

# Асинхронность в Python

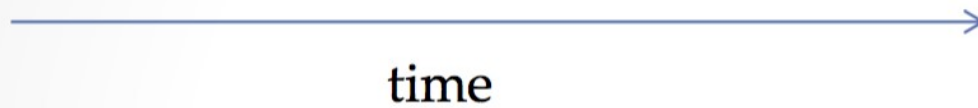
библиотека `asyncio`

`Async` & `Await`



**Асинхронное программирование** — это особенность современных языков программирования, которая позволяет выполнять операции, не дожидаясь их завершения.

# Programming Models



Task 1

Task 2

Task 3





# Понятия параллелизма, concurrency, поточности и асинхронности

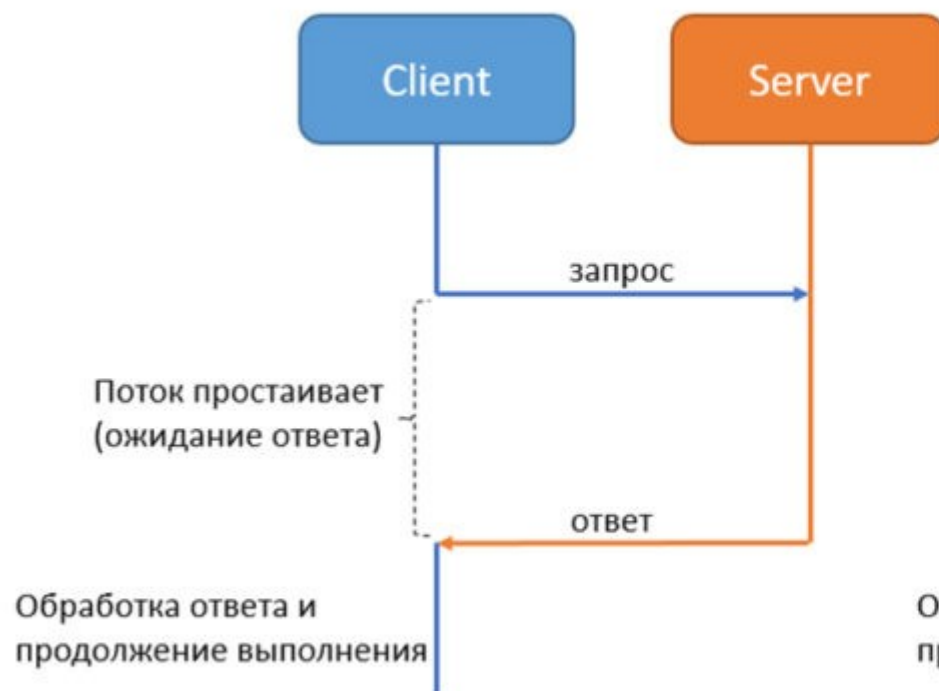
Параллелизм — это выполнение нескольких операций за раз. Многопроцессорность — один из примеров. Отлично подходит для задач, нагружающих CPU.

Concurrency — более широкое понятие, которое описывает несколько задач, выполняющихся с перекрытием друг друга.

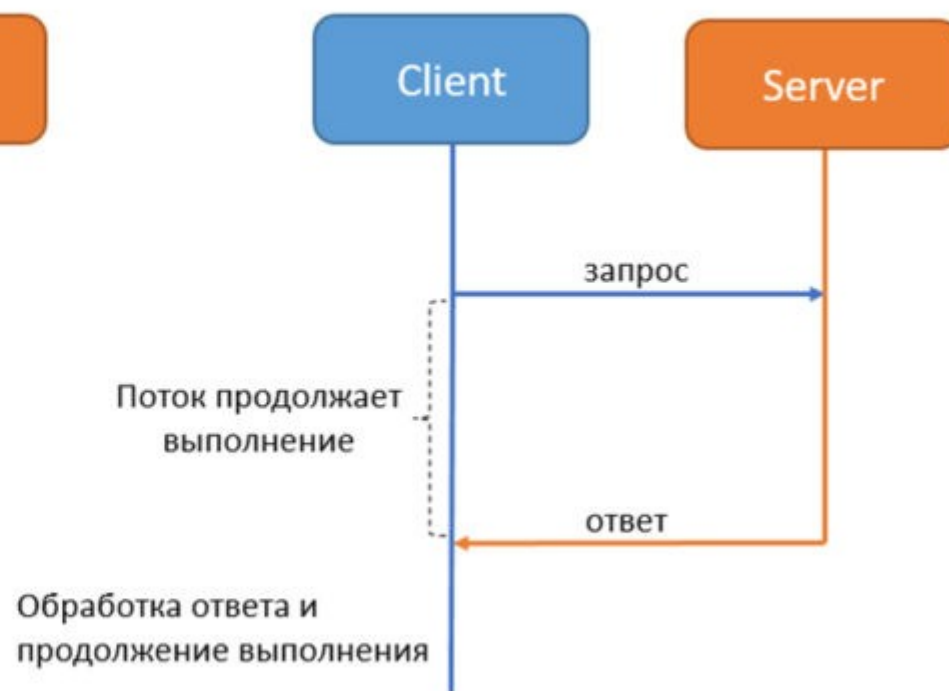
Поточность (поток) — это отдельный поток выполнения. Один процесс может содержать несколько потоков (threads), где каждый будет работать независимо. Отлично подходит для IO-операций.

**Асинхронность** — однопоточный, однопроцессорный дизайн, использующий многозадачность. Другими словами, асинхронность создает впечатление параллелизма, используя один поток в одном процессе.

### Синхронное выполнение



### Асинхронное выполнение





# Гипотетическая задача

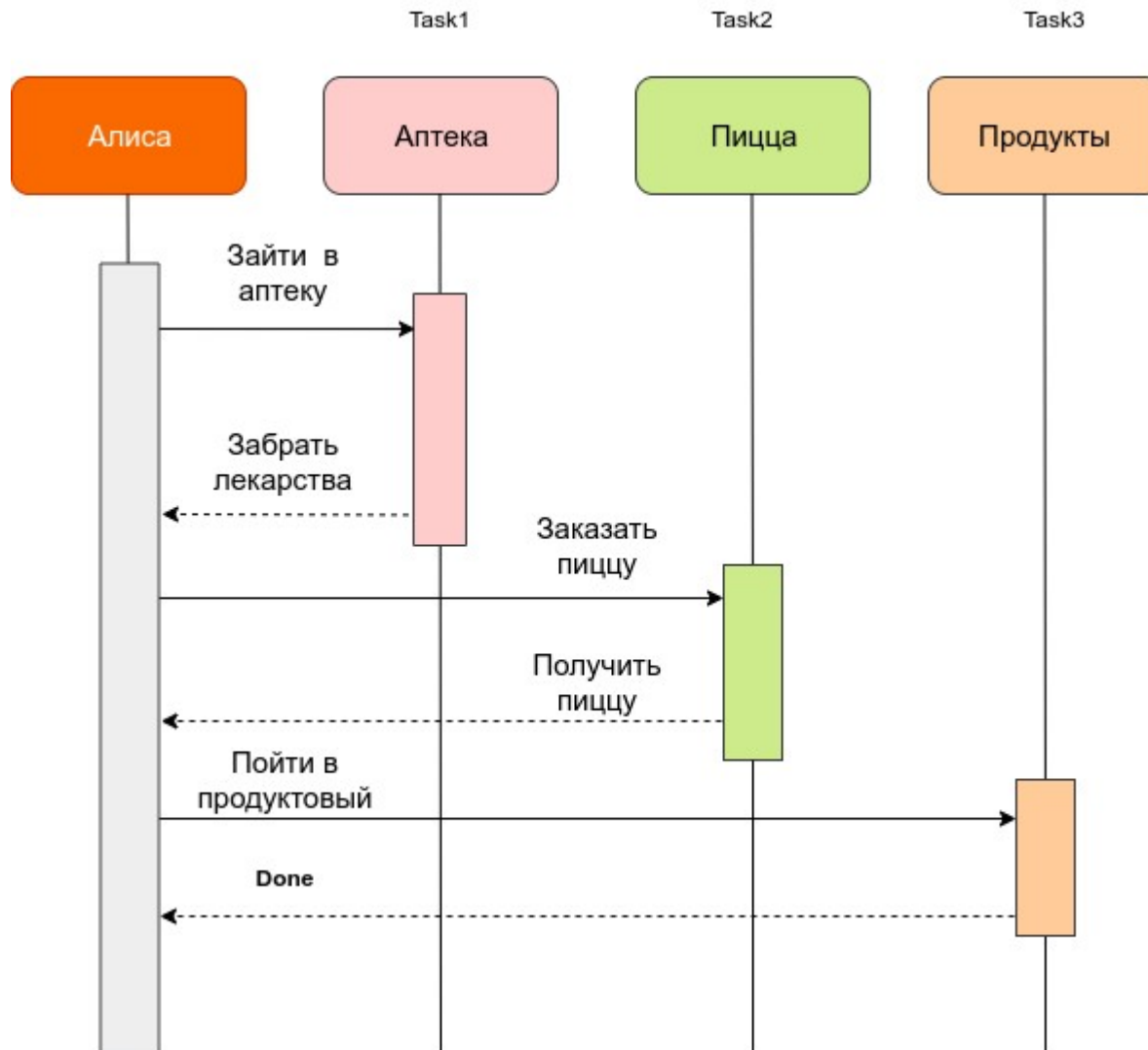
В нашем вымышленном примере Алиса должна выполнить несколько поручений

- пойти в аптеку, чтобы получить лекарство по рецепту
- заказать пиццу на ужин в Bob's Pizza
- купить кое-какие продукты в продуктовом магазине

Каждая из трех задач занимает некоторое время.

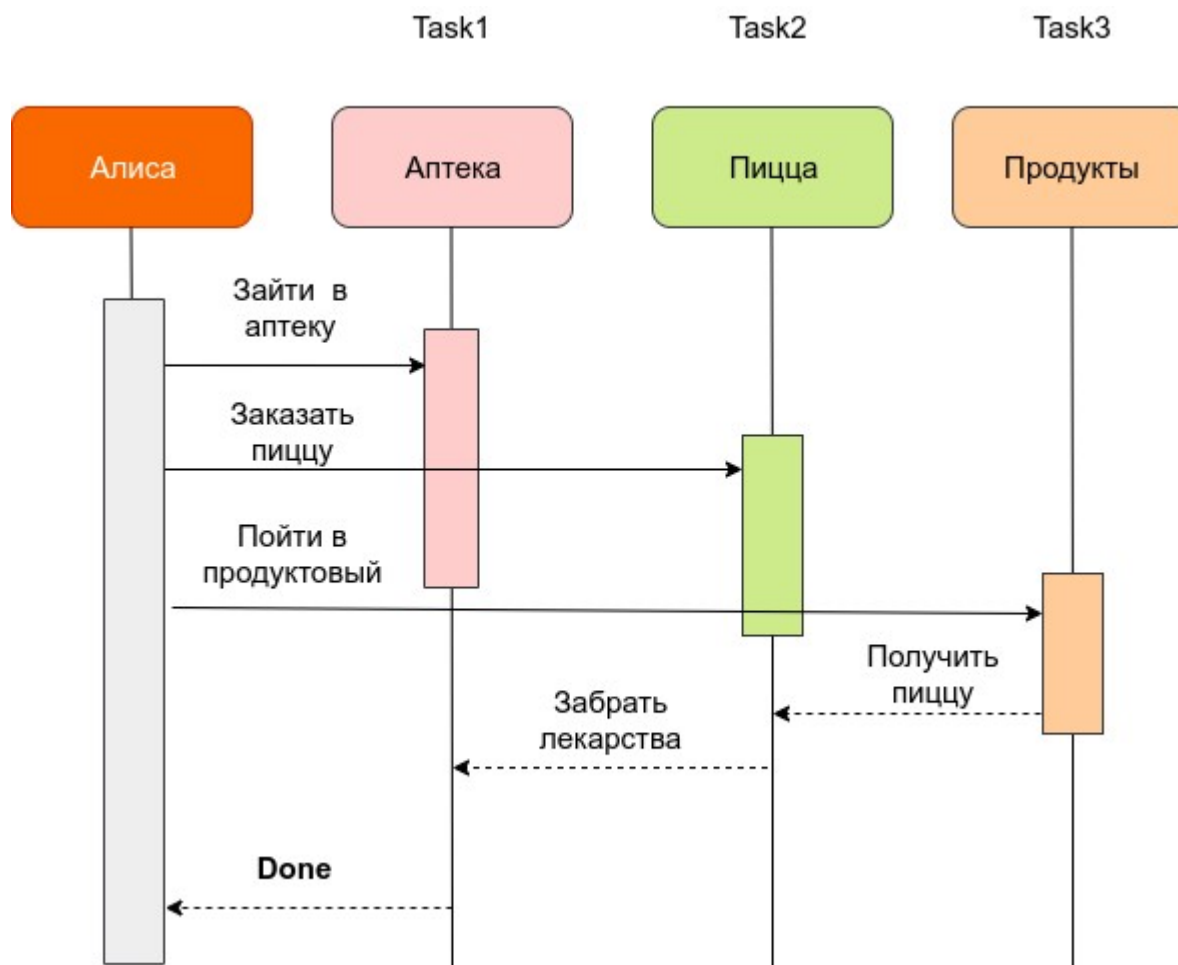
# Sequential

Последовательное выполнение задач Алисой



# Concurrency

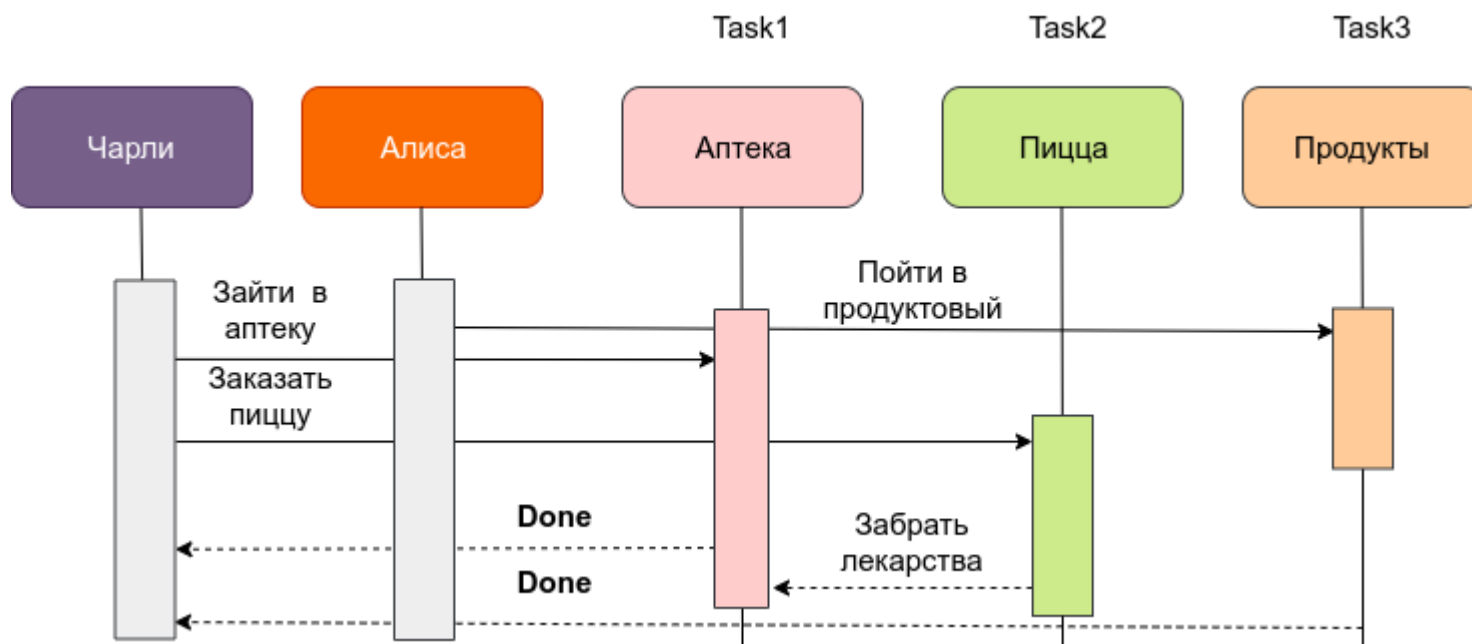
Более оптимальным подходом было бы пойти в аптеку и оставить рецепт, затем отправиться в пиццерию ,сделать заказ на вынос, затем пойти в продуктовый магазин, чтобы выбрать продукты. После того, как куплены продукты вернуться за пиццей и затем забрать лекарства по рецепту.





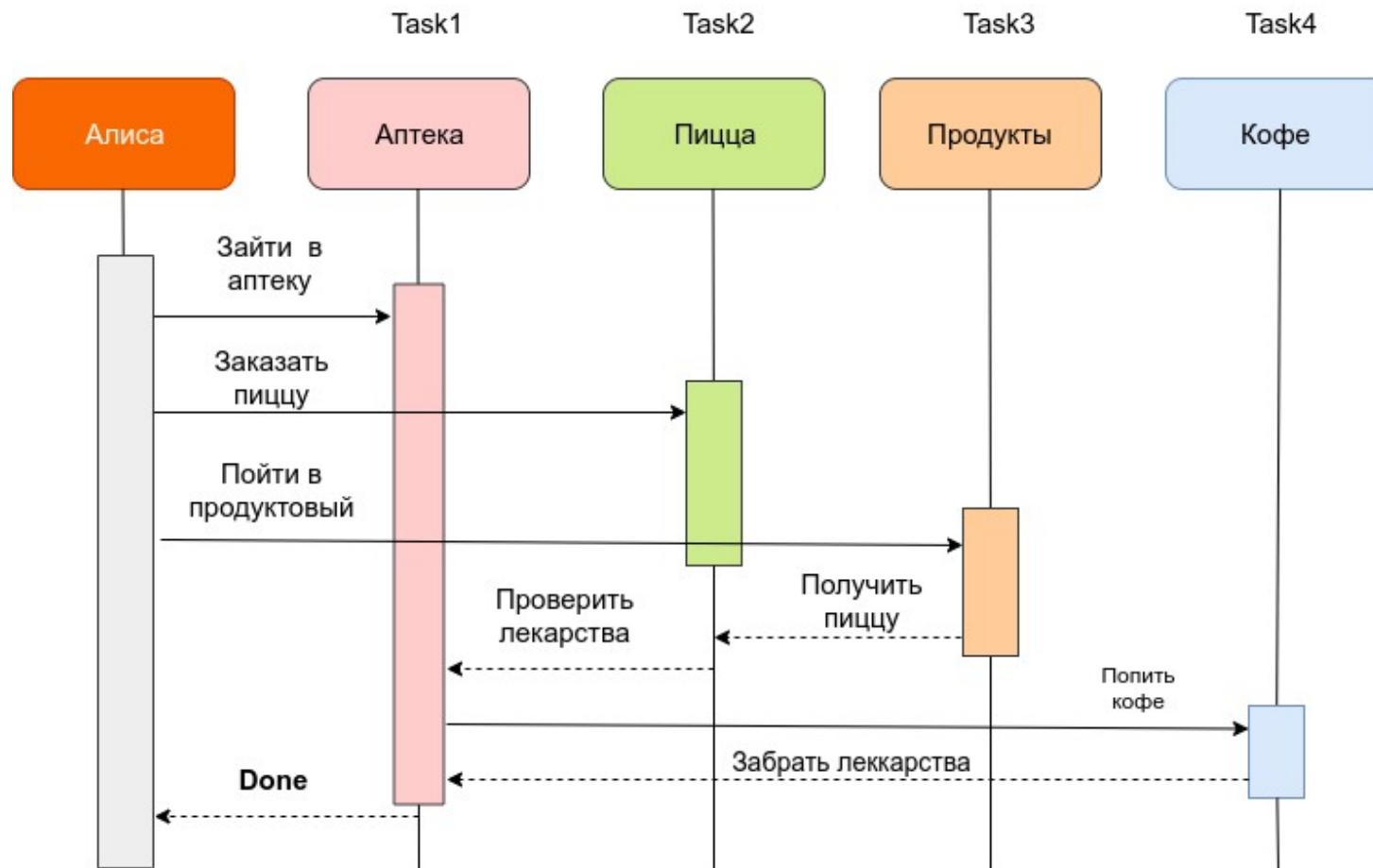
# Parallelism

Если Алиса получает помощь от Чарли, чтобы получить лекарства и пиццу, пока она покупает продукты, это и есть пример параллелизма.



# Asynchronous

Асинхронность — это более простая парадигма параллелизма, в которой используется один поток в одном процессе, а также совместная вытесняющая многозадачность, позволяющая различным задачам по очереди выполняться. Если задача блокируется, она уступает место другой готовой задаче для продвижения вперед.





# Что такое asyncio ?

```
import asyncio
```

asyncio – это библиотека Python, которая используется для запуска параллельного кода с использованием **async/await**. Это основа для асинхронной среды Python, которая предлагает библиотеки подключений, сетевые и веб-серверы, распределенные очереди задач базы данных, высокую производительность и т. д.



## Ключевые понятия

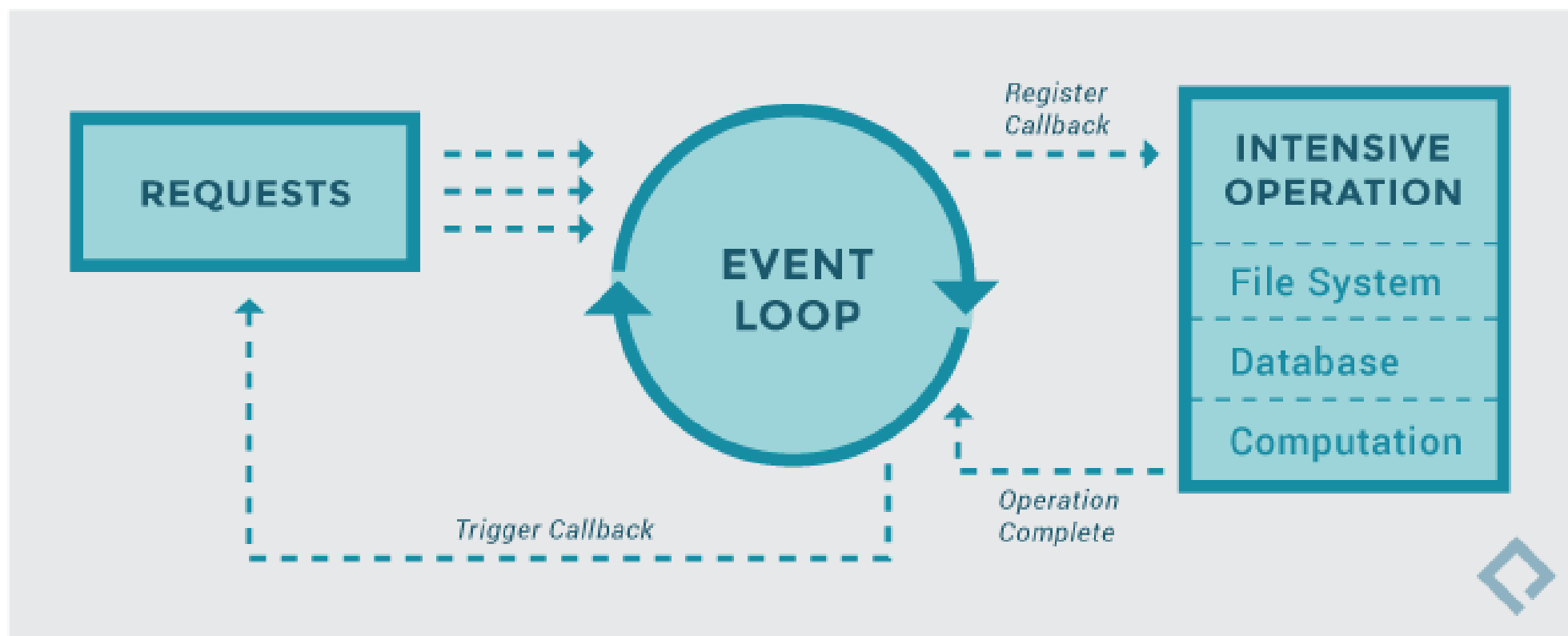
- цикл событий (**event loop**) по большей части всего лишь управляет выполнением различных задач: регистрирует поступление и запускает в подходящий момент
- корутины — специальные функции, похожие на генераторы python, от которых ожидают (await), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий
- футуры — объекты, в которых хранится текущий результат выполнения какой-либо задачи. Это может быть информация о том, что задача ещё не обработана или уже полученный результат; а может быть вообще исключение



## Что такое event loop ?

Event loop позволяет организовать логику "когда произошло А, сделай В". Проще говоря, event loop наблюдает за тем, не произошло ли "что-то", за что он отвечает, и если это "что-то" случилось, он вызывает код, который должен обработать это событие. Python включил event loop в стандартную библиотеку в виде `asyncio` начиная с версии Python 3.4.

# Цикл событий





## Сопрограммы (coroutine)

Сопрограммы (coroutine) в Python – это особый тип функций, которые сознательно передают контроль вызывающему, но не завершают свой контекст в процессе, а вместо этого поддерживают его в состоянии ожидания. Они извлекают выгоду из возможности хранить свои данные в течение всего срока службы и, в отличие от функций, могут иметь несколько точек входа для приостановки и возобновления выполнения..

Для определения сопрограммы асинхронная функция использует ключевое слово **async**. От которых ожидают (**await**), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий (также известный как **event loop**).

Для запуска сопрограммы нужно запланировать его в цикле событий. После этого такие сопрограммы оборачиваются в задачи (Tasks) как объекты Future.



# Как приступить к циклическому программированию на основе событий

Ключевое слово `async` идет до `def`, чтобы показать, что метод является асинхронным. Ключевое слово `await` показывает, что вы ожидаете завершения сопрограммы.





# Запуск асинхронных функций

```
import asyncio
```

```
async def main():
```

```
    print('hello')
```


```
    await asyncio.sleep(1)
```

```
    print('world')
```

```
asyncio.run(main())
```

```
main() → <coroutine object main at 0x1053bb7c8>
```

`asyncio` предоставляет функцию **run()** для выполнения асинхронной функции и всех других сопрограмм, вызываемых оттуда, например `sleep()` в функции `main()`.



# Рассмотрим особенности работы с `asyncio`

- Запуск цикла событий `event loop`
- Вызов функций `async/await`
- Создание задач для запуска в `event loop`
- Ожидание выполнения нескольких задач
- Закрытие цикла после выполнения конкурентных задач



## Что под капотом ?

```
import time
async def main():
    print(f"{time.ctime()} Hello!")
    await asyncio.sleep(1.0)
    print(f"{time.ctime()} world!")

loop = asyncio.get_event_loop()
task = loop.create_task(main())
loop.run_until_complete(task)

pending = asyncio.all_tasks(loop=loop)
for task in pending:
    task.cancel()

loop.close()
```

# Что под капотом ?

```
import time
async def main():
    print(f"{time.ctime()} Hello!")
    await asyncio.sleep(1.0)
    print(f"{time.ctime()} world!")
```

#Перед тем как запустить задачу нужно получить even\_loop

```
loop = asyncio.get_event_loop()
```

# Запланировать выполнение задачи

```
task = loop.create_task(main())
```

# Работать пока не завершиться задача

```
loop.run_until_complete(task)
```

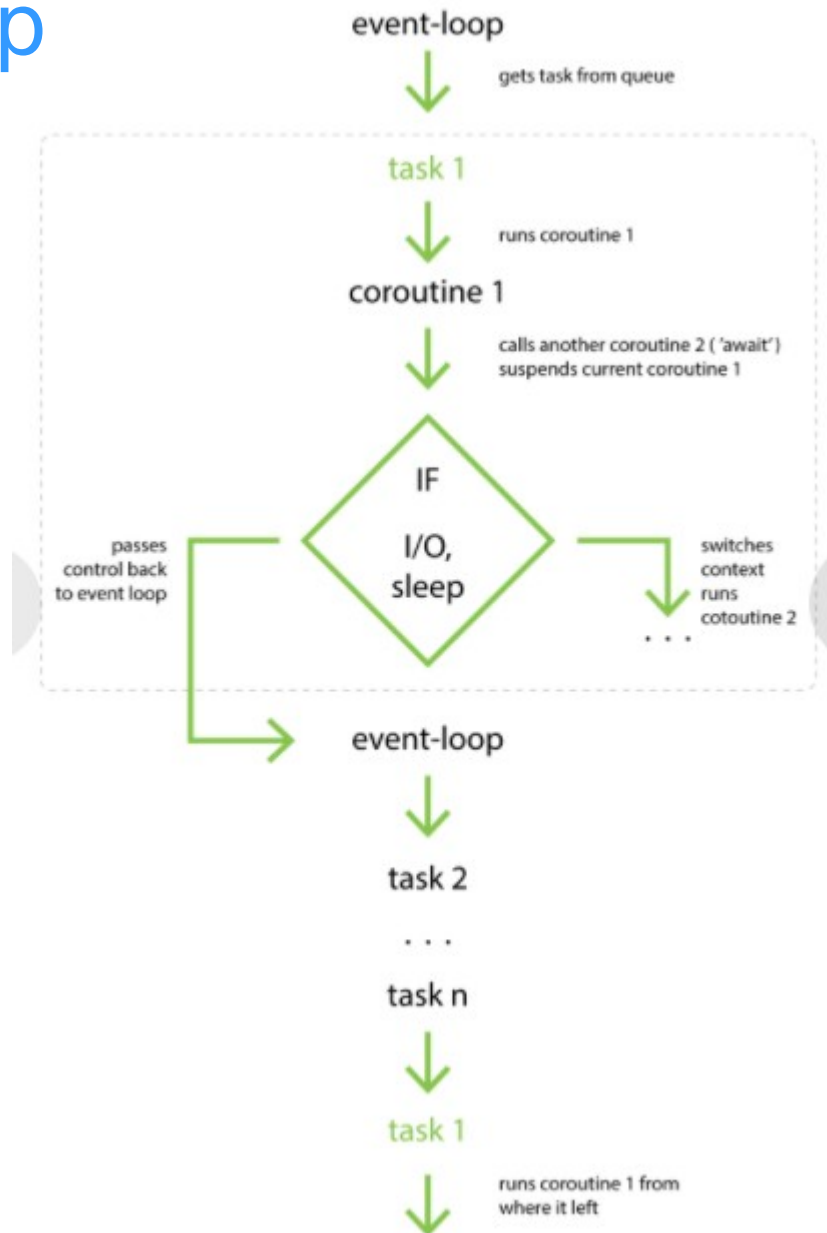
# Получение всех незавершенных задач

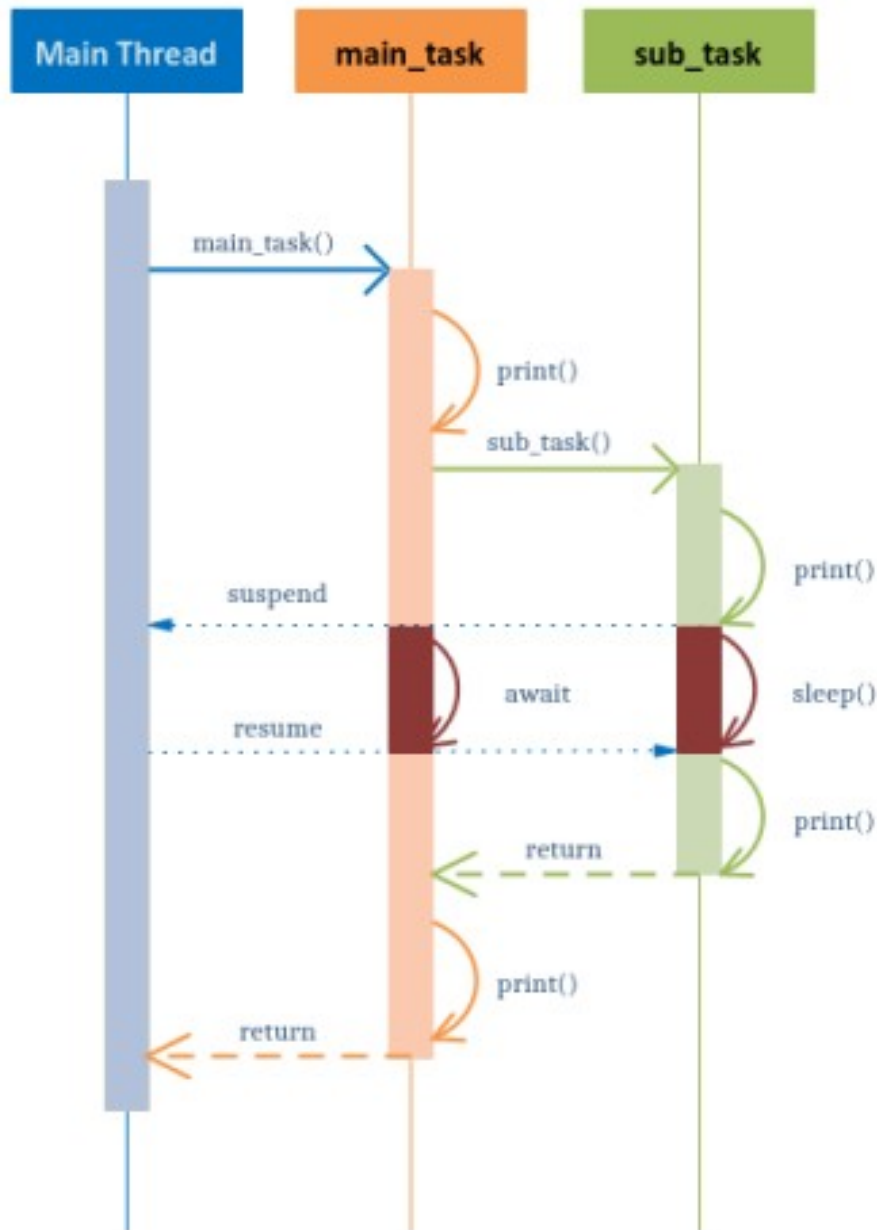
```
pending = asyncio.all_tasks(loop=loop)
```

```
for task in pending: # итерируем задачи и явно их завершаем
    task.cancel()
```

```
loop.close() # завершение цикла события
```

# Event-loop





```
async def sub_task():
    print(f'{time.ctime()} sub_task A')
    await asyncio.sleep(0.5)
    print(f'{time.ctime()} sub_task B')
```

```
async def main_task():
    print(f'{time.ctime()} main_task C')
    await sub_task()
    print(f'{time.ctime()} main_task D')
```

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main_task())
    loop.close()
```

```
Fri Jun 10 16:55:04 2022 main_task C
Fri Jun 10 16:55:04 2022 sub_task A
Fri Jun 10 16:55:04 2022 sub_task B
Fri Jun 10 16:55:04 2022 main_task D
```



# Асинхронные библиотеки async HTTP

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

async def call_url(url):
    print('Starting {}'.format(url))
    response = await aiohttp.get(url)
    data = await response.text()
    print('{}: {} bytes: {}'.format(url, len(data),
data))
    return data

futures = [call_url(url) for url in urls]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(futures))
```



## Заключение

- процессорное переключение контекста: Asyncio является асинхронным и использует цикл событий. Он позволяет переключать контекст программно;
- состояние гонки: поскольку Asyncio запускает только одну сопрограмму и переключается только в точках, которые вы определяете, ваш код не подвержен проблеме гонки потоков;
- взаимная/активная блокировка: поскольку теперь нет гонки потоков, то не нужно беспокоиться о блокировках. Хотя взаимная блокировка все еще может возникнуть в ситуации, когда две сопрограммы вызывают друг друга, это настолько маловероятно, что вам придется постараться, чтобы такое случилось;
- исчерпание ресурсов: поскольку сопрограммы запускаются в одном потоке и не требуют дополнительной памяти, становится намного сложнее исчерпать ресурсы.





# Документация

- <https://docs.python.org/3/library/asyncio-task.html>
- [https://docs.aiohttp.org/en/stable/web\\_quickstart.html](https://docs.aiohttp.org/en/stable/web_quickstart.html)