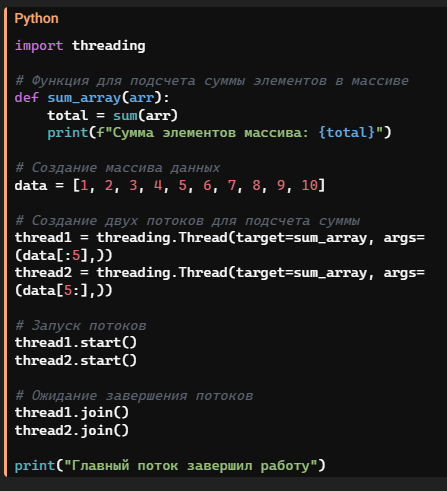
**Python threading**

Многопоточное программирование на Python – это способ организации выполнения нескольких потоков одновременно в рамках одного процесса. При использовании многопоточности в Python, можно выполнять несколько задач параллельно, что может значительно ускорить выполнение программы, особенно в случаях, когда процессор имеет несколько ядер.

Для реализации многопоточности в Python часто используется модуль threading, который позволяет создавать и управлять потоками выполнения. За счет многопоточности можно добиться повышения производительности программы и эффективного использования ресурсов компьютера. Однако, необходимо учитывать особенности работы с потоками, такие как возможные проблемы с синхронизацией доступа к общему ресурсу и потенциальные проблемы с блокировками.

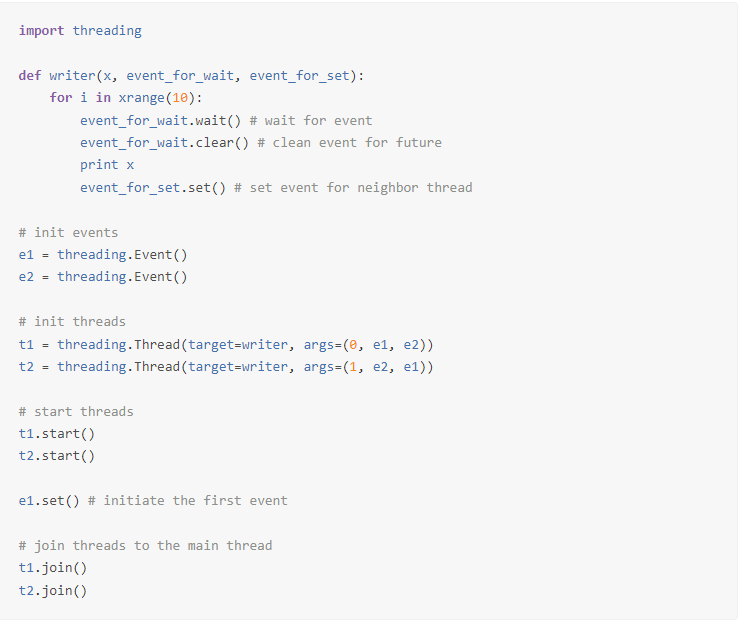
Многопоточное программирование на Python можно реализовать с помощью модуля threading. Вот пример простой программы, которая параллельно выполняет задачу по подсчету суммы элементов в массиве:



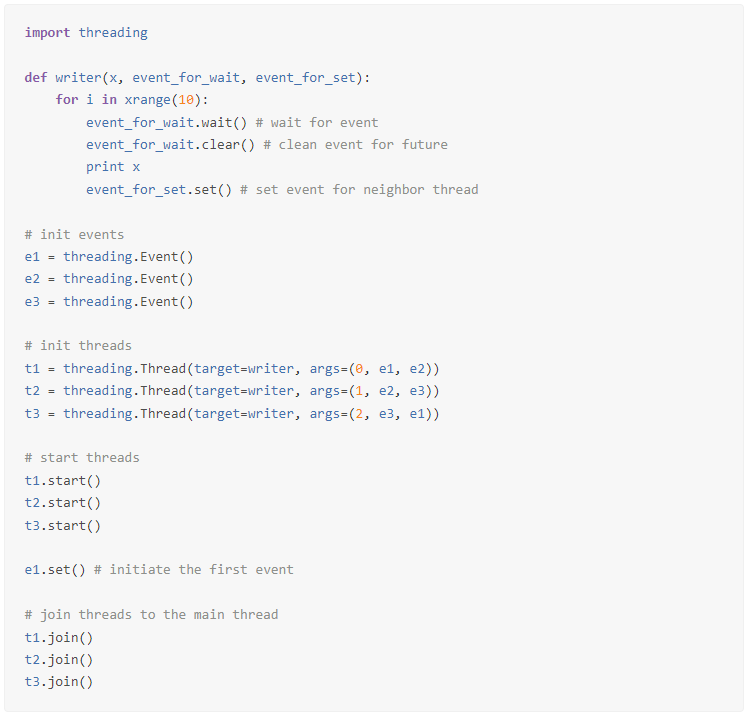
В этом примере создаются два потока, каждый из которых выполняет функцию sum\_array на разных подмассивах данных. После запуска потоков ожидается их завершение. В результате вы увидите сумму всех элементов массива и сообщение о завершении выполнения основного потока.

Помните, что в Python есть GIL (Global Interpreter Lock), который ограничивает одновременное выполнение нескольких потоков с использованием CPU. Поэтому для выполнения параллельных задач, требующих CPU, часто рекомендуется использовать модуль multiprocessing вместо threading.

Рассмотрим пример программы, которая запускает 2 потока. Один поток пишет десять “0”, другой — десять “1”, причем строго по-очереди.



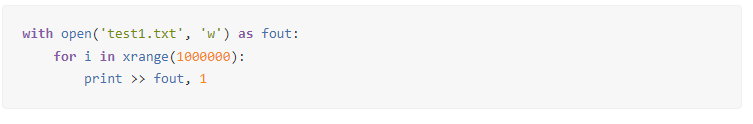
Никакой магии и voodoo-кода. Код четкий и последовательный. Причем, как можно заметить, мы создали поток из функции. Для небольших задач это очень удобно. Этот код еще и достаточно гибкий. Допустим у нас появился 3-й процесс, который пишет “2”, тогда код будет выглядеть так:



Мы добавили новое событие, новый поток и слегка изменили параметры, с которыми  
стартуют потоки (можно конечно написать и более общее решение с использованием, например, MapReduce, но это уже выходит за рамки этой статьи).  
Как видим по-прежнему никакой магии. Все просто и понятно. Поехали дальше.

**Global Interpreter Lock**

Существуют две самые распространенные причины использовать потоки: во-первых, для увеличения эффективности использования многоядерной архитектуры cоврменных процессоров, а значит, и производительности программы;  
во-вторых, если нам нужно разделить логику работы программы на параллельные полностью или частично асинхронные секции (например, иметь возможность пинговать несколько серверов одновременно).  
  
В первом случае мы сталкиваемся с таким ограничением Python (а точнее основной его реализации CPython), как Global Interpreter Lock (или сокращенно GIL). Концепция GIL заключается в том, что в каждый момент времени только один поток может исполняться процессором. Это сделано для того, чтобы между потоками не было борьбы за отдельные переменные. Исполняемый поток получает доступ по всему окружению. Такая особенность реализации потоков в Python значительно упрощает работу с потоками и дает определенную потокобезопасность (thread safety).  
  
Но тут есть тонкий момент: может показаться, что многопоточное приложение будет работать ровно столько же времени, сколько и однопоточное, делающее то же самое, или за сумму времени исполнения каждого потока на CPU. Но тут нас поджидает один неприятный эффект. Рассмотрим программу:



Эта программа просто пишет в файл миллион строк “1” и делает это за ~0.35 секунды на моем компьютере.

Рассмотрим другую программу:



Эта программа создает 2 потока. В каждом потоке она пишет в отдельный файлик по пол миллиона строк “1”. По-сути объем работы такой же, как и у предыдущей программы. А вот со временем работы тут получается интересный эффект. Программа может работать от 0.7 секунды до аж 7 секунд. Почему же так происходит?  
  
Это происходит из-за того, что когда поток не нуждается в ресурсе CPU — он освобождает GIL, а в этот момент его может попытаться получить и он сам, и другой поток, и еще и главный поток. При этом операционная система, зная, что ядер много, может усугубить все попыткой распределить потоки между ядрами.  
  
UPD: на данный момент в Python 3.2 существует улучшенная реализация GIL, в которой эта проблема частично решается, в частности, за счет того, что каждый поток после потери управления ждет небольшой промежуток времени до того, как сможет опять захватить GIL   
  
«Выходит на Python нельзя писать эффективные многопоточные программы?», — спросите вы. Нет, конечно, выход есть и даже несколько.

**Многопроцессные приложения**

Для того, чтобы в некотором смысле решить проблему, описанную в предыдущем параграфе, в Python есть модуль [subprocess](http://docs.python.org/library/subprocess.html). Мы можем написать программу, которую хотим исполнять в параллельном потоке (на самом деле уже процессе). И запускать ее в одном или нескольких потоках в другой программе. Такой способ действительно ускорил бы работу нашей программы, потому, что потоки, созданные в запускающей программе GIL не забирают, а только ждут завершения запущенного процесса. Однако, в этом способе есть масса проблем. Основная проблема заключается в том, что передавать данные между процессами становится трудно. Пришлось бы как-то сериализовать объекты, налаживать связь через PIPE или друге инструменты, а ведь все это несет неизбежно накладные расходы и код становится сложным для понимания.  
  
Здесь нам может помочь другой подход. В Python есть модуль [multiprocessing](http://docs.python.org/library/multiprocessing.html). По функциональности этот модуль напоминает **threading**. Например, процессы можно создавать точно так же из обычных функций. Методы работы с процессами почти все те же самые, что и для потоков из модуля threading. А вот для синхронизации процессов и обмена данными принято использовать другие инструменты. Речь идет об очередях (Queue) и каналах (Pipe). Впрочем, аналоги локов, событий и семафоров, которые были в threading, здесь тоже есть.  
  
Кроме того в модуле multiprocessing есть механизм работы с общей памятью. Для этого в модуле есть классы переменной (Value) и массива (Array), которые можно “обобщать” (share) между процессами. Для удобства работы с общими переменными можно использовать классы-менеджеры (Manager). Они более гибкие и удобные в обращении, однако более медленные. Нельзя не отметить приятную возможность делать общими типы из модуля ctypes с помощью модуля multiprocessing.sharedctypes.  
  
Еще в модуле multiprocessing есть механизм создания пулов процессов. Этот механизм очень удобно использовать для реализации шаблона Master-Worker или для реализации параллельного Map (который в некотором смысле является частным случаем Master-Worker).  
  
Из основных проблем работы с модулем multiprocessing стоит отметить относительную платформозависимость этого модуля. Поскольку в разных ОС работа с процессами организована по-разному, то на код накладываются некоторые ограничения. Например, в ОС Windows нет механизма fork, поэтому точку разделения процессов надо оборачивать в:



Впрочем, эта конструкция и так является хорошим тоном.

Для написания параллельных приложений на Python существуют и другие библиотеки и подходы. Например, можно использовать Hadoop+Python или различные реализации MPI на Python (pyMPI, mpi4py). Можно даже использовать обертки существующих библиотек на С++ или Fortran. Здесь можно было упомянуть про такие фреймфорки/библиотеки, как Pyro, Twisted, Tornado и многие другие