Пишем read, send и read_all

Методы чтения из сокета чанка (read), записи в сокет (send) и чтения всех данных из сокета (read_all) очень похожи на метод _connect, который мы рассматривали ранее.

Пример:

```
def send(self, sock, data: bytes):
    f = Future()

    def sent():
        self._selector.unregister(sock)
        sock.send(data)
        f.set_result(None)

    self._selector.register(sock, selectors.EVENT_WRITE, sent)
    yield f
    return None

def read(self, sock):
    f = Future()

    def _read():
        self._selector.unregister(sock)
        data = sock.recv(4096)
        f.set_result(data)

    self._selector.register(sock, selectors.EVENT_READ, _read)
    chunk = yield f
    return chunk

def read_all(self, sock) -> str:
    response = bytearray()
    chunk = yield from self._read(sock)
    while chunk:
        response += chunk
        chunk = yield from self._read(sock)
    return response.decode('ascii')
```

(пример можно найти в examples/coroutines/fetcher.py)

read

Посмотрим на функцию read() более внимательно — в последних двух строках происходит небольшая магия.

Нам необходимо убедиться в том, что сокет доступен на чтение (EVENT_READ), и только в этот момент мы пытаемся вычитать что-то из сокета. Мы делаем sock.recv(4096), т.е 4096 байт. Не факт, что мы столько прочитаем — главное, что прочитаем хотя бы часть.

В read() все то же самое, как в connect из предыдущих примеров. Мы объявили какой-то callback, создали Future, в которую засетим результат, а результатом у нас будут прочитанные байты.

Зарегистрировали callback на on_readable и передали событие EVENT_READ: попросили ОС вызвать нам on_readable, когда сокет станет доступен на чтение. Сделали yield для этой Future и сказали, что получим из нее чанк.

В Task в методе send() передается future_result — результат предыдущей выполненной Future.

Когда Future завершится, продолжится исполнение с последнего yield, и все, что передается в send, передается в качестве возвращаемого значения из yield. Так устроены генераторы в Python. Т.е. мы получили из этой Future результат и послали его в тот же генератор. Генератор поставил результат на место последнего yield, то есть сюда: *chunk = yield f*.

Соответственно, при следующем движении генератора yield f заменится на некоторый набор байт, который присвоится к chunk. А затем мы просто сделаем *return chunk*.

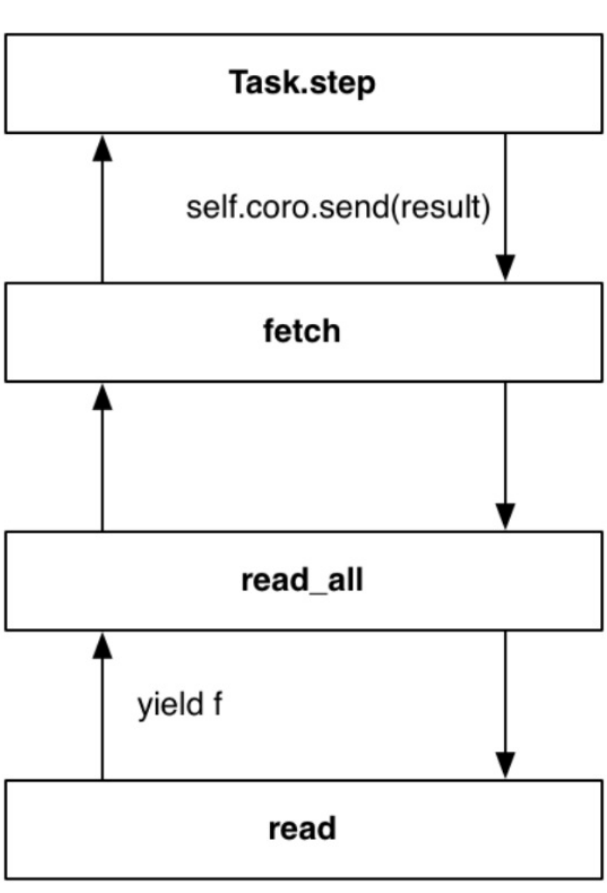
Мы могли бы просто вернуть в конце return f.result, но это не очень читаемо, правда?

read_all

Код получается более привычный: просто дописан синтаксический сахар yield from.

Код очень простой и похож на линейный: мы делаем read и получаес chunk: *chunk = yield from read(sock)*. До тех пор, пока есть чанки, мы все это суммируем и делаем read еще раз. Обычный бесконечный цикл до тех пор, пока мы не вычитаем все содержимое из сокета и не вернем результат.

На диаграмме указана последовательность выполнения действий:



У нас есть Task.step, который делает self.coro.send(result), в котором записан None. Когда дергается fetch, все это идет по цепочке, опускаясь до первого yiled, который произойдет в функции read. Далее все это дело поднимается обратно до Task.step, который засетит callback: «Когда Future завершится, вызови меня (step) еще раз».

Далее все засыпает на select. *Это самый важный момент.* Наше приложение спит на select и ждет события на сокетах. Как только оно просыпается, вызывается какая-то внутренняя функция, в данном случае — callback-функция _read.

Future завершилась благодаря вызову set_result, вызывается callback, чтобы еще раз вызвать step. Снова делается send и продолжает исполнение по цепочке до ближайшего yield. Сделался yield, и она опять пошла наверх и ждет. Т.е. как только сделался yield, мы ждем: приложение условно приостанавливается.

На этом закончим с низкоуровневыми объяснениями.
Давайте разберем в следующем видеоуроке, как с этим работать.