

Asynchronous Python

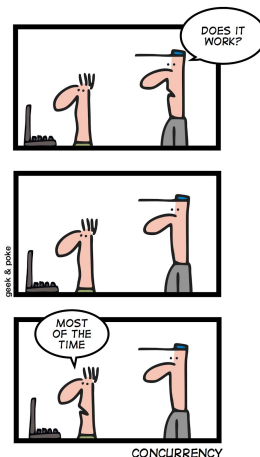
Асинхронне програмування - це потокова обробка програмного забезпечення / користувальницького простору, де програма, а не процесор керує потоками і перемиканням контексту.

Для чого потрібна асинхронність?

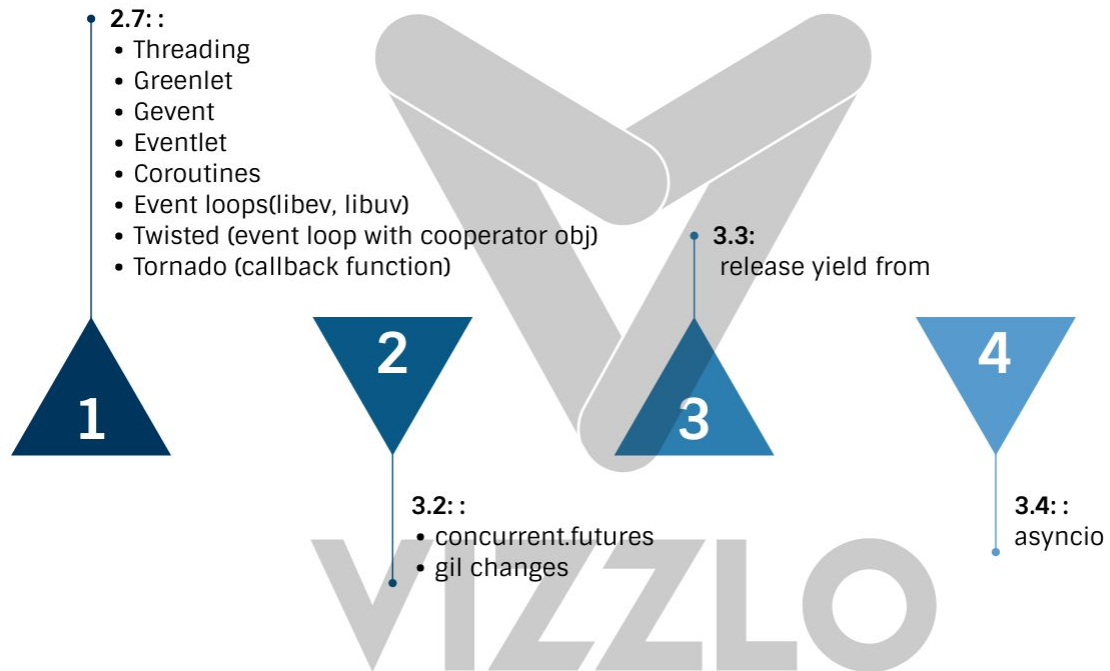
Кращий перформанс програми з ІО операціями

Контекст виконання переключається пайтон кодом а не ОС

SIMPLY EXPLAINED



Istoricheskaja stranichka



Основні Проблеми Конкаренсі

Race condition

(невизначеність паралелізму)

“плаваюча” помилка, яка виникає в випадкові моменти часу і може не виникнути при спробі її локалізувати.

Помилка проектування багатопоточної системи або програми,

при якій робота залежить від того в якому порядку виконуються частини коду.



Deadlock

Ні один тред не може продовжити виконання роботи, всі в режимі очікування.

Процесорне переключення контексту

Вичерпання ресурсу

Основні концепції в concurrency в Python

threads

coroutines

event loop

futures

callback functions

Корутіни

```
def grep(pattern):  
    while True:  
        line = yield  
        if pattern in line:  
            print(line)
```

```
g = grep("python")
```

```
next(g)
```

```
g.send('pty')
```

```
# >>>None
```

```
# A few moments later
```

```
g.send('python is')
```

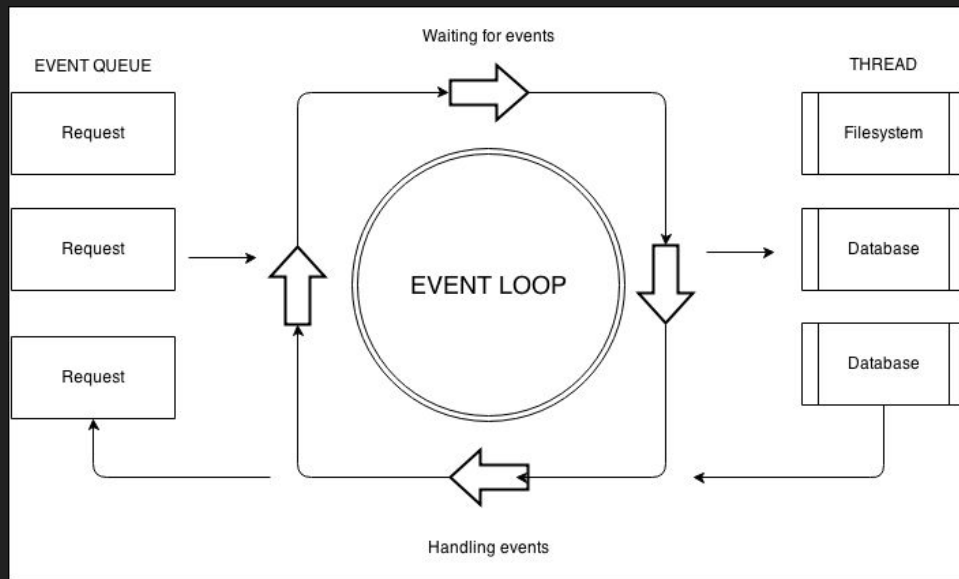
```
# >>>python is
```

- Генератор який приймає аргументи
- Зберігає свій стан між викликами
- Виконання може прерватись і продовжитись до виконання певних умов
- Поки виконююча корутіна не прервалась на очікування події, всі інші корутіни очікують навідь якщо подія яку вони чекають вже відбулась

Event loop

Event loop — черга подій/завдань і цикл, який витягує завдання з черги і запускає їх. Ці завдання називаються корутінами.

Все що робить івент луп це дістає таск з корутіни, або продовжує вионання таски після того як вона була в очікуванні. В деяких випадках розкидує таски по тредам) далі буде



Callback function - функція яка виконається одразу по закінченню іншого коду.

```
import tornado.ioloop
from tornado.httpclient import AsyncHTTPClient

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org']

def handle_response(response):
    if response.error:
        print("Error:", response.error)
    else:
        url = response.request.url
        data = response.body
        print('{}: {} bytes: {}'.format(url, len(data), data))

http_client = AsyncHTTPClient()
for url in urls:
    http_client.fetch(url, handle_response)

tornado.ioloop.IOLoop.instance().start()
```

Метод `fetch` замість повернення об'єкта викликає функцію з результатом (коллбек). В цьому випадку `handle_response`. При виконанні `fetch` запускає HTTP запит а потім обробляє відповідь в Event Loop.

Callback http

```
dtiur : htop — Konsole <2>
File Edit View Bookmarks Settings Help

 1  [||||]          5.4%]   Tasks: 90, 172 thr; 3 running
 2  [||||]          4.8%]   Load average: 1.03 0.66 0.64
 3  [|||||]         12.7%]   Uptime: 1 day, 20:13:19
Mem[|||||||]       1.15G/7.46G]
Swp[|]             0K/8.80G]

 PID USER      PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
17545 dtiur       20   0  407M 30412 9888  S   0.0  0.4   0:00.22 python callback_tornado_exmpl.py
17587 dtiur       20   0  407M 30412 9888  S   0.0  0.4   0:00.00 python callback_tornado_exmpl.py
17588 dtiur       20   0  407M 30412 9888  S   0.0  0.4   0:00.00 python callback_tornado_exmpl.py
17589 dtiur       20   0  407M 30412 9888  S   0.0  0.4   0:00.00 python callback_tornado_exmpl.py
17590 dtiur       20   0  407M 30412 9888  S   0.0  0.4   0:00.00 python callback_tornado_exmpl.py

F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice -F8Nice +F9Kill F10Quit
```

Якщо вбити тред, всі треди помруть.

Якщо в коді не вбити івент луп він буде жити =)

Реалізація в різних лібах

concurrent.futures (concurrent_futures.py)

Модуль concurrent.futures дозволяє писати асинхронний код дуже легко. ThreadPoolExecutor і ProcessPoolExecutor підтримують пул потоків або процесів. Ми відправляємо наші завдання в пул, і він запускає завдання в доступному потоці / процесі. Повертається об'єкт Future, який можна використовувати для запиту і отримання результату по завершенні завдання.

Приклад ThreadPoolExecutor:

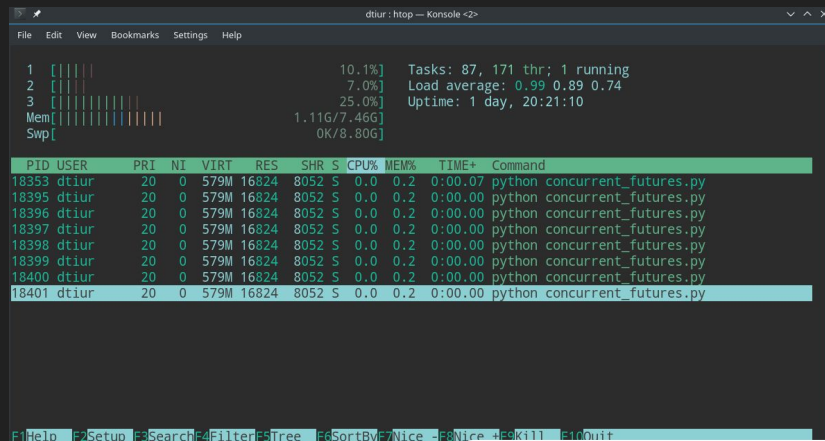
```
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def return_after_5_secs(message):
    sleep(5)
    return message

pool = ThreadPoolExecutor(3)

future = pool.submit(return_after_5_secs, ("hello"))
print(future.done())
print(future.done())
print(future.result())
```

output:
False
False
hello



Greenlets (зелені потоки)

Реалізація потоків на програмному рівні. ОС не знає про потоки :D

Green threads емулюють багатопоточне середовище, не покладаючись на можливості ОС по реалізації потоків. Управління ними відбувається в просторі користувача, а не просторі ядра, що дозволяє їм працювати в умовах відсутності підтримки вбудованих потоків.

По замовчуванню грінлети не мають івент лупа або екзекутора

Грінлети це C extension для звичайного інтерпретатора CPython

(базово являється корутіною написаною як C екстеншин)

може бути швидше за asuncіо по замовчуванню тому ще не уникає race condition

Greenlets example (greenlet_expl.py)

```
from greenlet import greenlet
```

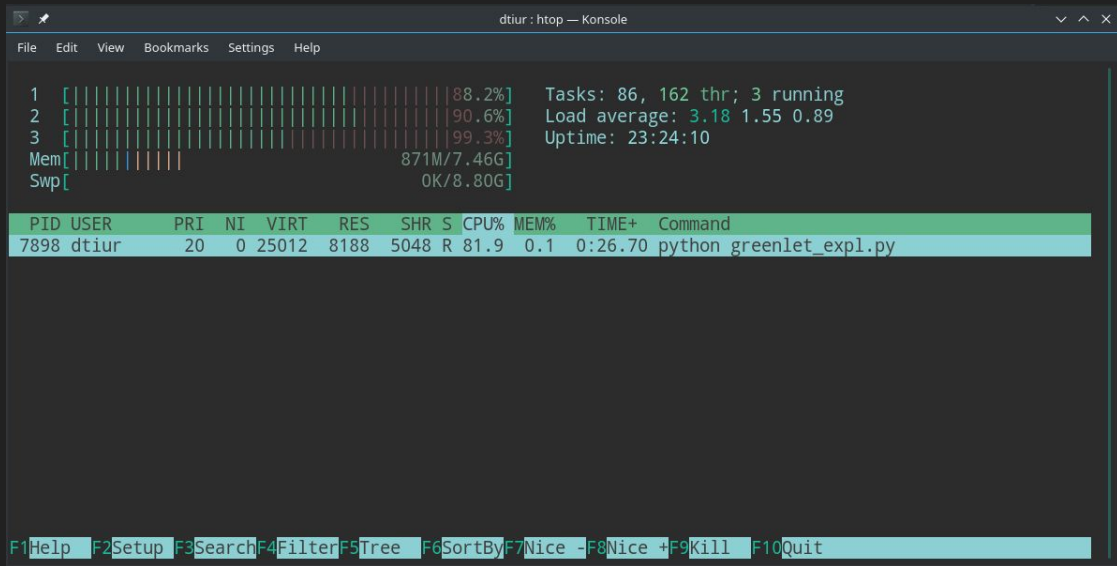
```
def test1():  
    while True:  
        print(1)  
        gr2.switch()  
        print(3)  
        gr2.switch()
```

```
def test2():  
    while True:  
        print(2)  
        gr1.switch()  
        print(4)  
        gr1.switch()
```

```
gr1 = greenlet(test1)  
gr2 = greenlet(test2)  
gr1.switch()
```

Output:

```
1  
2  
3  
4  
1  
2  
3  
4  
1  
2  
3  
...  
...
```



Gevents (gevent_expl.py)

Ліба побудована на Greenlets з використанням event loop (libev or libuv).

Екзекутор gevents сам переключає контекст виконання при першій же можливості (при роботі з мережею і ІО операціях).

В коді треба явно вбивати процес виконання, щоб він не став зомбі)

Monkeypatching

Можливість підміни методів і значень атрибутів класів програми під час виконання.

Gevent може пропатчити в стандартній лібі більшість блокуючих системних викликів, включно з socket, ssl, threading, select. Робить їх не блокуючими, а отже доступними для асинхронності.

Може привести до неочікуваної поведінки програми, або до покращення :D

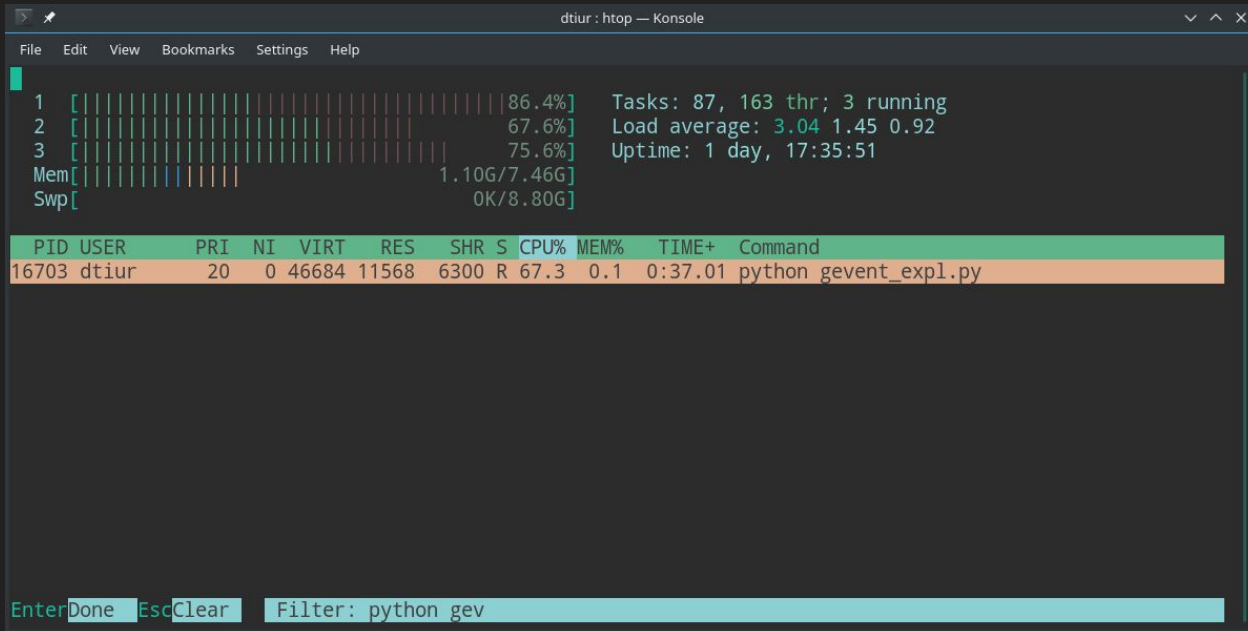
Gevents example

```
import gevent
```

```
def foo():  
    while True:  
        print('Running in foo')  
        gevent.sleep(0)  
        print('Explicit context switch to  
foo again')
```

```
def bar():  
    while True:  
        print('Explicit context to bar')  
        gevent.sleep(0)  
        print('Implicit context switch  
back to bar')
```

```
gevent.joinall([  
    gevent.spawn(foo),  
    gevent.spawn(bar),  
])
```



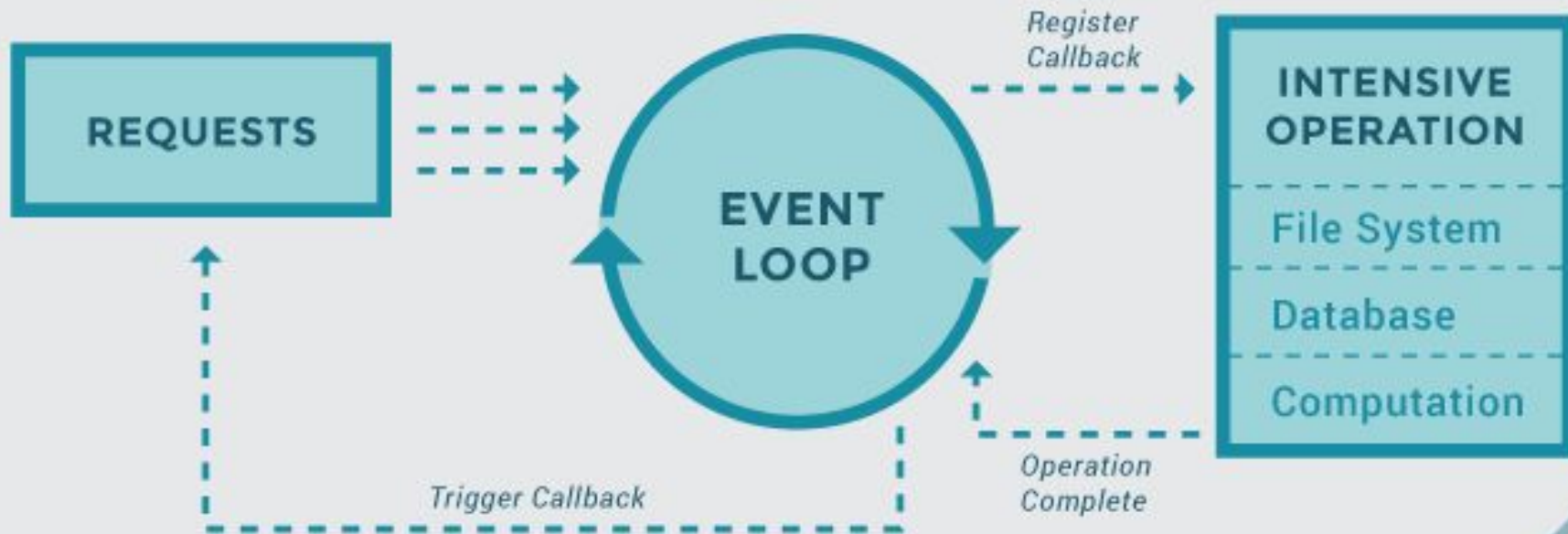
The screenshot shows a terminal window titled "dtiur : htop — Konsole". The top section displays system statistics: CPU usage at 86.4%, 67.6%, and 75.6% for three cores; memory usage at 1.10G/7.46G; and swap usage at 0K/8.80G. System tasks are 87, 163 threads, with 3 running. Load average is 3.04, 1.45, 0.92. Uptime is 1 day, 17:35:51. Below this is a table of running processes. The first row is highlighted in orange and shows a process with PID 16703, user dtiur, priority 20, 0 NI, 46684 VIRT, 11568 RES, 6300 SHR, S state, 67.3 CPU%, 0.1 MEM%, 0:37.01 TIME+, and command python gevent_expl.py. The bottom of the window shows a search bar with "Filter: python gev" and "Enter Done Esc Clear" buttons.

| PID | USER | PRI | NI | VIRT | RES | SHR | S | CPU% | MEM% | TIME+ | Command |
|-------|-------|-----|----|-------|-------|------|---|------|------|---------|-----------------------|
| 16703 | dtiur | 20 | 0 | 46684 | 11568 | 6300 | R | 67.3 | 0.1 | 0:37.01 | python gevent_expl.py |

Output:
Running in foo
Implicit context switch back to bar
Explicit context to bar
Explicit context switch to foo again
Running in foo
Implicit context switch back to bar

Asncio (example `asnc_exmpl.py`)

Юзає івент лупи, корутіни, `concurrency.future`



Some additional pages

Mutex

What is difference between Semaphore and Mutex

Mutex



Now person will give it's key to next person and next person will enter into toilet.



Mutex

A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section



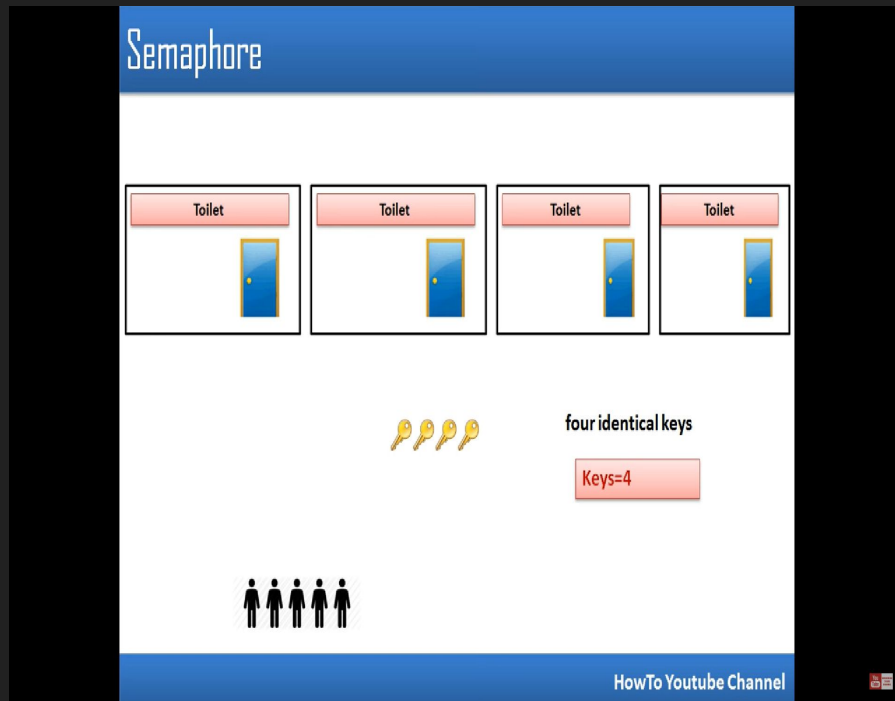
Thread-1

Thread-2

Thread-3

Lock(mutex)
Do work
Unlock(mutex)

Semaphore



```
semaphore = threading.BoundedSemaphore()
```

```
semaphore.acquire() # уменьшает счетчик доступа к ресурсу
```

```
semaphore.release() # увеличивает счетчик
```

```
max_connections = 10
```

```
semaphore = threading.BoundedSemaphore(max_connections)
```

```
#Если не передавать в BoundedSemaphore параметр,
```

```
#семафор будет инициализирован 1 (и таким образом станет
```

```
#обычной блокировкой)
```

Co-routine v/s Callback

```
import asyncio

@asyncio.coroutine
def just_print_messages():
    while True:
        print('Just print')
        yield from asyncio.sleep(1)

def main():
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(just_print_messages())
    finally:
        loop.close()

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

```
import asyncio

def just_print_messages(loop):
    print('Just print')
    loop.call_later(1, just_print_messages, loop)

def main():
    loop = asyncio.get_event_loop()
    try:
        loop.call_soon(just_print_messages, loop)
        loop.run_forever()
    finally:
        loop.close()

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