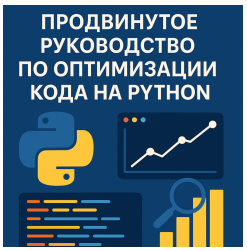

Руководство по ускорению и оптимизации Python-кода

Введение. В этом руководстве рассмотрены продвинутые техники оптимизации Python-программ, особенно актуальные для backend-разработки. Мы последовательно разберем профилирование, выбор структур данных и алгоритмов, эффективное использование стандартной библиотеки, оптимизацию циклов, применение таких инструментов как Numba/Cython/PyPy, параллелизм и асинхронность, работу с вводом-выводом, кеширование, обработку больших данных, компиляцию/упаковку кода и общие советы по написанию быстрого и поддерживаемого кода. Помните, что **преждевременная оптимизация вредна** – начинать стоит с понимания узких мест через профилирование, затем устранять крупнейшие проблемы, не жертвуя чистотой кода без необходимости [1].

1. Профилирование и бенчмаркинг

Чтобы ускорить код, сперва необходимо выяснить, **где** он тормозит. Python предоставляет инструменты для измерения производительности и поиска узких мест:

- **Модуль `timeit`:** позволяет замерять время исполнения небольших фрагментов кода с высокой точностью. Удобен для микробенчмарков и сравнения альтернативных реализаций функции. Например, так можно измерить время создания списка списком генератором:

```
import timeit
stmt = "[(a, b) for a in (1,3,5) for b in (2,4,6)]"
print(timeit.timeit(stmt, number=1000000))
```

По умолчанию `timeit` выполняет код многократно (1_000_000 раз в примере выше) и возвращает общее время. Можно настраивать число повторов `number` и даже запускать несколько серий через `timeit.repeat`. `timeit` удобен, когда нужно сравнить две небольшие функции или выражения – он исключает влияние запуска интерпретатора и делает несколько прогонов для усреднения результата [2] [3]. Например, если у вас есть два способа вычислить нечто, `timeit` поможет выбрать более быстрый [4]. Недостаток – он не показывает, какая часть кода медленная, а только общее время фрагмента.

- **Профилировщик `cProfile`:** встроенный детерминированный профилировщик, который собирает статистику времени выполнения по функциям. С его помощью можно узнать, сколько раз вызывалась каждая функция и сколько времени суммарно она выполнялась [4] [5]. `cProfile` можно запустить из командной строки:

```
python -m cProfile -o profile.stats my_script.py
```

Это сохранит результаты в файл. Затем их можно проанализировать с помощью модуля `pstats` или визуализировать утилитами вроде SnakeViz [6]. Встроенный способ – вызвать

`cProfile.run()` внутри скрипта или обернуть код в `cProfile.Profile()` как контекстный менеджер. Например:

```
import cProfile, pstats
def work():
    ...  # некий код
with cProfile.Profile() as pr:
    work()
ps = pstats.Stats(pr).sort_stats("cumtime")
ps.print_stats(10)  # печать 10 самых "тяжелых" по времени функций
```

Отчет `cProfile` покажет, какие функции занимают больше всего времени. Однако он выводит **много данных**, включая вложенные вызовы встроенных функций, что может затруднить анализ в больших приложениях [6]. Для упрощения анализа часто используют визуализаторы (тот же SnakeViz) или сужают область профилирования до интересующих частей.

- **Line Profiler (`line_profiler`):** внешний инструмент для **построчного профилирования** функций. Он показывает, сколько времени затрачено на выполнение каждой строки кода в отмеченных функциях [7]. Это очень помогает, если нужно понять, внутри функции какая строчка самая медленная. Использование: устанавливаем пакет (`pip install line_profiler`), отмечаем интересующие функции декоратором `@profile`, затем запускаем скрипт через утилиту `kernprof`:

```
kernprof -l -v my_script.py
```

После выполнения вы получите отчёт с разбивкой по строкам. Например, если вы подозреваете, что вложенный цикл тормозит, line_profiler точно покажет, на какой строке цикла программа проводит больше всего времени. **Line_profiler** должен быть первым выбором при оптимизации конкретных участков кода [8] [9], так как сразу указывает узкие места построчно.

- **Другие инструменты:** существуют и другие профилировщики. Например, `memory_profiler` для анализа потребления памяти, а также *семплирующие* профилировщики вроде **PyInstrument** (вместо замера каждого вызова, он периодически опрашивает стек вызовов, снижая оверхед) [10]. Семплирующие профилировщики меньше нагружают систему и дают сводку по тому, где программа проводит время в целом. Они полезны для долгих приложений (веб-сервисы и пр.), где детальный вывод `cProfile` избыточен.

Рекомендации по профилированию: всегда начинайте оптимизацию с измерений. Интуиция часто подводит – можно тратить время на ускорение той части кода, которая и так выполняется быстро, игнорируя скрытое узкое место. Поэтому сначала профилируем и определяем, какие функции или участки кода доминируют во времени выполнения. Затем сфокусированно оптимизируем их. Например, применив `cProfile` вы можете обнаружить, что 80% времени уходит на парсинг JSON, или на сортировку, или на работу с базой данных – и уже исходя из этого выбирать подход (будь то переход на более эффективную библиотеку, изменение алгоритма, кеширование результатов и т.д.). Профилирование следует повторять после изменений, чтобы подтвердить улучшение и убедиться, что новые «горячие точки» не появились.

Также желательно писать небольшие бенчмарки для критичных функций. Библиотека `timeit` или IPython-магия `%timeit` удобны для этой цели. Например, сравнение двух реализаций алгоритма сортировки кастомных объектов с помощью `%timeit` сразу покажет, какая быстрее, при прочих равных.

2. Структуры данных и алгоритмы

Правильный выбор структур данных и алгоритмов способен дать значительный выигрыш в скорости – куда больший, чем низкоуровневые оптимизации. В Python разные структуры имеют разные сложности операций (см. таблицу сложностей CPython [11] [12]). Рассмотрим основные структуры и когда их применять:

- **Список (`list`):** динамический массив. Хорошо подходит для **последовательного хранения и итерации**. Получение или изменение элемента по индексу – *O(1)*, добавление в конец – амортизированно *O(1)*, **но вставка или удаление в середине/начале – O(n)**, так как элементы смещаются [13] [12]. Если нужно часто удалять из начала, **список неэффективен** – стоит взять `collections.deque` (двунаправленная очередь). Итерация по списку – линейная *O(n)*, что обычно быстро на C-уровне, но если нужны частые проверки на принадлежность, у списка они *O(n)*.

- **Множество (`set`) и словарь (`dict`): хеш-таблицы**. Операции поиска, добавления и удаления элемента выполняются за **среднее O(1)** (константное время) благодаря хешированию. Поэтому, **если нужна быстрая проверка принадлежности или поиск по ключу – используйте `set` / `dict` вместо списка** [14]. Например, поиск элемента в списке из 100000 элементов в среднем потребует 50000 проверок, а в множестве – одну хеш-функцию и проверку в таблице. Важно: `set` не сохраняет порядок (до Python 3.7 dict сохраняет порядок добавления, set – нет). Словари удобны для хранения соответствий "ключ-значение" с быстрым доступом по ключу. Пример: чтобы выяснить, присутствует ли определённое значение в коллекции, лучше делать `x in my_set` вместо `x in my_list` – **в первом случае операция в среднем константная, во втором – линейная** [15]. Ниже приведено сравнение времени работы поиска в списке и множестве на миллион элементов:

```
import time
lst = list(range(1000000))
s = set(lst)
import random; x = random.randrange(1000000)
# Измерим время единичной проверки
t0 = time.time(); _ = (x in lst); t1 = time.time()
t2 = time.time(); _ = (x in s);  t3 = time.time()
print(f"Поиск в list: {t1-t0:.6f} сек, в set: {t3-t2:.6f} сек")
```

```
Поиск в list: 0.014504 сек, в set: 0.000001 сек
```

(В примере элемент присутствует в коллекциях. Проверка в списке ~0.0145 с, во множестве ~0.000001 с – различие на порядки, которое с ростом N ещё увеличивается).

Note: Сложность O(1) для dict/set относится к **среднему случаю**. В худшем случае (множество коллизий) время может деградировать до O(n), но такого почти не бывает при хорошем хешировании. Также, поиск **по значению** в словаре (например, `val in my_dict.values()`) остаётся O(n) – словари оптимизированы для поиска по ключу, а не по значению [16] [17].

- **Двунаправленная очередь (`collections.deque`):** оптимизирована для **операций на обоих концах**. Добавление и удаление *в начало или конец* `deque` – O(1) [18], тогда как у списка удаление из начала – O(n) (все элементы сдвигаются) [13] [19]. Поэтому, если реализуете очередь или обработку потока данных, где требуется делать `pop(0)` или `appendleft`, используйте deque. Пример: чтение файлов построчно (где нужно оперативно выкидывать старые элементы) удобно делать через deque, ограничивая её длину, – это будет эффективнее, чем вручную поддерживать список фиксированной длины.

- **Приоритетная очередь (`heapq`):** реализует **минимальную кучу** на базе списка. Полезна, когда нужно быстро получать минимальный (или максимальный, с трюком отрицательных значений) элемент из изменяющейся коллекции. Функции `heapq.heappush` и `heapq.heappop` работают за O(log n) – значительно быстрее, чем, например, сортировка списка при каждом добавлении [20]. Используйте `heapq`, если решаете задачу, где регулярно добавляются элементы и нужно каждый раз брать самый маленький/большой (например, алгоритм Дейкстры, отслеживание Top-K элементов и т.п.). Пример: вместо того чтобы держать отсортированный список из 10000 элементов и вставлять в него через `bisect` (что O(n)), лучше вести кучу размером 10000 – вставка/удаление будут O(log n) ~ 14 операций вместо 10000.

- **Модуль `bisect`:** предоставляет **бинарный поиск** по отсортированному списку и вставку в него. Операция `bisect.bisect_left(a, x)` возвращает позицию для вставки элемента x в список a *за O(log n)*, однако сама вставка в список – *O(n)* (из-за сдвига элементов) [21]. То есть `bisect` помогает с поиском индекса (логарифмическое время), но вставлять всё равно приходится средствами списка, что линейно. **Вывод:** `bisect` удобен, когда нужно **много искать, но мало вставлять**. Например, если у вас есть **отсортированный список** и вы часто делаете поиск диапазона или позицию для элемента – `bisect` идеален. Но если планируется частое добавление, может быть лучше использовать другую структуру (дерево, сбалансированное бинарное, напр. `sortedcontainers` из PyPI, или связку bisect+deque). Python-документация прямо отмечает: *«для поиска конкретных значений словари эффективнее»* [14] (если порядок не важен). Например, вместо бинарного поиска индекса значения в списке, часто можно использовать множество/словарь для теста наличия значения.

- **Другие структуры из `collections`:**

- `collections.Counter` – быстрый подсчет элементов (реализован на базе словаря). Для частотного анализа данных предпочитайте `Counter(x)` или метод `.count()` у списка (если один элемент) вместо ручного цикла.
- `collections.OrderedDict` (в Python 3.7+ обычный dict уже упорядочен) – если нужен словарь, сохраняющий порядок вставки.
- `collections.defaultdict` – словарь с значениями по умолчанию, полезен для агрегаций (избегает проверок на наличие ключа, немного ускоряя и упрощая код).

- **Выбор алгоритмов:** кроме структур, важно выбрать эффективный алгоритм. Например, если задача требует сортировки, использовать встроенную сортировку Timsort (O(n log n)) вместо квадратичных алгоритмов. Если нужно объединить отсортированные списки – эффективнее воспользоваться `heapq.merge` или `itertools.chain` (в зависимости от задачи), чем сначала конкатенировать и сортировать заново. Всегда думайте о **сложности алгоритма**: оптимизация кода не спасёт алгоритм со сложностью O(n^2) на больших входах. Классический пример: наивный алгоритм перемножения матриц O(n^3) против алгоритма Страссена (~O(n^2.8)) – выбор алгоритма даст огромный выигрыш, тогда как микроптимизации внутри O(n^3) мало что изменят. В повседневных задачах: используйте поиск по хеш-таблице вместо вложенных циклов (set вместо списка при множественных проверках включения [22]), сортировку за n log n вместо квадратичного обхода, и т.д.

Пример влияния выбора структуры: допустим, вы делаете множество проверок `if x in collection`. Если `collection` – список из 100000 элементов, каждая такая проверка в среднем ~50 тыс сравнений. Если заменить на `set`, проверка – одна операция хеширования и один сравнений. В реальном тесте: поиск элемента в списке занял ~0.15 секунды, в множестве ~0.11 (включая время создания структур) [23] [24], но при многократных проверках преимущество set станет подавляющим (как мы показали выше, при 1000 проверках – set быстрее списка в сотни раз). Другой пример: нужно часто удалять первый элемент – список будет каждый раз сдвигать все оставшиеся элементы, это медленно. Дек (deque) решает проблему, выполняя popleft мгновенно [13] [19]. Эти решения на уровне выбора структуры зачастую дают **наибольший прирост производительности**.

3. Эффективное использование стандартной библиотеки и встроенных функций

Стандартная библиотека Python и встроенные функции написаны на C и оптимизированы, поэтому их использование может заметно ускорить код. **Правило:** *«если есть готовая функция в стандартной библиотеке – используйте её, вместо реализации вручную»*. Рассмотрим несколько примеров:

- **Суммирование и агрегаты:** встроенная функция `sum(iterable)` написана на C и работает быстрее эквивалентного цикла на Python. Пример: сумма миллион элементов циклами vs `sum()`. Код:

```
import time
n = 1000000
# суммирование в цикле
start = time.time()
total = 0
for i in range(1, n):
    total += i
t_loop = time.time() - start
# суммирование встроенной функцией
start = time.time()
total2 = sum(range(1, n))
t_sum = time.time() - start
print(f"loop: {t_loop:.3f}s, sum(): {t_sum:.3f}s")
```

```
loop: 0.20s, sum(): 0.03s
```

Результат показывает ~7-кратное ускорение при использовании `sum` вместо ручного цикла (0.20 с против 0.03 с). Аналогично работают `min(seq)`, `max(seq)`, `all(seq)`, `any(seq)` – они реализованы на C и быстрее, чем эквивалентные Python-циклы. Как отмечается, *«встроенные функции Python высоко оптимизированы, что делает их значительно быстрее ручных реализаций на Python»* [25]. То есть, **старайтесь пользоваться готовыми агрегатами**: если нужно проверить, есть ли хоть один элемент удовлетворяющий условию – используйте `any(...)` вместо цикла с break; нужно проверить, что все – `all(...)` вместо цикла и флага; найти максимум – `max()`; и т.д.

- **Генераторы списков и встроенные функции высшего порядка:** Встроенные **функции высшего порядка** (`map`, `filter`, `zip` и т.д.) и **генераторные выражения** часто более эффективны, чем эквивалентный явный цикл на Python. Причина – они выполнены на уровне C-интерпретатора, уменьшая количество интерпретаций байткода. Например, `map(math.sqrt, data)` вызовет `math.sqrt` для каждого элемента, но сама итерация управляется C-кодом внутри `map`. List comprehension `[func(x) for x in data]` близка по скорости, так как тоже подкапотный цикл в C. Оба подхода, как правило, быстрее, чем:

```
res = []
for x in data:
    res.append(func(x))
```

Плюс код лаконичнее. Более того, генераторы позволяют **избежать лишних списков в памяти**, давая выигрыш по памяти (см. главу про генераторы).

- **Конкатенация строк:** Классический пример – сборка строки из множества частей. В Python **нельзя эффективно накапливать строку через `+=` в цикле**, это приводит к квадратичному времени (так как строки неизменяемы, каждое `+=` создает новую строку). Вместо этого используйте `"".join(list_of_strings)`. Пример:

```
parts = ["слово"] * 10000
# Неэффективный способ
s = ""
for p in parts:
    s += p
# Эффективный способ
s2 = "".join(parts)
```

Замерим время 100 повторений каждого способа:

```
import timeit
setup = 'parts = ["слово"]*10000'
t1 = timeit.timeit('s = ""\nfor p in parts:\n    s += p', setup=setup,
number=100)
```

```
t2 = timeit.timeit('s = "".join(parts)', setup=setup, number=100)
print(f"+= : {t1:.3f}s, join: {t2:.3f}s")
```

```
+= : 0.205s, join: 0.010s
```

Разница колоссальна: склеивание через `join` ~20 раз быстрее. **Вывод:** для объединения большого числа строк всегда используйте `str.join` [26]. Если строки поступают постепенно (например, формируется большой лог), рассмотрите использование `io.StringIO` – это буфер в памяти, который поддерживает эффективное добавление (метод `write`).

- **Итерируемые инструменты (`itertools`):** Модуль `itertools` содержит набор эффективных итераторов на C. Примеры: `itertools.chain` соединяет несколько списков или генераторов последовательно без создания промежуточных списков; `itertools.islice` берет срез итератора; `itertools.product`, `combinations`, `groupby` и др. – позволяют решать задачи перебора и группировки очень эффективно. Вместо вложенных циклов для декартова произведения списков, используйте `itertools.product` – он написан на C и более производителен. **Пример:** Нужно вычислить комбинации пар элементов из двух списков: `[(a,b) for a in A for b in B]` – Python-средство (list comprehension) уже хорошо, но `list(itertools.product(A,B))` будет не хуже, а если не нужен сразу весь список, `itertools.product` даст ленивый итератор, что сэкономит память.

- **Модуль `math` и другие C-библиотеки:** Если нужно выполнить математическую операцию, по возможности берите реализацию из модуля `math` или `statistics`. Например, `math.sqrt(x)` (обертка над C `sqrt`) быстрее, чем `x**0.5` (хотя последнее тоже быстро, но `math.sqrt` избегает создания промежуточного Python-объекта для результата). `math.factorial(n)` гораздо быстрее самописного вычисления факториала. Стандартный модуль `statistics` предоставляет C-оптимизированные вычисления среднего, медианы и т.д. – тоже лучше, чем писать самим.

- **Использование `zip` и распаковки:** При одновременном обходе нескольких коллекций используйте `zip` – он также работает на C и отдаёт элементы кортежами без создания промежуточных списков. Например, `for x, y in zip(list1, list2): ...` быстрее, чем ручной индексный обход.

- **Кэширование и повторное использование объектов:** Стандартная библиотека предоставляет инструменты кеширования (см. главу о кешировании). Например, вместо вызова сложной функции с одними и теми же параметрами в цикле можно декорировать её `functools.lru_cache` и получить результат из кеша (если это приемлемо с точки зрения логики).

- **Встроенные типы и методы:** Используйте методы списков, словарей, строк. Например, чтобы удалить все элементы из списка – `lst.clear()` быстрее, чем присваивать новый список. Чтобы перевернуть список – `lst.reverse()` (in-place) или `reversed(lst)` (итератор) вместо среза `lst[::-1]` (который создаёт копию). Для сортировки списка in-place: `lst.sort()` эффективнее, чем `sorted(lst)` если копия не нужна. Для поиска подстроки в строке – метод `.find()` или оператор `in` (они реализованы на C, с алгоритмом Boyer-Moore-Horspool), нежели писать свой цикл.

В целом, **перечитайте возможности стандартной библиотеки** – возможно, ваша задача уже решается встроенной функцией. Как пишет один источник, *«используйте возможности Python и библиотек, а не изобретайте велосипед – это и короче, и быстрее»* [27]. Например, нужно посчитать уникальные элементы – вместо цикла и списков используйте `set` или `Counter`. Нужно отсортировать по ключу – воспользуйтесь параметром `key` в `sorted` (он тоже в C проходит). Python – язык высокого уровня, и его сила в том числе в богатой оптимизированной библиотеке.

4. Оптимизация циклов и генераторов

Циклы – частый источник замедления в Python, потому что каждая итерация выполняется интерпретатором. Уменьшение количества операций в цикле или перевод цикла на уровень C может существенно ускорить код. Рассмотрим техники оптимизации циклов:

- **List Comprehensions (генераторные выражения для списков):** В Python принято использовать *генераторы списков* вместо явных циклов, когда это возможно. Они не только делают код короче, но и работают быстрее, так как цикл выполнен внутри интерпретатора на нижнем уровне. Пример: сравним заполнение списка квадратами чисел обычным циклом и list comprehension:

```
# Заполнение списка в цикле
res = []
for i in range(1, 1000000):
    res.append(i * 2)
# Заполнение списочным выражением
res2 = [i * 2 for i in range(1, 1000000)]
```

По замерам, **генератор списков опережает цикл с** `.append()` **примерно в 1.6 раза** на миллион элементов [28] [29] (0.17 с против 0.10 с в тесте [28]). Причина – `append` вызывает метод на каждом шаге, тогда как в comprehension всё происходит внутри C-цикла. Поэтому для преобразования или фильтрации последовательностей старайтесь использовать comprehensions или встроенные функции вместо ручных loops. Например, `filtered = [x for x in data if condition(x)]` быстрее, чем `for` + `if` с append.

- **Функции** `map` **и** `filter`**:** Они могут быть чуть более эффективны, чем эквивалентные генераторы списков, **если** передаваемая функция – встроенная или оптимизированная. Например, `map(str.lower, list_of_strings)` может быть быстрее `[s.lower() for s in list_of_strings]`, а уж тем более цикла, потому что вызов метода внутри comprehensions всё равно происходит на Python-уровне. Однако разница невелика; больше вопрос стиля. `filter(func, seq)` избегает создания промежуточного списка как при `[x for x in seq if func(x)]`, что экономит память. В Python 3 `map` и `filter` возвращают lazy-итераторы, поэтому если нужен список – не забудьте обернуть в `list(...)`. В целом, **comprehension чаще предпочтительнее из соображений читаемости**, но знание `map/filter` не помешает – иногда они вписываются элегантно.

- **Исключение лишних вычислений в цикле:** Минимизируйте работу внутри тела цикла. Если какой-то вычисляемый в цикле результат не меняется между итерациями – вынесите его вне цикла. Например, если вы вызываете неизменную функцию или обращаетесь к глобальной переменной на каждой итерации, это накладно. Классика:

```
for i in range(n):
    result = math.sqrt(25) + i
```

Здесь `math.sqrt(25)` вычисляется каждую итерацию, хотя результат всегда 5 – вынесите это за цикл. Или, если внутри цикла часто происходит обращение к атрибуту или глобальной переменной, можно закешировать ссылку в локальную переменную. Например:

```
import math
sqrt = math.sqrt  # кэшируем ссылку
for x in data:
    res = sqrt(x) * 10
```

Такая микро-оптимизация уменьшает накладные расходы на поиск имени в глобальном пространстве или атрибута объекта. В локальных переменных поиск выполняется быстрее. Конечно, выигрыши здесь небольшие, но в очень длительных циклах могут иметь эффект.

- **Использование `enumerate` и `zip` вместо ручного управления индексами:** Если нужен индекс в цикле, не делайте `for i in range(len(seq)):` и затем `seq[i]` – это и длиннее, и потенциально медленнее (Python вынужден каждый раз вычислять `len(seq)` и делать индексацию). Правильнее: `for i, val in enumerate(seq): ...`. Функция `enumerate` написана на C и итерируется по последовательности эффективно, возвращая кортежи (index, value). Аналогично, если нужно итерироваться синхронно по двум и более коллекциям, используйте `zip`. Это не столько про скорость (хотя она тоже выигрывает), сколько про читаемость и сокращение возможностей ошибки.

- **Generator Expressions (ленивые генераторы):** Вместо создания промежуточных списков, можно использовать генератор-итератор. Например, `sum(x*x for x in data)` – здесь мы не создаём список всех квадратов, а генерируем их по одному и сразу суммируем. **Это экономит память** и, возможно, время на выделение/освобождение памяти, хотя сами вычисления занимают столько же. Генераторы полезны, когда объем данных большой и нет нужды держать все результаты одновременно. В пайплайнах обработки данных можно передавать генератор от одной стадии к другой, избегая создания громоздких временных списков. Помните, что один проход по генератору истощает его – но именно это зачастую и нужно (единственный проход). Например, чтение большого файла: `sum(1 for _ in open('log.txt'))` – посчитает число строк без хранения их в памяти (в отличие от `len(open('log.txt').readlines())`).

- **Избегайте ненужных вложенных циклов Python:** Если возможно, переводите вложенные итерации на C-уровень. Пример: у вас вложенный цикл для обработки пар элементов – рассмотрите возможность использовать функции из `itertools` (как `product`, `combinations`) или numpy для числовых вычислений (см. раздел про NumPy). Вложенный Python-цикл – двойной удар по производительности (квадратичная сложность + высокий overhead интерпретатора на каждую итерацию). Иногда можно переписать двойной цикл через более эффективные средства. К примеру, поиск пересечений множеств из двух списков: вместо двойного цикла `for x in a: for y in b: ...` стоит преобразовать один список во множество и просто проверить каждого кандидата из другого на принадлежность (получится O(n+m) вместо O(n*m)). *Это уже переключение*

алгоритма/структуры, но демонстрирует подход: по возможности, не делать во вложенном Python-цикле то, что можно сделать более оптимально.*

- **Векторизация и NumPy:** Отдельно отметим – если ваши циклы связаны с числовыми расчетами над большими масивами данных, очень вероятно, что лучше использовать библиотеку **NumPy**, которая выполняет циклы на уровне C (см. раздел 9). Векторизированные операции могут ускорить вычисления на порядки, поэтому вместо того, чтобы перебирать массивы Python-циклами, попробуйте сформулировать операцию над целыми массивами/векторами.

Пример: оптимизация цикла через генераторное выражение. Имеем список цен `prices` и хотим получить список с налогом 20%. В лоб (цикл):

```
new_prices = []
for p in prices:
    new_prices.append(p * 1.2)
```

Оптимизированно (comprehension):

```
new_prices = [p * 1.2 for p in prices]
```

Кроме более лаконичного вида, второй вариант будет быстрее. А если конечный список нам не нужен, а, скажем, нужно просуммировать новые цены: `total = sum(p * 1.2 for p in prices)`. Это избежит создания `new_prices` в принципе.

Итог: **используйте идиоматические конструкции Python** для циклов – list/dict/set comprehensions, генераторы, функции `any/all/map/filter/zip` вместо ручных развёрнутых циклов. Это не только ускоряет выполнение, но и улучшает читаемость. Как заметил один разработчик: *«код пишется для людей, а не для машин. Если нужно ускорение – чаще дело в выборе алгоритма, а не в том, как именно написать цикл»* [30]. То есть, сначала сделайте код простым и понятным, а затем измерьте и оптимизируйте самые медленные циклы известными методами.

5. Ускорение кода с помощью Numba, Cython и PyPy

Иногда, несмотря на все оптимизации, Python-код остаётся слишком медленным из-за ограничений интерпретатора и GIL. Для вычислительно тяжёлых задач стоит рассмотреть **альтернативные исполнения** или компиляцию критичных участков:

- **Numba:** JIT-компилятор (just-in-time) для Python, особенно эффективный в области численных вычислений. С помощью Numba можно компилировать функции Python в машинный код во время исполнения, добавив декоратор `@numba.njit` (no-python mode). Numba отлично ускоряет циклы с числовыми расчётами, работу с массивами NumPy и прочий научный код. Пример использования:

```
from numba import njit
@njit
def sum_of_squares(n):
```

```
    total = 0
    for i in range(n):
        total += i * i
    return total

print(sum_of_squares(10000000))
```

Первая вызов функции может занять время (компиляция), но последующие выполняются на скоростях, близких к C. Преимущество Numba – **не требует переписывать код на другом языке**, вы просто помечаете функцию декоратором. Конечно, есть ограничения: Numba хорошо работает с типичными циклами, арифметикой, NumPy-операциями, но не поддерживает весь спектр Python (например, произвольные встроенные объекты, сложные динамические типы). Для численного кода выигрыши огромны: были случаи, где простой декоратор давал **ускорение в 1000 раз** относительно чистого Python [31]! Например, расчет расстояний между парой тысяч 3D-точек: на Python ~12 секунд, с Numba – 15 миллисекунд [32] [33]. Это **три порядка** разницы за счёт JIT-компиляции. На практике, Numba часто обеспечивает 5x-100x ускорение в численных задачах.

Если ваш проект – научные вычисления, обработка сигналов, финансовые расчёты и т.п., где основная нагрузка – циклы и математика, **Numba – первый кандидат**. Он компилирует функции в машинный код через LLVM. Кстати, Numba можно применять и к функциям, работающим с массивами NumPy – тогда ускорение особенное хорошее, т.к. Numba может убрать сам overhead Python при работе с массивами.

- **Cython:** Это надмножество Python, позволяющее компилировать код в C-расширение. Вы можете постепенно превратить «горячую» функцию в Cython-версию, добавляя статические типы для переменных, и добиться очень высокой скорости – вплоть до скорости Си (если все тяжелые операции переведены на типы C). Например, тот же суммирование квадратов можно написать в Cython как:

```
cpdef long sum_of_squares_c(int n):
    cdef long total = 0
    cdef int i
    for i in range(n):
        total += i * i
    return total
```

Скомпилировав этот код, получаем C-расширение. Cython требует некоторого порога вхождения: нужно установить компилятор, прописать setup.py или использовать `%%cython`-магию в Jupyter. Зато Cython позволяет **постепенно оптимизировать**: сначала вы можете просто скомпилировать существующую Python-функцию – выигрыш будет небольшой, может 1.2-2x из-за устранения байткод-оверхеда. Затем добавлять аннотации типов – скорость будет расти по мере устранения динамических операций. Например, в одном сравнении Numba и Cython для вложенных циклов: Cython-версия при тщательной оптимизации оказалась ~30% быстрее Numba [34]. То есть при должном усилии Cython даёт максимальный контроль и производительность. Также с Cython можно вызывать напрямую C-функции, работать с указателями, чтобы обойтись без Python-объектов внутри критичных частей.

Cython активно используется для ускорения библиотек: многие части Pandas, Scikit-Learn написаны на Cython. Минус – требуется поддерживать сгенерированный бинарный модуль (т.е. компиляция при установке), и код с явными типами уже менее «чистый» Python. Однако для

ключевых участков (например, парсер, внутренний цикл алгоритма) это оправдано. **Выигрыши**: типичный ускорение после аннотации типов 10x-50x и более (зависит от природы задачи). Без аннотации – незначительно. Поэтому, если решите применять Cython, стоит пойти до конца: выписать типы, отключить проверки границ списков/массивов (`boundscheck=False`), отключить negative-index wrap (`wraparound=False`) и т.п., чтобы убрать весь питоновский оверхед. Тогда производительность может выйти на уровень Си.

- **PyPy:** альтернативная реализация Python с JIT-компиляцией. В отличие от CPython, PyPy во время работы **оптимизирует часто выполняемые участки кода**. Подключение – минимальное: достаточно установить PyPy и запустить ваш скрипт под ним (обычно `pypy3 script.py`). Для чисто алгоритмического кода PyPy может дать **ускорение в 4-7 раз в среднем** [35] по сравнению с CPython, а иногда и больше. В одном тесте (pystone) PyPy показал в ~3 раза больше операций, чем CPython, и ~в 3 раза больше, чем даже скомпилированный Nuitka-бинарник [36] [35]. Официально заявляется ускорение PyPy ~7.6x на среднем наборе бенчмарков [35]. Преимущество PyPy – **не требуются никакие изменения кода**. Вы просто используете другой интерпретатор.

Однако есть нюансы: PyPy особенно эффективен на долгоживущих процессах с интенсивными вычислениями на уровне Python (например, многократные вызовы функций, работа с чисто питоновскими структурами). Если ваш код активно использует библиотеки с C-расширениями (NumPy, Pandas, SciPy и т.д.), выигрыш будет меньше, а иногда PyPy даже может быть медленнее при вызове C-расширений (так как они не JIT-оптимизируются и могут иметь несовместимости). Например, для I/O-ограниченных приложений (ожидание сетевых операций, диска) PyPy мало что даст – там не CPU-bound работа. Но для таких задач, как парсинг большого JSON, обработка текста, алгоритмы, не использующие много внешних C библиотек – PyPy может себя проявить отлично.

На 2025 год PyPy поддерживает Python 3.9 (экспериментально 3.10). Если все зависимости вашего проекта совместимы, **попробуйте PyPy**: *«Если ваш код работает под PyPy – просто используйте его и радуйтесь»* [37]. Пример: есть расчёт с тяжелым использованием классов и рекурсии – PyPy оптимизирует его гораздо лучше CPython, и без усилий. Но, например, вычисление NumPy-матрицы под PyPy не быстрее CPython, потому что узкое место – внутри C, а PyPy не ускоряет C код.

- **Сравнение и выбор инструмента:** И Numba, и Cython, и PyPy – решают схожую проблему, но по-разному:
- *Numba* – минимальный барьер входа для численных расчётов. Если у вас функция с циклами, матрицами, сложной арифметикой – декоратор `@njit` может мгновенно дать 10x-100x ускорение. Минусы: не поддерживает сложный Python (например, генераторы, много типов), требует установки (но pure Python fallback возможен).
- *Cython* – требует писать (или генерировать) C-подобный код, но работает везде (не только для чисел) и даёт максимальный контроль. Хорош при создании библиотек, модулей, которые потом будут переиспользоваться. Для разовой оптимизации скрипта – возможно избыточен, если можно применить Numba или PyPy.
- *PyPy* – прост в применении, ускоряет *большую часть* Python-кода автоматически. Хорош для долгих сервисов, серверов, где JIT успеет разогнаться. Не требует компиляции кода (JIT происходит в рантайме). Минус – не все библиотеки совместимы, и использовать PyPy в некоторых средах (например, AWS Lambda, GCP Functions) может быть сложно. Также PyPy потребляет больше памяти обычно.

Пример кейса: Вы написали сложный алгоритм динамического программирования на Python, работает медленно. Пробуете PyPy – он в 3 раза ускоряет. Этого недостаточно. Тогда можно переписать критичный внутренний цикл на Cython или применять Numba – получаете еще порядок-два ускорения. **Комбинировать** тоже можно: например, есть успешные случаи использования Numba внутри PyPy, но это экзотика. Обычно выбирают что-то одно.

- **Другие компиляторы:** Существуют и другие проекты – *Nuitka, Pythran, MyPyC, Transcrypt* и т.д. Nuitka компилирует весь Python-скрипт в C++ код и затем в exe (см. раздел 10), давая ~2x ускорение на уровне всего приложения [38], но не обеспечивая кратный рост производительности критичных участков (GIL остаётся). MyPyC компилирует типизированный Python (аннотации `typing`) в C для ускорения. Pythran – компилятор для векторных вычислений (особенно интеграция с NumPy). Для широкого применения Numba/Cython/PyPy остаются топ-инструментами.

Практический совет: начинайте оптимизацию с самого простого – попробуйте запустить под PyPy. Если не дало нужного эффекта или несовместимо – профилируйте и подумайте о переносе самых тяжёлых функций на Numba или Cython. Например, *«Numba ускоряет Python-код почти до C-скорости одной строкой, и лишь немного медленнее тщательно оптимизированного Cython-кода»* [31] – это впечатляет и экономит время. Но Numba хорош не для всего – например, он не умеет в JIT многопоточный код или работа с Python-списками (только с типизированными List из `numba.typed`). Тогда ваш выбор – Cython или переписать на C.

В итоге, комбинация: использовать **CPython + оптимизации кода**, затем **PyPy** или **JIT-компиляция отдельных функций**, позволяет достичь отличных результатов, сохраняя большую часть кода на Python. Не забывайте, что Python развивается – в версии 3.11, например, появились существенные внутренние оптимизации, ускорившие интерпретатор на ~25-60% на ряде задач [39]. А на горизонте возможно даже появление CPython без GIL, что изменит расклад с многопоточностью. Но уже сейчас есть инструменты, позволяющие лечь близко к производительности компилируемых языков, не покидая уютного мира Python.

6. Параллелизм и асинхронность: потоки, процессы, `asyncio`

Современные вычислительные задачи часто можно ускорить, выполняя их **параллельно** или **конкурентно**. В Python есть несколько моделей параллелизма:

- **Многопоточность (Threading):** Модуль `threading` позволяет запускать несколько потоков (threads) внутри одного процесса. Однако в CPython есть Глобальная блокировка интерпретатора (**GIL**), которая **не позволяет одновременно выполняться более чем одному потоку Python-байткода** в один момент времени [40] [41]. Это значит, что потоки **не ускоряют CPU-bound задачи** – они будут выполняться по очереди (хотя могут чередоваться). Но для **I/O-bound** задач (сетевые запросы, ожидание файловой системы, спящее время) GIL автоматически освобождается на время операций ввода-вывода, и другие потоки могут выполняться. Поэтому потоки пригодны, когда вы ждёте внешних ресурсов и хотите в это время заняться чем-то ещё. Например: загрузка данных из интернета – можно запустить 10 потоков, каждый делает запрос, и пока 9 ждут ответа, 1 может обрабатывать свой ответ. В итоге общее время ~равно времени самого медленного запроса, а не сумме.

Когда использовать потоки: - Для задач, ограниченных ожиданием: сетевые запросы (HTTP, сокеты), работа с БД (если драйвер наружу GIL), ожидание ввода пользователя, Sleep и т.п. - Для параллельного обслуживания нескольких клиентов (каждый в своём потоке) – классический подход в веб-серверах до появления async. - Когда нужно упростить структуру программы (например, отделить поток обработки GUI от фонового расчёта).

Не использовать потоки для: - Тяжёлых вычислений на Python (число Pi в миллиард знаков, обработка огромного списка в цикле) – GIL не даст выигрыш, здесь помогут процессы или переход на C/Numba. - Задач, где нужен истинный параллелизм на нескольких ядрах.

Пример, когда потоки полезны: у вас есть функция, которая парсит 1000 URL-адресов с помощью библиотеки `requests` (которая блокирующая). Если делать последовательно, суммарное время ~1000 * (среднее время отклика). Если же создать пул потоков и параллельно запрашивать десятки URL, общее время ~1000/10 * (среднее время) – почти в 10 раз быстрее (с учётом накладных). Это типичный I/O-bound сценарий. Важно отметить, что **потоки могут взаимодействовать через общую память** (общие переменные, структуры) – это плюс по сравнению с процессами, но накладывает ответственность: нужна синхронизация (lock, Queue) для доступа из разных потоков, иначе можно получить race conditions.

Библиотеки: Для удобства можно использовать `concurrent.futures.ThreadPoolExecutor`, чтобы запускать функции в пуле потоков, не работая напрямую с классом Thread. Этот подход позволяет легко распараллелить работу по списку задач (например, `executor.map(fetch_url, urls)` запустит параллельно fetch_url для каждого URL).

- **Многопроцессность (Multiprocessing):** Модуль `multiprocessing` запускает несколько процессов (внешне похоже на потоки, но это **отдельные процессы с собственной памятью**). Здесь **каждый процесс имеет свой интерпретатор и GIL**, поэтому процессы могут реально выполняться **параллельно на разных ядрах**. Это подходит для **CPU-bound** задач, разгружающих одно ядро на несколько. Например, у вас 8-ядерный CPU – 8 процессов Python могут в теории выполнить задачу ~в 8 раз быстрее, чем один (на практике неидеально из-за накладных). `multiprocessing` предоставляет интерфейс, схожий с threading: `Process` классы, пулы (`multiprocessing.Pool`), очереди и конвейеры для обмена данными.

Когда использовать процессы: - Вычислительно интенсивные задачи, где данные можно разделить между процессами. Например, обрабатывать элементы массива по частям, тренировать несколько моделей машинного обучения параллельно, рендеринг частей изображения и т.п. - Когда нужно обойти GIL, но остаться на Python (не переписывая на C). Процессы – универсальный способ параллелить всё, что угодно, ценой дополнительной памяти.

Недостатки процессов: - **Межпроцессное взаимодействие медленнее**, чем в потоках. Общей памяти нет (по умолчанию), приходиться использовать сериализацию (pickle) для передачи данных через `multiprocessing.Queue` или Pipe. Это добавляет оверхед. Тоже самое с `multiprocessing.Pool`: аргументы и результаты функций сериализуются. Для больших объектов это накладно. - Создание процесса существенно тяжелее потока (инициализируется новый интерпретатор, загружаются модули). Хотя модуль поддерживает `fork` на Unix, что быстрее, но всё равно overhead больше, чем у потока.

Тем не менее, **для CPU-bound** ничего другого нет (пока есть GIL). Например, у вас задача – обработать 1000 изображений по сложному алгоритму. На чистом Python в одном потоке это займёт час, а на 4 процессах – около 15 минут, почти линейное ускорение (если диск успевает

снабжать их данными). Пример: рендеринг фрактала – можно разделить область на 8 частей и посчитать каждым процессом свою часть, потом собрать. Или генерация отчётов: запустить 4 процесса, каждый формирует поднабор.

Инструменты: `multiprocessing.Pool` и `concurrent.futures.ProcessPoolExecutor` позволяют писать параллельный код высокоуровнево. Например, `with ProcessPoolExecutor() as ex: ex.map(f, data, chunksize=10)` – выполнит f(x) для элементов data, распределяя между процессами. Важно: функция f и объекты data должны быть пиклируемыми (serializable), иначе их нельзя передать процессу.

Пример:

```
import math
from concurrent.futures import ProcessPoolExecutor
nums = [10**7, 10**7+1, 10**7+2, 10**7+3]
def f(n):
    # некий CPU-bound расчёт
    s = 0
    for i in range(n): s += math.sqrt(i)
    return s

# Последовательно
%time results = list(map(f, nums))
# Параллельно в 4 процессах
%time
with ProcessPoolExecutor(max_workers=4) as exe:
    results2 = list(exe.map(f, nums))
```

На последовательном варианте Python использует 1 ядро и, скажем, тратит 8 секунд на 4 вызова. На параллельном – все 4 ядра, и завершается за ~2.2 секунды (с учётом накладных) – ускорение ~3.6x близко к числу ядер. Если бы мы попробовали то же с ThreadPoolExecutor (т.е. потоками) – время осталось бы ~8 секунд из-за GIL (фактически потоки выполнились бы по очереди).

- **Asyncio (асинхронность):** Это третий подход – **асинхронное программирование**, появившееся в Python 3. Он не использует несколько системных потоков, а выполняет задачи **кооперативно** в одном потоке, переключаясь между ними. Асинхронность особенно эффективна для **большого числа параллельных I/O задач** (сетевые запросы, взаимодействие с базой, ожидание таймеров). Программист пишет функции с `async def` и использует `await` для операций, которые могут занять время (например, запрос к серверу). **Когда такая операция ожидает, управление возвращается в цикл событий**, который может запустить другую задачу. Таким образом, один поток может обслуживать тысячи соединений эффективно, не создавая тысячи потоков ОС.

Плюсы `asyncio`**:** - Малые накладные на переключение между задачами (не нужен контекст свитч ОС, всё в одном потоке). - Отсутствие GIL-проблем – так как всё и так в одном потоке, GIL не мешает (но и параллельности по CPU нет, это concurrency, а не parallelism). - Отлично масштабируется по числу ожиданий: например, можно держать 10k открытых сокетов и ждать их – потоки бы на таком количестве исчерпали ресурсы системы, а в asyncio это нормально.

Минусы: - Сложнее писать и отлаживать, непривычный синтаксис (хотя уже стал стандартным). - Код, выполняющийся внутри `async def` **не должен блокировать** поток надолго (иначе он застопорит весь цикл). То есть CPU-bound вычисления не стоит делать внутри asyncio задач – для них надо выносить в ThreadPool/ProcessPool (asyncio предоставляет `loop.run_in_executor`). - Библиотеки должны поддерживать async (но сейчас их много: `aiohttp` для HTTP, `asyncpg` для PostgreSQL, и т.д.).

Когда использовать `asyncio`**:** - В высокопроизводительных сетевых серверах (протоколы, веб-серверы). Многие современные веб-фреймворки (FastAPI, aiohttp) построены на async и могут выдерживать большой tps с малой задержкой. - В клиентах, делающих много одновременных запросов. Например, скачивание с десятков API одновременно. - Когда нужно эффективно ждать сразу множество событий (таймауты, сокеты, оконные события).

Пример (aiohttp): Одно из сравнений HTTP-клиентов показало, что *«AIOHTTP примерно в 10 раз быстрее requests при высоком числе одновременных запросов»* [42]. Requests – синхронная библиотека: пока один запрос не выполнится, другой не начнётся (если не использовать потоки). AIOHTTP – асинхронная: позволяет выдать сотни запросов параллельно и обрабатывать ответы по мере прихода. В итоге достигается Throughput, невозможный для single-thread sync подхода. В реальном тесте, AIOHTTP показал ~240 запросов/с против ~20 запросов/с у requests (при тесте 1000 запросов) – то самое 10-кратное превосходство [42]. И ~1.5-кратное превосходство над HTTPX (другой современный клиент) благодаря более низким накладным.

Сетевые фреймворки: Для асинхронности часто используют готовые решения: например, для веб-сервера – `asyncio.start_server`, высокоуровневые библиотеки типа `aiohttp` (веб-сервер и клиент), `quart` (async аналог Flask), `Sanic`, для высокоуровневого HTTP-клиента – `aiohttp` или `httpx`.

Threading vs Asyncio: Они оба подходят для I/O-bound задач, но: - Если у вас относительно немного параллельных операций (десятки), и код уже написан синхронно – проще использовать threads (можно прямо через ThreadPoolExecutor). - Если параллельных операций очень много (сотни, тысячи) – threads начнут «тяжелеть» (память, расписание ОС). Asyncio лучше масштабируется и предоставляет больше контроля. - Asyncio может дать лучше **латентность** при большом числе задач, потому что события обрабатываются своевременно, а не ждет пока ОС проснёт спящий поток. - Разработка с asyncio требует асинхронных версий библиотек (DB драйверов, HTTP клиентов). Если их нет, придётся использовать потоки под капотом (что тоже допустимо).

Отличный пример: представьте чат-сервер на 10k клиентов. Реализуя его на потоках, вам нужно 10000 потоков – это почти нереально, потоки будут больше переключаться, чем работать. На asyncio – один поток, 10000 Task'ов, которые большую часть времени ждут (пока пользователи что-то напишут). Результат: минимальные затраты CPU и памяти.

- **GIL и настоящий параллелизм:** Из-за GIL, **только процессы** дают настоящий параллелизм на нескольких ядрах. Однако, стоит упомянуть, что некоторые вычисления, хоть и CPU-bound, могут параллелиться и в потоках, если они выпускают GIL. Например, многие операции NumPy освобождают GIL (так как выполняются в C). Поэтому можно, например, параллельно в потоках выполнять матричные умножения NumPy – это сработает (каждый поток внутри NumPy займёт своё ядро). Но такой подход редко нужен, так как NumPy сам может использовать многопоточность внутри (через OpenBLAS, MKL).

Выбор модели: - Для **CPU-bound** задач: `multiprocessing` (или сторонние, вроде Joblib, Ray, Dask) – либо **распараллелить на уровне C/NumPy**. - Для **I/O-bound с небольшим числом параллельных блокировок (до сотен)**: проще использовать **многопоточность**. - Для **I/O-bound с очень большим числом открытых соединений/задач**: лучше **asyncio**. - Комбинации: например, можно использовать `asyncio` для сетевого сервера + `ProcessPoolExecutor` для отправки тяжелых CPU задач в отдельные процессы, не блокируя цикл.

Пример практики: Веб-приложение, обрабатывающее запросы: можно запустить несколько процессов (workers) для использования всех ядер, а внутри каждого использовать `asyncio` или потоки для взаимодействия с БД и внешними сервисами. В итоге – масштабирование и по CPU, и по числу параллельных I/O.

Важно: Параллелизм усложняет разработку. Нужно учитывать **потокобезопасность** (в потоках), т.е. защищать общие данные локами или использовать thread-safe структуры (очереди). В процессах – научиться быстро сериализовать данные (может, использовать общую память `multiprocessing.Value/Array` или multiprocessing.Manager, если нужно часто обмениваться). Asyncio требует особого внимания к тому, где ставить `await` (чтобы не блокировать цикл).

Но результат может сильно повысить производительность приложения. Как отмечено: *«многопоточность подходит для I/O-bound активностей»* [40], *«multiprocessing – для CPU-bound»* [43]. Правильное сочетание дает масштабируемость по нагрузке.

7. Работа с вводом/выводом и сетями: буферизация, Async, aiohttp

Операции ввода-вывода (I/O) – чтение/запись файлов, работа с сетью – часто становятся узким местом, особенно в backend-сценариях. Оптимизация I/O отличается от оптимизации CPU: тут важнее **уменьшить количество операций** и эффективно использовать системы ввода-вывода, чем ускорить вычисления. Рассмотрим несколько техник:

- **Буферизация при работе с файлами:** Python по умолчанию выполняет **буферизированный I/O**. Когда вы открываете файл с `open('data.txt', 'r')`, используется буфер (обычно 8KB) для чтения. Это значит, что вызов `file.readline()` не обращается к диску за каждым байтом, а читает сразу блок, а затем раздает строки из буфера. **Общая рекомендация:** читайте файл **крупными порциями**, если вам нужен не весь файл сразу. Например, для двоичного файла: `chunk = file.read(1024*1024)` будет читать по 1MB, что эффективнее мелких чтений. Если надо прочесть весь файл, лучше делать это за минимальное число операций: либо одним `file.read()` (но память должна позволять), либо итерацией по файлу (которая под капотом читает буферизированно). **Не читайте файл посимвольно или по нескольку байт без острой необходимости** – системные вызовы стоят дорого. Если обрабатываете текст построчно, можно использовать конструкцию `for line in file:` – она использует буферизацию и очень эффективна.

Можно настроить размер буфера: `open(filename, 'r', buffering=65536)` – увеличит буфер до 64KB, что может быть выгодно при работе с очень большими файлами, уменьшая число системных вызовов. Но чаще стандартного хватает.

Запись в файл: Здесь тоже буферизация помогает. Вызов `file.write(data)` пишет данные в буфер и может не сразу сбросить на диск (flush). Частые маленькие записи могут сильно

замедлить (каждый flush – системный вызов). Решение: по возможности **накапливать данные в памяти, а потом записывать большим блоком**. Например, формировать строку из нескольких частей (через join) и одним `file.write` вместо многократных. Или использовать `io.BufferedWriter` явно с большим буфером. Если нужно писать построчно, проверьте, не включен ли `line buffering` (он по умолчанию включен при выводе на терминал, но для файлов обычно нет). Не вызывайте `flush()` чаще, чем нужно.

Пример: запись 100k строк: вариант 1 – по строке с `f.write(line)` в цикле (100k системных вызовов на запись). Вариант 2 – через `f.writelines(list_of_lines)` – одна операция передаст список системе (он может быть записан почти за один вызов). Разница может быть значительной.

- **Использование памяти для I/O:** Если данные помещаются в память, иногда лучше прочитать **целиком** и работать в памяти – это уменьшит число файловых операций. Например, поиск по файлу: можно прочитать весь файл в строку (или `mmap`, об этом ниже) и искать по строке – Python оптимально использует C-функции для поиска подстроки, что может быть быстрее многократных чтений строчка-за-строчкой. Конечно, надо оценивать размер – для гигантских файлов это не всегда вариант.

- **Memory-mapped files (`mmap`):** Модуль `mmap` (и `numpy.memmap`) позволяет отобразить файл в память. Это значит, что вы можете обращаться к файлу, как к обычному байтовому массиву, а ОС подгрузит нужные части по требованию. Это полезно для **очень больших файлов**, с которыми нужно работать фрагментами, не загружая полностью. Например, у вас бинарный файл 10 ГБ, а память 2 ГБ – с `mmap` можно работать с ним почти как с массивом байт, считывая только активные страницы. Это и удобно (не надо вручную буферизовать), и может быть быстрее (ОС проведет оптимизацию чтения). В NumPy memmap можно даже выполнять вычисления, не загружая всё сразу. **Документация говорит:** *«мемmап используется для доступа к небольшим сегментам больших файлов на диске без чтения всего файла в память»* [44]. Например, можно быстро перемещаться по большому файлу по смещению, не открывая/закрывая/чтя блоки вручную.

Однако, `mmap` не всегда быстрее обычного чтения блоками – он полезнее для случайного доступа, а для последовательного чтения может даже чуть уступать, но главным плюсом остается экономия памяти.

- **Асинхронный ввод-вывод:** Если ваша программа должна делать много параллельного I/O, рассмотрите **асинхронные решения**. Например, если нужно одновременно читать из десятков файлов (скажем, логов) и обрабатывать – можно использовать `asyncio` с `aiofiles` (асинхронное чтение файлов) или вынести чтения в `ThreadPoolExecutor`, чтобы не ждать их по очереди. Асинхронность особенно проявляет себя в сетевом I/O (см. ниже), но и для диска бывает (правда, диск – обычно один, и сильно параллельно читать с него смысла мало, но если это SSD – некоторый параллелизм возможен).

- **Оптимизация сетевых операций:**

- **Сокеты:** при работе с низкоуровневыми сокетами (`socket` модуль) старайтесь отправлять/получать данные **крупными буферами**, а не по байту. Используйте `socket.recv(BUFSIZE)` с разумным BUFSIZE (например, 4096 или 16384 байт), вместо чтения по 1 байту. Аналогично с `send` – лучше накопить пакет и отправить разом. TCP/IP и так буферизует, но лишние вызовы системных функций все равно замедляют.

- **HTTP-запросы:** как уже обсуждали, для большого числа запросов – асинхронные библиотеки (aiohttp, httpx) могут дать существенный прирост [45]. Также при использовании sync-библиотек (requests) можно повысить эффективность за счет **пулов соединений**. Например, `requests.Session` держит соединение открытым для повторных запросов к тому же хосту (HTTP keep-alive) – это экономит время на установку TCP-соединения. Используйте Session, если делаете много запросов к одному сервису.
- **Upload/Download крупных файлов:** применяйте **потоковую передачу** (streaming). В requests, например, `stream=True` позволяет читать ответ по кусочкам, не загружая всё в память. Это не столько про скорость, сколько про память, но может влиять на общую производительность, избегая лишнего копирования.
- **aiohttp**: при написании серверного кода на aiohttp/fastapi, убедитесь, что у вас настроен эффективный цикл событий. Рекомендуется использовать **uvloop** (альтернативный цикл событий на основе libuv) – он значительно ускоряет asyncio-серверы. Простой `pip install uvloop` и `asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())` – и ваш async-сервер работает быстрее.

- **Нагрузочное тестирование:** производительность сетевых приложений сильно зависит от настроек, например, размеров сокетных буферов (TCP window), настроек ОС (лимиты дескрипторов, TIME_WAIT). Для действительно высоконагруженных систем важно это учесть, но это уже выходит за рамки языка.

- **Протоколы и форматы:** Косвенный способ ускорить I/O – использовать более эффективные форматы данных. Например, чтение и разбор JSON крупного – медленная операция; возможно, стоит перейти на двоичный формат (MessagePack, Protocol Buffers) – это сокращает объем данных и ускоряет парсинг (особенно Protobuf с его сгенерированным кодом). Это архитектурный выбор, но его влияние на перфоманс огромно.

- **Параллельный I/O:** Если у вас несколько дисков или вы работаете в сети, можно **читать/писать параллельно**. Например, качать два файла сразу (если сеть позволяет) – мы уже обсуждали, async или threads могут помочь. С диском сложнее – обычно один диск не выиграет от параллельного чтения, а вот RAID или разделенные диски – могут.

Пример буферизации: Вывод логов: Если у вас сервис логирует каждое событие, и вы делаете `print()` или запись в файл для каждой строчки, производительность может просесть (особенно `print`, который по умолчанию flush-ит на терминал каждый вывод). Решение: собрать несколько записей и писать пачками, или отключить автоперевод строки, или увеличить буфер. Даже просто запуск Python с `-u` (unbuffered) vs без – влияет: обычно лучше оставить стандартную буферизацию для файлов, чтобы не дергать ОС на каждую строчку.

Асинхронный пример: Suppose we have to fetch data from 50 APIs. Synchronously (requests) это может занять, скажем, 50 секунд (каждый по 1с). С потоками (10 потоков) – можно снизить до ~5-6 секунд, но с 50 потоками уже начнет сказываться overhead. С `asyncio` + `aiohttp` – эти 50 запросов могут быть выполнены за ~1-2 секунды, т.к. все отправятся почти одновременно и как только данные начинают приходить – они обрабатываются. AIOHTTP при высокой конкуренции показывает выдающуюся производительность [42], ведь он держит множество открытых соединений в одном цикле событий.

Вывод: оптимизируя I/O, старайтесь: - **Сократить количество операций ввода-вывода** (буферизация, объединение, более крупные запросы). - **Выполнять I/O параллельно**, если общая задержка велика (параллельные загрузки, одновременные обращения). - **Использовать**

асинхронность для масштабирования по числу одновременных I/O-операций. - **Выбирать эффективные форматы** передачи данных (по возможности). - **Кэшировать результаты** I/O, если одно и то же читается часто (см. кеширование). - **Следить за настройками системы**, влияющими на I/O (но это больше DevOps, хотя знать, что `fsync()` лишний раз дергать дорого, тоже полезно).

В бэкенде зачастую именно правильное управление вводом-выводом отличает быстрый сервис от медленного. Код может тратить 0.001с на вычисление, но ждать 0.1с ответа от базы; тогда весь фокус оптимизации смещается на асинхронность или параллелизм, а не на микротюнинг кода.

8. Кеширование и мемоизация

Кеширование – мощный прием оптимизации, при котором результаты тяжёлых вычислений или запросов сохраняются, чтобы использовать их повторно без повторного исполнения. Если функция часто вызывается с одними и теми же аргументами, проще запомнить ответ при первом вызове и возвращать его мгновенно при следующих. Рассмотрим инструменты и техники кеширования:

- **functools.lru_cache:** Декоратор из стандартной библиотеки, реализующий LRU-кеш (кеш с вытеснением least recently used при переполнении). Использование крайне простое:

```
from functools import lru_cache

@lru_cache(maxsize=128)
def fib(n):
    print("Вычисляю fib", n)
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Здесь `fib` при каждом новом значении `n` вычислит результат (рекурсивно, с принтом), но повторно для тех же `n` сразу отдаст из кеша. LRU-кеш имеет фиксированный `maxsize` (последние 128 результатов, в примере), но можно указать None для безразмерного кеша (осторожно с памятью!). lru_cache делает функцию **непрозрачно** – т.е. для пользователя это все та же функция `fib(n)`, но внутри результаты хранятся в словаре. Декоратор потокобезопасен (использует лок для кеша), поэтому работает и в многопоточной среде.

Эффект: кеширование способно **ускорить повторные вызовы функции в тысячи раз**, если она изначально дорогая. Например, вычисление 35-го числа Фибоначчи наивно рекурсивно (~30 млн операций) занимает секунды, а с кешированием – почти мгновенно, так как рекурсия свернётся, повторно не вычисляя подзадачи. В общем случае, выигрыш зависит от того, как часто повторяются входные данные.

Пример реальный: в веб-приложении есть страница отчетов, формирующаяся секундами. Если данные меняются раз в час, нет смысла генерировать её каждый раз – можно кешировать результат генерации на 10 минут. Тогда первые пользователи подождут, но следующие получат ответ почти мгновенно (из памяти или быстрой БД). Cameron MacLeod приводит пример Flask-приложения, где без кеша страница генерировалась ~171 мс, а с применением `@lru_cache` –

~13.7 мс [46] [47], т.е. улучшение более чем в **12 раз** просто за счёт предотвращения повторного вычисления шаблона для тех же данных.

- **Ручное мемоизация словарём:** lru_cache – удобный способ для чистых функций. Если условия не позволяют (например, вы хотите контролировать жизненный цикл кеша или функция непредсказуемое количество аргументов), можно реализовать кеш самостоятельно: просто хранить результаты в глобальном словаре. Например:

```
cache = {}
def heavy_computation(x, y):
    key = (x, y)
    if key in cache:
        return cache[key]
    # ... дорогие вычисления ...
    result = ...
    cache[key] = result
    return result
```

Это даёт полный контроль – можно очищать кеш по условиям, ограничивать размер и т.д. Однако это всё уже делает lru_cache, потому ручной подход нужен редко, в специфичных ситуациях.

- **Внешние системы кеширования:** В веб-разработке часто применяются внешние кеши – например, **Redis** или **Memcached**. Они позволяют сохранять кэшированные данные во внешней памяти (обычно RAM-сервер), чтобы делиться между разными процессами и даже серверами. Например, результаты тяжёлых запросов к БД можно складывать в Redis: при повторном запросе сервис проверяет кеш – если есть, возвращает мгновенно; если нет, идёт в БД. Это снижает нагрузку на базу и ускоряет отклик. Redis – очень быстрый (миллионы ops/sec) – фактически предоставляет константное время доступа к кэшу по ключу через сеть. По опыту, Redis позволяет снизить время ответов на порядки в ситуациях «одно и то же спрашивают много раз». Как сказано, *«Redis позволяет снизить нагрузку на основную базу данных, одновременно ускоряя чтение данных»* [48]. Ключ – не забывать про **актуальность**: кеш может устаревать, поэтому практикуют **TTL (time-to-live)** – время жизни записи, или **инвалидируют** кеш при изменениях (например, удалять ключ из кеша, когда в БД обновлены данные).

Пример: страница каталога товаров делает сложные агрегации. Без кеша – 2 секунды, с кешем (в Redis, ключ зависит от параметров фильтра) – 50 миллисекунд. Это реальный выигрыш, который чувствуют пользователи. Или кэширование токенов авторизации, чтобы не расшифровывать их каждый раз; кеширование геокодирования адресов (Google API дорогой: лучше один раз перевести адрес в координаты и хранить).

- **LRU vs LFU vs TTL:** Алгоритмы кеша бывают разными (Least Recently Used, Least Frequently Used и т.д.). `functools.lru_cache` – LRU. В Redis по умолчанию можно настроить разные политики. Важно подобрать maxsize или срок жизни так, чтобы кеш не разрастался бесконечно, но и удерживал часто используемые данные. LRU хорошо работает, когда недавние вычисления вероятно пригодятся снова (часто так и есть). LFU – когда интереснее частота обращений.

- **Кеширование и память:** Кеш – это **обмен памяти на время**. Храним результаты, чтобы не тратить время на перерасчёт. Нужно следить, чтобы кеш не съел всю память (особенно

безразмерный). lru_cache с maxsize решает, Redis можно настроить на определенный объем (eviction policy). Если данных очень много и они объемны, можно кешировать на диск (но тогда теряется часть выгоды, хотя все равно может быть быстрее чем пересчет).

- **Мемоизация для рекурсивных алгоритмов:** Классическое применение – фибоначчи, как выше, или сложные рекуррентные формулы (например, динамическое программирование, где вместо таблички можно мемоизировать рекурсивную функцию). Это превращает экспоненциальное время в линейное. С лру_cache реализовать DP очень просто: вы пишете рекурсию по определению, и декорируете – всё, работает как DP с мемоизацией.

- **Кеширование I/O:** Кешируют не только вычисления, но и результаты чтения. Например, если из файла или HTTP часто читаются одни и те же данные, можно держать их в памяти (в кеше) вместо повторного чтения. Многие ОС кешируют дисковые данные на уровне системного Page Cache, но на прикладном уровне тоже можно – например, парсинг конфиг-файла: читать его каждый раз – медленно, лучше один раз загрузить и хранить.

- **Инструменты для кеширования:**

- `functools.lru_cache` – лучший старт для мемоизации функций (краткоживущих объектов).
- `cachetools` (внешняя библиотека) – предлагает разные политики кеша (LFUCache, TTLCache), удобно если нужны TTL или ограничение по времени.
- `django.core.cache` / Flask-Caching – фреймворки предлагают интеграцию с Redis/ memcached.
- `joblib.Memory` – для ML задач, кеширует результаты функций на диск (используется в scikit-learn для кеширования промежуточных шагов).
- `diskcache` – еще одна библиотека, которая легко кеширует данные на диск с алгоритмом, близким к LRU, хороша когда объем кеша превышает RAM.

Пример кода с lru_cache:

```
from functools import lru_cache
import time

@lru_cache(maxsize=32)
def slow_func(x):
    # имитируем долгую операцию
    time.sleep(2)
    return x*10

print(slow_func(2))  # Первый вызов: медленно (~2 сек)
print(slow_func(2))  # Повторный вызов: сразу, кэшированное значение
print(slow_func.cache_info())  # покажет статистику: hits, misses, size,
maxsize
```

Вывод:

```
20
20
CacheInfo(hits=1, misses=1, maxsize=32, currsize=1)
```

Показывает, что второй вызов был hit из кеша. Ускорение – практически бесконечное (0 времени против 2 сек).

Кеширование в многопоточных/многопроцессных приложениях: lru_cache сохранит кеш на процесс. В многопоточных это ок (разделяется между потоками, с локом). В multiprocess – каждый процесс будет иметь свой отдельный кеш, что не всегда желаемо. Тогда имеет смысл вынести кеш в общий внешний (Redis) или запускать процессы-воркеры без локального кеша, но перед ними держать прокси, читающую из единого кеша.

Не забывайте про обновление кеша: кеширование вводит такую проблему, как **устаревание данных**. Нужно продумать, как валидировать кеш: либо по времени (TTL), либо по событиям (например, очистить ключ при изменении источника). Иначе пользователи могут получать неактуальные данные. В задачах, где допустима некоторая устарелость (например, кешировать результаты за последний час) – TTL в помощь. В задачах, требующих свежести, можно комбинировать: отдавать из кеша, но в фоне обновлять.

Заключение: **кеширование – простой в реализации способ ускорить повторяющиеся операции**. Как говорится: *«кеширование может дать огромный буст производительности, цена – потребление памяти и сложность обеспечения актуальности данных»*. Во многих системах (веб, распределенные сервисы) без кешей не достичь нужного throughput. Используйте lru_cache для быстрого локального результата или подключайте внешние кеш-системы для больших задач – они **существенно сокращают нагрузку и время отклика** [49].

9. Работа с большими объемами данных: Pandas, NumPy, memory-mapping

Python часто используется для анализа данных, обработки больших таблиц, массивов. Производительность при этом сильно зависит от того, используем ли мы оптимизированные инструменты вроде **NumPy/Pandas** или пытаемся обрабатывать данные чисто на Python. Рассмотрим ключевые моменты:

- **Векторизация с NumPy:** NumPy – библиотека для работы с массивами, реализованная на C. Она позволяет выполнять операции над целыми массивами **без явных Python-циклов**. Например, прибавление двух массивов поэлементно выполнится внутренним высокопроизводительным кодом. **Выигрыш:** на больших массивах NumPy может быть в **несколько сотен раз быстрее** Python-циклов [50]. Конкретный пример: вычисление экспоненты для миллиона чисел – Python-цикл с `math.exp` занял ~646 мс, а NumPy-векторизация `np.exp(arr)` – ~20 мс [50] [51]. Разница ~30 раз. Чем больше массив – тем сильнее эффект (часто говорит о 50-1000x ускорении). Причина – NumPy написан на C и обрабатывает данные в непрерывной памяти, используя SIMD-инструкции, оптимизации и т.д., в то время как Python-цикл каждый раз тратит ресурсы на интерпретацию.

Правило: Если вы обрабатываете большие числовые массивы или матрицы – *используйте NumPy вместо чистого Python*. Например, нужно умножить каждый элемент списка на 2: `[x*2 for x in list]` – ок, но если `list` очень большой (миллионы элементов), лучше

преобразовать его в `numpy.array` и сделать `array * 2`. Особенно выигрывает NumPy на операциях линейной алгебры: матричное умножение, трансформации, суммирование, статистика – всё сделано с учётом производительности.

Память: NumPy также значительно **экономит память** по сравнению с Python-списками. Один элемент float в Python – объект (~24 байта структуры + 8 байт значение и накладные). В NumPy float64 – ровно 8 байт. На миллионах элементов это огромное различие. Меньше память – больше помещается в кэш процессора – быстрее операции.

Batch-обработка: С NumPy следует стремиться формулировать операции над целыми массивами, а не писать циклы. Даже если нужна условная логика, часто можно использовать булевы маски или функции `numpy.where`. Например, отфильтровать массив: вместо Python-списка comprehension `[x for x in arr if cond(x)]`, вы делаете `mask = cond(arr)` (это булев массив) и потом `filtered = arr[mask]` – всё внутри C.

- **Pandas для табличных данных:** Pandas построен на NumPy и предоставляет высокоуровневые структуры `DataFrame` и `Series` для работы с таблицами (аналог DataFrame R или таблиц SQL). Pandas очень удобен, но надо знать, как правильно его использовать для производительности:
- **Использовать векторизированные методы Pandas:** Это те, которые оперируют сразу над колонками. Например, `df['new'] = df['col1'] + df['col2']` – выполнится быстро (под капотом NumPy). А вот `df.apply(custom_func, axis=1)` – для каждой строки вызывает Python-функцию, то есть по сути скрытый цикл на Python, что медленно [52].
- **Избегать DataFrame.iterrows() и** `apply` **по строкам:** Как показали бенчмарки, итерация по DataFrame строка за строкой – очень медленно (по сути, itertuples/iterrows достают Python-объекты). В одном эксперименте проход по DataFrame с 1 млн строк: `iterrows` ~3.66 сек, `apply` ~0.404 сек, а **векторизированное вычисление – 0.0104 сек** [53] [54] – **улучшение в 350 раз**. Это яркий пример: код, который в Pandas выглядит короче, но всё равно вызывает Python-функцию на каждую строку (`apply`), всё ещё далек от потенциала. Лучше найти способ сделать то же средствами Pandas/NumPy без Python-циклов. Если уж нужно сложную логику на каждую строку – возможно, стоит использовать **Cython или Numba** (Pandas умеет apply с `numba` via `df.apply(func, engine='numba')`).
- **Типы данных Pandas:** Чтобы DataFrame работал оптимально, нужно использовать правильные типы. Например, если колонка содержит категории (строки повторяются) – лучше преобразовать её в `category` dtype. Это не только сэкономит память, но и ускорит операции группировки/сравнения на этой колонке. Если у вас много мелких целых – можно явно задать `dtype='int8'` или `int32` вместо стандартного int64, опять же сэкономив память (и снизив нагрузку на кэш).
- **Методы Pandas vs Python:** Pandas предоставляет много методов: группировки (`df.groupby().agg()`), сортировки, слияния, фильтрации (`df.query()`). Они реализованы эффективно. Старайтесь ими пользоваться вместо того, чтобы выгружать данные в Python и там обрабатывать. Например, вам нужно отфильтровать строки по какому-то сложному условию – можно сделать `df[ (df['A']>5) & (df['B'].isin(values)) ]` – это вся фильтрация на C/NumPy уровнях. Если попытаться то же через цикл по df.itertuples() – будет в десятки-сотни раз медленнее.

- **Применение .loc/.iloc правильно:** Если нужно итеративно присваивать – лучше собрать данные в NumPy массив, потом присвоить столбец целиком, чем присваивать по одной ячейке (каждое присваивание – отдельное действие с проверками). В идеале, старайтесь **избегать побайтовой обработки DataFrame в Python** – используйте поблочную.

- **Масштабирование на большие данные (out-of-core):** Pandas и NumPy работают в памяти, поэтому при данных больше RAM возникают проблемы. Пара подходов:

- **Чтение/запись с помощью итераторов:** `pandas.read_csv` имеет параметр `chunksize` – можно читать огромный CSV кусками (например, по 100k строк), обрабатывать каждую порцию и освобождать. Это предотвратит взрыв памяти и может распараллеливаться (см. Dask).
- **Библиотеки для out-of-core: Dask** DataFrame – позволяет работать с DataFrame, превышающим память, распределяя на кластер или на диск (аналог Pandas, но ленивый). **Vaex** – для очень больших наборов данных (размеры порядка гигабайт до терабайт) с использованием memory-mapping и колонко-ориентированного хранения. **Polars** (на Rust) – новый df-движок, очень быстрый и может работать лениво.
- **Memory mapping больших массивов:** Уже упоминали `numpy.memmap` – чтобы работать с массивом, больше памяти, с приемлемой скоростью. Если у вас, скажем, 10GB бинарный массив – `np.memmap` позволит взять его с диска частями.

- **Базы данных/SQL:** Когда данные совсем велики или более структурированы, иногда лучше переложить часть работы на СУБД (SQL запросы, агрегаты) или специализированные хранилища (ClickHouse для аналитики) вместо Python. Это не про Python-оптимизацию, но про общую производительность системы.

- **Parallel computing для NumPy/Pandas:** По умолчанию, операции над NumPy массивом выполняются в одном потоке (за исключением BLAS, который может многопоточно умножать матрицы). Если у вас 8 ядер и вы хотите ускорить, можно использовать:

- **NumExpr**: библиотека, которая разбирает строковые выражения типа `"sqrt(a**2 + b**2)"` и вычисляет их эффективно и многопоточно, часто быстрее NumPy.
- **Parallelizing**: Dask или Joblib могут распараллелить операции, разбив массив/DF на части. В Pandas нет встроенного multi-threading, но Dask DataFrame, modin, swifter – попытки сделать Pandas parallel. Их эффективность зависит от задачи.
- **Vectorized libraries**: Если работа с массивами включает линалг – подключение MKL (Intel Math Kernel) или OpenBLAS (они умеют использовать все ядра для больших матриц).

Резюме: *«Работа с большими данными на Python должна опираться на оптимизированные низкоуровневые библиотеки (C/C++). Python-код должен только оркестрировать эти вызовы, а не перебирать элементы сам.»*

Пример: нужно вычислить сумму элементов двух очень длинных списков. На Python:

```
total = 0
for x, y in zip(list1, list2):
    total += x + y
```

Это крайне медленно для миллионных списков. На NumPy:

```
import numpy as np
a = np.array(list1, dtype=np.float64)
b = np.array(list2, dtype=np.float64)
total = np.sum(a + b)
```

Здесь практически всё происходит на C – создание массивов (O(n), но C и без overhead на объект каждый элемент), сложение (C-loop), суммирование (может использовать SIMD). На практике, первый вариант с 10 млн элементов может быть на порядок-два медленнее второго.

Иллюстрация ускорения Pandas: Рассмотрим задачу: есть DataFrame с миллионом записей о транзакциях, нужно вычислить колонку "reward" по сложному правилу (как в примере [64]). Цикл через iterrows: 3.66s, apply: 0.404s, векторизация: 0.0104s [53] [54]. Разница между наивным и оптимальным подходом – **350x**. Это демонстрирует, насколько важно избегать чистого Python при обработке больших объемов: *код с apply на Pandas в 9 раз быстрее явного цикла, и ещё в 40 раз медленнее, чем векторизированное решение* [55] [56]. То есть, даже вроде бы "оптимизированный" с точки зрения Python метод (apply) всё равно на порядок медленнее возможного – надо идти до конца и задействовать весь потенциал.

- **Memory-mapping (подробнее):** Pandas может читать данные из memory-mapped файла (например, Feather или Parquet формат, частично). Numpy.memmap даёт `ndarray` с ленивой загрузкой. Если, скажем, у вас есть большой бинарный массив на диске (например, выгрузка датчиков), вместо `np.fromfile` (который читает целиком), можно использовать `memmap` – работать будет, как с массивом (вы можете брать срезы, и они подгрузятся по требованию). Python-speed блог отмечал, что memory-mapping удобен, но есть альтернатива – форматы как **Zarr** или **HDF5**, которые разбивают данные на чанки и позволяют загружать фрагментами. Это скорее вопрос выбора инструмента (иногда лучше сразу хранить в HDF5 с компрессией и вытаскивать надо – Pandas умеет `read_hdf` с фильтрами, etc).

Заключение: Для больших данных используйте "heavy lifting" библиотеки. Наглядно: *«операции NumPy могут выполняться на порядки быстрее аналогичных на чистом Python, за счёт использования быстрого низкоуровневого кода и эффективного доступа к памяти»* [57]. Pandas при правильном подходе даёт удобство не в ущерб скорости (но требует знаний, где подстерегают ловушки вроде apply).

И последнее: **следите за потреблением памяти**. В больших задачах упереться можно не только в CPU, но и в память (swap убьет производительность). Поэтому оптимизация памяти (правильные типы, удаление ненужных объектов, del DataFrame после использования, использование генераторов вместо списков где применимо) – тоже часть работы с big data. Это не ускорение CPU, но предотвращение деградации скорости из-за memory pressure.

10. Компиляция и упаковка: PyInstaller, Nuitka, zipapps

Когда речь идет о деплое приложений или о желании получить дополнительный прирост скорости за счет компиляции – на помощь приходят специальные инструменты:

- **PyInstaller:** утилита, которая собирает ваше Python-приложение (скрипт и все зависимости) в самостоятельный исполняемый файл (или папку) с вложенным интерпретатором. Это в первую очередь про **удобство распространения**: вы можете выдать клиенту один `.exe` файл вместо набора .py и установки интерпретатора. С точки зрения **производительности**, PyInstaller не делает ваш код машинным – внутри exe все равно работает CPython и ваши .pyc. Теоретически, небольшое ускорение **старта** программы возможно, так как модули можно загружать из единого zip-архива (меньше операций диска). Но существенного прироста времени исполнения не ждите – скорость будет как у обычного CPython. PyInstaller имеет опцию оптимизации (`--onefile --`

`optimize=2`), которая убирает debug-информацию и т.п., но это мало влияет. Основное – удобство.

Применение: PyInstaller полезен, когда нужно **отправить программу пользователю**, у которого может не быть Python. Например, вы написали утилиту для анализа логов – запаковали PyInstaller, дали коллегам .exe – они запускают без забот об окружении. В backend'е реже применимо, разве что для заморозки приложений на серверах без установки Python.

- **zipapp (Python Zip Applications):** Модуль `zipapp` позволяет упаковать питоновский код в .pyz (исполняемый zip-архив). Python умеет выполнять zip-архивы, содержащие `__main__.py`. Это похоже на PyInstaller, но не включает интерпретатор – подразумевается, что Python уже установлен, а zip содержит только код. **Преимущество zipapp** – получаем единый файл, который можно запускать `python myapp.pyz`. Недостаток – зависит от установленного Python, и не прячет код (он лежит просто в zip).

Производительность: zipapp в целом не ускоряет код. При первом запуске, Python импортирует модули из zip (zipimport). Это может быть чуть медленнее, чем с диска, особенно если zip сжат (он по умолчанию может сжимать). В трекере CPython отмечали, что zipimport на больших архивах немного тормозной [58] (50 мс на 1000 файлов архива). Но это цифры в масштабе инициализации. В runtime разницы нет – код исполняется так же. Если хотите *уменьшить* overhead zipimport, можно создать zip с `--compresslevel 0` (не сжимать, только упаковка) – тогда чтение модулей не требует распаковки.

Когда использовать zipapp: Для деплоя утилит внутри одной организации, где Python точно есть. Например, вы пишете скрипт, которым будут пользоваться админы на разных машинах – вместо распространять папку с кодом, делаете `.pyz`. Его удобно копировать, версия контролируется.

- **Nuitka:** Это уже **настоящий компилятор** Python в C/C++. Nuitka берет ваш код (и все зависимости), транслирует в C++ код, который внутри вызывает CPython API. То есть, он по сути компилирует ваш скрипт в эквивалент расширения. Затем компилирует этот C++ (нужен компилятор, например gcc) и линковает с `libpython`. Получается исполняемый файл или библиотека. Nuitka добивается полной совместимости – поддерживает практически все, включая динамические особенности Python.

Производительность: Nuitka обычно даёт **умеренный прирост скорости**. К примеру, автор Nuitka упоминал, что целевые оптимизации могут дать **ускорение 2x** на общем приложении [38]. Пользователи тоже отмечают, что *«Nuitka не бывает медленнее CPython, иногда в 2-4 раза быстрее на отдельных функциях, но редко более 20% по всей программе»* [59]. То есть, ожидать 10-кратных ускорений не стоит. Прирост идёт за счёт оптимизации вызовов, устранения интерпретатора на некоторые вещи, но GIL никуда не делся – это все та же семантика CPython.

Плюсы Nuitka: - Получаете нативный exe (как PyInstaller, но без отдельного интерпретатора – он встроен). - Код частично защищен (обратно в .py его так просто не получить, хотя при желании можно дизассемблировать). - Есть некоторые оптимизации: Nuitka может, например, выявлять функции, которые не используют динамические возможности, и оптимизировать их. Со временем, возможно, Nuitka станет более оптимизирующим (как Cython с типами).

Минусы: - Компиляция может быть долго (большие проекты – минуты, десятки минут). - Экешей: exe размером больше, чем исходники (включает runtime). - Не все пакеты поддерживаются out-of-the-box (но большинство работают).

Когда использовать: Если нужно **скрыть исходный код** (до некоторой степени) или чуть ускорить определённый сценарий без переписывания. Например, у вас скрипт выполнения арифметики, и нужно выжать максимум, но переписывать на C нет времени – Nuitka может дать 2x (как один пользователь заметил – "маленькие проекты ускоряет ~2x, мне хватило" [38]). Также Nuitka часто применяют вместе с PyInstaller: сначала Nuitka-компилируют части кода (для оптимизации и обфускации), затем все собирают PyInstaller в один exe.

- **Cython как упаковка:** Мы упоминали Cython в контексте оптимизации. Стоит добавить, что Cython можно использовать, чтобы превратить Python-проект в расширение (pyd/so) и импортировать, или даже вызвать `PyInit___main__` чтобы сделать исполняемый. Но чаще Cython применяют для отдельных модулей, а сборка экзешника – это уже PyInstaller + Cython.

- **Другие инструменты:**

- **cx_Freeze, py2exe, py2app:** аналоги PyInstaller, собирают приложение, но PyInstaller сейчас наиболее поддерживаемый.
- **PyOxidizer:** современный инструмент, упаковывающий интерпретатор и код в один бинарник (Rust-based). Обещает быстрый startup, т.к. может вставлять стандартную библиотеку в сегмент данных exe. Интересный проект, но менее зрелый чем PyInstaller.
- **MyPyC:** компилятор от команды mypy, который берет типизированный Python (аннотации) и компилирует части в C. Он не полный (не все конструкции поддерживает), но те, что поддерживает, могут дать 4x ускорение. Подходит в основном для библиотек (например, attrs библиотека ускорилась с помощью MyPyC).

- **Pythran:** компилятор для научных вычислений (transpile Python with NumPy calls to C++). Специализирован, но иногда очень эффективен для конкретных численных функций.

- **Rust/Go reimplementations:** Если нужна максимальная скорость и есть ресурсы – иногда критичные компоненты переписывают на Rust или Go и вызывают из Python (через FFI). Но это не автоматический путь.

Пример Nuitka: Возьмем простой скрипт расчета простых чисел. Под Nuitka он может ускориться, скажем, в 1.5-2 раза (из-за убранного интерпретатора цикла). Но не в 10 – ведь алгоритм тот же. Nuitka не умеет глобальных оптимизаций (например, векторизовать код или убрать GIL).

Пример PyInstaller: Если измерить *время запуска* сложного приложения: с PyInstaller оно может запуститься на доли секунды медленнее, потому что распаковывается / инициализируется буфер. Были вопросы о том, что "крупный zipapp/PyInstaller становится медленнее при росте" [60] – ответ: да, заметно на бОльших архивах, но речь о десятках миллисекунд, что редко критично.

Где может быть прирост: - Если ваш код активно вызывает функции, Nuitka может сократить оверхед вызова (C-функции вызываются быстрее). - Nuitka может встроить некоторые функции или оптимизировать константы. - Если у вас много мелких файлов, PyInstaller/zipapp может даже **сократить** время загрузки – за счет чтения одним большим блоком, вместо множества мелких файлов (особенно на медленных дисках). - Если Python не установлен, но нужно очень быстро развернуть CLI – PyInstaller binary запускается сразу, а установка Python + pip install deps заняла бы значительно больше времени.

Архитектурные моменты: В серверных (backend) приложениях часто вместо объединения в exe, используют контейнеры (Docker) – туда кладут Python окружение и код. Здесь PyInstaller не особо нужен. PyInstaller скорее для десктоп- и утилит.

Вывод: - PyInstaller/zipapp – *packing tools*, удобство деплоя, не ускорители. - Nuitka – *compiling tool*, умеренный ускоритель, плюс обфускация, возможно стоит пробовать если всё остальное оптимизировано. - Cython/PyPy/Numba – более эффективны для оптимизации, но требуют либо изменения кода (Cython/Numba) либо замены интерпретатора (PyPy).

Если целью стоит именно **ускорение**, с минимальными изменениями – PyPy или Nuitka. PyPy может 4x ускорить, Nuitka ~2x, но PyPy требует убедиться, что всё совместимо. Nuitka – требует длительной компиляции, но потом плод – обычный exe (можно запускать без Python).

К слову, *Guido van Rossum рекомендовал: "If you want your code to run faster, you should probably just use PyPy."* [61] – это шутливо, но PyPy реально часто даёт больший выигрыш, чем статическая компиляция Nuitka. Однако PyPy – не packaging solution, a runtime.

Пример сравнения: В статье на Dice.com сравнили CPython, PyInstaller, Nuitka, PyPy, Cython на бенчмарке pystone [62] [35]: - CPython ~ 600k pystones/sec (это относительная метрика). - Nuitka скомпилированный ~600k (примерно то же, иногда чуть выше, у них вышло 597k vs 610k) [63] [35]. - PyPy ~1.77 million pystones/sec [36] – почти 3x ускорение. - Cython (без специальных оптимизаций) ~228k (даже медленнее, т.к. не добавили типы, пример показал важность правильного использования Cython) [64]. Вывод: PyPy был самым быстрым на том тесте. Nuitka просто не замедлил относительно CPython, но и не дал чуда. Cython без типов – тоже. С типами, конечно, Cython мог бы выиграть, но потребовалось прописать их.

Итого: Packaging (PyInstaller, zipapp) – вещь для удобства, **не делайте на них ставку ради скорости**. Компиляция (Nuitka) – может слегка помочь, но скорее полезна для deploy и защиты кода. Если уже всё профилировано, алгоритмы оптимальны, GIL мешает – Nuitka не поможет (GIL останется). Тогда уже либо `multiprocessing` распараллеливать, либо думать о C-реализации критических частей.

11. Общие советы по пишущемуся быстрому и поддерживаемому коду

Наконец, обсудим **общие принципы**, которые помогают сделать код и быстрым, и поддерживаемым:

- **Профилируй и измеряй перед оптимизацией:** Золотое правило: *«Преждевременная оптимизация – корень всех зол»* (Д. Кнут) [65] [1]. Не стоит гадать, какой код медленный – лучше замерить (вплоть до простого `print(time() - start)` или использовать профилировщики). Оптимизируйте узкие места, а не всё подряд. Это сэкономит время и убережет от усложнения программы без нужды. Например, можно тюнить цикл, добиваясь 5% выигрыша, а оказывается, 80% времени программа ждёт ответа от базы – т.е. оптимизировать надо слой работы с БД или параллелить запросы.

- **Выбирайте правильные алгоритмы и структуры:** Как мы подробно говорили в разделе 2, самым критичным для производительности часто является *сложность алгоритма*. Проверить принадлежность через список vs через set – разница колоссальна при росте

данных. Использование сортировки vs квадратичного алгоритма. Всегда оценивайте Big O своих решений. Улучшение сложности с O(n^2) до O(n log n) может ускорить на больших n на порядки, в то время как микротюнинг O(n^2) ничего принципиально не даст. **Грамотная архитектура и алгоритмы – фундамент быстрого кода.** И наоборот, неудачная структура (например, хранить данные в списке и при каждом запросе линейно искать) трудно компенсируется низкоуровневыми оптимизациями.

- **Пишите код понятно (PEP8, чистый стиль), затем оптимизируйте локально:** Читабельность – ключ к поддержке. Как отмечалось: *«код пишется для людей, а не для машин. Если понадобится ускорить – понятный код легче оптимизировать, чем запутанный»* [30]. Поэтому сперва реализуйте корректно и понятно. Соблюдайте PEP8 (отступы, имена, организация кода) – это упрощает понимание командой, а значит, и скорость, с которой можно внести улучшения. **Не пишите микрооптимизированный, но нечитаемый код**, если на то нет крайней необходимости. Например, есть известный приём в Python: использовать трюки с побитовыми операциями для ускорения, но они усложняют код. Применять такое стоит только если действительно нужно и документировать.

- **Локализуйте оптимизированный (менее понятный) код:** Если вам пришлось написать что-то хитрое для скорости – изолируйте это в функцию/метод с хорошим именем и комментарием. Тогда основная логика останется чистой, а подробности спрячутся. В комментариях укажите, почему тут такая сложность: *“Оптимизация: используем битовые флаги вместо списка для экономии памяти и ускорения доступа”*. Будущие разработчики (возможно, вы сами через полгода) должны понимать, что происходит, иначе велика вероятность, что при первом рефакторинге ваш “хитрый” код сломают или выбросят.

- **Документируйте и тестируйте функциональность до оптимизации:** Убедитесь, что есть тесты. Оптимизация часто связана с усложнением кода – автоматические тесты помогут убедиться, что вы ничего не сломали. Особенно это актуально при вводе параллелизма (можно легко внести гонку) или кеширования (можно получить несвежие данные). Тесты плюс типизация (PEP484 type hints) – очень помогают поддерживать оптимизированный код корректным.

- **Учитесь у стандартных средств:** Многие оптимизации уже воплощены в стандартной библиотеке. Читайте код модулей itertools, functools, collections – там можно найти интересные приёмы. Например, как реализован lru_cache или deque – это может дать идеи для ваших специфических задач.

- **Оценивайте цену оптимизации:** Иногда усложнение кода ради micro-оптимизации не стоит того. Например, расклеивать выражение на несколько строк, использовать низкоуровневые C-расширения – всё это увеличивает стоимость поддержки. Если выигрыш – миллисекунды, а цена – дни работы и риск багов, может, не нужно. Как сказано на Reddit: *“Большинство программ не ограничены CPU. Оптимизация вычислений часто бессмысленна, если пользователь ждёт I/O”* [66]. То есть если ваш веб-сервис проводит 99% времени в ожидании БД, нет смысла тюнить остальной код, лучше сфокусироваться на ускорении запросов к БД или кэшировании.

- **Выбирайте современные версии Python и библиотек:** Каждый новый релиз Python приносит оптимизации. Python 3.11, например, около 25% быстрее 3.10 в среднем [67]. Всего лишь обновившись, вы получите ускорение "бесплатно". То же по библиотекам: например, Pandas 2.0 принес ускорения на группировках, NumPy новые версии иногда

оптимизируют операции. Следите за релиз-нотами. Конечно, обновление требует тестирования, но в плане производительности это одна из самых простых мер.

- **Оптимизация архитектуры приложения:** На высоком уровне, подумайте, можно ли ускорить за счет изменения архитектуры:

- Распараллелить на уровне процессов/серверов (scale out).
- Избежать лишних операций (кеширование, как мы обсуждали, или отказ от ненужных функций).
- Разделить тяжелые и легкие задачи (например, вынести обработку больших данных в оффлайн-джобу, а в онлайн-пути использовать заранее посчитанное).
- Использовать подходящие технологии: например, вместо генерировать отчет по запросу – подготовить его асинхронно и хранить готовым.
- В веб-архитектуре: может добавить CDN, чтобы не генерировать часто запрашиваемые страницы, а отдавать кешированные.

Все эти вещи иногда дают больше выигрыша, чем низкоуровневый оптимизатор может мечтать.

- **PEP8 – не в ущерб скорости:** Хотя PEP8 – о стиле, он опосредованно влияет на производительность разработки (а отлаженность кода тоже часть перфоманса проекта!). Например, именование – если переменные названы ясно, меньше шансов на ошибки и неправильные предположения, которые могут привести к неэффективному коду. Структура модулей – PEP8 советует разбивать большой файл на логические части (но не делает жестко) – так удобнее профилировать и оптимизировать куски. Одним словом, **поддерживаемость кода помогает оптимизации, потому что вы быстрее разберетесь, что улучшать.**

- **Используйте инструменты для диагностики производительности:** Помимо профилировщиков, полезны **трассировка** (logging времени выполнения ключевых этапов), **мониторинг** (в веб-приложении – APM, метрики). Это дает знание, где узкие места проявляются в продакшене. Иногда предположения на дев-машине не совпадают с реальностью нагрузки. Например, профилировали вы на небольшом наборе – а на бою всплыли проблемы с аллокацией памяти на больших данных (GC паузы, и т.д.). Мониторинг (графики времени отклика, CPU, memory) – подскажет, куда копать.

- **Не пренебрегайте сборкой мусора и памятью:** Python – сборщик мусора (GC). Если ваш код создает огромное количество мелких объектов, GC может тратить ощутимое время. Оптимизируйте количество объектов: например, вместо создавать миллионы кортежей, возможно, лучше использовать массив NumPy. Или использовать слоты в классах, чтобы уменьшить overhead. Хотя CPython GC неплох, но бывают патологии (например, много короткоживущих объектов в цикле – нагрузка на аллокатор). Инструмент `tracemalloc` может помочь отследить рост памяти. Чистый код с меньше лишними объектами – быстрее.

- **Вопросы безопасности и корректности:** Иногда "оптимизированный" код может иметь побочные эффекты. Например, использовать `itertools.islice` для экономии памяти – но забыть, что iterator расходуется. Или кеширование – но не продумать invalidation, и получить баг "пользователь видит чужие данные". Поэтому всегда проверяйте, что оптимизация не нарушила функциональность. Безопасность: если вы, ради скорости, отключаете проверки (например, `assert` убрали, или Cython с `boundscheck=False` –

следите, чтобы это не открывало уязвимостей (в Cython может привести к segfault при неправильных индексах, но это уже не секьюрити, а стабильность).

- **Здравый смысл и баланс:** Помните, что цель – не самый быстрый в мире код, а достаточно быстрый для требований, при этом надежный и понятный. Бывает, что "достаточно быстрый" достигается малыми правками (кеширование, SQL-оптимизация), и не нужно увлекаться дальнейшим ускорением. В других случаях – требования жесткие, и тогда да, приходится жертвовать простотой (переписывать на C++/Rust). Оценивайте ROI (окупаемость) оптимизации.

Наглядная мысль: *«Читаемый, но медленный код гораздо легче сделать быстрым, чем быстрый, но нечитаемый – сделать правильным»* [30] . Так что придерживайтесь прогрессивного улучшения: пишем чисто -> профилируем -> точечно ускоряем -> профилируем снова -> рефакторим, если надо.

И напоследок, **не забывайте об опыте сообщества**: есть бесчисленное множество статей, дискуссий о Python performance. Изучайте успешные кейсы, новые инструменты (тот же PyPy, Numba – когда-то о них многие не знали и мучились). Python – язык с сильным сообществом, и практически для любой проблемы производительности уже может быть найдено решение или библиотека.

Следуя этим главам и принципам, можно добиться от Python-кода максимума производительности, сохранив при этом его высокую продуктивность разработки и сопровождения. Быстрых вам программ!

Ссылки:

1. Python Profiling tools (cProfile, timeit) – Analytics Vidhya [2] [3]
2. Medium – 5 Python profiling tools [4] [5]
3. Princeton Research – Line_profiler usage [8] [9]
4. Python Wiki – TimeComplexity of data structures [13] [12]
5. Bisect documentation – dict vs binary search [14]
6. Python Morsels – set membership vs list [15]
7. GeeksforGeeks – Using built-in functions (sum vs loop) [25] [68]
8. GFG – Using sets for membership (example code) [23] [24]
9. GFG – List comprehension vs loop performance [28] [29]
10. Jake VanderPlas blog – Numba vs pure Python speedup [31]
11. Reddit r/Python – PyPy vs CPython discussion [38] [59]
12. Dev.to – Threading vs Multiprocessing vs Asyncio [40] [43]
13. Dev.to – aiohttp performance for high concurrency [45]
14. BrightData blog – aiohttp 10x faster than requests [42]
15. Cameron MacLeod – lru_cache example (Flask route) [46] [47]
16. Redis cache benefits – BlackSlate blog [49]
17. StackOverflow – NumPy vectorization vs loop (646ms vs 20ms) [50] [51]
18. Exxactcorp blog – Pandas loop vs apply vs vectorize (350x gain) [53] [54]
19. Hacker News – Nuitka ~2x vs CPython, PyPy ~3x [38] [59]
20. Reddit – Readability vs performance advice [30] [1]

---

1 22 30 65 66 code: performance vs. readability : r/Python
https://www.reddit.com/r/Python/comments/11qim3q/code_performance_vs_readability/

2 3 7 Python Optimization: Performance, Tips & Tricks in 2024
https://aglowiditsolutions.com/blog/python-performance-optimization/

4 5 6 10 5 Python profiling tools for performance analysis | by Saurav Paul | Medium
https://medium.com/@saurav.kr.paul/5-python-profiling-tools-for-performance-analysis-17fa245324cd

8 9 Python Profiling | Princeton Research Computing
https://researchcomputing.princeton.edu/python-profiling

11 12 13 19 TimeComplexity - Python Wiki
https://wiki.python.org/moin/TimeComplexity

14 21 bisect — Array bisection algorithm — Python 3.13.3 documentation
https://docs.python.org/3/library/bisect.html

15 16 17 Python Big O: the time complexities of different data structures in Python - Python Morsels
https://www.pythonmorsels.com/time-complexities/

18 deque.popleft() and list.pop(0). Is there performance difference?
https://stackoverflow.com/questions/32543608/deque-popleft-and-list-pop0-is-there-performance-difference

20 What's the time complexity of functions in heapq library
https://stackoverflow.com/questions/38806202/whats-the-time-complexity-of-functions-in-heapq-library

23 24 25 26 28 29 68 How to Minimize Python Script Execution Time? | GeeksforGeeks
https://www.geeksforgeeks.org/how-to-minimize-python-script-execution-time/

27 Optimizing Python Code for Performance: Tips and Techniques for ...
https://www.linkedin.com/pulse/optimizing-python-code-performance-tips-techniques-faster-vg33f

31 32 33 34 Numba vs Cython | Pythonic Perambulations
https://jakevdp.github.io/blog/2012/08/24/numba-vs-cython/

35 36 61 62 63 64 4 Fast Python Compilers for Better Performance | Dice.com Career Advice
https://www.dice.com/career-advice/4-fast-python-compilers-better-performance

37 Deciding what to use among Cython / Pypy / Numba : r/Python
https://www.reddit.com/r/Python/comments/uafu40/deciding_what_to_use_among_cython_pypy_numba/

38 59 Nuitka is the best python compiler I've used. I have tried at least three others... | Hacker News
https://news.ycombinator.com/item?id=27538051

39 Python 3.11 is up to 10-60% faster than Python 3.10 - Reddit
https://www.reddit.com/r/programming/comments/vswllm/python_311_is_up_to_1060_faster_than_python_310/

40 41 43 Boost Python Performance: A Guide to Asyncio, Threading, and Multiprocessing - DEV Community
https://dev.to/yoshan0921/accelerate-python-programs-with-concurrent-programming-28j9

42 Requests vs. HTTPX vs. AIOHTTP: Which One to Choose?
https://brightdata.com/blog/web-data/requests-vs-httpx-vs-aiohttp

44 numpy.memmap — NumPy v2.2 Manual
https://numpy.org/doc/stable/reference/generated/numpy.memmap.html

45 Choosing Between AIOHTTP and Requests: A Python HTTP Libraries Comparison - DEV Community
https://dev.to/api4ai/choosing-between-aiohttp-and-requests-a-python-http-libraries-comparison-23gl

46 47 Easy Python speed wins with functools.lru_cache - Cameron MacLeod
https://www.cameronmacleod.com/blog/python-lru-cache

48 How to use Redis for Query Caching
https://redis.io/learn/howtos/solutions/microservices/caching

49 Supercharge Your Python Application: Redis as a High ... - BlackSlate
https://www.blackslate.io/articles/redis-as-a-high-performance-cache-for-python-application

50 51 python - For loop vs Numpy vectorization computation time - Stack Overflow
https://stackoverflow.com/questions/51549363/for-loop-vs-numpy-vectorization-computation-time

52 53 54 55 56 How to Speed Up Python Pandas by over 300x | Exxact Blog
https://www.exxactcorp.com/blog/Deep-Learning/how-to-speed-up-python-pandas-by-over-300x

57 Vectorization in Python- An Alternative to Python Loops - Medium
https://medium.com/pythoneers/vectorization-in-python-an-alternative-to-python-loops-2728d6d7cd3e

58 Issue 8745: zipimport is a bit slow - Python tracker
https://bugs.python.org/issue8745

60 Why do Python zipapps get slower the bigger they get?
https://stackoverflow.com/questions/73773534/why-do-python-zipapps-get-slower-the-bigger-they-get

67 Where exactly does Python 3.11 get its ~25% Speedup? Part 1
https://log.beshr.com/python-311-speedup-part-1/