ДИПЛОМНАЯ РАБОТА

*Тема:*  ***Сравнение различных подходов к реализации асинхронного программирования: asyncio, threading и multiprocessing***.

Автор: Михайлюков Вадим Николаевич.

2024

Содержание.

|  |  |
| --- | --- |
| 1. Введение.  2. Описание (**asyncio, threading и multiprocessing**).  3. Исходные данные и модель для исследования.  4. Структура проекта.  5. Методика и результаты исследования.  6. Итоги и выводы.  7. Заключение.  8. Приложение (Тексты программ) | - 2  - 2  - 6  - 7  - 8  -11  -12  -13 |

*Задача:* Реализовать асинхронные задачи с использованием *asyncio, threading и multiprocessing*, сравнить их производительность и уместность для различных типов задач.

**1. Введение.**

Большое многообразие программных средств, библиотек, приложений Python ставит разработчика ПО перед выбором такого подхода к разработке, который на заданной вычислительной базе, требуемых (минимальных, либо заранее заданных) временных ресурсах позволил бы реализовать поставленную задачу.

Сказанное в полной мере относится к решению практических задач по анализу данных, собранных в результате какой-либо деятельности. При этом, условно разделив решение подобной задачи на этап сбора данных, ввода их в программный модуль и непосредственно обработку данных, рассмотрим далее последние два вопроса, предполагая, что данные для обработки уже сформированы в известном разработчику виде и формате.

**2. Описание** (**asyncio, threading и multiprocessing**).

Итак, рассматривая задачу анализа данных, будем предполагать, что данные, которые предстоит анализировать, имеются и содержатся в виде файлов известного вида и находятся в определенной директории жесткого диска либо облака (хотя это и не принципиально - и с таким же успехом они могут быть размещены в сети).

Поскольку время выполнения задачи ввода/вывода данных зависит от многих параметров, в том числе от объема данных, времени доступа к объекту хранения данных, скорости реакции самого объекта хранения, такие задачи, как правило, являются по своей сути асинхронными. А если каждое такое обращение в режиме ввод/вывод будет сопровождаться блокировкой других шагов выполнения программы - общее время выполнения программы будет, вообще говоря, непрогнозируемым.

Для преодоления таких эффектов в Python созданы такие средства как

**asyncio, threading и multiprocessing.**

**аsyncio** – библиотека, которая предоставляет инфраструктуру для написания параллельного кода с использованием концепции асинхронного программирования. Она позволяет эффективно обрабатывать многочисленные задачи ввода-вывода (например, сетевые операции или чтение/запись из файлов) без необходимости создавать множество потоков или процессов.

Ключевые концепции asyncio:

- Событийный цикл (Event Loop).

Это ядро **asyncio**, которое отвечает за планирование и выполнение задач (корутин). Работает по принципу однопоточной многозадачности (single-threaded concurrency). Постоянно опрашивает очередь событий и выполняет соответствующие задачи.

- Корутины (Coroutines).

Это специальные функции, выполнение которых можно приостанавливать на определенных точках с помощью ключевого слова await. Могут ждать асинхронные события (ввод-вывод, сигналы или другие корутины). Определяются с помощью async def.

- Задачи (Tasks).

Представляют собой обертки над корутинами для их планирования и выполнения. Создаются с помощью  asyncio.create\_task(coroutine)  или  asyncio.ensure\_future(coroutine).

Задачи можно отменять, объединять с другими задачами и отслеживать на предмет исключений. Задачи добавляются в событийный цикл и выполняются в порядке их готовности.

**asyncio**  работает следующим образом:

- Создается событийный цикл, который будет управлять выполнением задач.

- Определяются корутины, которые выполняют некоторую работу и могут приостанавливаться на операциях ввода-вывода с помощью ключевого слова await.

- Корутины оборачиваются в задачи с помощью функции asyncio.create\_task().

- Задачи планируются для выполнения в событийном цикле с помощью функции asyncio.run() или добавляются в цикл вручную.

- Событийный цикл начинает выполнять задачи по очереди. Когда задача достигает операции ввода-вывода (например, await asyncio.sleep()), она приостанавливается, и событийный цикл переключается на другую задачу.

Когда операция ввода-вывода завершается, задача возобновляется и продолжает выполнение до следующей операции ввода-вывода или завершения.

asyncio позволяет эффективно использовать один поток и один процесс для выполнения множества задач с операциями ввода-вывода, избегая блокировки и простоев из-за ожидания завершения этих операций.

Важно отметить, что asyncio оптимален именно для I/O-bound задач (сеть, файловые операции и т. д.), и не подходит для CPU-bound операций (вычислительно-интенсивных задач), поскольку в этом случае он не сможет переключаться на другие задачи во время блокировки. Для такого рода задач лучше использовать многопоточность или многопроцессорность.

**threading -** многопоточное программирование — это мощный инструмент для увеличения производительности в Python. Он позволяет одновременно выполнять несколько задач внутри одного процесса. Это особенно полезно в многозадачных приложениях, где несколько операции должны выполняться параллельно.

Thread – это отдельный поток выполнения. И, несмотря на то, что Thread является элементом многопоточного программирования, это не означает, что в программе могут работать две и более подпрограммы одновременно. Разные потоки на самом деле не работают одновременно. Различные задачи, внутри потоков выполняются на одном ядре, а операционная система управляет, когда программа работает с каким потоком.

Ответственность за такое управление несет GIL (Global Interpreter Lock ). Суть GIL заключается в том, что выполнять байт код может только один поток. GIL не дает нескольким потокам исполнять одновременно код Python. Поэтому получается, что разные потоки не будут выполняться параллельно, а программа просто будет между ними переключаться, выполняя их в разное время. Однако, если в каком-то потоке есть некое «ожидание» (пакетов из сети, запроса пользователя, пауза типа time.sleep), то в такой программе потоки будут выполняться как будто параллельно, поскольку во время таких пауз ресурсы процессора можно передать другому потоку.

Стандартная библиотека Python предоставляет библиотеку [threading](https://docs.python.org/3/library/threading.html), которая содержит необходимые классы для работы с потоками. Основной класс в этой библиотеке - Thread, который отвечает за создание, управление и мониторинг потоков. Поток можно создать на базе функции, либо реализовать свой класс – наследник Thread и переопределить в нем метод run().

Чтобы запустить отдельный поток, нужно создать экземпляр потока Thread и затем запустить его с помощью метода .start():

Поскольку потоки выполняется на одном процессоре, они хорошо подходят для ускорения некоторых задач, но не для всех. Задачи, которые требуют значительных вычислений ЦП и тратят мало времени на ожидание внешних событий, очевидно, используя многопоточность, не будут выполняться быстрее.

То есть, в каждый конкретный момент времени, будет выполняться только один поток, несмотря на то, что в нашем распоряжении может быть многоядерный процессор (или многопроцессорный сервер), плюс ко всему, будет тратиться время на переключение между задачами.

В Python все стандартные библиотечные функции, которые выполняют блокирующий ввод-вывод, освобождают GIL, это дает возможность поработать другим потокам, пока ожидается ответ от ОС.

Потоки отлично подходят для задач, которые связаны с операциями ввода-вывода:

- Подключение к оборудованию и подключение по сети в целом

- Работа с файловой системой

- Скачивание файлов по сети

**multiprocessing -** **(м**ультипроцессинг) - это использование нескольких процессов для выполнения задач параллельно.

Процесс — это независимая единица выполнения в операционной системе. Каждый процесс имеет собственный набор ресурсов и изолирован от других процессов. Процесс включает в себя один или несколько потоков, которые являются подмножеством процесса. Потоки разделяют ресурсы этого процесса, такие как память и файловые дескрипторы.

Модуль multiprocessing в Python позволяет создавать процессы и управлять ими. Каждый такой процесс выполняется в своем собственном пространстве памяти и имеет свой собственный интерпретатор Python, у каждого процесса свой GIL. Это означает, что выполнение каждого процесса не зависит от выполнения других процессов. Однако, такая изоляция процессов идет в ущерб скорости их создания и коммуникации между ними.

Основные концепции мультипроцессинга:

1. *Параллельное выполнение*: Мультипроцессинг позволяет запускать несколько процессов одновременно, что позволяет использовать многопроцессорные системы более эффективно и улучшает производительность программы.

2. *Работа с потоками*: Каждый процесс в мультипроцессинге имеет свой собственный поток выполнения, независимый от других процессов. Это позволяет избежать проблем с глобальной блокировкой интерпретатора (GIL), которая присутствует в многопоточных приложениях.

3. *Использование ресурсов*: Каждый процесс имеет свое собственное пространство памяти и ресурсы, что обеспечивает изоляцию между процессами и предотвращает конфликты доступа к данным.

4. *Модуль multiprocessing*: Для работы с мультипроцессингом используется встроенный модуль `multiprocessing`, который предоставляет API для создания и управления процессами. Он аналогичен модулю `threading`, но использует отдельные процессы вместо потоков для параллельного выполнения кода.

**3. Исходные данные и модель для исследования.**

В качестве модели для исследования выберем задачу анализа биржевых сводок торгов по группам биржевых кодов (ценных бумаг).

Ценные бумаги выпускаются партиями, от десятка до несколько миллионов штук. Каждая такая партия (выпуск) имеет свой торговый код на бирже – тикер.

Все бумаги из одной партии (выпуска) одинаковы в цене, поэтому говорят о цене одной бумаги. У разных выпусков бумаг - разные цены.

Каждая биржевая сделка характеризуется:

- тикер ценнной бумаги

- время сделки

- цена сделки

- обьем сделки (сколько ценных бумаг было куплено)

В ходе торгов цены сделок могут со временем расти и понижаться. Величина изменения цен называется волатильностью.

В нашем исследовании данные сводок по биржевым сделкам ценных бумаг собраны в одной директории на жестком диске в виде отдельных файлов (\*\*\* .csv). Имя каждого файла, например TICKER\_BRJ9.csv, характеризует сделку по ценной бумаге BRJ9, а строки содержимого файлов - параметры сделки (тикер, время сделки, цена сделки, количество проданных бумаг). Количество таких сделок в каждом файле (по каждому тикеру) разное

Задача состоит в расчете волатильности сделки по каждому тикеру за все зафиксированные в файле торги.

Волатильность (%) рассчитывается по формуле:

***vol(%) = ((max.price – min.price) / ((max.price + min.price) /2)) \* 100***

Где: *- max.price* - максимальная цена сделки;

*- min.price* - минимальная цена сделки;

Требуется вывести данные по 3 тикера с минимальной волатильностью,

3 тикера с максимальной волатильностью и все тикеры с нулевой волатильностью.

**4. Структура проекта.**

Как уже было сказано, объектом исследования будет время, необхо-димое для выполнения операций, обозначенных на рис.1 индексами 1-5. При этом блоки 1 – 4 должны реализовать одну и ту же задачу - получение исходных данных из Data source, возможно, их какую-то предварительную обработку, и доставку полученных результатов определенной структуры в блок окончательной обработки 5. В каждом блоке эта задача решается по-своему.

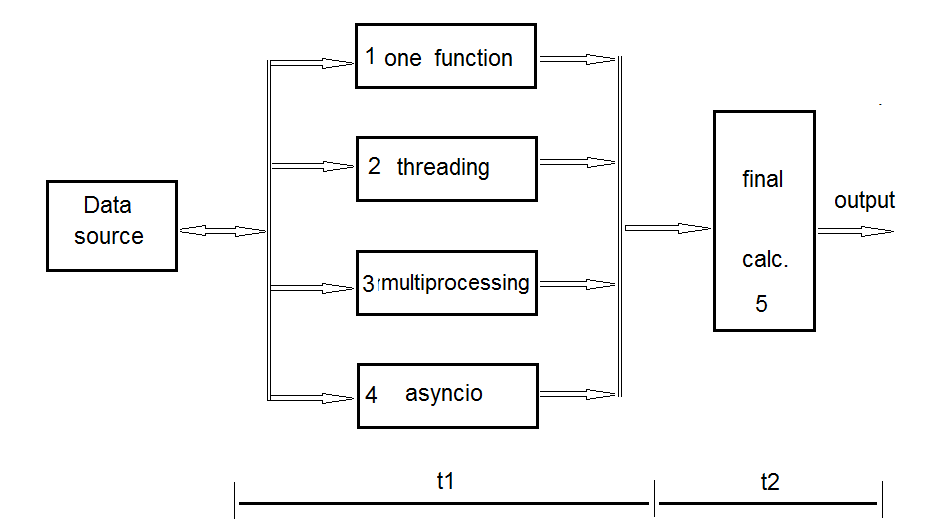


Рис. 1

Предстоит:

- разработать программы, реализующие ввод данных из известной директории жесткого диска, методами функционального программирования, с помощью потоков (threading), многопроцессорной структуры (multiprocessing) и asyncio (1 – 4);

- разработать программу результирующих расчетов (5);

- оценить время выполнения операций 1 – 4 (t11- 4) для каждого из вариантов и 5 (t2).

Структура данных в блоке Data source:

Файл - TICKER\_ALM9.csv (один из файлов, содержащихся в папке).

|  |  |  |
| --- | --- | --- |
| 1 | SECID,TRADETIME,PRICE,QUANTITY | заголовок |
| 2 | ALM9,12:59:44,9415.4500,1 | данные по сделке |
| 3 | ALM9,12:59:44,9416.4000,1 | * “ - |
| 4 | ALM9,14:35:19,9622.4000,1 | * “ - |
| 5 | ALM9,14:35:19,10119.4200,1 | данные по сделке |
| …. | ......... | …… |
| 75 | ALM9,11:23:04,9754.4700,2 | данные по сделке |
| 76 | ALM9,11:23:04,9655.9400,1 | данные по сделке |

Количество записей в каждом файле произвольное – от одного до нескольких десятков тысяч. Количество файлов в директории – тоже произвольное.

Структура данных на выходе блоков 1-4:

Результатом работы блоков 1-4 является таблица, в которой в каждой строке содержится следующая информация (через запятую):

- тикер;

- средняя цена по всем сделкам данного тикера;

- рассчитанная волатильность по всем сделкам данного тикера;

|  |  |
| --- | --- |
| 1 | AFH9,10441.0,17.26 |
| 2 | ALH9,9930.6,17.07 |
| 3 | AFM9,10314.61,9.52 |
| 4 | ALM9,9857.22,11.51 |
| 5 | BRG9,61.25,11.82 |
| . . . | . . . . |
| 111 | VBH9,3699.55,13.83 |
| 112 | VIH9,22.25,0.0 |

Количество строк таблицы равно количеству файлов в исходной директории (количеству тикеров, предоставленных для анализа). Эта таблица записывается в файл table.csv, который является исходным для блока 5.

|  |
| --- |
|  |

Структура данных на выходе блока 5 (выводится на консоль):

==== Нулевая волатильность: 14 позиций из 112 ====

CLM9, CYH9, EDU9, EuH0, EuZ9, JPM9, MTM9, O4H9, PDU9, PTU9, RIH0, RRG9, TRH9, VIH9,

==== 3 позиции с минимальной волатильностью: ====

RNU9 - 0.98 %

GOG9 - 0.97 %

CHM9 - 0.95 %

==== 3 позиции с максимальной волатильностью: ====

SiH9 - 24.39 %

PDM9 - 23.2 %

PDH9 - 22.69 %

**5. Методика и результаты исследования.**

Отправной точкой исследования была принята реализация синхронного подхода к решению поставленной задачи (блок 1), начиная с ввода данных в программу из блока Data source, и до получения результатов на выходе блока 5. Для более наглядной имитации вероятной задержки в канале обмена данными между их источником и программой, в программный модуль, отвечающий за этот обмен, была введена принудительная задержка 0,2 сек на этапе каждого обращения к источнику данных. Такая задержка применена для каждого варианта (1 – 4) решения задачи.

Для более корректного сравнения получаемых результатов коды программ (1 – 4) реализованы максимально идентично, с незначительными изменениями друг от друга, отражающими специфику применяемых подходов для каждого варианта.

Оценка временных затрат на выполнение задачи производилась по результатам усреднения 10 независимых испытаний. При этом окончательная обработка в блоке 5, одинаковая для всех вариантов 1 – 4, давала задержку в расчетах не более 0.0001сек, что на несколько порядков меньше, чем реализация задач 1 – 4. Поэтому в дальнейшем сосредоточим внимание на эффективности работы блоков 1 – 4.

Итак:

1. *Синхронный подход* к решению задачи - реализован в едином программном модуле D\_01. Время, которое потребовалось для решения задачи составило 28,1 сек.

1. *Многопоточный подход (threading),* модуль D\_02. При тех же условиях (**tdelay**= 0,2сек), позволил решить эту же задачу за 6,7 сек. Фактически это минимальное время для решения задачи на конкретном вычислительном средстве.

При этом интересно и важно, что увеличение задержки элементарного обращения к жесткому диску не приводит к пропорциональному увеличению времени работы программы. Так, для того, чтобы увеличить время работы многопоточной программы до исходной величины синхронной реализации задачи, потребовалось увеличить время элементарного обмена данными программы с источником данных практически в 100 (!) раз. В то время как для варианта D\_01 (синхронный вариант) при **tdelay**= 0.4 (увеличение в два раза), **tproc** составило уже 51 сек, т.е. тоже практически вдвое.

Динамика изменения времени работы программы **tproc** в зависимости от задержки обращения к диску **tdelay** , представлена в таблице:

Таблица 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **tdelay** | 0.0 | 0.2 | 0.5 | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 | 3.5 | 4.0 | 4.5 | 5.0 | 10.0 | 15.0 | 20.0 |
| **tproc** | 6,4 | 6,7 | 7,0 | 7,6 | 8.1 | 8.7 | 9.2 | 9.7 | 10.3 | 11.0 | 11.5 | 11.9 | 17.8 | 22.0 | 26.7 |

1. *multiprocessing,* модуль D\_03.

Как и ожидалось, если процесс один – время работы программы практически совпадает с вариантом синхронной реализации. Снижение временных затрат до показателей варианта *threading* происходит при организации 6 параллельных процессов с тенденцией незначительного плавного уменьшения при увеличении количества одновременно работающих процессов. Разумеется, можно организовать и большее количество процессов, однако при этом следует учитывать эффект гонки, преодоление которого также требует временных затрат.

Увеличение задержки доступа к источнику данных в этом варианте пропорционально сказывается на временных показателях работы программы в целом.

Результаты измерений временных затрат на решение задачи при использовании multiprocessing для различного числа процессов **Nproc** приведены в табл.2

Таблица 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Nproc** | 1 | 2 | 3 | 4 | 5 | 6 | 8 |
| **tdelay = 0.2** | 28.4 | 14.4 | 9.9 | 8.2 | 7.3 | 6.7 | 5.4 |
| **tdelay = 0.4** | 50.7 | 25.9 | 17.6 | 13.5 | 11.3 | 10.5 | 7.8 |

1. *Asyncsio*, модуль D\_04.

Модуль *get\_data\_,* который в терминологии *asyncio* является корутиной, - практически тот же, что и в предыдущих решениях. Отличие состоит лишь в том, что задержка обмена данных с источником формируется оператором *await asyncio.sleep(0.2), (*вместо *time.sleep(0.2))*, в результате которого выполнение текущей задачи приостанавливается на указанное время (имитация задержки в канале обмена данными), давая возможность работы следующей корутины, стоящей в очереди. Список корутин определен в функции *main*, а их запуск происходит по команде *asyncio.run(main()).*

Поскольку корутина *get\_data\_* у нас одна, но, в процессе решения задачи ей приходится извлекать разные файлы из источника их хранения Data source, логично обработку каждого такого файла (или группы файлов) представить как отдельную корутину и поставить ее в очередь наравне с другими такими же.

В нашем случае мы организовали стек из **N** таких корутин, которым проходим по всему списку файлов, подлежащих обработке.

Результаты зависисмости времени **tproc** от размеров стека N при tdelay = 0.2, приведены в табл. 3.

Таблица 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | 1 | 2 | 4 | 8 | 16 |
| **tproc** | 28,3 | 16,9 | 11,3 | 8,5 | 7,1 |

И здесь, как и в предыдущих вариантах, при N = 1 мы получили исходное время выполнения задачи, а увеличение очереди (числа корутин) ожидаемо приводит к экономии времени решения задачи.

1. **Итоги и выводы.**

Подводя итоги проведенного исследования решения конкретной практической задачи методами синхронного и асинхронного программирования можно сделать следующие выводы.

1. За счет применения рассмотренных методов асинхронного программирования удалось сократить время выполнения программы практически в 4 раза по сравнению с вариантом синхронной реализации программы.
2. Каждый из рассмотренных подходов асинхронного программирования способен реализовать близкие по эффективности результаты, однако:
   1. В многопроцессном варианте решения данной задачи приходится оптимизировать количество одновременно запускаемых процессов. А, поскольку всем процессам приходится оперировать с общими данными как на входе (*Data source*) так и на выходе (результат в файле *table*), необходимо еще и позаботиться о синхронизации доступа как со стороны входа, так и на выходе.
   2. Вариант с *asyncio*  также требует оптимизации количества корутин со стороны разработчика. И если в программе отдельные модули имеют явное различие в исходном алгоритме и структуре их реализации – тогда такое разделение более очевидно. Если же приходится организовывать корутины, реализующие один и тот же алгоритм, но с разными блоками данных - разработка такой программы более проблематична. Следует отметить еще одно свойство *asyncio*. Поскольку сценарий последовательности корутин статичен и определяется заранее прописанной логикой в программе, то, в случае возникновения проблем с выполнением какой-либо одной корутины (блокирующий вызов, например), - весь процесс останавливается и ждет разрешения этой проблемы (если это возможно). В противном случае выполнение всей программы останавливается.
   3. От необходимости разрешения подобных (2.1 и 2.2) проблем избавляет мультипоточная реализация задачи, в которой библиотека *threading* с помощью GIL берет на себя функции создания, управления и мониторинга потоков. В этом случае адаптивная настройка структуры потоков, работающих на единственном ядре (процессе), позволила наилучшим образом распорядиться возникающими в потоках простоями, связанными с задержками обмена данными **tdelay**. Если, к тому же, иметь в виду устойчивость варианта D\_02 к величине этих задержек, что иллюстрируют приведенные в табл.1 результаты (увеличение **tdelay**, в 10 раз привело к снижению производительности всего на 30%), то *именно вариант многопоточной реализации рассматриваемой задачи представляется предпочтительным.*
3. **Заключение.**

Все приведенные результаты, рассуждения и выводы справедливы для конкретной рассматриваемой задачи. Тем не менее, общие рекомендации по повышению эффективности вычислительных процессов и использованию аппаратных ресурсов остаются такими же, как и в большинстве литературных источников:

- задачи, связанные с замкнутыми сложными и громоздкими расчетами предпочтительнее реализовать в мультипроцессном варианте;

- если в задаче приходится обращаться к источникам хранения информации, будь то базы данных на сервере, любая информация на внешних носителях или в сети и на это уходит значительная часть временных ресурсов - тут место методов асинхронного программирования;

- для задач, включающих в себя и то и другое, возможно сочетание обоих подходов.

В целом, для выбора наиболее подходящей стратегии решения каждой конкретной задачи, можно было бы порекомендовать при разработке конкретного проекта провести исследования, подобные приведенным выше.

P.S.

*Автор не ставил перед собой задачу написать “красивый” код программы,*

*сосредоточив внимание на его функциональной адекватности рассматриваемому в каждом конкретном случае подходу.*

1. **Приложение.**

Тексты программ имеют обозначения, указанные на рис.1

**D\_01.py - (1 - one function)**

import csv

import os

import time, datetime

list = os.listdir("trades")

print(len(list), list)

print()

if os.path.exists('table.csv'):

os.remove('table.csv')

class StockManager():

def \_\_init\_\_(self):

self.list = list

super().\_\_init\_\_()

def get\_data1(self, path):

ticker\_list = []

with open(path, 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

ticker\_list.append(row)

ticker\_price\_list = []

for j in range(1, len(ticker\_list)):

ticker\_price\_list.append(float(ticker\_list[j][2]))

max\_price = max(ticker\_price\_list)

min\_price = min(ticker\_price\_list)

average\_price = round((max\_price + min\_price) / 2, 2)

volatility = round(((max\_price - min\_price) / average\_price) \* 100, 2)

x = (str(path))[14:18]

table = [x, average\_price, volatility]

# --------------------------------------------------

time.sleep(0.2) # имитация задержки обмена данными

with open('table.csv', 'a', newline='') as f:

csv.writer(f).writerow(table)

# --------------------------------------------------

def read\_file(self):

res\_table = []

with open('table.csv', 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

res\_table.append(row)

# print(len(res\_table), res\_table)

return res\_table

def zero\_volat(self,table):

self.table = table

zero\_list = []

for i in range(len(self.table)):

if (self.table[i][2]) == '0.0':

zero\_list.append(self.table[i])

print('==== Нулевая волатильность:', len(zero\_list),'позиций ====')

for i in range(len(zero\_list)):

print(zero\_list[i][0], end=', ')

print()

# return zero\_list

def min\_volat(self, table):

self.table = table

no\_zero\_list = []

for i in range(len(self.table)):

if (self.table[i][2]) != '0.0':

no\_zero\_list.append(self.table[i])

no\_zero\_list.sort(key=lambda x: float(x[2]))

print('==== 3 позиции с минимальной волатильностью: ====')

for i in range(2,-1,-1):

print(no\_zero\_list[i][0],' - ', no\_zero\_list[i][2])

def max\_volat(self, table):

self.table = table

no\_zero\_list = []

for i in range(len(self.table)):

if (self.table[i][2]) != '0.0':

no\_zero\_list.append(self.table[i])

no\_zero\_list.sort(key=lambda x: float(x[2]))

print('==== 3 позиции с максимальной волатильностью: ==== ')

for i in range(-1,-4,-1):

print(no\_zero\_list[i][0],' - ', no\_zero\_list[i][2])

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

list = os.listdir("trades") # формирование путей к файлам

for i in range(len(list)):

list[i] = 'trades/' + list[i]

manager = StockManager()

start = datetime.datetime.now()

for i in range(len(list)):

manager.get\_data1(list[i])

end = datetime.datetime.now()

print('Время подготовки данных: ',end-start)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

start = datetime.datetime.now()

table = manager.read\_file()

manager.zero\_volat(table)

manager.min\_volat(table)

manager.max\_volat(table)

end = datetime.datetime.now()

print('Время анализа данных: ', end-start)

**D\_02.py - (2 - threading)**

import csv

import os

import datetime, time

from threading import Thread

if os.path.exists('table.csv'): # удаление файла table.csv

os.remove('table.csv')

print('-------------------------------------------------------\*\*')

class StockManager(Thread):

def \_\_init\_\_(self, list):

self.list = list

super().\_\_init\_\_()

def get\_data\_(self, path): # - вычисления для одного тикера

self.path = path

ticker\_list = []

with open(self.path, 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

ticker\_list.append(row)

ticker\_price\_list = []

for j in range(1, len(ticker\_list)):

ticker\_price\_list.append(float(ticker\_list[j][2]))

max\_price = max(ticker\_price\_list)

min\_price = min(ticker\_price\_list)

average\_price = round((max\_price + min\_price) / 2, 2)

volatility = round(((max\_price - min\_price) / average\_price) \* 100, 2)

x = (str(path))[14:18]

table = [x, average\_price, volatility]

# --------------------------------------------------

time.sleep(0.2) # имитация задержки обмена данными

with open('table.csv', 'a', newline='') as f:

csv.writer(f).writerow(table)

def run(self):

self.list = list

self.get\_data\_(self.list[i])

# --------------------------------------------------

#print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

start = datetime.datetime.now()

list = os.listdir("trades")

for i in range(len(list)):

list[i] = 'trades/' + list[i]

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

start = datetime.datetime.now()

threads = []

for i in range(len(list)):

thread = StockManager(list[i])

thread.start()

threads.append(thread)

for thread in threads:

thread.join()

end = datetime.datetime.now()

print('Total processing time =', end - start)

print('\*\*\* table.csv file created !!! \*\*\*')

**D\_03.py - (3 - multiprocessing)**

import csv

import multiprocessing

import os

import datetime, time

if os.path.exists('table.csv'):

os.remove('table.csv')

print('-------------------------------------------------------\*\*')

def get\_data\_(path): # - вычисления для одного тикера

ticker\_list = []

with open(path, 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

ticker\_list.append(row)

# print(ticker\_list)

ticker\_price\_list = []

for j in range(1, len(ticker\_list)):

ticker\_price\_list.append(float(ticker\_list[j][2]))

max\_price = max(ticker\_price\_list)

min\_price = min(ticker\_price\_list)

# print('max price = ',max\_price, 'min price = ',min\_price)

average\_price = round((max\_price + min\_price) / 2, 2)

volatility = round(((max\_price - min\_price) / average\_price) \* 100, 2)

x = (str(path))[14:18]

table = [x, average\_price, volatility]

# --------------------------------------------------

time.sleep(0.2) # имитация задержки обмена данными

with open('table.csv', 'a', newline='') as f:

csv.writer(f).writerow(table)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

list = os.listdir("trades")

for i in range(len(list)):

list[i] = 'trades/' + list[i]

if \_\_name\_\_ == '\_\_main\_\_':

with multiprocessing.Pool(processes=4) as pool:

start = datetime.datetime.now()

pool.map(get\_data\_, list)

end = datetime.datetime.now()

print('Total processing time =', end - start)

print('\*\*\* table.csv file created !!! \*\*\*')

**D\_04.py - (4) - asyncio**

import csv

import os

import time, datetime

import asyncio

if os.path.exists('table.csv'):

os.remove('table.csv')

print('-------------------------------------------------------\*\*')

async def get\_data\_(path): # - вычисления для одного тикера

ticker\_list = []

with open(path, 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

ticker\_list.append(row)

ticker\_price\_list = []

for j in range(1, len(ticker\_list)):

ticker\_price\_list.append(float(ticker\_list[j][2]))

max\_price = max(ticker\_price\_list)

min\_price = min(ticker\_price\_list)

average\_price = round((max\_price + min\_price) / 2, 2)

volatility = round(((max\_price - min\_price) / average\_price) \* 100, 2)

x = (str(path))[14:18]

table = [x, average\_price, volatility]

# --------------------------------------------------

await asyncio.sleep(0.2) # имитация задержки обмена данными

with open('table.csv', 'a', newline='') as f:

csv.writer(f).writerow(table)

# ---------------------------------------

async def main():

for i in range(0, len(list), 8):

await asyncio.gather(get\_data\_(list[i]), get\_data\_(list[i+1]),

get\_data\_(list[i+2]), get\_data\_(list[i + 3]),

get\_data\_(list[i+4]), get\_data\_(list[i + 5]),

get\_data\_(list[i + 6]), get\_data\_(list[i + 7]))

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

list = os.listdir("trades")

for i in range(len(list)):

list[i] = 'trades/' + list[i]

print(len(list), 'list... = ', list)

print(' -- path list created --')

if \_\_name\_\_ == '\_\_main\_\_':

start = datetime.datetime.now()

asyncio.run(main())

end = datetime.datetime.now()

print('Total processing time =', end - start)

print('\*\*\* table.csv file created !!! \*\*\*')

**D\_05.py - (5 final calc.)**

import csv

import datetime, time

def read\_file():

res\_table = []

with open('table.csv', 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

res\_table.append(row)

# print(len(res\_table), res\_table)

return res\_table

def zero\_volat(table):

zero\_list = []

for i in range(len(table)):

if (table[i][2]) == '0.0':

zero\_list.append(table[i])

print('==== Нулевая волатильность:', len(zero\_list),'позиций из ', len(table), ' ====')

for i in range(len(zero\_list)):

print(zero\_list[i][0], end=', ')

print()

return zero\_list

def min\_volat(table):

no\_zero\_list = []

for i in range(len(table)):

if (table[i][2]) != '0.0':

no\_zero\_list.append(table[i])

no\_zero\_list.sort(key=lambda x: float(x[2]))

print('==== 3 позиции с минимальной волатильностью: ====')

for i in range(2,-1,-1):

print(no\_zero\_list[i][0],' - ', no\_zero\_list[i][2],'%')

return no\_zero\_list

def max\_volat(table):

no\_zero\_list = []

for i in range(len(table)):

if (table[i][2]) != '0.0':

no\_zero\_list.append(table[i])

no\_zero\_list.sort(key=lambda x: (float(x[2])), reverse=True)

print('==== 3 позиции с максимальной волатильностью: ==== ')

for i in range(3):

print(no\_zero\_list[i][0],' - ', no\_zero\_list[i][2],'%')

return no\_zero\_list