

# В поисках ответа, почему Python [не] умеет работать с CPU- bound задачами



Контур

Иван Жерновков

# Мой опыт

2г – трейдинговые алгоритмы и сервисы

2г – классическая backend разработка

2г - реализация алгоритмов (TSP)

3г – desktop приложения (front и back на Python)

Python, C++, C, Limited C



# Мой опыт, настоящий момент

Контур

Мой основной проект:

Edge L3/L4 balancer, немного L7 в прошлом

Backend для балансера

И сервисы вокруг

# План доклада

1

На правах  
предисловия

2

Проблема

3

Анализ

4

Выводы

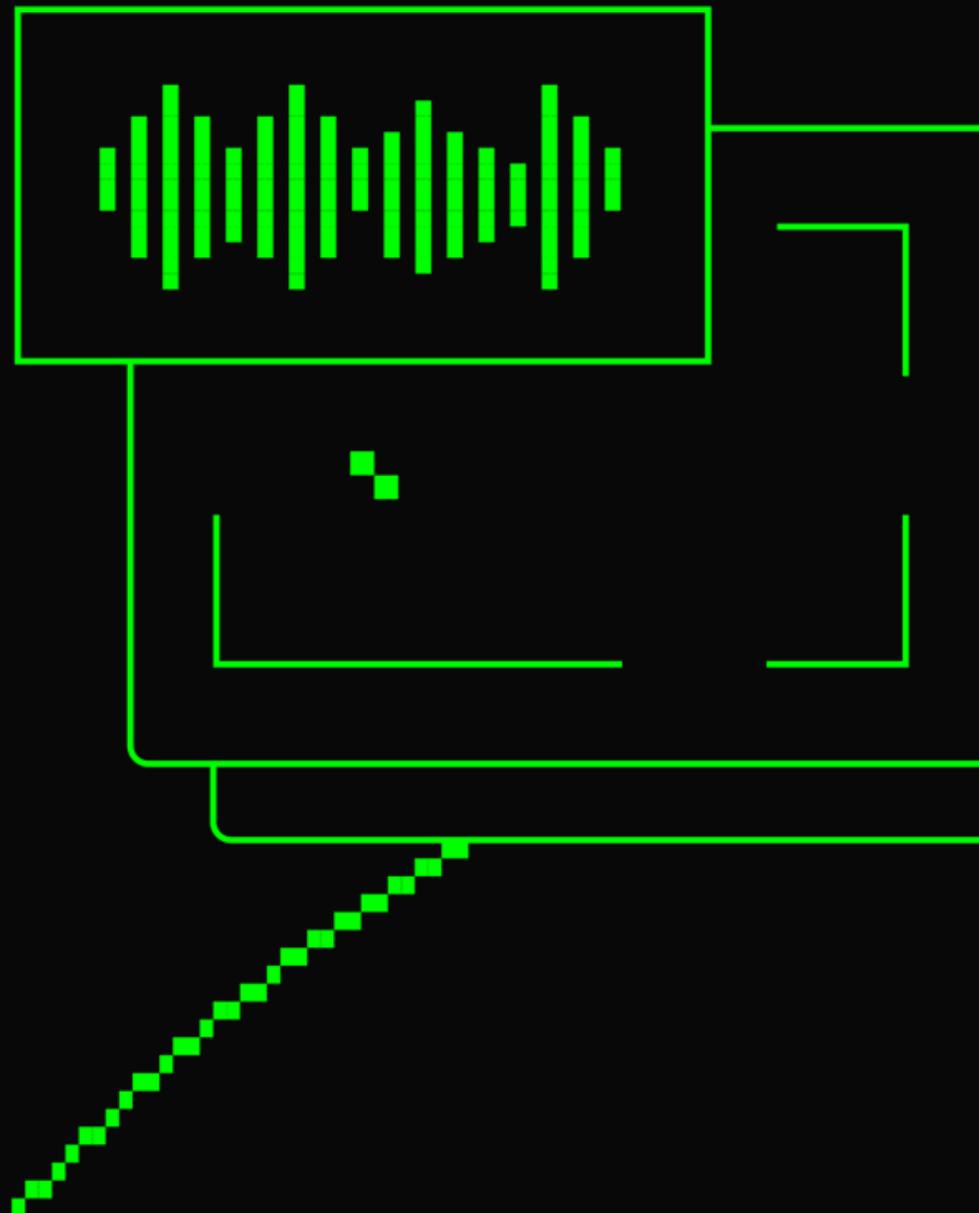
# Дисклеймер

Что нужно помнить на протяжении доклада:

- Будут QR коды. Много.
- Материал по ссылкам не прочесть за время демонстрации слайда, да и за время презентации скорее всего тоже.
- Все ссылки будут в конце.
- Так же будет много терминов, буду расшифровывать их в моменте. Но термины скорее всего будут пересекаться с чем то уже известным. Дайте мне чуть-чуть времени чтобы я успел рассказать о них.

- 1
- 2
- 3
- 4

# На правах предисловия

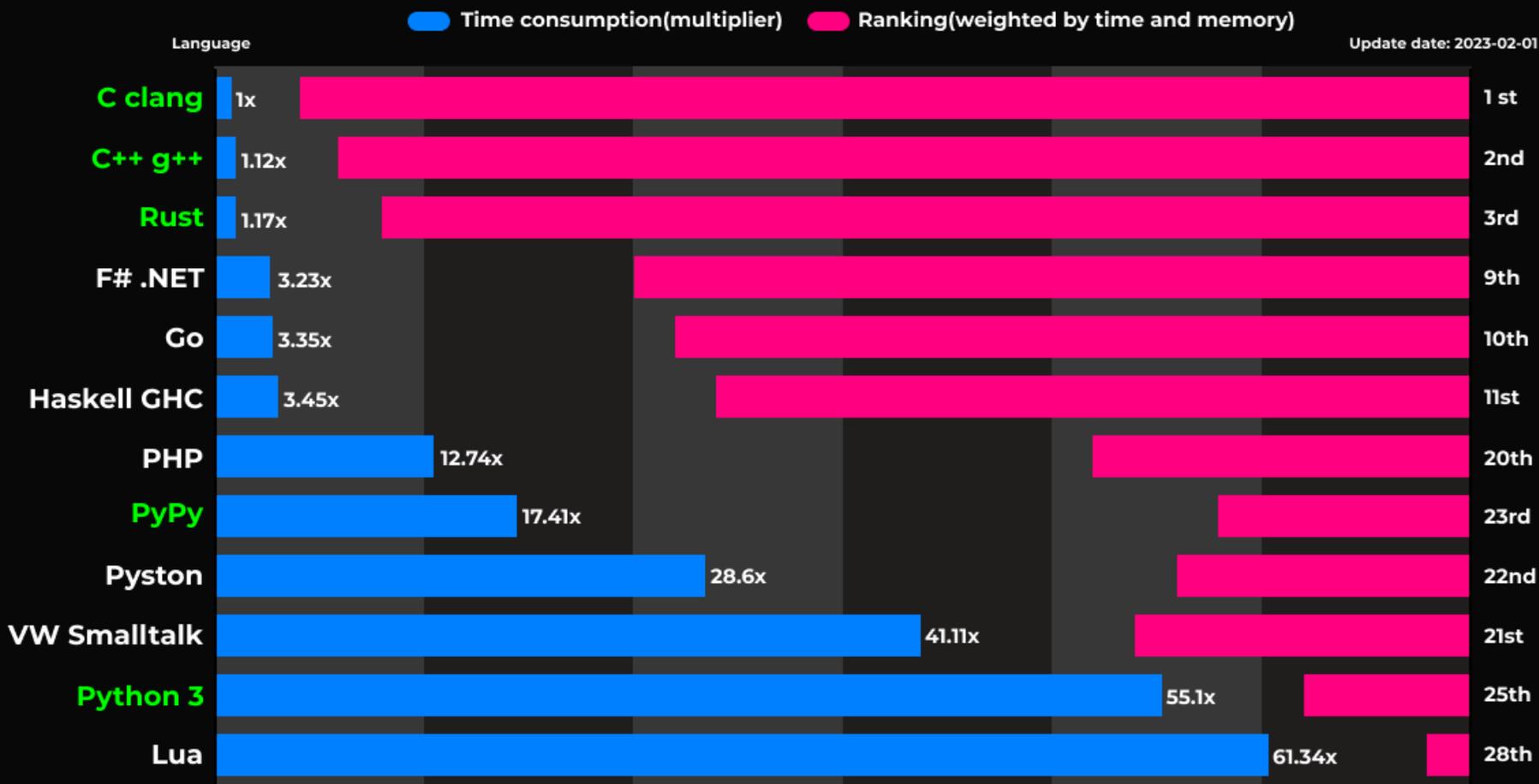


# Откуда мы знаем, что Python – медленный?

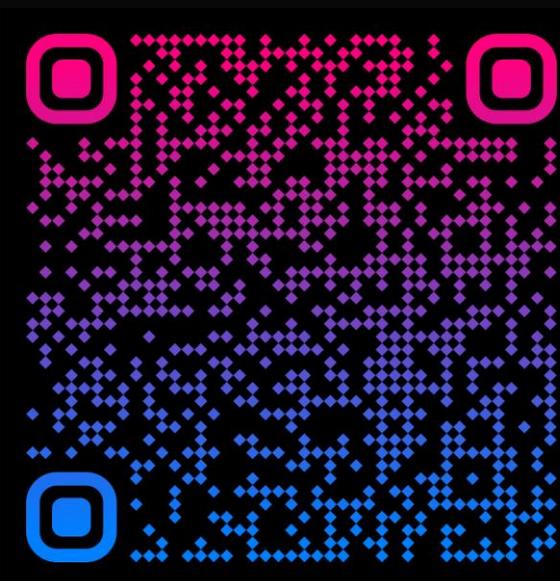
- Из учебной программы
- Из профессиональной литературы
- Рассказываем об этом на собеседованиях
- На конференциях постоянно разгоняем Python
- С ресурсов типа Benchmarks Game

# А что ещё мы знаем о скорости языков?

The Computer Language Benchmarks Game Visualization

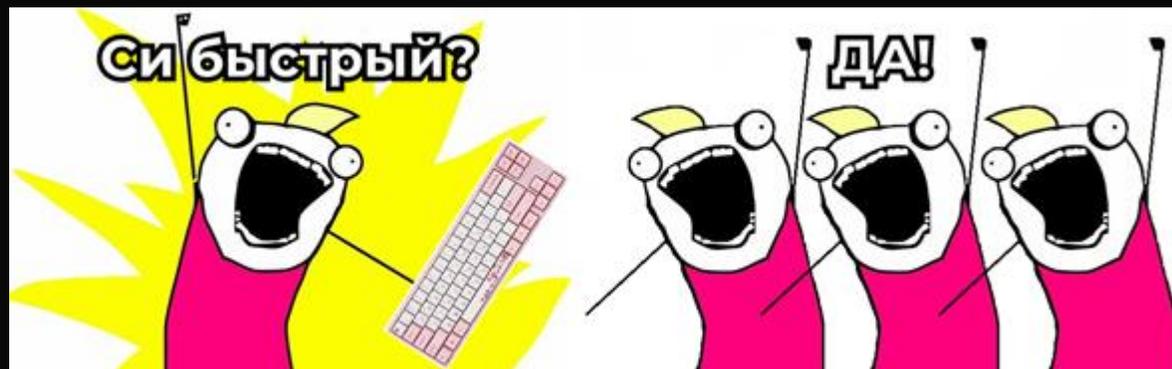


Бенчмарк



А что задевает докладчика?

# А что задевает докладчика?



# А что задевает докладчика?



# А что задевает докладчика?



А если взглянуть на РуРу?

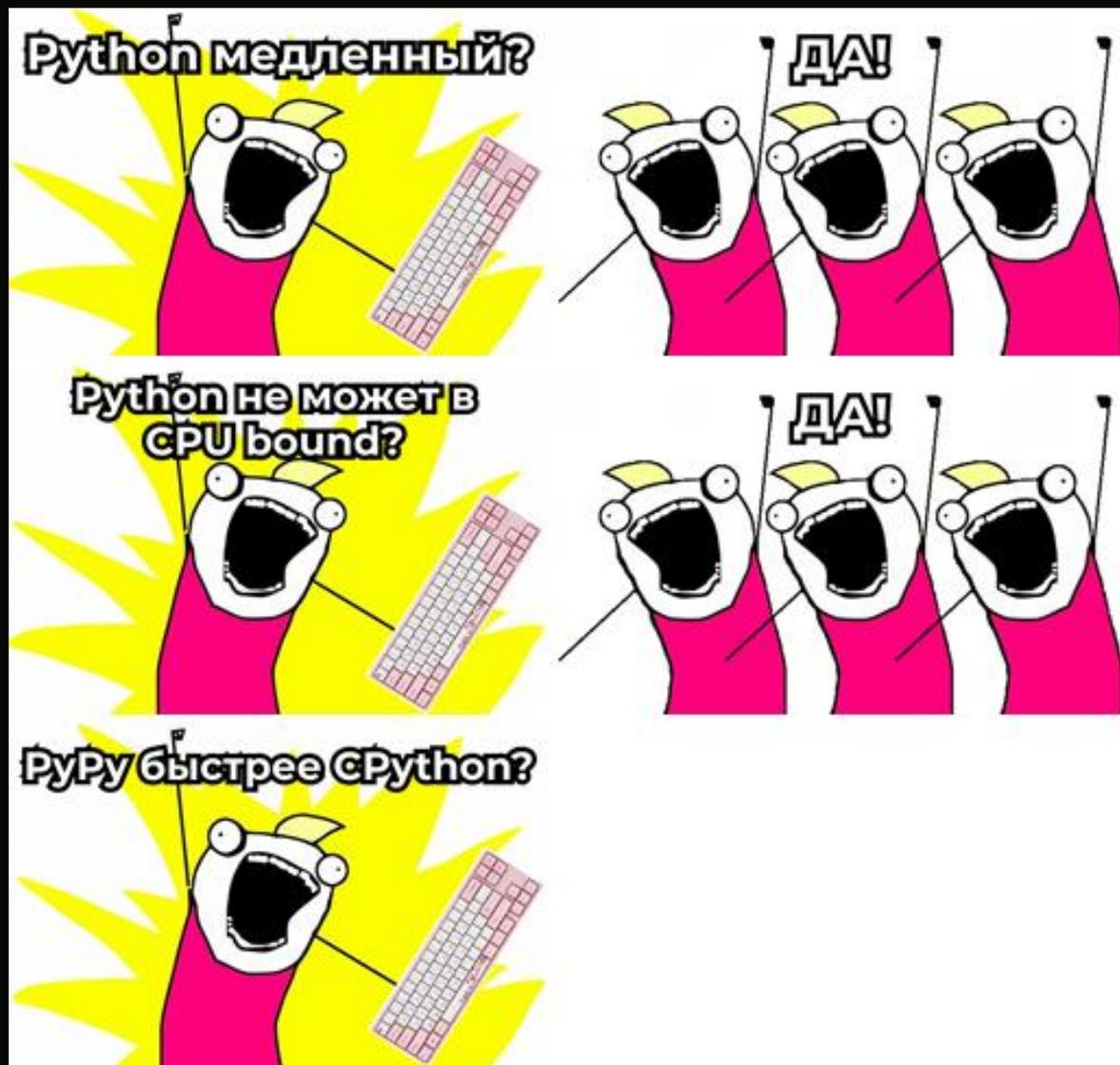
# А если взглянуть на PyPy?



# А если взглянуть на PyRu?



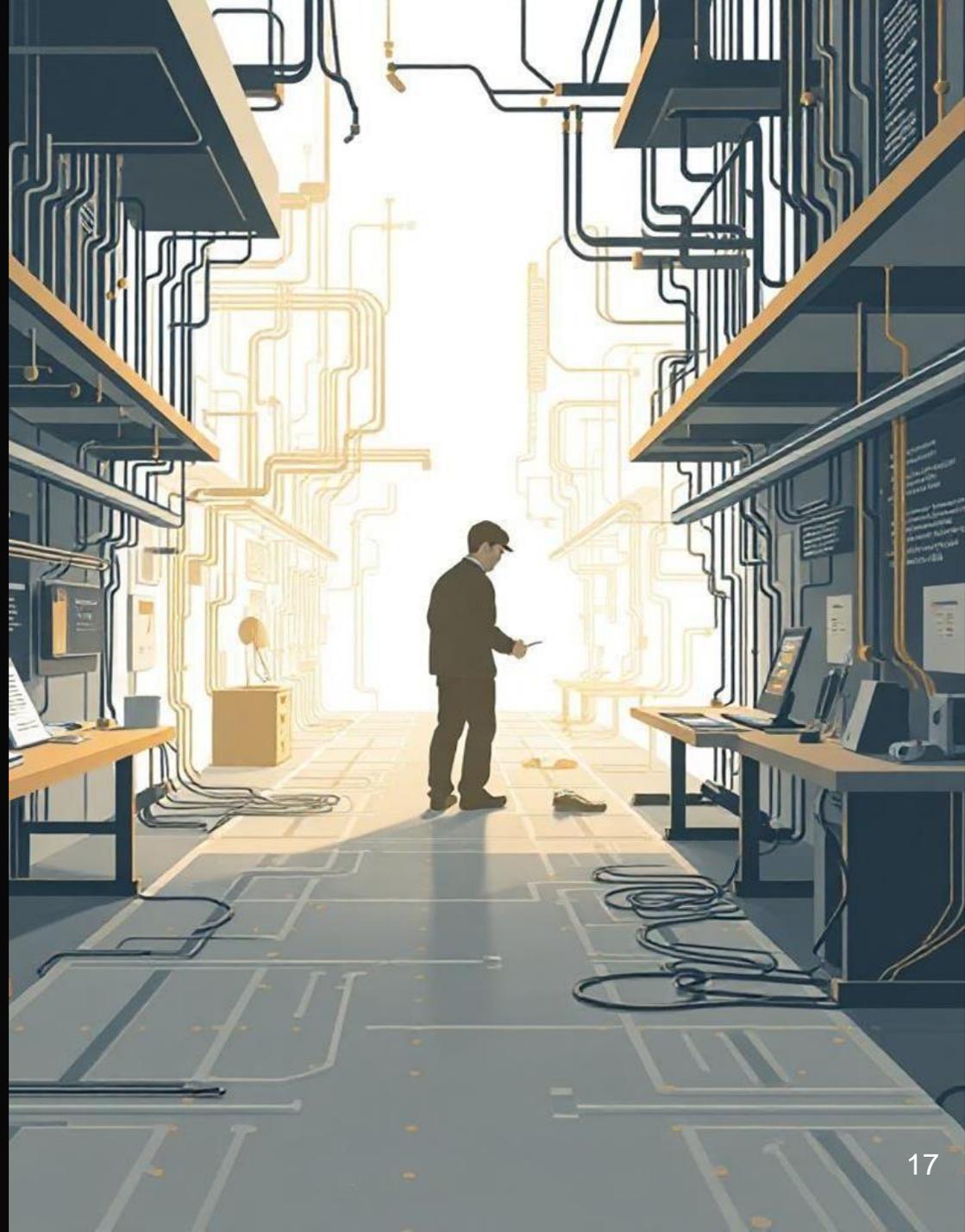
# А если взглянуть на PyPy?



# Инженерный путь

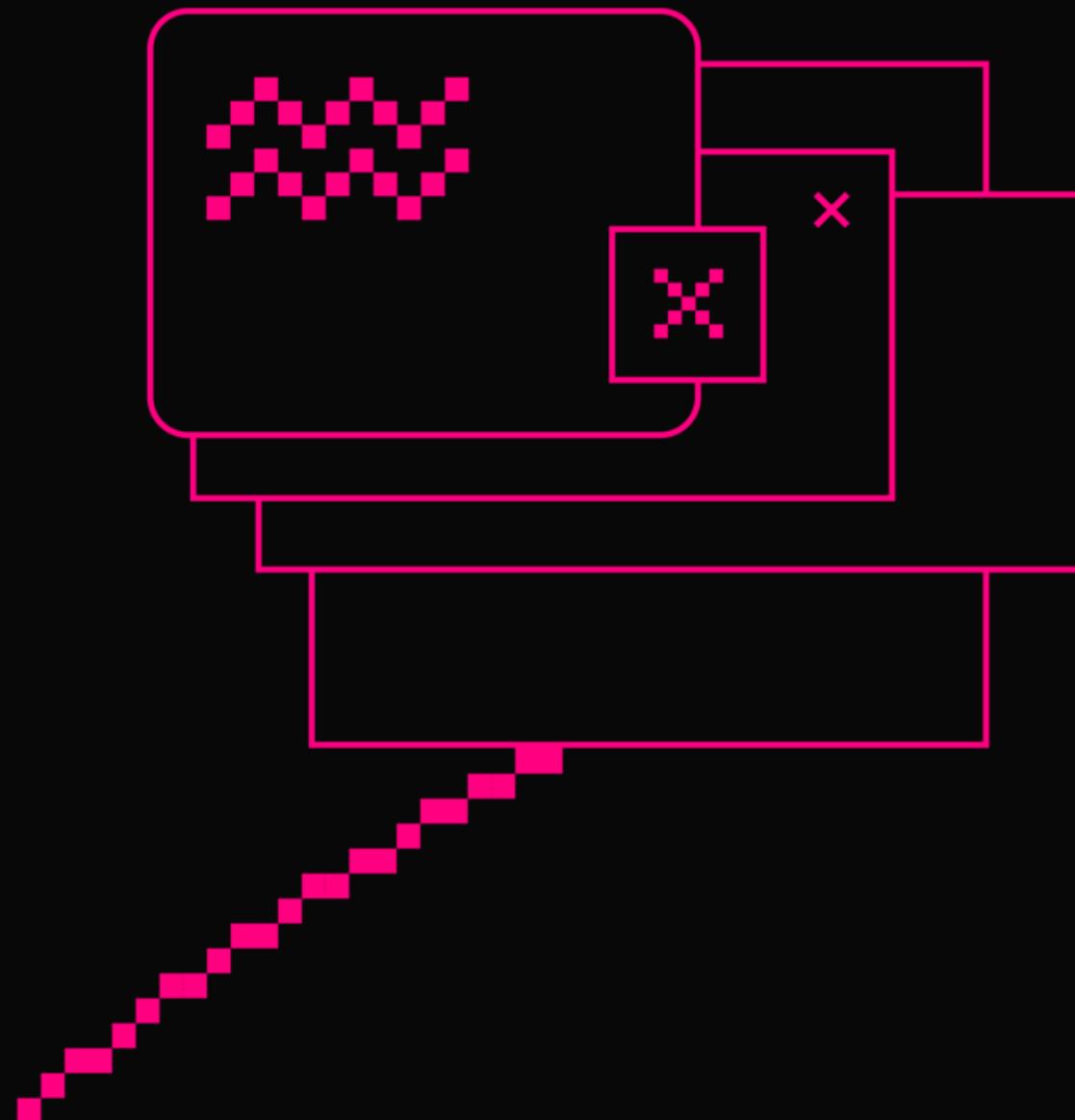
- Нет плохих или хороших технологий
- Технологический стек – это тоже инструмент. Умение отказаться от его применения так же ценно, как и умение его применять
- Практика – критерий истины
- И многое другое...

Инженерный путь – тонкая красная нить доклада, но явных отсылок на него не будет. Он сыграл свою роль в прокопке материала, уважим его, упомянув о нём, но далее только воспользуемся его плодами.



- 1
- 2
- 3
- 4

# Проблема



# Сервис на Python

- ~20к строк кода
- MVP не может запуститься, так как CI отваливается по таймату
- Проблема в проверке прав
- И нагрузочный тест провалился уже на 10% от запланированной нагрузки..
- А если приглядеться, то виноваты 14 строк кода, которые вывели CPU в стратосферу



# Что нам нужно посчитать?

Наши данные (псевдо код):

`[[10.40.40.40:1000 - 10.40.40.40:2000],`

`[10.40.40.50:1000 - 10.40.40.60:2000],`

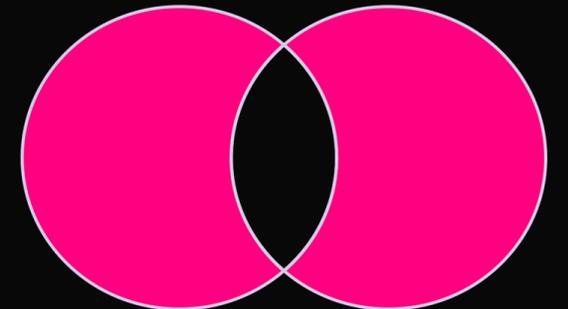
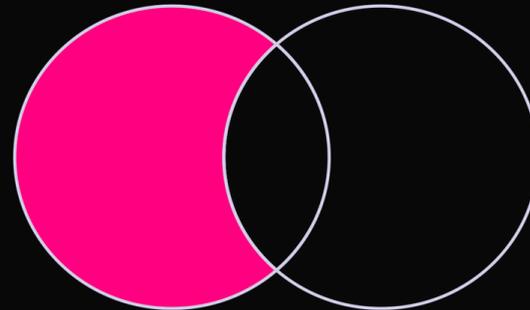
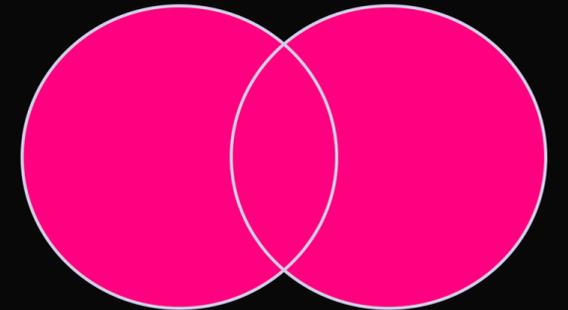
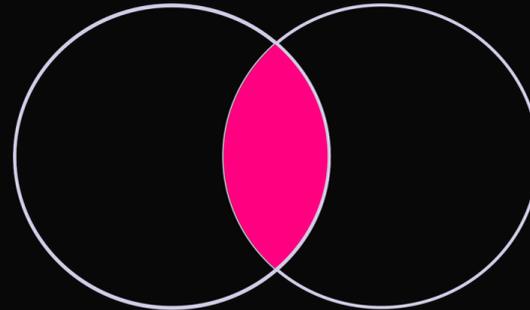
`[10.40.20.50:1000 - 10.40.30.60:2000],`

`[10.40.40.50:5000 - 10.40.40.60:9000],`

`[10.200.40.50:1 - 10.200.40.255:1024]]`

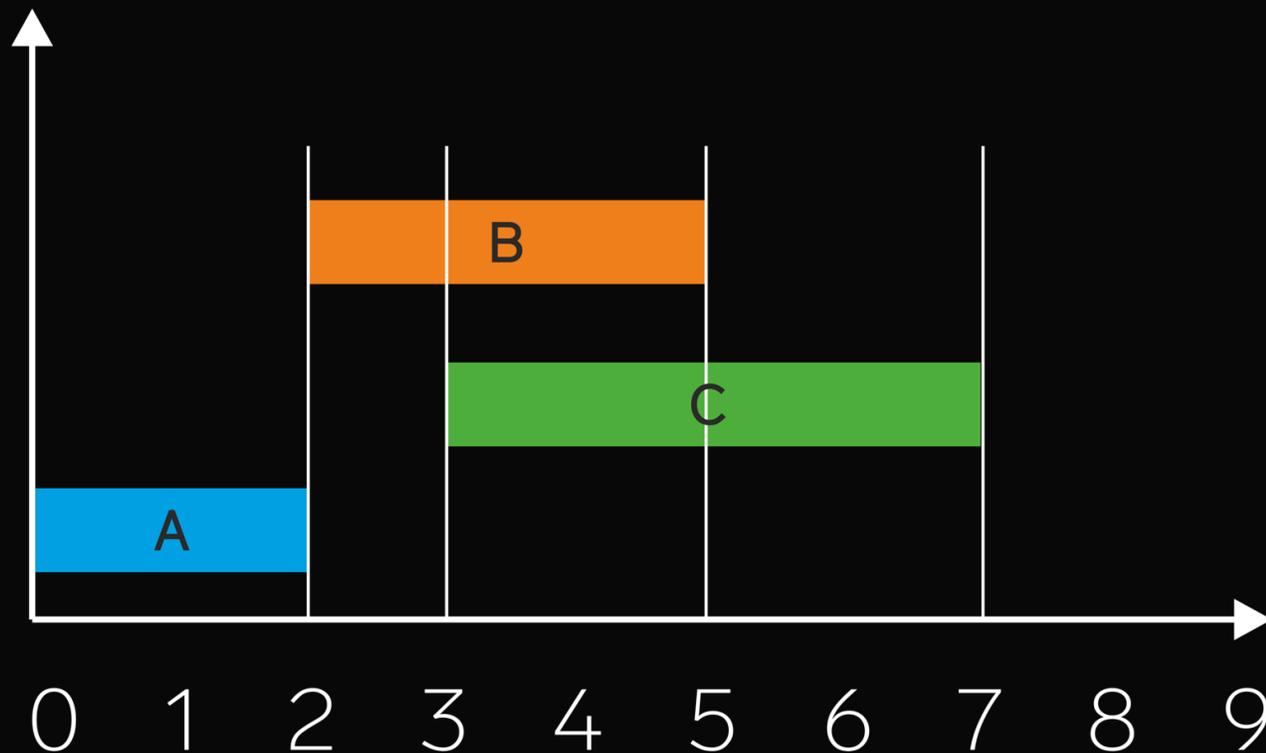
...

И операции над ними



# Или то же самое, но другими словами

Входные данные:  $[[0, 2], [2, 5], [3, 7]]$

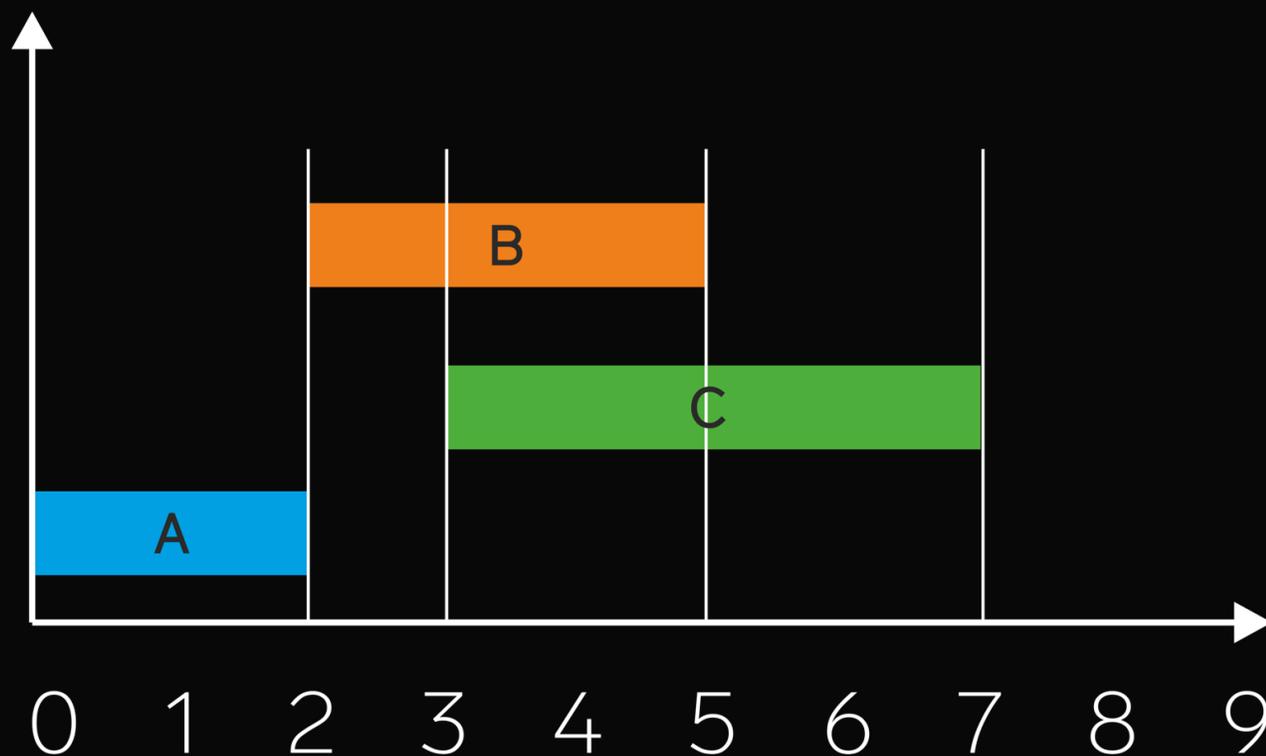


# Или то же самое, но другими словами

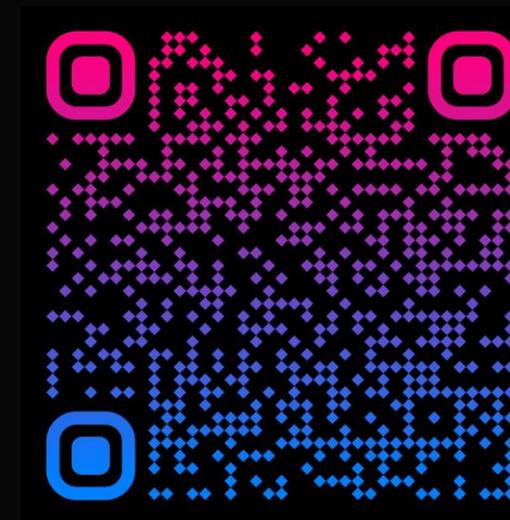
Входные данные:  $[[0, 2], [2, 5], [3, 7]]$

А и В =  $[[ ]]$

В и С =  $[[3, 5]]$



Логическое И

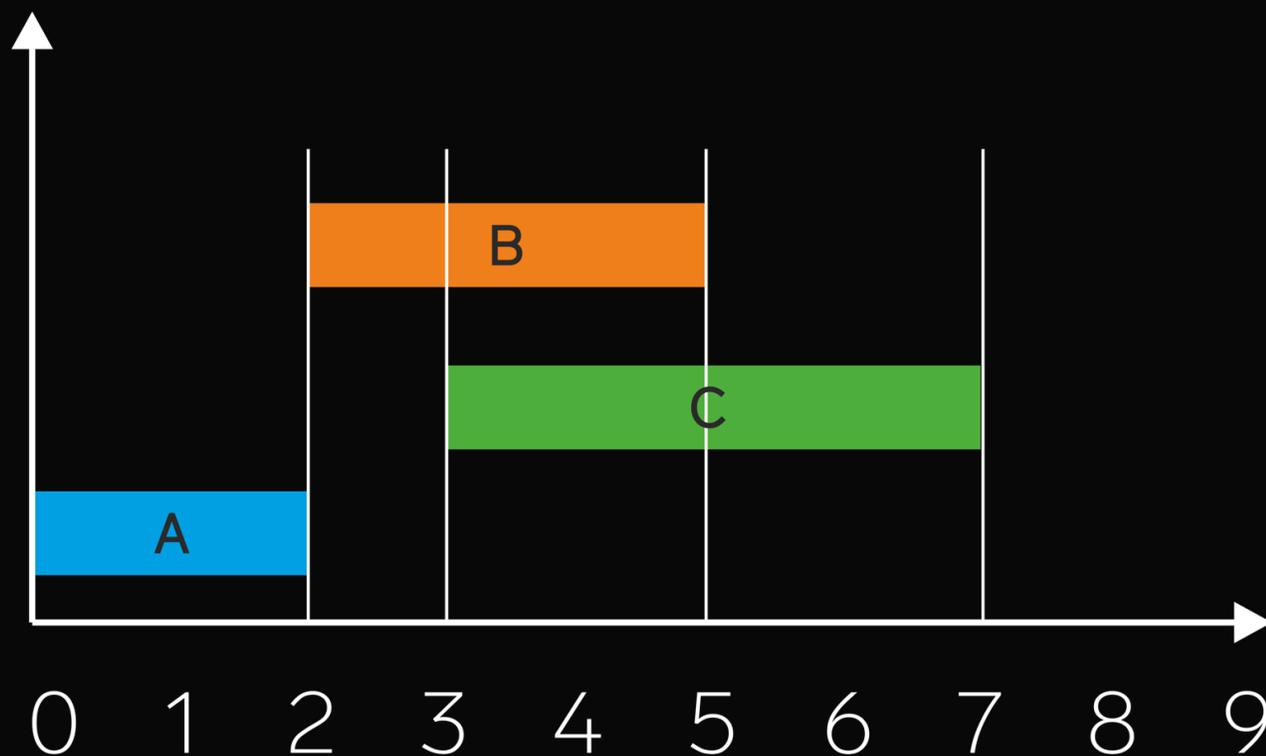


# Или то же самое, но другими словами

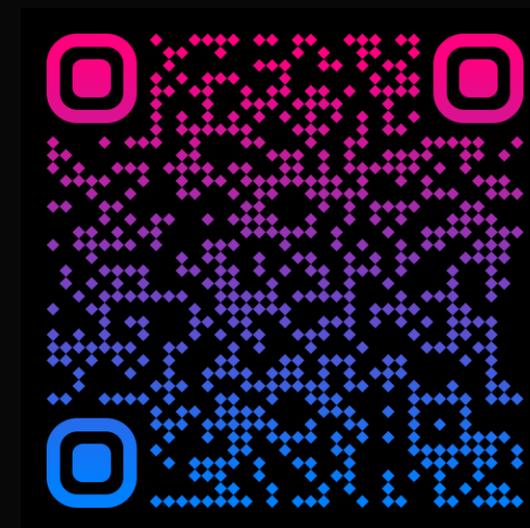
Входные данные:  $[[0, 2], [2, 5], [3, 7]]$

А или В =  $[[0, 5]]$

В или С =  $[[2, 7]]$



Логическое ИЛИ



# Как выглядела проблема в коде

- Никаких цепляющих глаз действий, которые имеют большую O-сложность
- Немного некрасивая работа с Pydantic первой версии
- И.. И всё?

Наверное, в такие моменты появляются мифы, что Python медленный

```
while self_iter < len(self.__root__) and other_iter < len(other.__root__):
    if other[other_iter].start <= self.__root__[self_iter].end and \
        self.__root__[self_iter].start <= other[other_iter].end:

        left = max(self.__root__[self_iter].start, other[other_iter].start)
        right = min(self.__root__[self_iter].end, other[other_iter].end)

        if left == right:
            joint.append(IpPort.parse_obj(left))
        else:
            joint.append(IpPortRange.parse_obj([left, right]))

    if self.__root__[self_iter].end > other[other_iter].end:
        other_iter += 1
    else:
        self_iter += 1

return IpPorts.parse_obj(joint)
```

# Недостатки данного кода

- Постоянно пересоздаём объекты
- Мы работаем с Pydantic, а значит, каждое создание объекта – это ещё и валидация
- Внутри валидации спрятан важный для алгоритма код, от которого зависит алгоритм – проверка на сортировку. С высокой O-сложностью

```
while self_iter < len(self.__root__) and other_iter < len(other.__root__):
    if other[other_iter].start <= self.__root__[self_iter].end and \
        self.__root__[self_iter].start <= other[other_iter].end:

        left = max(self.__root__[self_iter].start, other[other_iter].start)
        right = min(self.__root__[self_iter].end, other[other_iter].end)

        if left == right:
            joint.append(IpPort.parse_obj(left))
        else:
            joint.append(IpPortRange.parse_obj([left, right]))

    if self.__root__[self_iter].end > other[other_iter].end:
        other_iter += 1
    else:
        self_iter += 1

return IpPorts.parse_obj(joint)
```

# На что нужно обратить внимание

- Нет постоянных пересозданий объектов
- Нет работы с Pydantic, а значит, нет ненужной валидации
- Вся работа, которая делалась ранее в Pydantic, теперь делается нами
- Но мы снова с не самым лучшим алгоритмом

```
while self_iter < len(self.ranges) and other_iter < len(other.ranges):
    if other.ranges[other_iter].begin <= self.ranges[self_iter].end and \
       self.ranges[self_iter].begin <= other.ranges[other_iter].end:

        left = max(self.ranges[self_iter].begin, other.ranges[other_iter].begin)
        right = min(self.ranges[self_iter].end, other.ranges[other_iter].end)

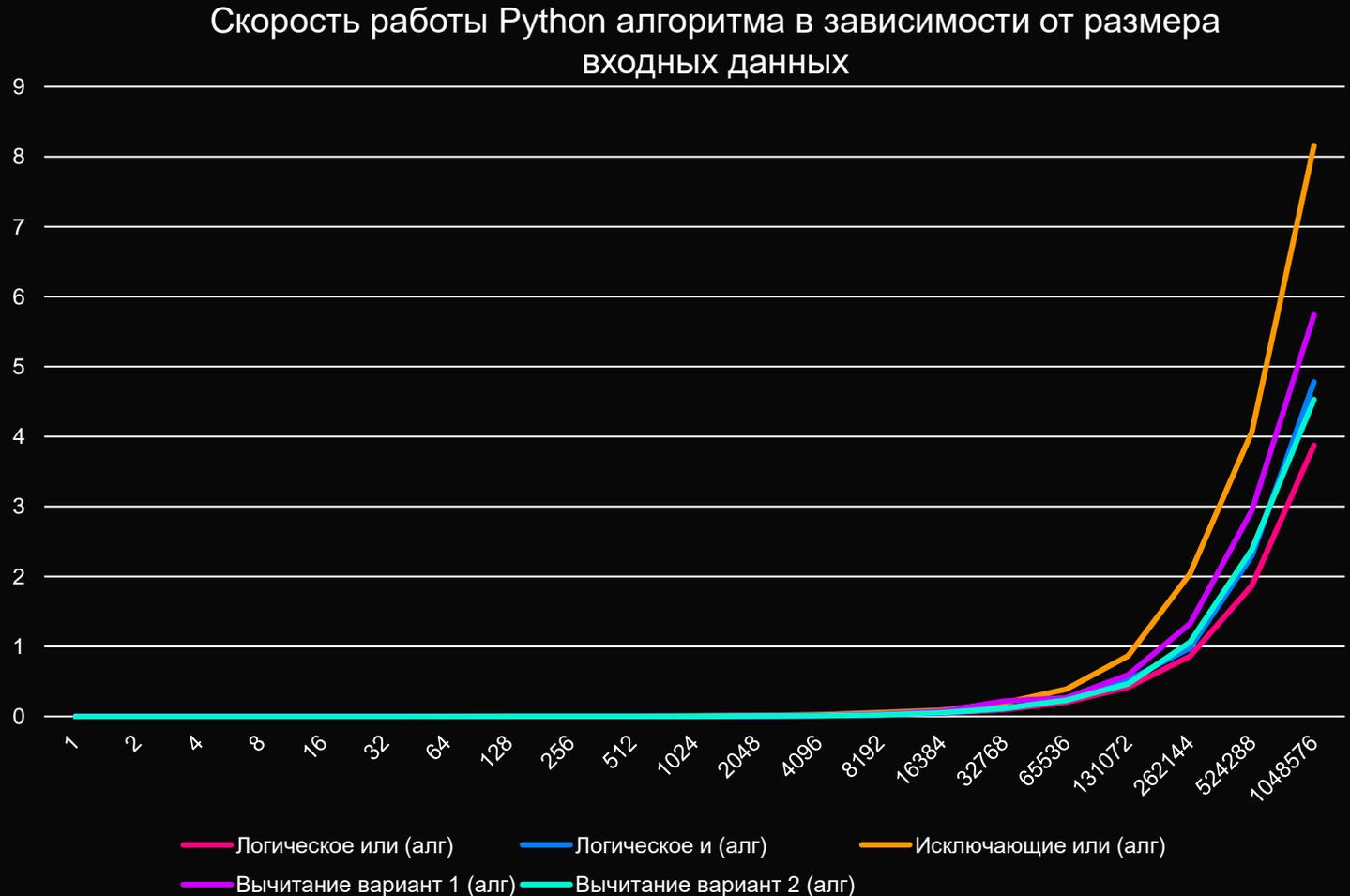
        ans.append(type_(left, right))

    if self.ranges[self_iter].end > other.ranges[other_iter].end:
        other_iter += 1
    else:
        self_iter += 1

return LogicalOperations(ans, type_=type_)
```

# На что нужно обратить внимание

- Нет постоянных пересозданий объектов
- Нет работы с Pydantic, а значит, нет ненужной валидации
- Вся работа, которая делалась ранее в Pydantic, теперь делается нами

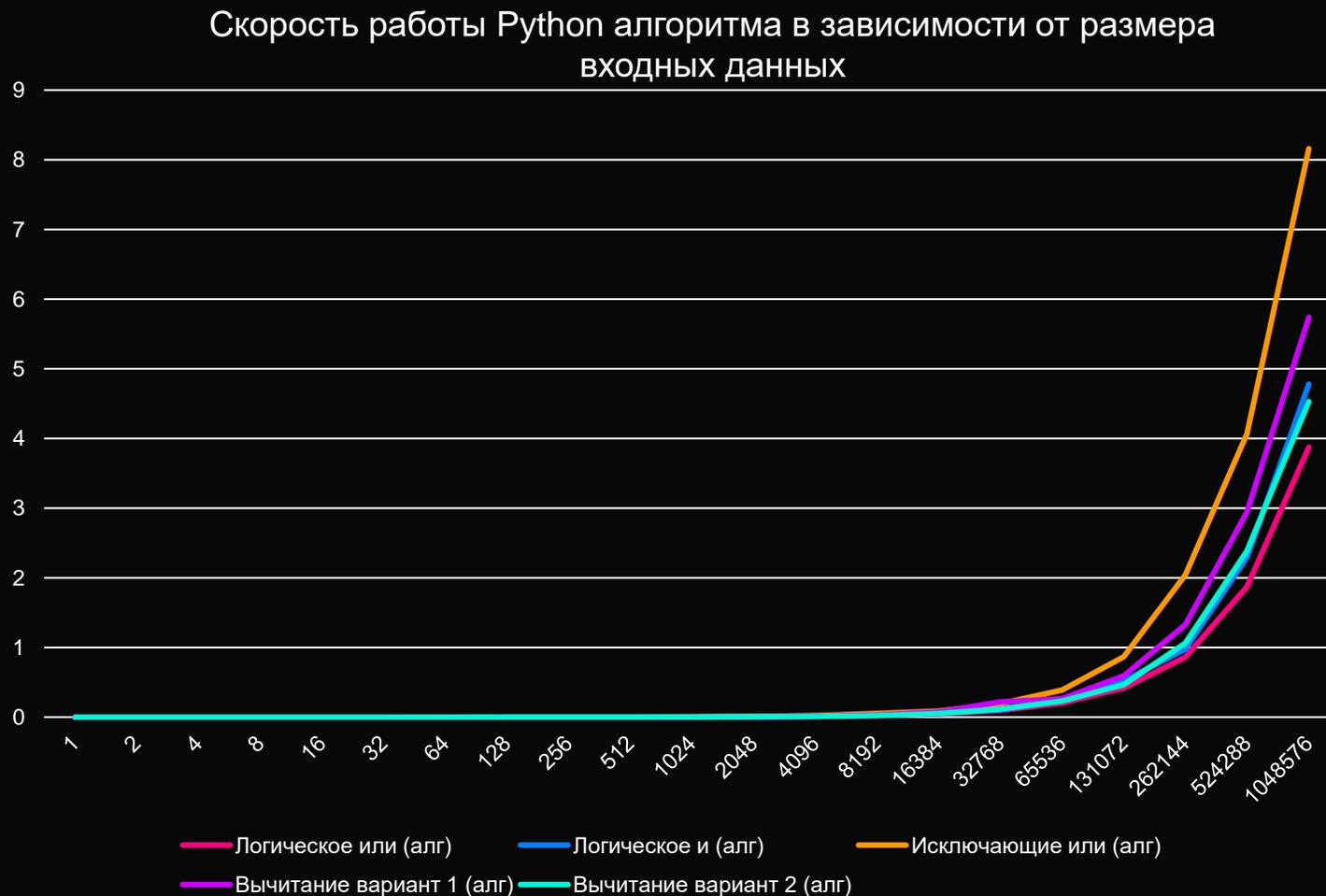


# За что мы боремся в итоге?

- Наша цель – успеть всё за 1с

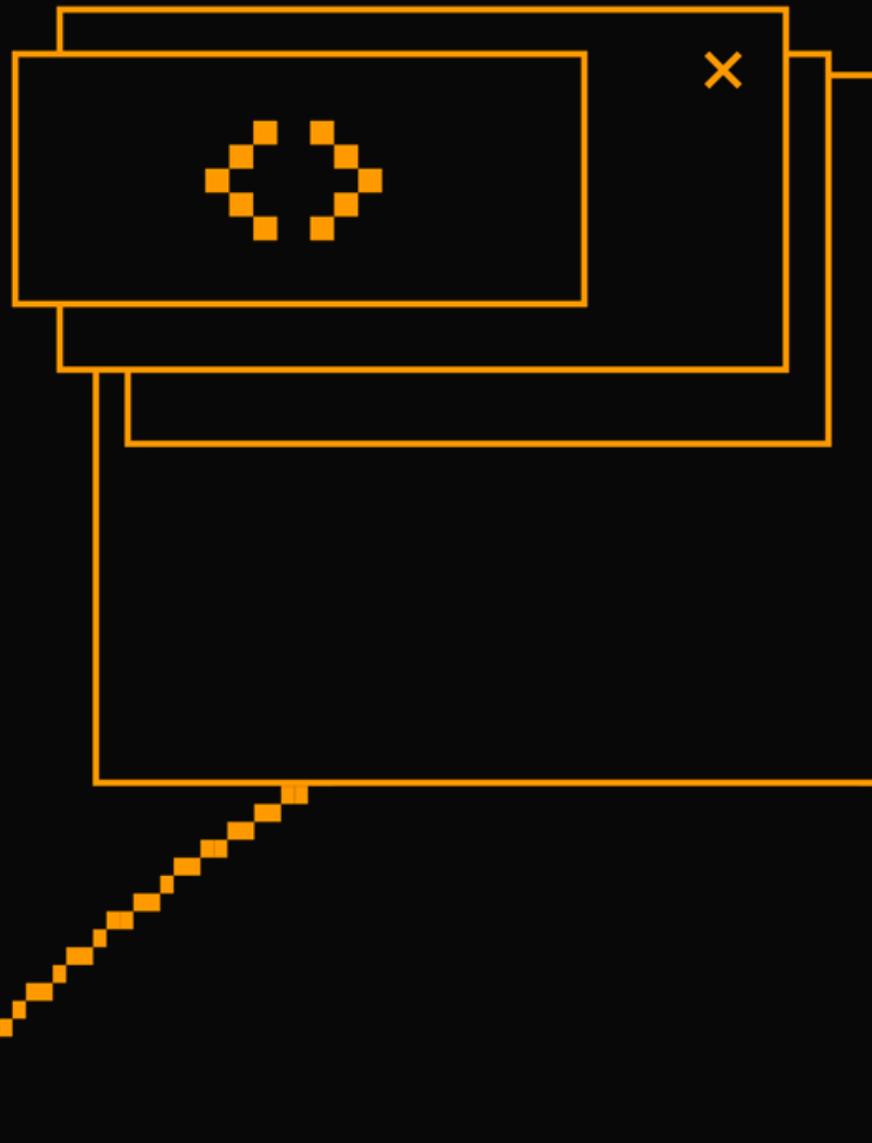
Легенда:

- Ось Y – время в секундах
- Ось X – размер входных данных
- Максимальное значение 1048576 пар IP:Port в двух множествах
- Или, другими словами, 32 Мб данных



- 1
- 2
- 3
- 4

# Анализ



# Как можно ускорить решение любой задачи?

- Использовать алгоритмы
- Использовать подходящие структуры данных



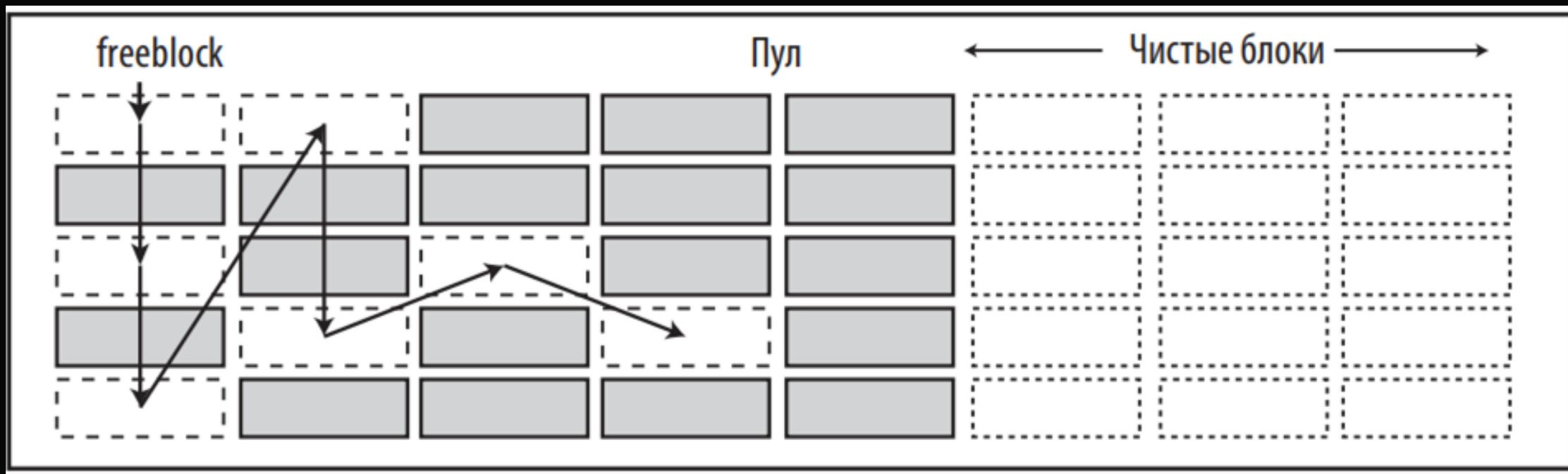
# Как устроена память на Python?



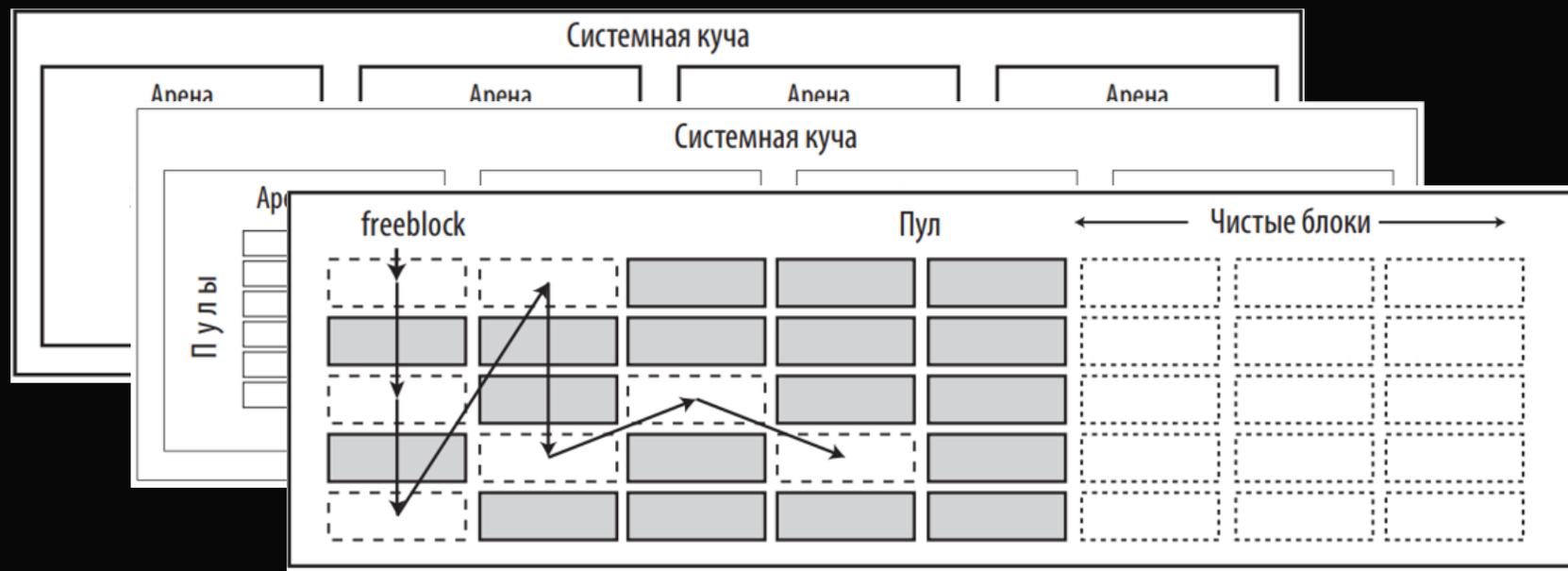
# Как устроена память на Python?



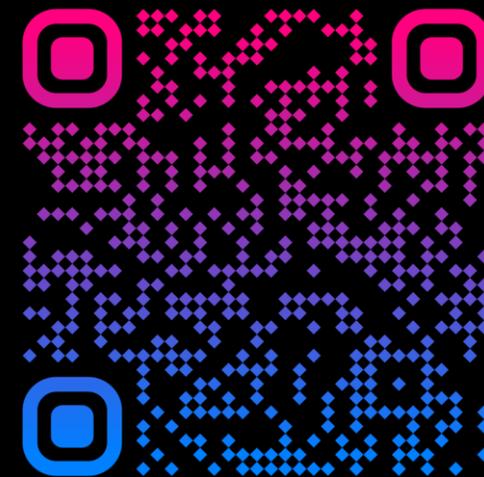
# Как устроена память на Python?



# Как устроена память на Python?



Про память  
в CPython



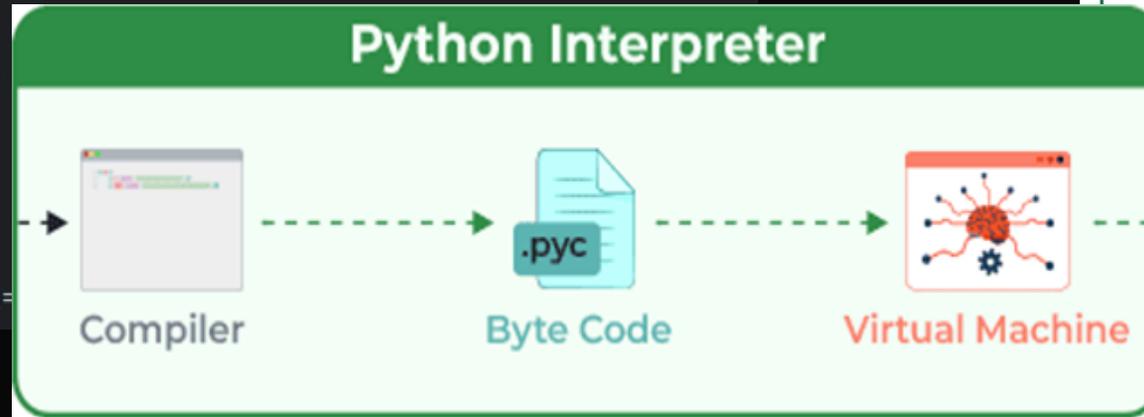
# Какие свойствами обладает память Python?

- Позволяет быстро создавать и удалять объекты
- Позволяет быстро обращаться к произвольным данным по ссылке на память в пуле
- Эффективно утилизирует память
- Хранит данные фрагментировано
- Изолирована от платформы

# Какие ещё особенности Python влияют на производительность?

```
while self_iter < len(self.ranges) and other_iter < len(other.ranges):  
    if other.ranges[other_iter].begin <= self.ranges[self_iter].end and \  
        self.ranges[self_iter].begin <= other.ranges[other_iter].end:  
  
        left = max(self.ranges[self_iter].begin, other.ranges[other_iter].begin)  
        right = min(self.ranges[self_iter].end, other.ranges[other_iter].end)  
  
        ans.append(type_(left, right))  
  
    if self.ranges[self_iter].end >  
        other_iter += 1  
    else:  
        self_iter += 1  
  
return LogicalOperations(ans, type_=  

```



```
sub    sp, sp, #32  
str    x0, [sp,24]  
str    x1, [sp,16]  
str    w2, [sp,12]  
ldr    x0, [sp,24]  
ldrb   w0, [x0]  
asr    w0, w0, 2  
adrp   x1, cb64  
add    x1, x1, :lo12:cb64  
sxtw   x0, w0  
ldrb   w1, [x1,x0]  
ldr    x0, [sp,16]  
strb   w1, [x0]  
ldr    x0, [sp,16]  
add    x0, x0, 1  
ldr    x1, [sp,24]  
ldrb   w1, [x1]  
and    w1, w1, 3  
lsl    w2, w1, 4  
ldr    x1, [sp,24]
```

# Что в итоге нужно учесть создавая алгоритмы на Python?

С точки зрения дизайна языка:

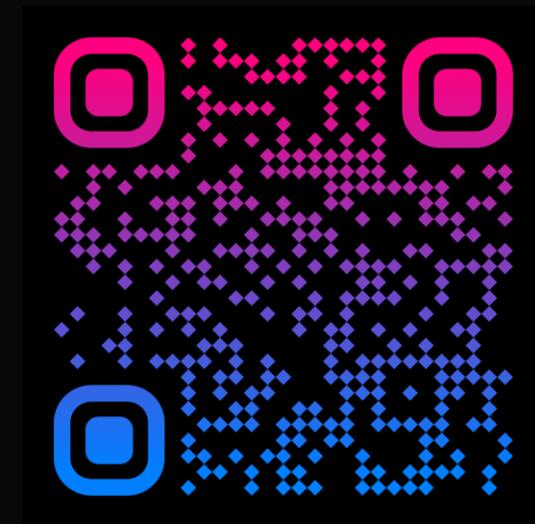
- Мы ограничены в выборе инструментов: быстрым созданием и удалением данных, а так же обменом указателей на данные
- А так же небольшими объёмами предвыделенной памяти. Так как не обязательно, что аллокатор будет успевать отдавать запрашиваемые объёмы памяти

# Что в итоге нужно учесть создавая алгоритмы на Python?

С точки зрения взаимодействия с железом:

- Невозможность пользоваться аппаратными возможностями для разгона кода. Например нельзя воспользоваться выравниванием памяти из наличия виртуальной машины
- И из-за неё же, мы не можем взаимодействовать с процессором напрямую. Например через 3DNow, AVX, SSE и т.д.

Что такое конвейер процессора

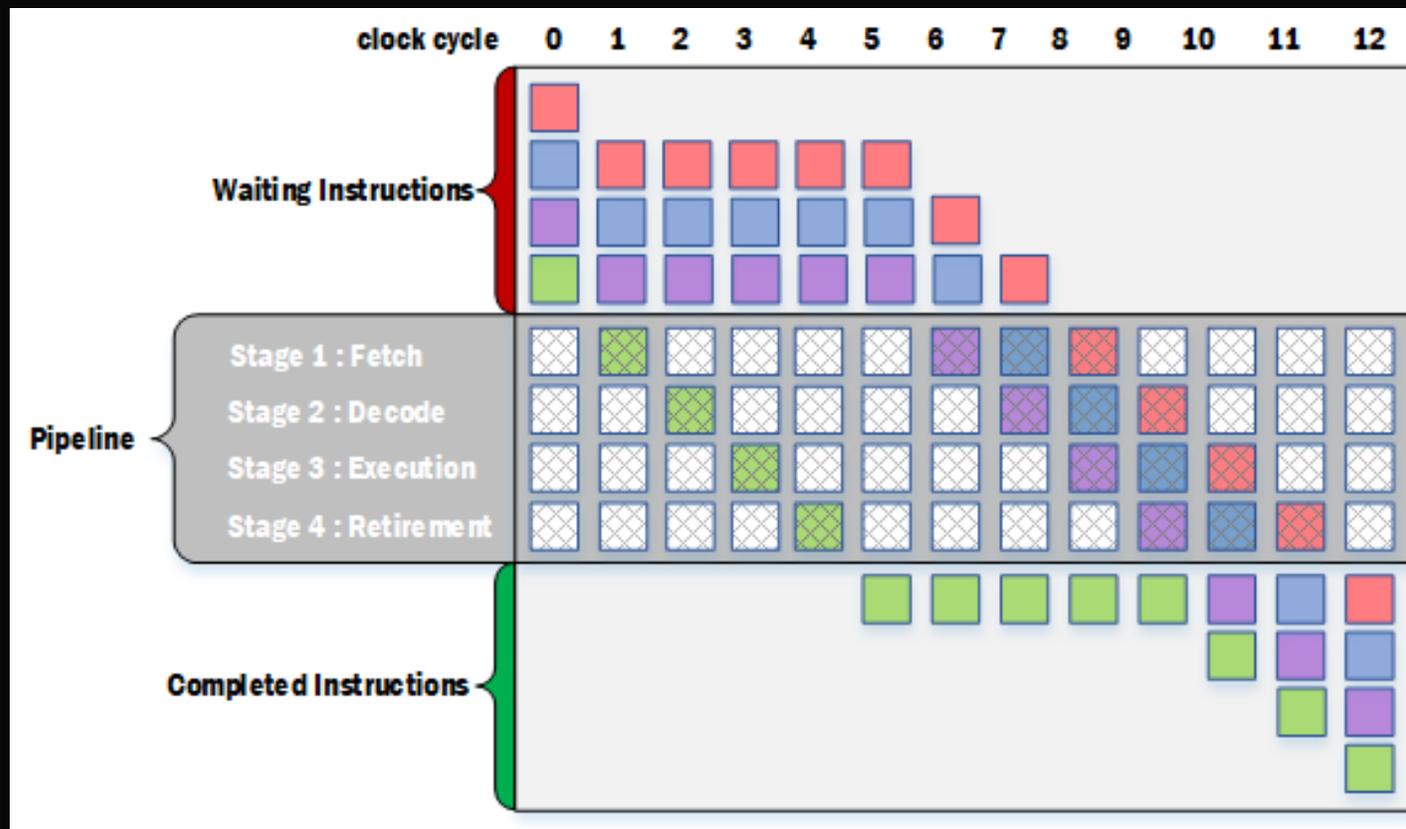


# Немного про конвейер и оценку производительности

IPC он же Instructions Per Cycle это  
главная характеристика  
эффективности выполнения кода  
на CPU

Применим знания к картинке:

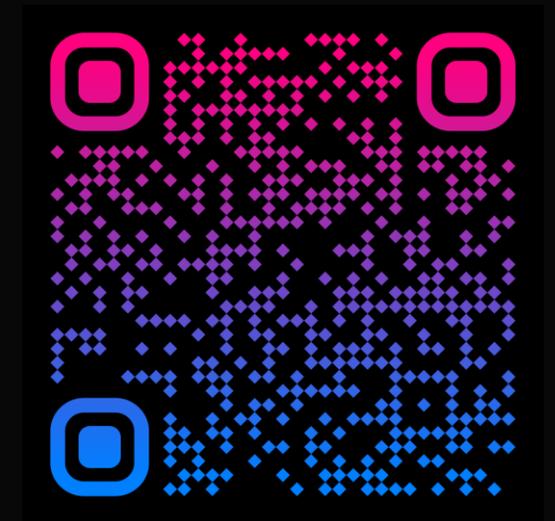
- IPC равно 0.33
- Или 4 инструкции за 12 циклов



# Что мешает конвейеру CPU разогнаться?

- Branch misprediction
- Pipeline stalls
- Input/Output
- Cache misses

О branch miss prediction  
в нашем блоге



# А для чего нужно знать про конвейр?

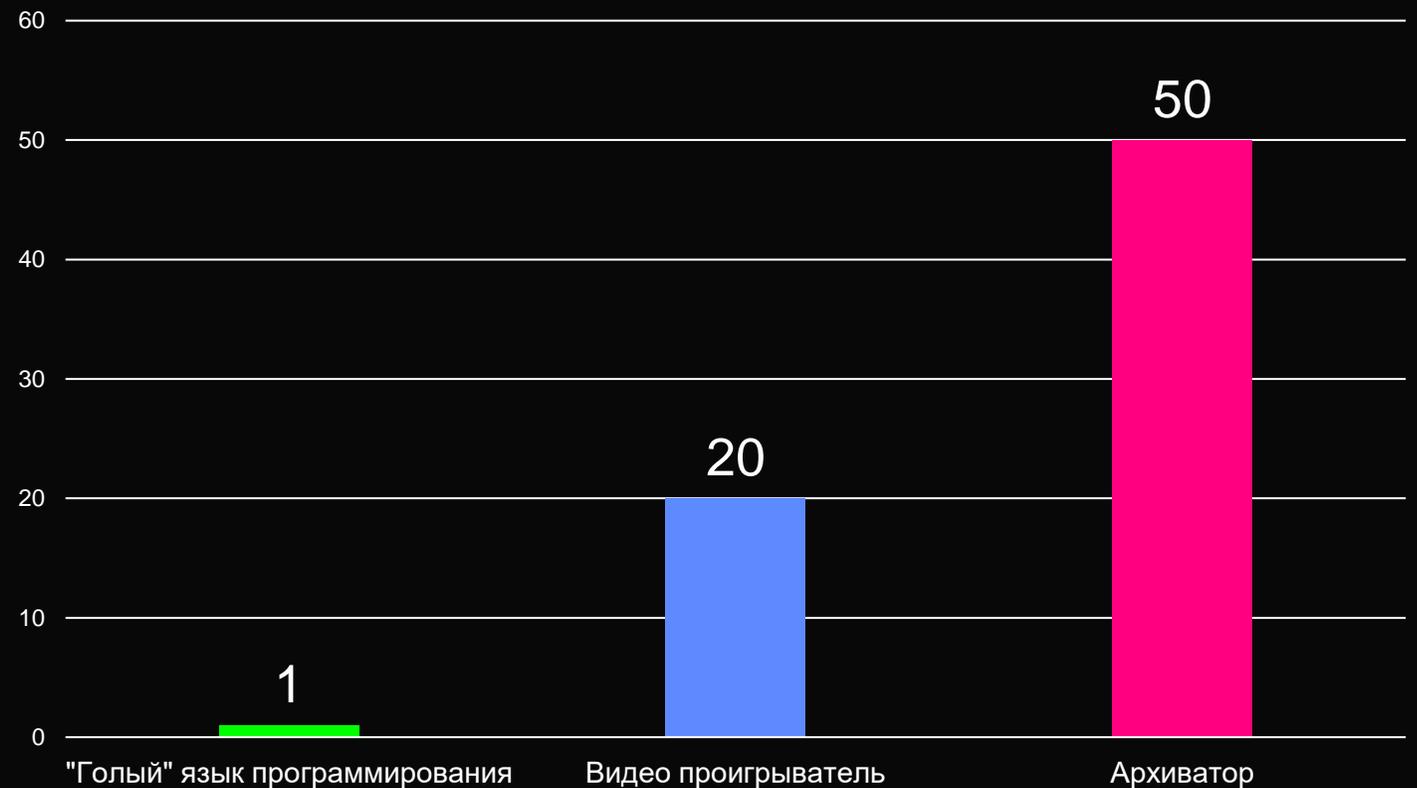
IPC при использовании расширений процессора



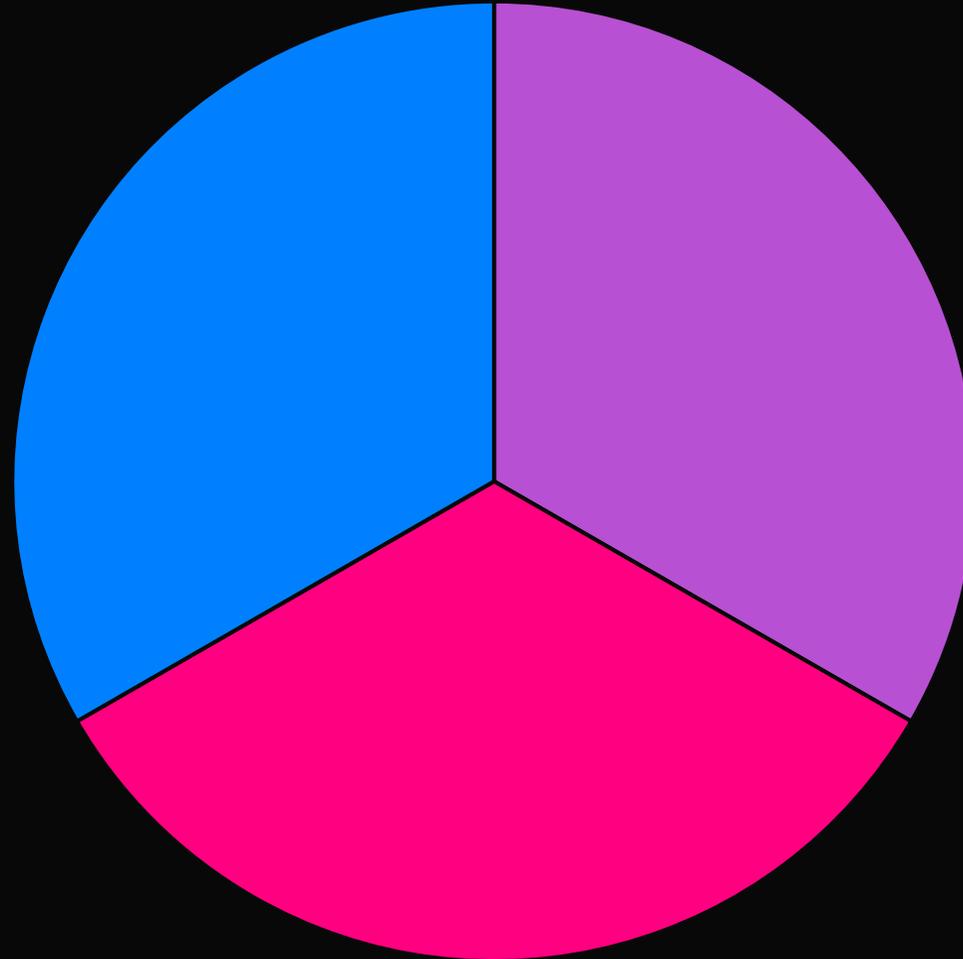
# И как использовать данные знания?

- Работа в один поток
- Использование обычных инструкций процессора
- Или...
- Если мы просто пишем код на каком либо язык, то получим зелёную колонку

IPC при использовании расширений процессора



# Что мешает разогнаться Python?



■ Виртуальная машина ■ Абстрагирование от железа ■ Ошибки конвейера CPU

А что, если бы в Python была возможность работать с данными, хранящимся непрерывно и без виртуальной машины?

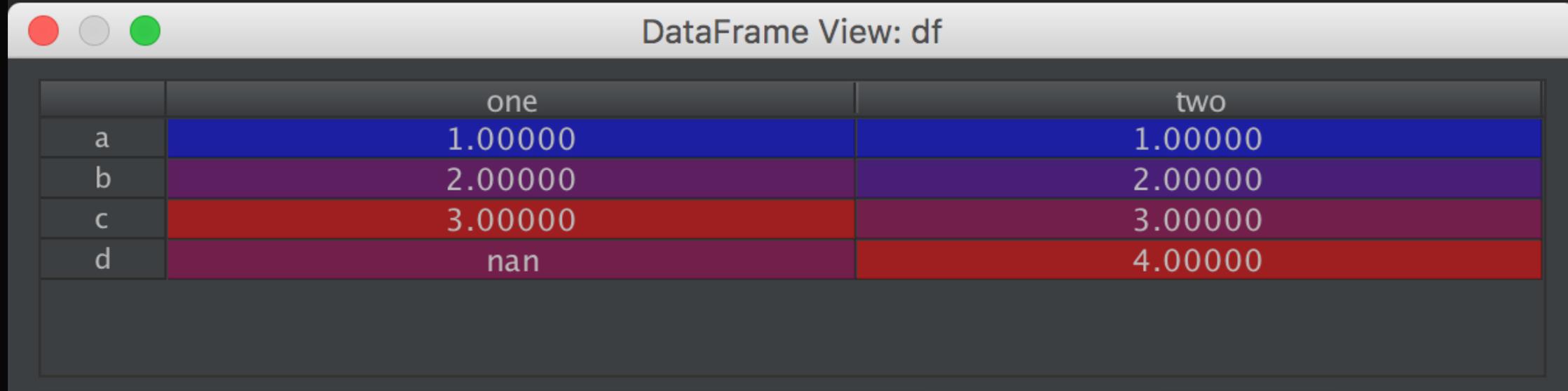
# Возможность разумеется есть

```
import pandas as pd

s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
     'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
```



The screenshot shows a window titled "DataFrame View: df" with a table containing the following data:

|   | one     | two     |
|---|---------|---------|
| a | 1.00000 | 1.00000 |
| b | 2.00000 | 2.00000 |
| c | 3.00000 | 3.00000 |
| d | nan     | 4.00000 |

# Решение

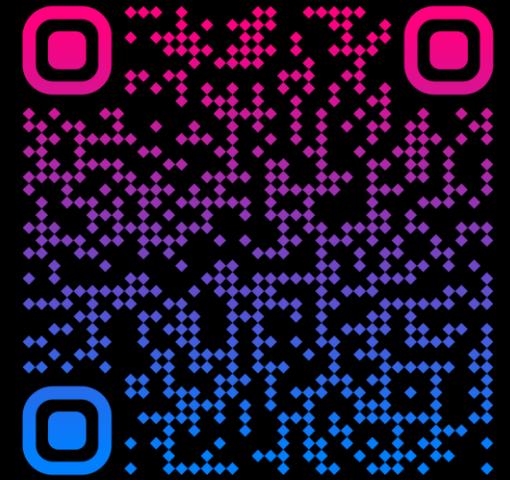
```
def __and__(self, other): # izh
    if isinstance(other, LogicalOperationsNP):
        intervals = np.concatenate([x for x in (self.ranges, other.ranges)], axis=0)
        intervals.sort(axis=0)

        intervals = pd.DataFrame(intervals, columns=['start', 'end'])
        intervals['intersects'] = (intervals.end + 1 - intervals.start.shift(-1)) > 0
        intervals['start'] = intervals['start'].shift(-1)

        ans = intervals[(intervals['intersects'] == True)][['start', 'end']] # noqa

        return LogicalOperationsNP(ans.values.astype('int64'))
    else:
        return NotImplemented
```

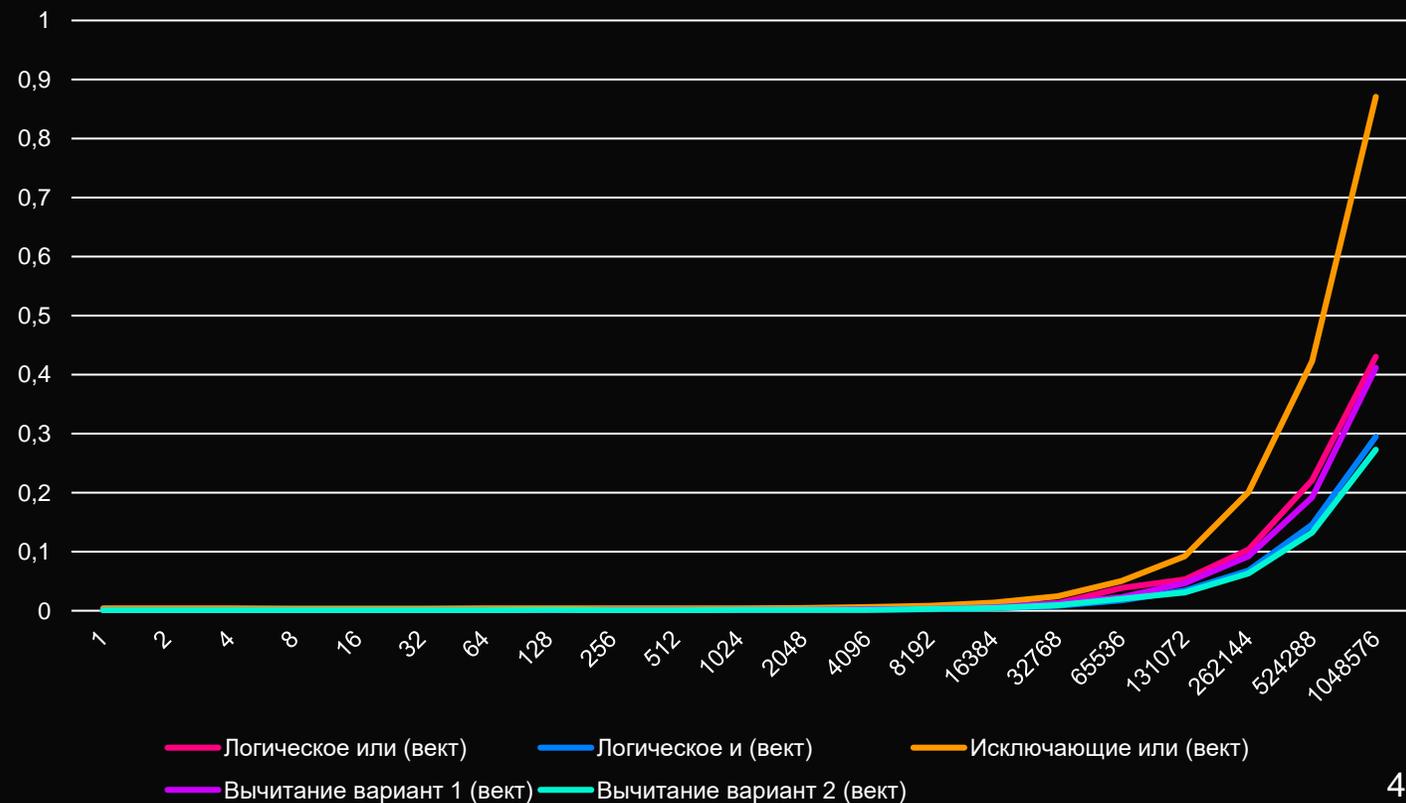
Код на Python



# Что с производительностью?

- 10% нагрузки мы считаем  $< 0,05$  с, а раньше сервис падал на ней
- 100% нагрузки за  $\sim 0,9$  секунд
- Цель успеть всё за 1с достигнута

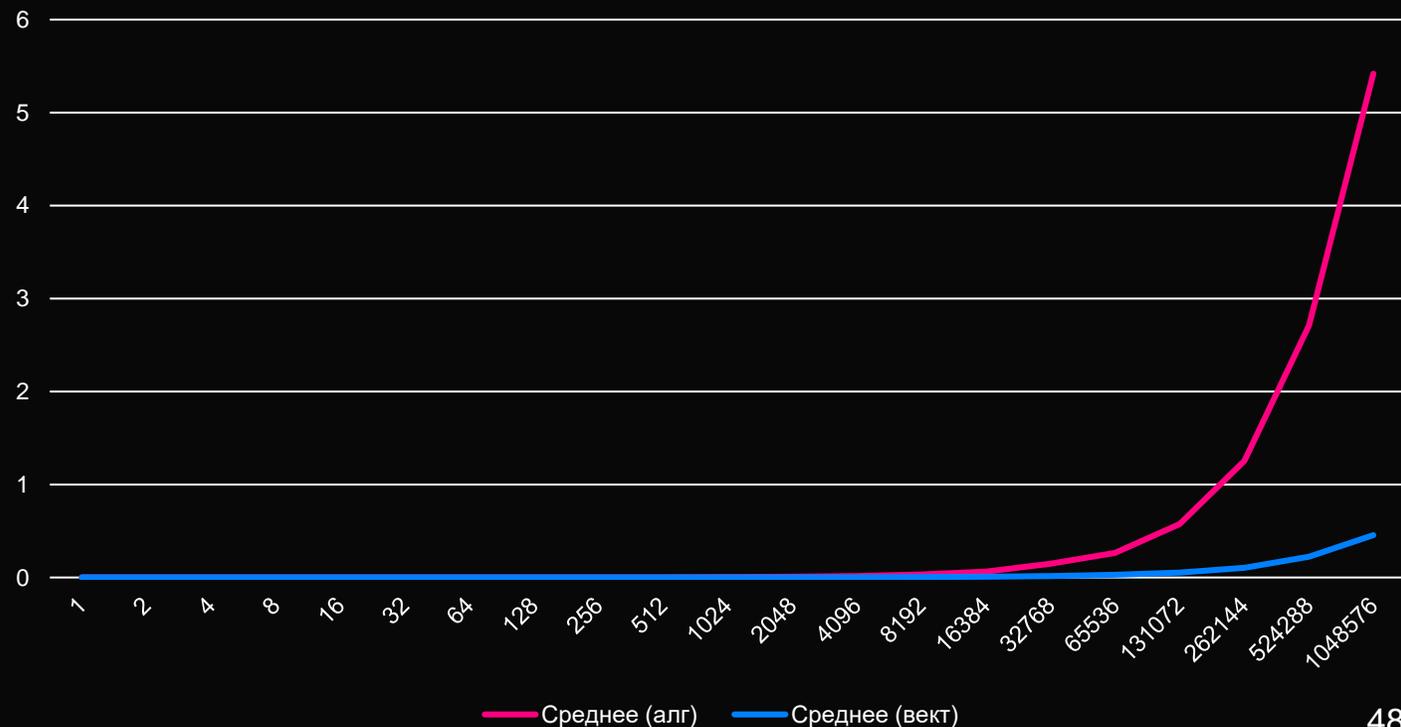
Скорость работы "векторизованного" Python алгоритма в зависимости от размера входных данных



# Какие выводы?

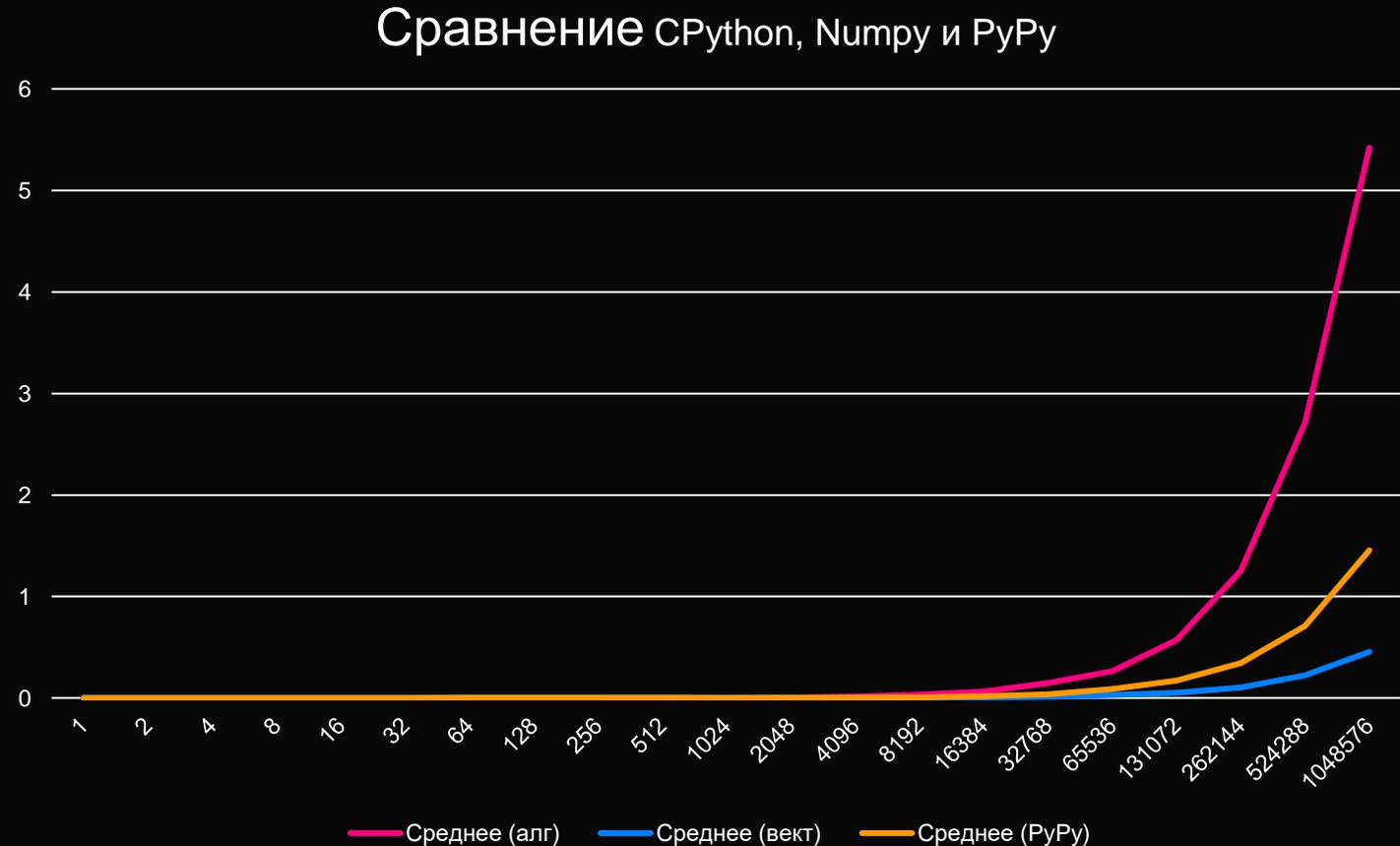
- Если поменять способ хранения данных в памяти, Python превращается в числодробилку
- Увеличение производительности больше, чем на порядок реально
- Особенно, если наши данные уместятся в кеше CPU

Сравнение средних показателей обычного и векторизованного алгоритмов



# А если бы я использовал PyPy?

- PyPy не работает с NumPy. Поэтому добавился только один график
- Но одинаковый алгоритм PyPy щёлкает быстрее чем CPython



# Смоделируем ситуацию #1

...когда мы переписали проект или модуль на другой стек, например на C++ с помощью питониста который когда то немного знал этот язык:

- Код почти полностью повторяет код на Python, но не делает ничего лишнего (помним о связывании рук за спиной)

```
LogicalOperations<T> operator&&(const LogicalOperations<T>& other_ranges) const {
    auto own_index = 0;
    auto other_index = 0;

    vector<T> ans;
    while(own_index < ranges.size() and other_index < other_ranges.ranges.size()) {
        if(other_ranges.ranges[other_index].begin <= ranges[own_index].end and
           ranges[own_index].begin <= other_ranges.ranges[other_index].end) {
            auto left = max(ranges[own_index].begin, other_ranges.ranges[other_index].begin);
            auto right = min(ranges[own_index].end, other_ranges.ranges[other_index].end);

            ans.emplace_back(T(left, right));
        }

        if(ranges[own_index].end > other_ranges.ranges[other_index].end) {
            other_index++;
        } else {
            own_index++;
        }
    }

    return std::move(LogicalOperations<T>(std::move(ans)));
}
```

# Смоделируем ситуацию #2

...а так же мы переписали код на Rust силами интересующегося языком питониста:

- И он тоже постарался написать его так, чтобы он повторил алгоритм и не делал ничего лишнего
- И да, он получился ну почти один в один такой же

```
impl BitAnd for Op {
    type Output = Self;
    fn bitand(self, rhs: Self) -> Self::Output {
        let mut ranges: Vec<Box<dyn Range>> = vec![];
        let range_of = self.get_range_of();

        let mut l_cnt: usize = 0;
        let mut r_cnt: usize = 0;

        while l_cnt < self.ranges.len() && r_cnt < rhs.ranges.len() {
            if rhs.ranges[r_cnt].begin() <= self.ranges[l_cnt].end()
                && self.ranges[l_cnt].begin() <= rhs.ranges[r_cnt].end() {
                let left: u64 = max(self.ranges[l_cnt].begin(), rhs.ranges[r_cnt].begin());
                let right: u64 = min(self.ranges[l_cnt].end(), rhs.ranges[r_cnt].end());
                ranges.push(build_from(range_of.clone(), left, right));
            }

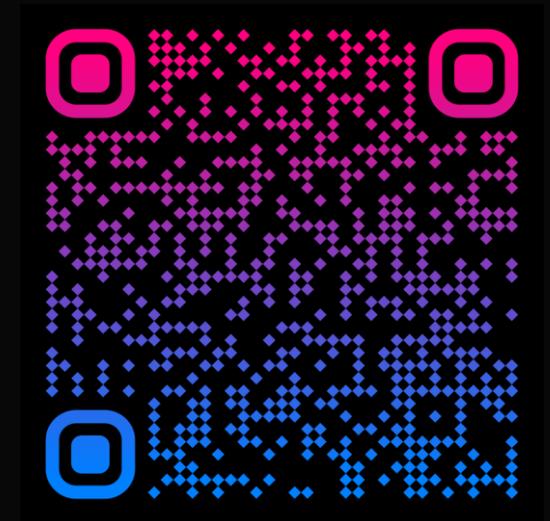
            if self.ranges[l_cnt].end() > rhs.ranges[r_cnt].end() {
                r_cnt += 1;
            } else {
                l_cnt += 1;
            }
        }

        Op {ranges, range_of: Some(range_of)}
    }
}
```

# Как же мы его писали?

- Я потратил много времени на повторение материала, изучил неизвестные мне темы, сильно углубил то что было известно
- Мой коллега, помогший с кодом на Rust, тоже не знает пройдёт ли его код ревью опытного разработчика
- При этом решение на питру мне далось примерно в два раза быстрее чем на C++.
- Будем честны, предущий пункт это не показатель, мы все разные и с разной скоростью разбираемся с разными технологиями
- Но что важно, это то, что я не уверен, что код корректный. Так как его не ревьювили опытные разработчики. А значит, будет не совсем корректно нести переписанный модуль в прод

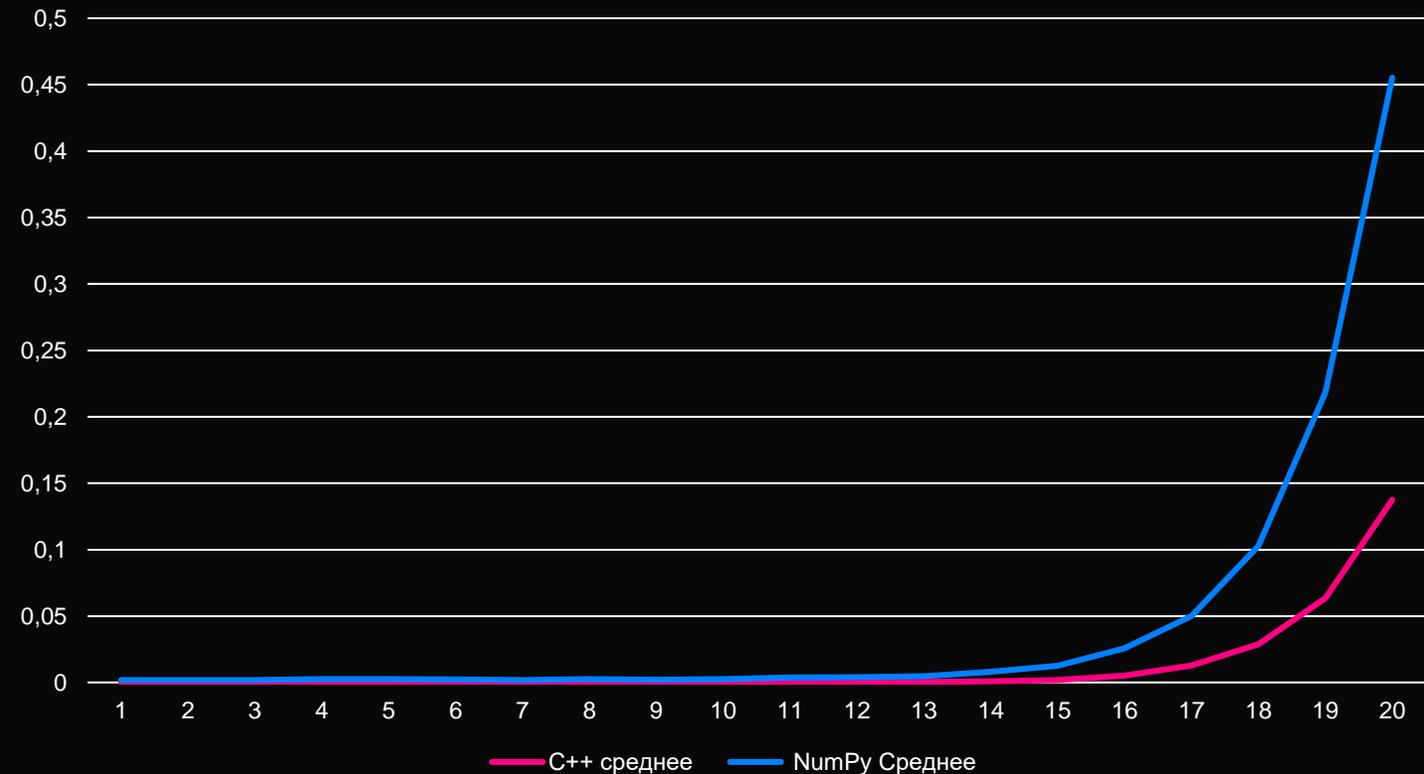
Репозиторий  
с кодом



# Что мы получили с точки зрения производительности?

- C++ быстрее Python в 3 раза
- Но в C++ данные хранятся в том виде в котором рассчитываются, а в Python есть конвертация из Pydantic. Гандикап не в пользу Python

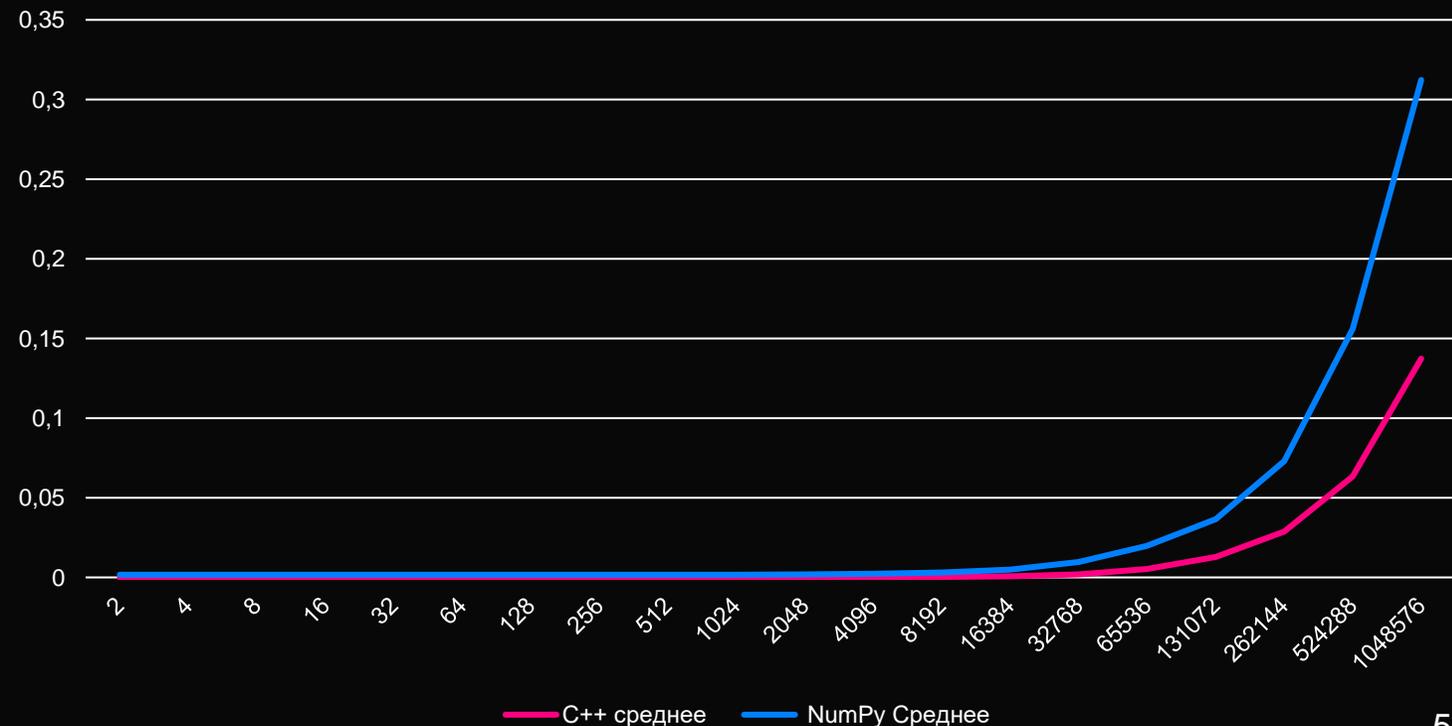
## Сравнение NumPy и C++



# Производительность, если сравнивать только обработку данных

- Без гандикапа C++ быстрее Python всё-таки только в 2 раза

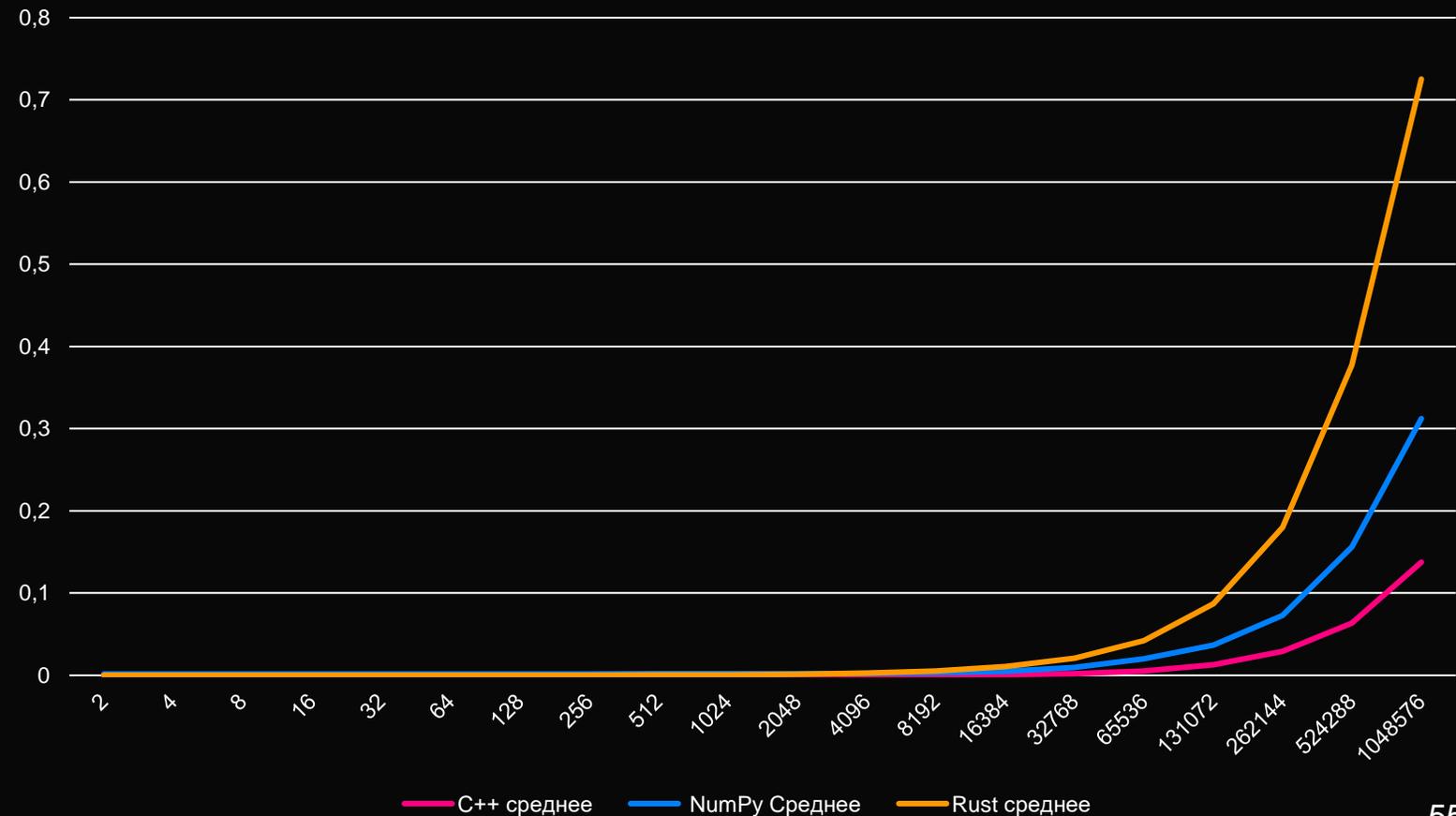
Сравниваем NumPy без конвертации из Pydantic и C++



# Производительность в сравнении с Rust

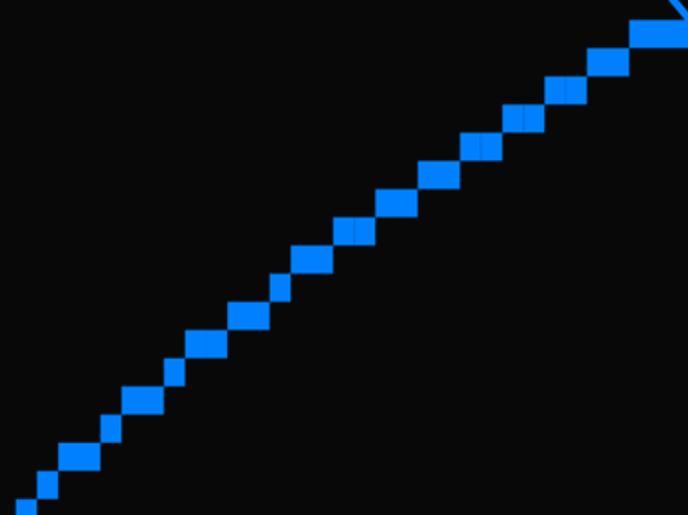
- Rust оказался медленнее Python более чем в 2 раза...
- Но возможно там есть ошибки

Сравниваем NumPy, C++ и Rust



- 1
- 2
- 3
- 4

# Выводы



# Для чего сравнили языки?

- Потому что Python якобы не может в CPU-bound задачи и проигрывает другим языкам в производительности
- В нашей команде возникло предложение переписать модуль на другой стек
- А ещё потому что неоднократно слышали истории про отказ от Python в виду его медлительности
- И потому что никогда в этих историях не фигурировала работа с расширениями процессоров, как и у нас



# Так может ли Python в CPU-bound задачи?

- Да, но только если сам разработчик умеет «готовить» CPU-bound задачи. Сам по себе язык это инструмент, который с разной эффективностью помогает решать те или иные задачи
- Но Python не догонит Си в скорости выполнения. В прочем как и Си никогда не догонит Python во многих других аспектах

# Стоит ли переходить с Python для решения CPU-bound задач?

- Если вы хотите пользоваться только конструкциями языка то нет. Прирост будет, но стоит ли жертвовать достоинствами языка?
- Скорее всего нет, если у вас есть IO в пайплайне обработки данных. Это как Феррари в пробке. Она всё равно движется со скоростью потока из-за светофоров.
- Однозначно стоит если вам нужно написать архиватор. На длительной дистанции нужно скинуть всё лишнее.

PS Из-за батареек Python иногда будет даже быстрее дефолтной реализации алгоритма на другом языке. Вспомним Rust

# Тогда как понимать фразу, что Python не может в CPU-bound?

- Из-за VM Python не сможет дать прямое управление расширениями процессора, тогда как язык без такой абстракции на это способен
- Но в батарейках всё есть. Нужно только ими воспользоваться
- И всё
- Никакого специального неотключаемого ручного тормоза для CPU в Python нет

Расширения процессоров



# И если нужна скорость и нет желания отказываться от Python

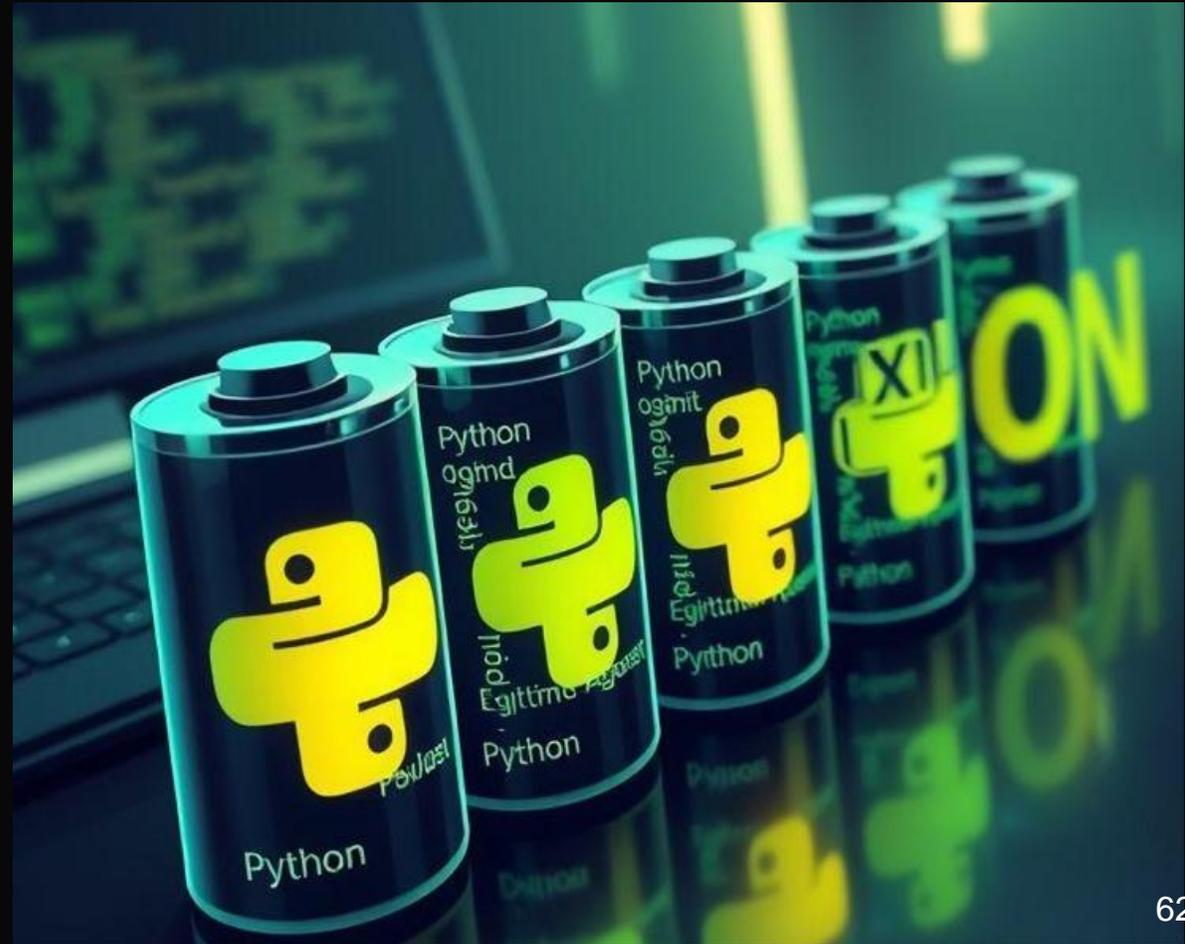
- В Python полно батареек, которые могут предоставить необходимую модель памяти
- А так же доступ к расширениям процессора
- И это может быть даже проделано под капотом той или иной батарейки



# Корректно ли считать батарейки на Си частью Python?

Как считает автор да, корректно:

- Если вам не нужно знать Си чтобы с ними работать или вам не нужен для этого разработчик
- Не все батарейки для Python написаны на Си. Код numpy написан ещё и на Fortran
- На Си (были) написаны компиляторы и интерпретаторы почти для всего, но только для Python сообщества это оказалось столь важным
- Да и сам Python написан на Си
- А теперь есть ещё и Jit который переписывает Python на Си «руками» самого Python



# А что с бенчами?

- Большинство бенчей между языками программирования, чтобы добиться «равных условий», связывают языкам руки за спиной и не используют их на полную катушку.
- Рассуждения о медлительности языков в таком контексте – вопрос только того, на сколько они хорошо чувствуют себя в тех или иных рамках.

```
from math import sqrt
import sys

def eval_A(i, j):
    return 1.0/((i+j)*(i+j+1)/2+i+1)

def eval_A_times_u(N, u, Au):
    for i in range(N):
        Au[i]=0
        for j in range(N): Au[i]+=eval_A(i,j)*u[j]

def eval_At_times_u(N, u, Au):
    for i in range(N):
        Au[i]=0
        for j in range(N): Au[i]+=eval_A(j,i)*u[j]

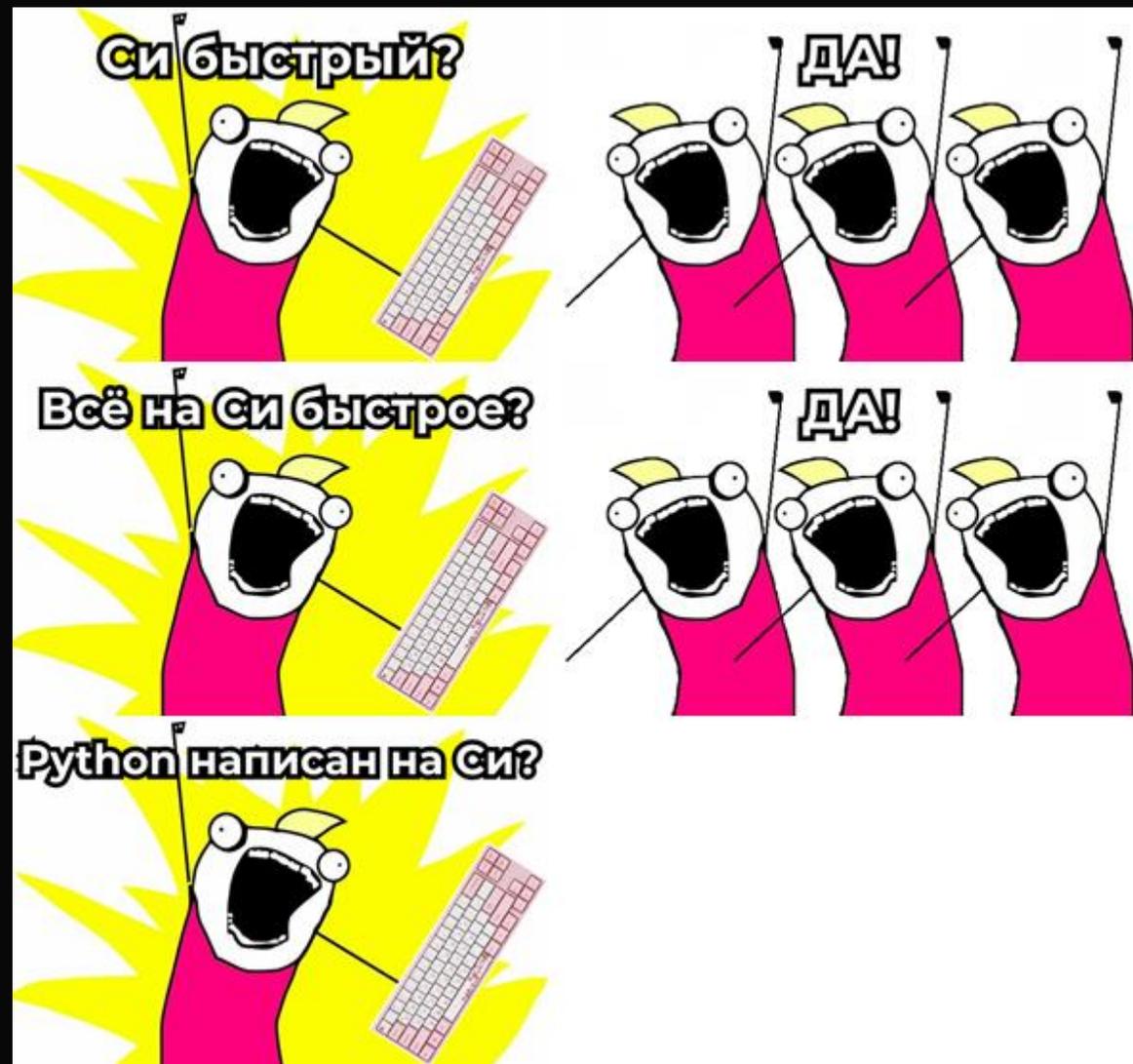
def eval_AtA_times_u(N, u, AtAu):
    v=[0]*N; eval_A_times_u(N,u,v); eval_At_times_u(N,v,AtAu)

def main(n):
    u=[1]*n
    v=[0]*n
    for i in range(10):
        eval_AtA_times_u(n,u,v)
        eval_AtA_times_u(n,v,u)
    vBv=vv=0
    for i in range(n): vBv+=u[i]*v[i]; vv+=v[i]*v[i]
    print("%.9f" % sqrt(vBv/vv))

if __name__ == '__main__':
    main( int(sys.argv[1]) if len(sys.argv) > 1 else 100 )
```

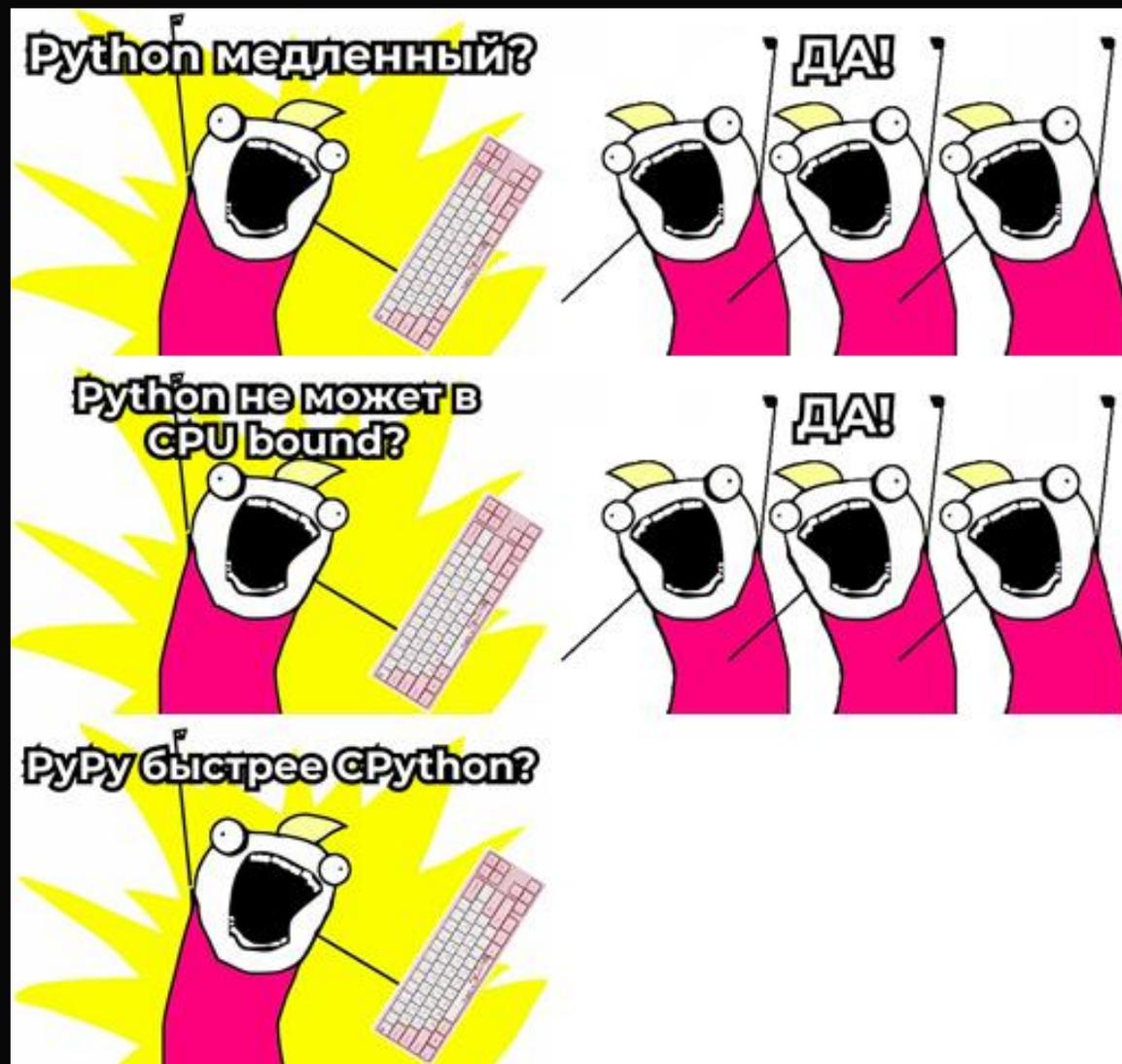
# Как подходить к написанию бенчей?

- Кажется что единственный правильный подход это подход из спидранов: any%
- Иначе получится мем



# Корректно ли сравнивать ЯП?

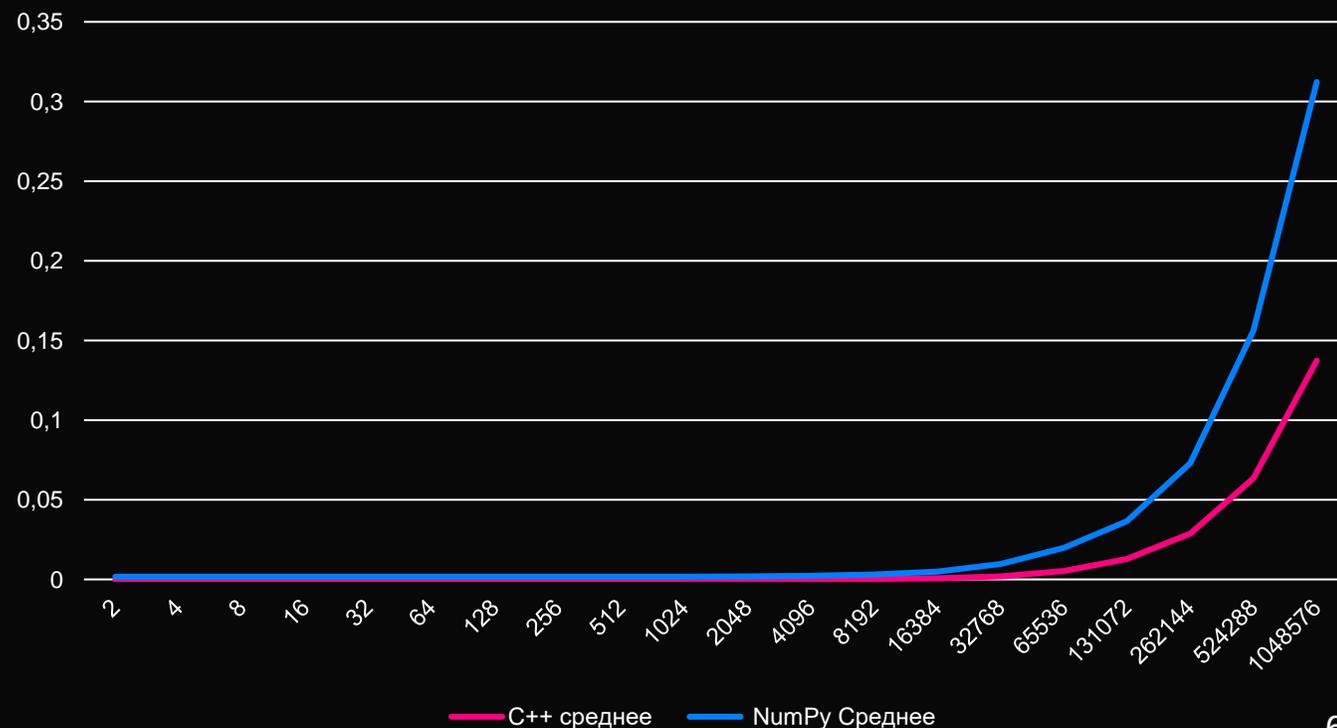
- Как мне кажется корректно, если использовать правило any%, то всё ок. Иначе снова можно получить мем
- Сравним PyPy и CPython. – видно, что написанный на Python интерпретатор быстрее написанного на Си.
- Но и это сравнение некорректно, так как PyPy частично написан на limited Си.
- Как и большинство абстрактных сравнений.
- **Call to action:** давайте смотреть на язык как на инструмент.



# Что нужно унести с этого доклада?

- Запомним этот график
- Форсим мемы
- Python быстрый

Сравниваем только работу с данными  
NumPy и C++



# Если вы где-то увидели ошибку

- В алгоритмах и коде
- В данных
- В графиках
- И т.д.

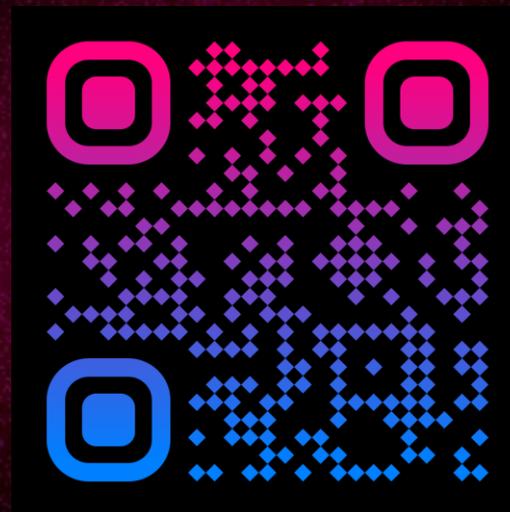
Call to action!

Дайте знать и мы это поправим, или выступите с опровержением. Инженерный путь к этому взывает 😊

Спасибо за внимание!

Контур

Технологии  
в Контуре



Спасибо за внимание!

Контур

Все ссылки

