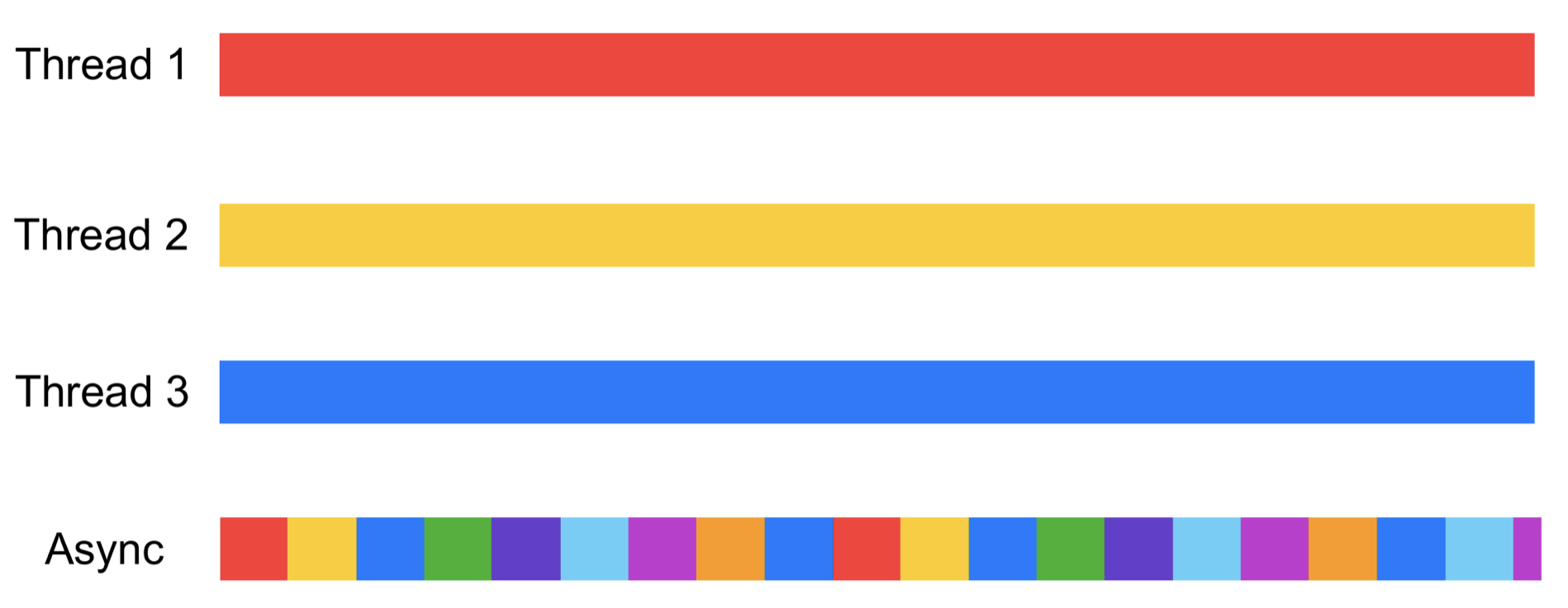
Прежде чем осмыслить это, читайте: 10. Блокирующие и неблокирующие операции.

## **Параллелизм и конкурентность**

**Параллелизм –** когда несколько задач выполняются одновременно(параллельно). На картинке Thread 1-3. Такая форма обычно контролируется системой

**Конкурентность** – когда несколько задач выполняются **совместно, но не одновременно**. Такую форму использует **асинхронность**(на картинке async). Задача может разбиваться на несколько подзадач.

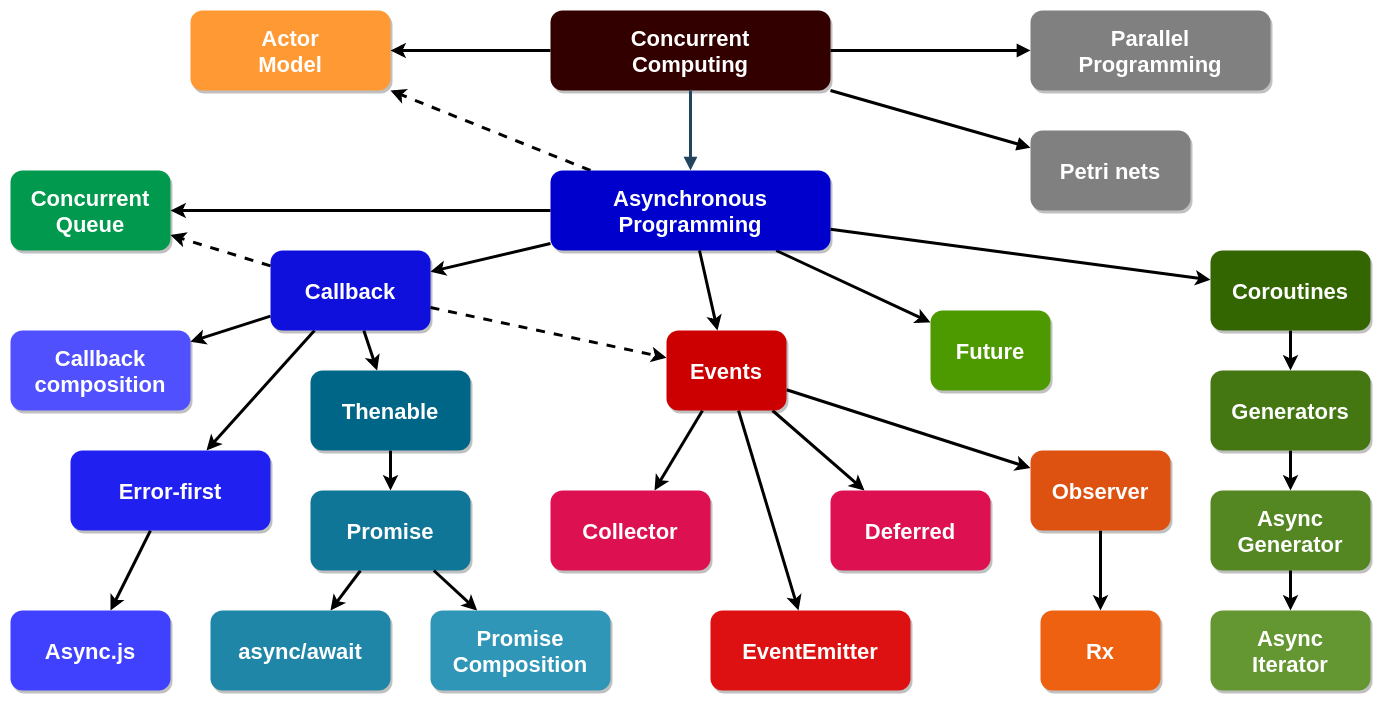


Параллелизм подразумевает конкурентность (Читайте про GIL, замки, гонки и т.д.). Но конкурентность не всегда подразумевает параллелизм.

## **Асинхронное программирование**

**Асинхронный код** убирает блокирующие операции из основного потока.

**Асинхронные приложения** **производительные** из за того, что приложение берет на себя ответственность переключения задач.



**Асинхронный код –** это набор абстрактных парадигм, которые вместе дают асинхронное (конкурентное) выполнение.

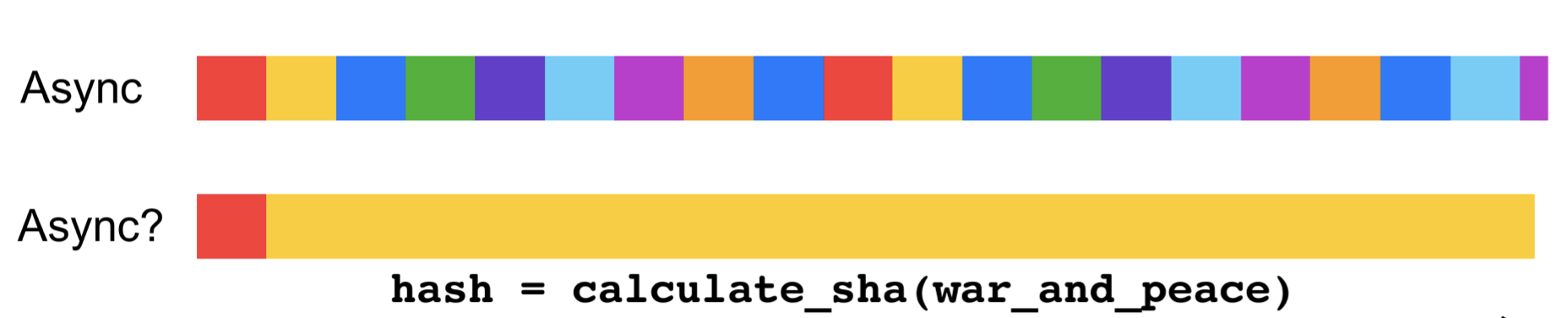
**Асинхронность** достигается с помощью **Event Loop** – цикл событий.

В **Event Loop** кладуться **coroutines (**сопрограммы или корутины), которые выполняются в нем. **Coroutines** способны приостанавливаться в тот момент, когда **Event Loop** прикажет это сделать.

**Task** (задача) – позволяет запускать **coroutine** на фоне.

**Future** (футура) – будущий результат выполнения **coroutine**. Возвращает футуру, значение которой еще не подсчитано. Программа может или подождать(допустим в бесконечном цикле) выполнения футуры или пойти заниматься другими делами. **Future** – наследник **Task**.

**Проблемы** **асинхронного программирования**:

* Синхронный и асинхронный код не живут вместе. Решения:
  + Найти в интернете асинхронную реализацию
  + Написать самому асинхронную реализацию
  + **ThreadPool**
* Не для всего есть асинхронные интерфейсы (допустим открыть файл асинхронно можно). Решение:
  + **ThreadPool**
* **CPU-bound** задачи. **Event-loop** кто то должен вращать, если его кто-то заблокирует – встанет все приложение. Решения:
  + **Process Pool**
  + Отдельный микросервис
* **RPS** может быть ограничен не только сервером (например медленный интернет). Решение только 1 – думать над архитектурой. There is no magic here.

Когда использовать **асинхронный код**:

* Микросервисы (**I/O-bound**, не **CPU-bound**)
* Долгоживущие соединения (**websocket**, раздача файлов)
* Производительная инфраструктура (шардирование базы, кеши, write-heavy очереди)
* Экономия ресурсов серверов

Когда **не** использовать **асинхронный код**:

* **CPU-bound**
* Боттлнек в инфраструктуре (1 инстанс базы), но можно пережить, если нагрузка не упирается в небеса (<1800 RPS), а инстанс выдерживает.
* Вы очень богаты и можете позволить себе сколько угодно железа для сервера

Все дальше относится к asyncio!

Асинхронное программирование – набор парадигм, а asyncio это Python библиотека помогающая их использовать для решения своих проблем.

Что есть **asyncio**?

* Библиотека для написания конкурентных задач в **Python** 3.4+
* Учитывает опыт **Twisted, Tornado, Tulip** (PEP 3156), **greenlet** и прочих, оглядывается на **Curio** и **Trio**
* Привнес синтаксис **async** / **await** (**Python** 3.5+)
* Прослойка между расширениями, работающими на функциях обратного вызова и **async** / **await**

Что такое стандарт **asyncio**?

* Фундамент для асинхронных фреймворков
* Базовые абстракции (**Future/Coroutine/Task/AbstractEventLoop**)
* Высокоуровневый API
  + Сопрограммы (**coroutine, generator coroutine**), задачи (**Task**)
  + **Streams**, примитивы для синхронизации, **Queues**
  + **API** для работы с процессами и межпроцессного взаимодействия
* Низкоуровневый API
  + **loop.\*, asyncio.Future**
  + Транспорты и протоколы

**Awaitable-объекты**

* Можно использовать в выражении **await**
* Использование объекта в выражении **await** означает, что текущая **сопрограмма** переключит **контекст** и будет ожидать, пока выражение не будет выполнено
* Существует 3 встроенных типа **awaitable** объектов: **coroutine, Task, Future**

Что такое **сопрограмма**?

Методика связи программных модулей друг с другом по принципу

кооперативной многозадачности: модуль приостанавливается в

определённой точке, сохраняя полное состояние (включая стек

вызовов и счётчик команд), и передаёт управление другому. Тот, в

свою очередь, выполняет задачу и передаёт управление обратно,

сохраняя свои стек и счётчик. © Wikipedia

**Запуск сопрограмм:**

* **asyncio.run()**:
  + запуск event loop
  + ожидания окончания работы асинхронной функции
  + завершение работы event loop и отмена всех порожденных асинхронных задач
* **await** – запуск асинхронного кода с явным переключением контекста
* **asyncio.create\_task()** – запуск задачи в фоновом режиме

**asyncio Coroutine** вPython **–** генератор, а с 3.5+ объект который ведет себя как генератор

* Объект, который имеет ряд инструкций, умеет хранить свое состояние и может переключать контекст (передавать управление)
* Является более обобщенной формой подпрограмм (интерпретатор может выходить в подпрограмму в 1 точке и выходить в другой. В сопрограммах может быть несколько точек входа, выхода и возврата).

**asyncio Coroutine function** в Python:

* Функция, возвращающая объект coroutine
* Определяется ключевыми словами async def
* Может содержать ключевые слова await, async for, async with

**asyncio Task** – сопрограмма, запущенная или запланированая для запуска в цикле событий и контексте:

* Позволяет запускать задачи в фоновом режиме
* Создается с помощью **asyncio.create\_task()** или **loop.create\_task()**, которые оборачивают сопрограмму в объект Task и планирует ее выполнение в цикле событий на ближайшее время
* Абстракция, позволяющая отменить/прервать выполнение сопрограммы с помощью **.cancel()**

**asyncio Future**

* Специальный низкоуровневый **awaitable** объект, представляющий конечный результат выполнения асинхронной операции
* Позволяет использовать низкоуровневый код, реализованный на функциях обратного вызова с высокоуровневым кодом на async/await
* Создается с помощью **loop.create\_future()**

**asyncio.sleep()** приостанавливает действие текущей задачи на

указанное время, позволяя выполнять другие задачи

*Если указан параметр result, его значение возвращается*

*вызывающему объекту по завершении работы сопрограммы*

**asyncio.gather()** запускает указанные **awaitable** объекты в

конкурентном режиме и возвращает результаты выполнения в том же

порядке

Оборачивает объекты **coroutine** в **asyncio Task**

В случае отмены **asyncio.gather()** отменяются все запущенные (но

еще не завершенные) задачи

**asyncio.shield()** защищает awaitable объект от отмены

**asyncio.wait\_for()** ожидает выполнения задачи в течение указанного времени, если задача не успевает выполнится - она отменяется и бросается asyncio.TimeoutError

* Если кто-то отменяет wait\_for(), то и обернутый им awaitable обьект тоже отменяется
* Отмену задачи можно предотвратить с помощью asyncio.shield()

**asyncio.as\_completed()** запускает awaitable объекты, возвращает итератор по результатам в порядке выполнения (сначала - самые быстро вычисленные)

**asyncio.current\_task()** вернет выполняющуюся в данный момент задачу или None

**asyncio.all\_tasks()** вернет все незаконченные задачи, запущенные в цикле событий

**asyncio Policy** (политика)

* Глобальный объект для каждого процесса, отвечает за выбор, настройку и управление циклом событий
* Определяет понятие контекста и управляет отдельным циклом событий для каждого контекста (по умолчанию контекст - текущий поток)
* По умолчанию используется **DefaultEventLoopPolicy**, использует **SelectorEventLoop** на **\*nix** и **ProactorEventLoop** на **Windows**
* Есть альтернативные **WindowsSelectorEventLoopPolicy** и **WindowsProactorEventLoopPolicy**
* Можно получить текущую с помощью **asyncio.get\_event\_loop\_policy()**
* Настраивается с помощью **asyncio.set\_event\_loop\_policy(policy)**

**Цикл событий**

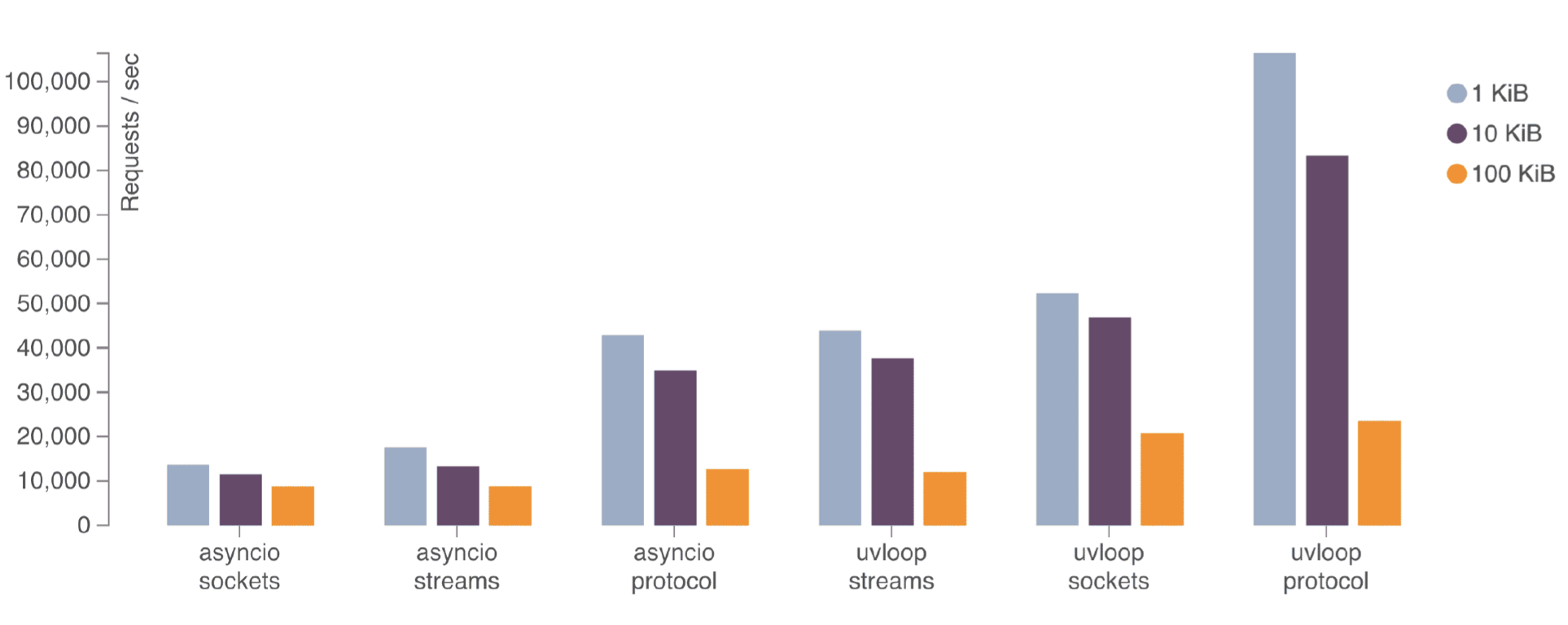
* Ядро любого **asyncio** приложения
* Выполняет асинхронные задачи и функции обратного вызова из очереди, выполняет сетевой **I/O**, управляет выполнением подпроцессов.

**SelectorEventLoop** использует модуль **selectors**, который включает в себя

* SelectSelector
* PollSelector
* EpollSelector
* DevpollSelector
* KqueueSelector
* DefaultSelector

**Умеет**:

* Планировать обратных вызовов
* Открывать сетевые подключения, в т.ч. защищенные (**TLS**)
* Создавать сетевые серверы
* Эффективно передавать файлы (**sendfile**)
* Мониторить файловые дескрипторы
* Напрямую работать с объектами **socket**
* Резолвить **DNS** (в потоках, потому что “unix плох”)
* Обрабатывать сигналы операционной системы (\*nix)
* Выполнять код в пулах потоков или процессов
* Выполнять подпроцессы



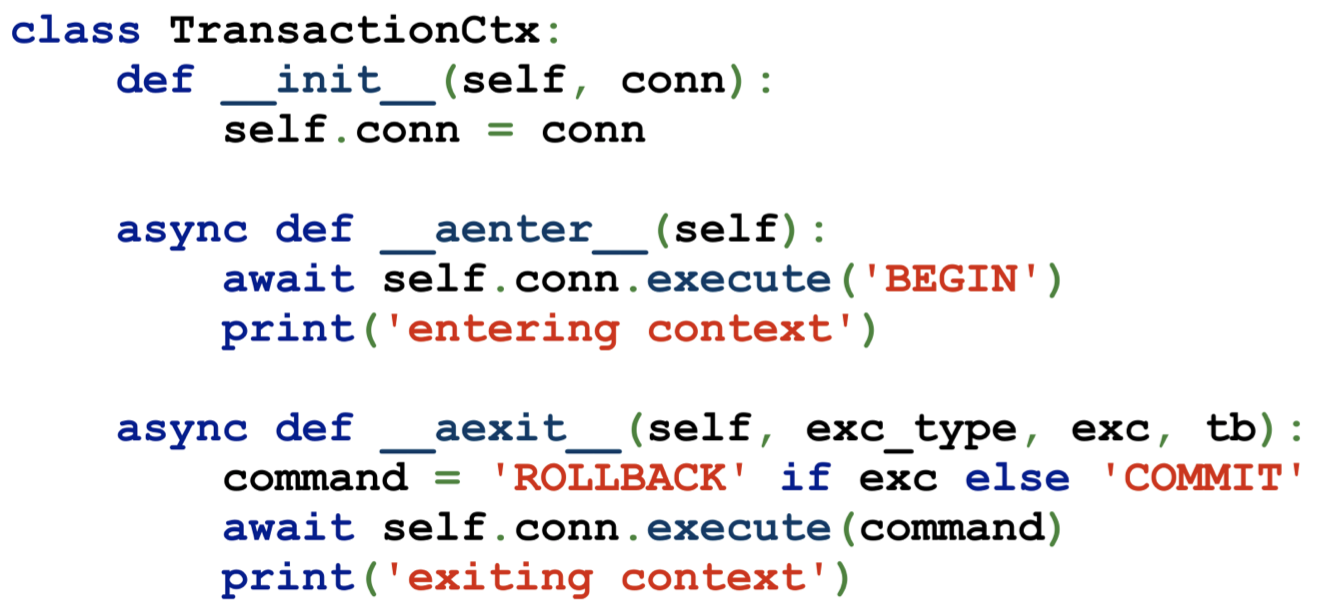
Альтернативные циклы: **uvloop**

* Реализован поверх **libuv**, стабильный
* Может дать хороший прирост производительности, если есть очень много сетевого **I/O**

**Асинхронный менеджер контекста:**

* Способен приостановить выполнение в **\_\_aenter\_\_** и **\_\_aexit\_\_** методах
* Как и с обычными менеджерами контекста, можно использовать несколько объектов с оператором async with
* PEP 492

Пример:

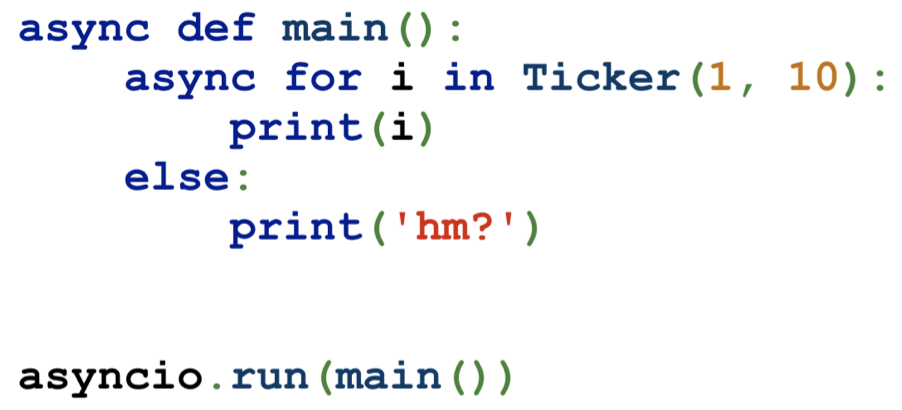
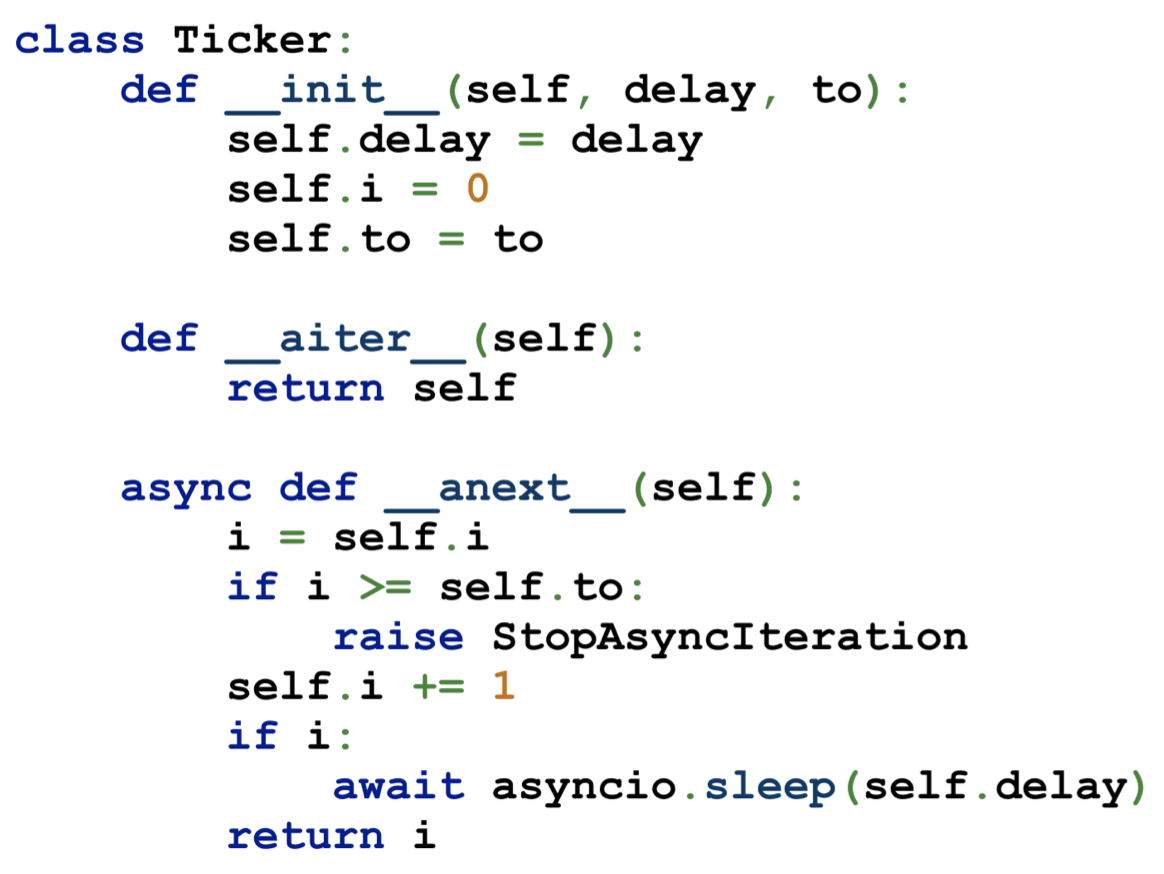
****

****

**Асинхронные итераторы:**

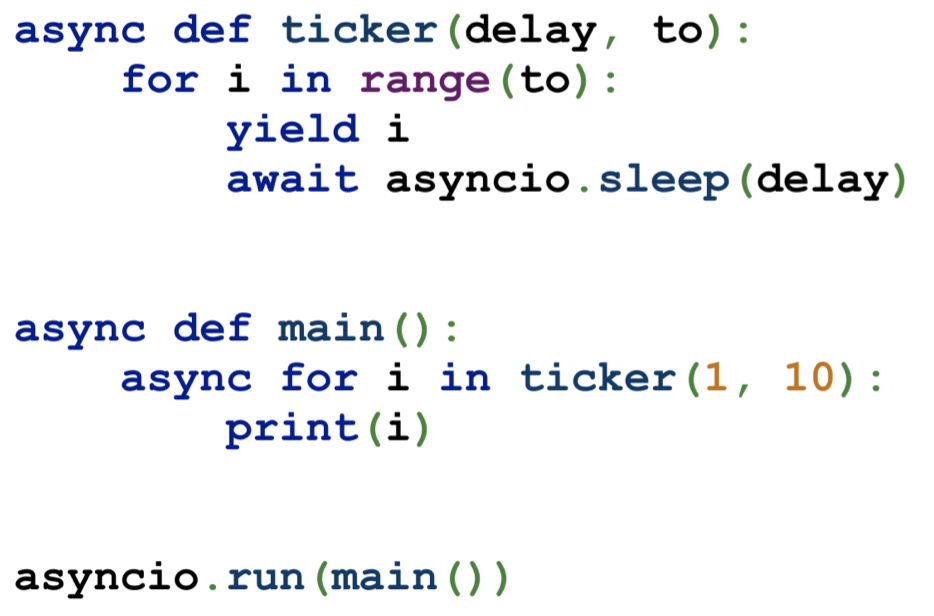
* Можно вызывать асинхронный код
* Итерируемый объект должен реализовывать метод **\_\_aiter\_\_**
* Итератор должен реализовывать асинхронный метод **\_\_anext\_\_**
* По завершении метод **\_\_anext\_\_** должен бросить исключение **StopAsyncIteration**
* PEP 492

Пример:

****

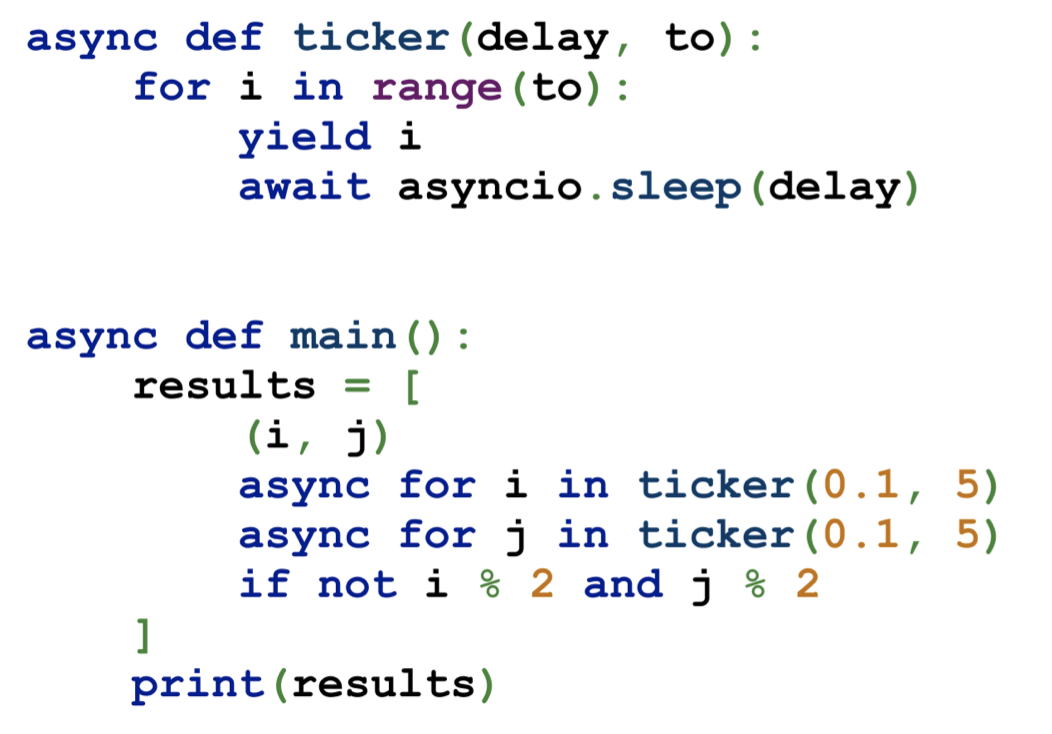
**Асинхронные генераторы**

* Асинхронная функция, в которой используется **yield**
* Вместо **send** и **throw** - асинхронные **asend()** и **athrow()**
* Можно использовать с **async for**
* Не поддерживают **yield from**
* PEP 525

****

**Asynchronous comprehensions**

* В python 3.6+ поддерживаются все compherensions:
  + Множетсво set(): {i async for i in agen()}
  + Список list(): [i async for i in agen()]
  + Словарь dict(): {i: i \*\* 2 async for i in agen()}
  + Генератор generator(): (i \*\* 2 async for i in agen())
* Можно сочетать с for и условиями if
* PEP 530

****