lasti - идентификатор последней выполненной инструкции,
# чтобы продолжить выполнение генератора со следующей инструкции при вызове
print(f'Последняя вызванная инструкция генератора: {g.gi_frame.f_lasti}')
# >>> -1 - генератор только создали, он еще не исполнялся

print('вызываем генератор первый раз')
print(f'a = {g.send(None)}')  # 1
# >>> генератор: я начал выполняться!
# >>> 1
print(f'Последняя вызванная инструкция генератора: {g.gi_frame.f_lasti}')
# >>> 6

print(f'a = {g.send(None)}')  # 2
# >>> 2
print(f'Последняя вызванная инструкция генератора: {g.gi_frame.f_lasti}')
# >>> 20

print(g.send(None))
# >>> raise StopIteration
```

(можно найти в *examples/generators/generator.py*)

Разберем, что происходит в примере выше:

1. Объявляем генератор — он делает два раза делает yield переменной *a.*
2. Создаем генератор — *заметьте, генератор еще не начинает свое выполнение.* Надпись: 'генератор: я начал выполняться!' на этом шаге не появится.
3. С помощью атрибута генератора *gi_frame.f_lasti* выведем адрес последней инструкции, которая была в нем выполнена. *Выведется -1*, то есть генератор еще не начал свое выполнение.
4. Выполним метод g.send(None), тем самым начав его выполнение. Нам выведется число *1.* То есть генератор дошел до первого yield и остановился, вернув значение *a.*
5. Выведем адрес последней выполненной инструкции — выведется число *6.* Т.е. при следующем вызове генератор продолжит свое выполнение именно с инструкции под номером *6,* а не с самого начала.
6. Выполним метод g.send(None), продолжив выполнение генератора. Нам выведется число *2.* Генератор дошел до второго yield, при этом он все еще помнит, что *a* изначально была равна *1.* То есть генератор хранит все данные о своем окружении даже после yield, хранит свой стек выполнения.
7. Вызовем метод g.send(None) еще раз, и произойдет исключение StopIteration. Генератору больше нечего выполнять, он завершился и теперь при каждом новом вызове будет кидать это исключение.

Теперь разберем, как использовать генераторы для написания асинхронного кода.

Future

Это сущность, которая является обещанием чего-то в будущем. Например, мы хотим подключиться к сокету: создадим Future сейчас и отдадим ее тоже сейчас. А значение в нее положим когда-то потом — когда подключение выполнится.

Суть Future в том, что мы разделяем факт *происхождения события* и *момент использования этого факта*.

Пример использования Future:

```
import selectors
import socket

from examples.coroutines.future import Future



class Fetcher:
    def __init__(self, selector: selectors.DefaultSelector):
        self._selector = selector

    # создаем неблокирующий сокет и вызываем подключение к нему
    def _connect(self, host: str, port: int):
        sock = socket.socket()
        sock.setblocking(False)

        try:
            sock.connect((host, port))
        except BlockingIOError:
            pass
        # инстанцируем Future
        f = Future()
        # callback-функция, которая будет вызвана после установки соединения
        def connected():
            print('connected')
            # удаляем "подписку" на события в этом сокете
            self._selector.unregister(sock)
            f.set_result(None)

        # регистрируем событие EVENT_WRITE на нашем сокете
        self._selector.register(sock, selectors.EVENT_WRITE, connected)
        # делаем yield нашей корутины
        yield f

        # возвращаем уже подключенный сокет
        return sock
```

(*полный пример можно посмотреть тут: examples/coroutines/fetcher.py*)

У нас есть функция Fetcher._connect, *которая является генератором* и пытается установить соединение с хостом. Мы создадим объект f = Future() и скажем, что в нем появится значение в тот момент, когда мы осуществим успешное подключение.

Зарегистрируем на селекторе наш файловый дескриптор, передадим туда on_connected и в этом on_connected установим результат. Например, f.set_result(None). Этот момент будет означать, что Future завершилась с каким-то результатом, в данном случае — None.

Давайте посмотрим, как устроена эта Future.

Пример Future:

```
class Future:
    def __init__(self, name: str):
        self.result = None  # результат выполнения Future
        self._callbacks = []  # добавленные к Future колбэки

    def add_done_callback(self, fn):  # метод добавления нового колбэка
        self._callbacks.append(fn)

    def set_result(self, result):  # метод установки результата выполнения Future
        self.result = result
        for fn in self._callbacks:  # вызов всех колбэков, зарегистрированных в этой Future
            fn(self)
```

(*можно посмотреть в examples/coroutines/future.py*)

В момент вызова self.set_result мы записываем результат нашей Future и вызываем все callback-функции, которые были зарегистрированы на эту Future. Т.е. мы можем при создании Future сказать: когда завершишься — т.е. когда появится какой-то результат — выполни такие-то действия.

Эти действия можно добавлять с помощью *add_done_callback*. Мы регистрируем какой-то callback на Future, почти так же, как регистрировали callback на сокет. И в тот момент, когда Future завершится, мы просто последовательно вызовем эти callback-функции.

В _connect мы еще не вызывали никаких add_done_callback, просто создали Future и сделали ее yield.

Если мы сейчас начнем просто дергать .send() у генератора fetcher(), то в первый раз нам вернется Future, а во второй раз вернется сокет, который может быть еще не готов к записи. Нам нужно, чтобы второй раз .send() дергался при наступлении события EVENT_WRITE или в callback-функции connected, которая как раз и будет вызвана при наступлении EVENT_WRITE.

Но для такой логики нам не хватает некоторой управляющей конструкции. Кто-то должен следить, выполнилась ли Future и продолжать выполнение генератора _connect.

Task

Пример Task:

```
from examples.coroutines.future import Future



class Task:
    def __init__(self, coroutine):
        self.coroutine = coroutine  # генератор, который возвращает Future
        f = Future()
        f.set_result(None)  # никакие callback не выполняются, так как мы только создали Future
        self.step(f)

    def step(self, future: Future):
        try:
            # вызываем генератор и отправляем в него результат текущей Future
            next_future = self.coroutine.send(future.result)  # нам возвращается новая Future
        except StopIteration:
            # если дошли до конца генератора, то заканчиваем выполнения
            return

            # просим новую Future вызывать этот метод (step) еще раз, после своего выполнения
        next_future.add_done_callback(self.step)
```

(можно посмотреть в examples/coroutines/task.py)

Сущность Task — задача, которую нам необходимо выполнить.

Фактически это и есть корутина, ведь корутина — это функция, которая чем-то управляется. Нам нужно двигать исполнение генератора до yield и после yield, т.е. кто-то должен итерировать по этой функции, чтобы она завершилась. И задача Task как раз в этом: завершить функцию-генератор, например, _connect().

Как использовать Task, Future и fetcher

Примерно так:

```
Task(Fetcher._connect('xkcd.com', 80))
```

Разберем, что происходит в строчке выше:

1. Мы создали генератор *Fetcher._connect()* — но он не начал выполняться, ведь это генератор
2. Мы создали экземпляр Task и передали в его __init__ наш генератор
3. Task создал начальную Future, которая сразу выполнилась. Это нужно, чтобы соответствовать интерфейсу метода step
4. Task вызвал метод step с выполненной Future
5. Метод step вызвал send у нашего генератора, *т.е. продвинул его выполнения до следующего yield.* Это очень важный момент: сейчас генератор Fetcher._connect() начал неблокирующее соединение сокета и зарегистрировал callback на событие открытия сокета на запись. А еще генератор вернул нам новую Future
6. Новая Future, возвращенная генератором, присвоилась в next_future
7. Мы повесили callback на эту новую Future, то есть попросили ее: «Когда ты выполнишься, вызови функцию step() и передай в нее себя»
8. Через некоторое время у нас установилось TCP-соединение. Предположим, что кто-то (Event Loop) ждал этого события и вызвал нашу callback-функцию connected()
9. Функция connected() отписала selector от подписки на события для этого сокета
10. Функция connected() вызвала f.set_result() — сделала Future завершенной
11. При вызове f.set_result() Future установила self._is_done = True и начала вызывать зарегистрованные callback-функции
12. **Еще один очень важный момент:** Future выполнилась и вызвала функцию step, как мы ей наказали в шаге 7
13. Все начало выполняться с пункта 5. Но теперь .send() уже не вернет очередную Future, а кинет ошибку StopIteration — генератор Fetcher._connect() закончил свое выполнение
14. Ура! Мы вышли из Task, и он выполнен до конца, причем асинхронно — мы не ждали в цикле и не блокировались, пока соединение будет установлено

Как выглядит Event Loop, который "продвигает" выполнение корутин мы уже видели в предыдущих уроках, приведем здесь код для наглядности:

```
import selectors

selector = selectors.DefaultSelector()

def loop():
    while True:
        events = selector.select()
        for event_key, event_mask in events:
            callback = event_key.data
            callback()



Task(Fetcher(selector)._connect('xkcd.com', 80))
loop()
```

Заметьте, что теперь в Fetcher при инстанцировании передается selector - это сделано для того, чтобы Fetcher в _connect мог зарегистрировать сокет на ожидания события EVENT_WRITE.

Теперь вам надо запустить код, поставить много точек останова и написать много print'ов, чтобы понять, как это все работает. Пример здесь: *examples/coroutines.* Перечитайте эту карточку еще и пройдите руками и глазами эти 14 этапов выше.

Мы все это делали, чтобы получить код, который выполняется асинхронно, но не превращается в CallbackHell.

Теперь мы можем выполнить какие-то действия после того, как действительно произошел on_connected. Не делать их в def on_connected, в каком-то смещенном контексте, который непонятно где находится в данный момент, а продолжить исполнение, выглядящее синхронно. Мы что-то сделали, сделали _connect, зарегистрировали, сделали yield.

Так мы создали ожидание какой-то Future, ожидание события в будущем. Оно произошло, и мы продолжили исполнение.

Мы отвязались от on_connected и всего, что происходит там. Просто установили результат, чтобы потом продолжить исполнение.

Вспомним: когда у нас не было self.sock.setblocking(False), у нас был просто connect, и мы печатали connected.

По факту все так и осталось: у нас есть connect, и мы печатаем connected, в одной процедуре. Мы не опустились внутрь какого-то callback, не сменили контекст исполнения, мы все так же находимся в той же самой функции, и там же мы выполнили print('connected'). Просто благодаря использованию такого асинхронного подхода у нас появляется возможность дождаться, пока эта Future не завершится. Мы дождались этого и продолжили исполнение.

Так устроен весь асинхронный мир: что-то происходит там, что-то тут, и все как-то между собой взаимосвязано. Все, что мы пытаемся сделать, — систематизировать этот процесс, чтобы он шел в линейном порядке. Благодаря этому механизму мы локализовали в одном месте эти разные непонятные события, которые происходят в разные моменты времени.

У нас есть функция on_connected в class Fetcher, которая выполнилась, когда ОС что-то сообщила. Дальше у нас просто синхронно выглядящий код. Мы последовательно выполняем операции и не задумываемся о том, как сложно это все устроено.

Теперь давайте систематизируем это все в одном месте.