

# Занятие 3

Профилирование и оптимизация кода на Python

# Ноутбуки занятия

Профилирование: [https://  
drive.google.com/file/d/  
1MuJ9v262ehrzJ42uN-6VPqjDsaAEQrfZ/  
view?usp=sharing](https://drive.google.com/file/d/1MuJ9v262ehrzJ42uN-6VPqjDsaAEQrfZ/view?usp=sharing)

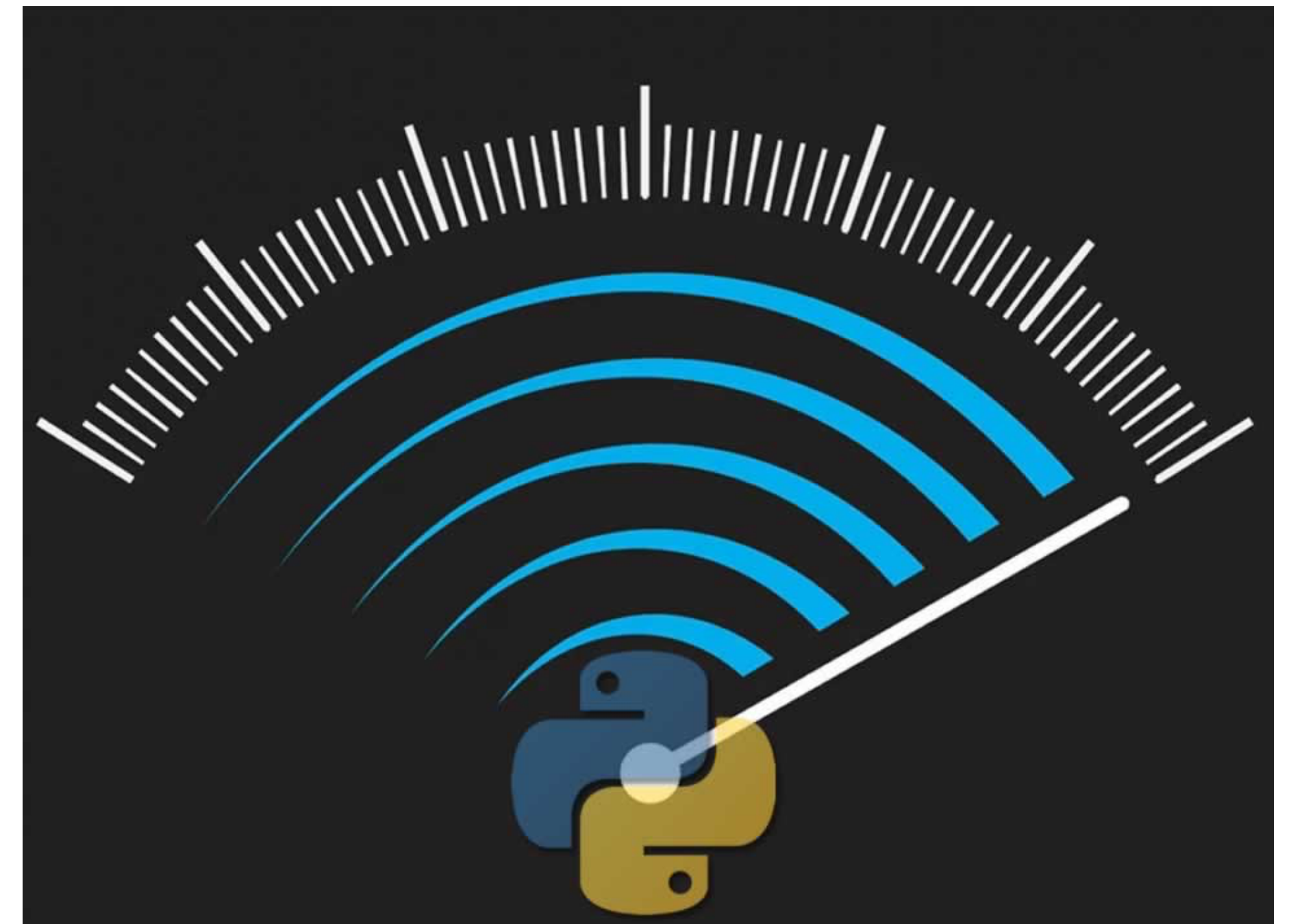
Оптимизация: [https://  
drive.google.com/file/d/  
1PkIq8yeYPSS9xhf2HC517bUFVDSVa8m2/  
view?usp=sharing](https://drive.google.com/file/d/1PkIq8yeYPSS9xhf2HC517bUFVDSVa8m2/view?usp=sharing)

# Сегодня мы научимся

- ▶ Искать `bottlenecks` — “узкие места” в коде, из-за которых код работает медленно или потребляет много памяти в моменте — при помощи `профилировщиков`.
- ▶ Одним декоратором ускорять вычисления в сотни раз (заодно разберёмся с тем, что такое JIT-компиляция и зачем она в Python).
- ▶ Распараллеливать вычисления при помощи библиотек `multiprocessing` и `joblib` (заодно поймём, почему это не всегда стоит делать).
- ▶ Эффективно решать IO-задачи при помощи “`многопоточности`” в Python (заодно поймём, почему она взята здесь в кавычки).

# План

- ▶ Python не очень быстрый.  
Почему?
- ▶ Профилирование кода на Python
- ▶ JIT-компиляция: numba, PyPy
- ▶ Параллельные вычисления в CPU-bound задачах. Библиотеки multiprocessing и joblib
- ▶ I/O-bound задачи.  
“Многопоточность” в Python.

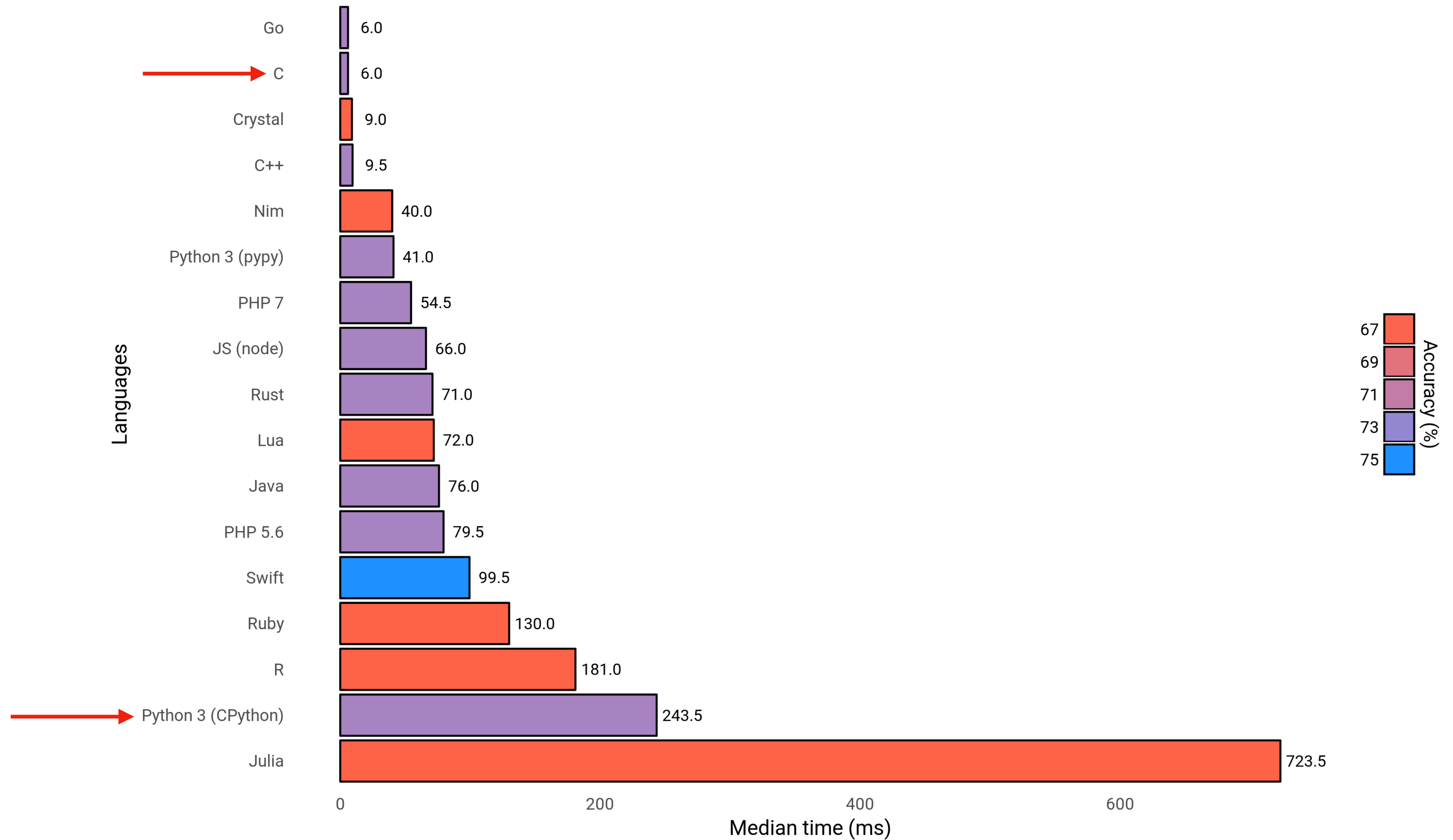


**Python не очень быстрый.  
Почему?**

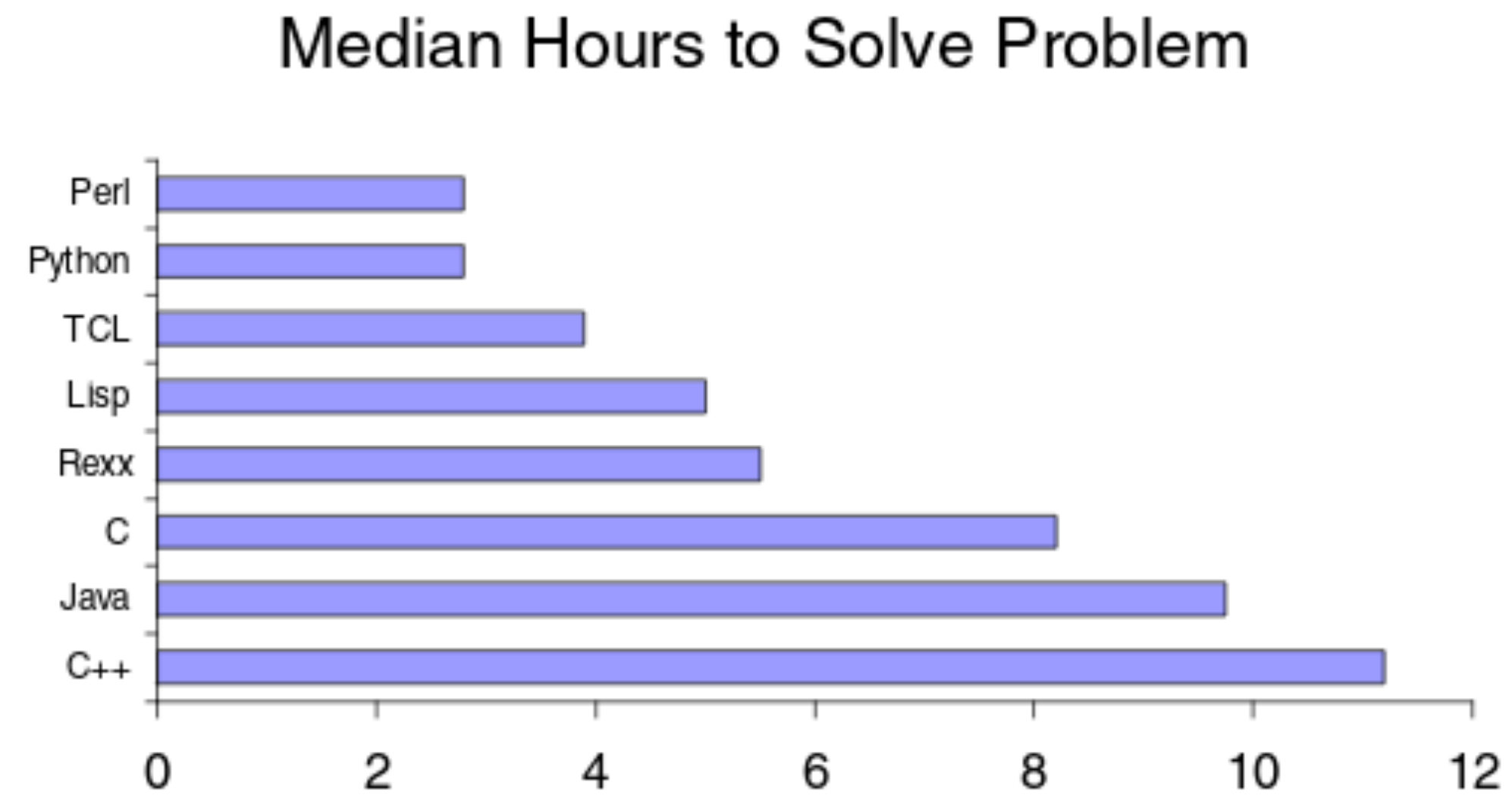
**Проблема:** Python хорош всем — удобством, лаконичностью, стандартной библиотекой, — но есть один существенный недостаток: он **очень** медленный.

# Speed comparison of various programming languages

Method: calculating  $\pi$  through the Leibniz formula x times



# Почему обычно это неважно?



Потому что Python **экономит** ваше время.



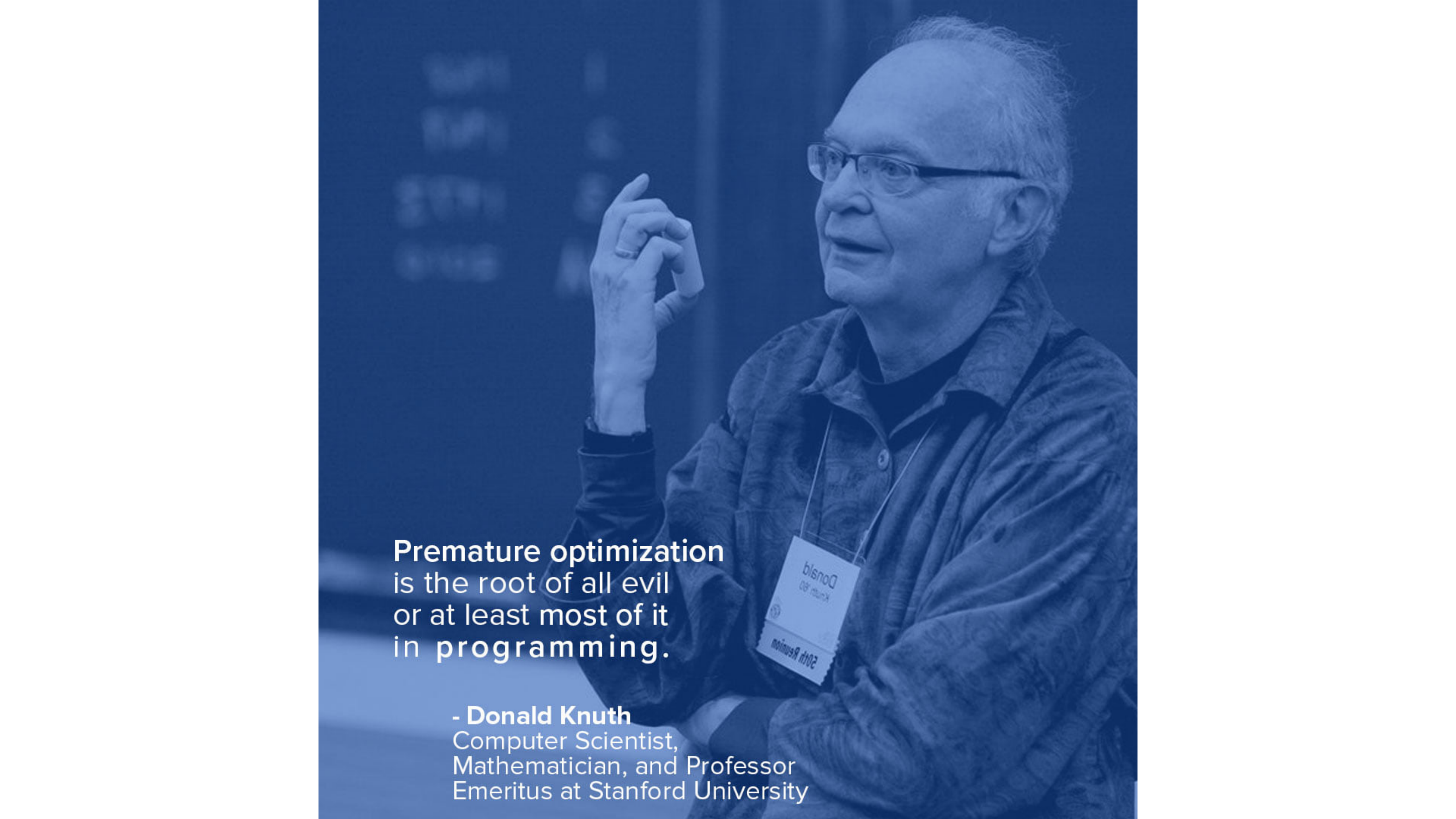
# Почему Python имеет право тормозить

- Времена, когда процессорное время и затрачиваемая память были самыми дорогими ресурсами, давно в прошлом.

Сейчас самый дорогой ресурс – **время разработчика**. Решить задачу хотя бы как-то гораздо важнее, чем решить её оптимально.

- В больших компаниях обычно используют микросервисную архитектуру: отдельные компоненты продукта обмениваются сообщениями и не зависят друг от друга.

Обычно **именно передача сообщений по сети это bottleneck**. Если принять такт процессора за секунду, то на передачу JSON-строки в среднем уходят **годы**.

A photograph of Donald Knuth, an older man with glasses, wearing a patterned jacket and a conference badge. He is gesturing with his right hand while speaking. The background is a blurred blue wall with some text. The entire image has a blue tint.

**Premature optimization**  
is the root of all evil  
or at least most of it  
in **programming**.

**- Donald Knuth**  
Computer Scientist,  
Mathematician, and Professor  
Emeritus at Stanford University

Больше аргументов в пользу того,  
что Python — замечательный язык  
программирования, можно  
прочитать на Medium:

[https://medium.com/pyslackers/  
yes-python-is-slow-and-i-dont-  
care-13763980b5a1](https://medium.com/pyslackers/yes-python-is-slow-and-i-dont-care-13763980b5a1)

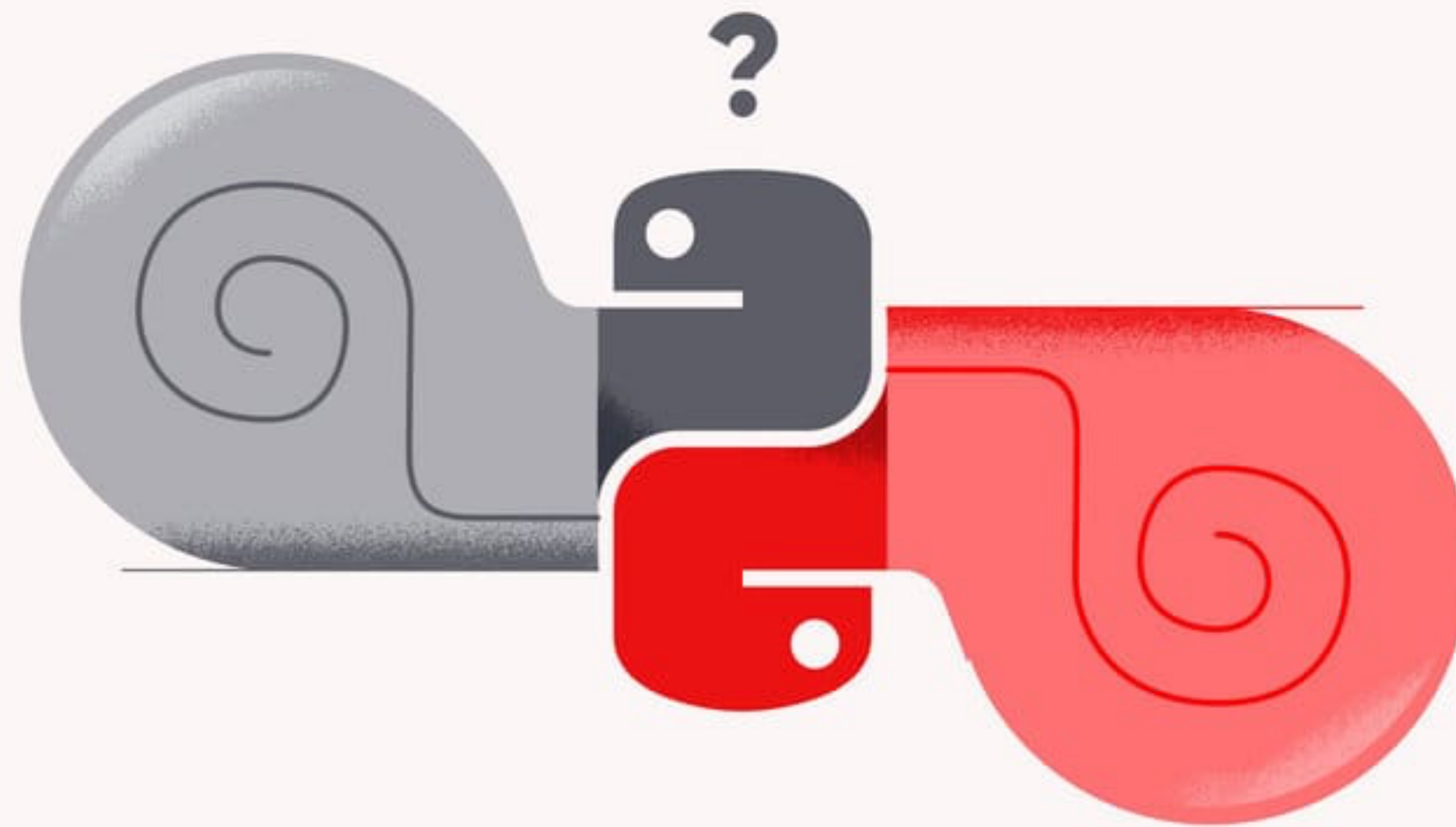
**Проблема:** ML-алгоритмы завязаны на численные методы и обработку данных, а потому очень требовательны к вычислительным ресурсам.

Более того: даже формирование признакового описания объекта может занимать много времени (больше, чем обмен сообщениями).

Поэтому в вашем случае скорость работы и используемая память имеют значение.



# Так почему же Python тормозит?



Источник: <https://www.monterail.com/blog/is-python-slow>

**Python – интерпретируемый язык программирования с динамической типизацией.**

Поэтому при выполнении любой команды приходится убеждаться её корректности. Даже в `for`- и `while`-циклах на десятки тысяч итераций. **Каждый раз.**

Python, в целом, сложно устроен. В нём даже поля и методы объектов можно менять на ходу.

Такой язык **очень трудно оптимизировать**. Поэтому интерпретатор Python никогда не сможет быть таким же быстрым, как, скажем, компилятор языка C.

Кроме того, механизм **сборки мусора** — очистки памяти устройства от ненужных данных — в Python **не может** работать в многопоточном режиме.

Более того, в Python встроен специальный механизм под названием **GIL** — **Global Interpreter Lock** — который за этим следит.

Как следствие, интерпретатор Python не может использовать базовое преимущество современных процессоров в CPU-bound задачах — их **многоядерность**.

# Что с этим делать?

- Модифицировать или заменить стандартный интерпретатор Python – `CPython` – на такой, который лучше адаптирован под оптимизацию вычислений (`PyPy`, `Jython`, `IronPython` etc). Стараться везде использовать специализированные инструменты в духе `numpy`.
- В CPU-bound задачах – запускать несколько `подпроцессов`, распределять по ним обработку данных (эдакий MapReduce). В I/O-bound задачах – использовать “потoki” (`threading`), в network-bound задачах – асинхронную обработку (`asyncio`, не затрагиваем на этом занятии).
- Реализовать вычислительно затратную часть на `C/C++` и реализовать для неё Python-интерфейс. Или просто использовать `Cython`.



Но перед тем, как начать  
что-то оптимизировать,  
давайте разберёмся, как  
определить проблему.

# Профилирование

Поиск “узких мест” в коде

# В этом разделе мы узнаем

- ▶ Что такое `bottleneck`, какие они бывают.
- ▶ Как, используя `%%time`, `%timeit` и `line_profiler`, понять, на что уходит время.
- ▶ Как, используя `memory_profiler`, понять, на что расходуются память.
- ▶ Как пользоваться профилировщиком, встроенным в PyCharm.

# Bottlenecks

- ▶ Bottleneck-ом — “бутылочным горлышком”, “узким местом” — принято называть компоненту программы, которая потребляет большую часть ресурсов (либо в моменте, либо интегрально).
- ▶ Типов узких мест столько же, сколько и ресурсов. Обычно это время, память, сеть и дисковый I/O (ввод/вывод).

Any improvements made anywhere besides the  
bottleneck are an illusion. — Gene Kim

**Важно** понимать, что оптимизация потребления одного ресурса обычно чревата **повышением** потребления какого-то другого!

Особенно это характерно для **времени** и **памяти**. Нужно оценивать поведение системы комплексно!

На этом занятии мы  
сосредоточимся на  
оптимизации скорости  
работы и расхода  
памяти как на наиболее  
актуальных для вас.

# Profiling

- ▶ **Profiling** (профилирование) — поиск узких мест в вашем коде, как правило динамический.
- ▶ Слово “**динамический**” в этом контексте означает, что **профайлер** запускает ваш код в специальном окружении и анализирует поведение вашей программы в реальном времени.
- ▶ **Статические** профайлеры существуют, но не для Python, так что в сложных ситуациях придётся полагаться исключительно на свой **опыт**.



# Пример работы профайлера

## Приближённое вычисление числа $\pi$ по методу Монте-Карло

Timer unit: 1e-06 s

Total time: 28.0353 s

File: <ipython-input-8-69dbc2813525>

Function: estimate\_pi at line 3

Line #	Hits	Time	Per Hit	% Time	Line Contents
3					def estimate_pi(n=1e7) -> "area":
4	1	7.0	7.0	0.0	in_circle = 0
5	1	1.0	1.0	0.0	total = n
6					
7	10000001	4026597.0	0.4	14.4	while n != 0:
8	10000000	4101831.0	0.4	14.6	prec_x = random()
9	10000000	4242937.0	0.4	15.1	prec_y = random()
10	10000000	8168531.0	0.8	29.1	if pow(prec_x, 2) + pow(prec_y, 2) <= 1:
11	7851624	3295817.0	0.4	11.8	in_circle += 1 # inside the circle
12	10000000	4199593.0	0.4	15.0	n -= 1
13					
14	1	2.0	2.0	0.0	return 4 * in_circle / total



Количество  
вызовов



Суммарное  
время



Доля от  
общего времени

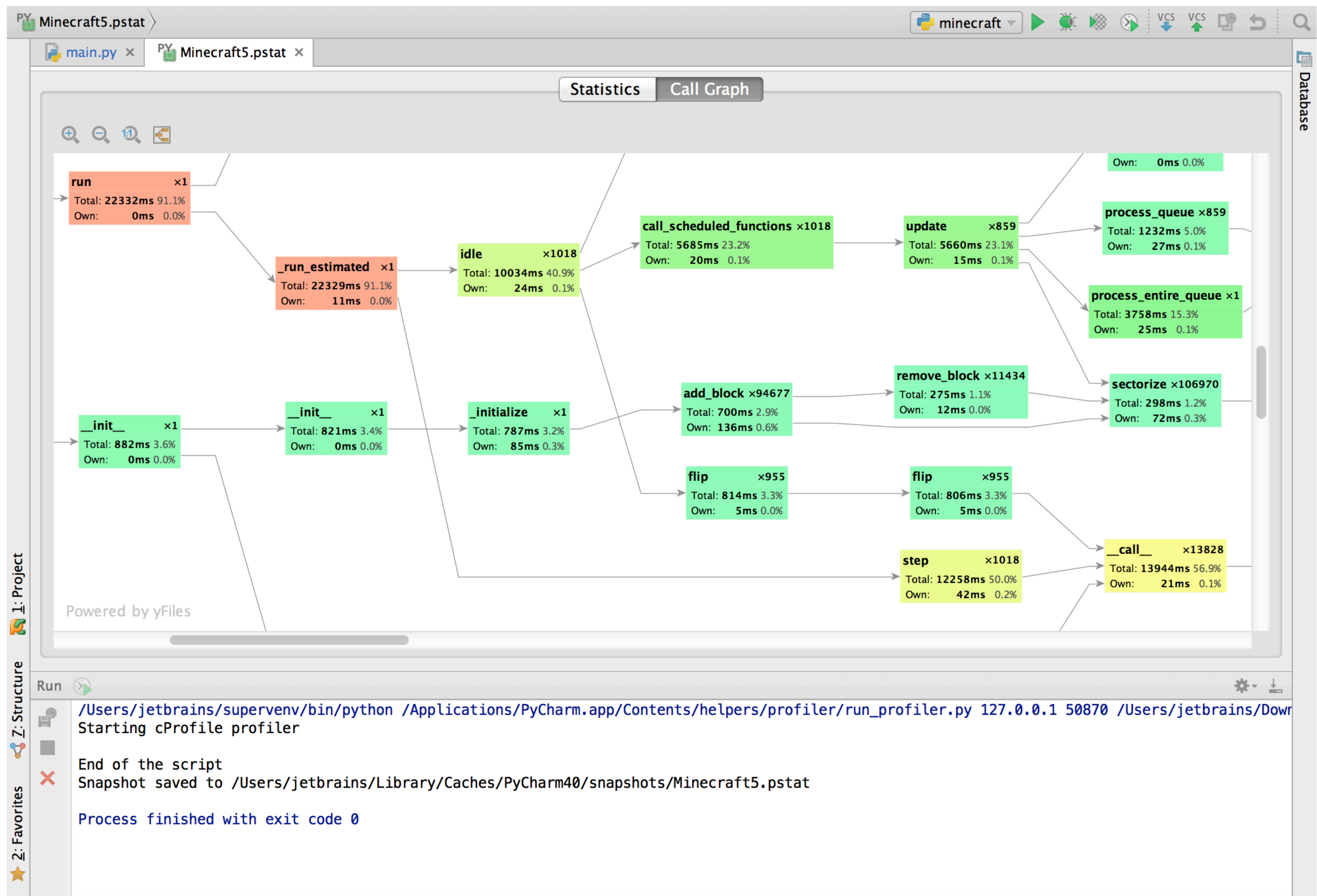
Мы будем обсуждать профайлеры в контексте **Jupyter-ноутбуков** и **консольных утилит**.

Желающие могут ознакомиться с тем, как это реализовано в **PyCharm**:

<https://www.jetbrains.com/help/pycharm/profiler.html>

<https://www.jetbrains.com/help/pycharm/v8-cpu-and-memory-profiling.html>

# Пример работы профайлера в PyCharm (граф вызовов)



**Смотри первый ноутбук занятия**  
**(про профилирование)**

# Recap

- ▶ `%timeit`, `%%timeit`, `%%time` – line и cell magics в Jupyter. Замеряют время работы строки /ячейки кода.
- ▶ `line_profiler` позволяет замерить время, приходящееся на каждую строчку кода. Запускается через `%lprun` или `kernprof -lv my_file.py` после навешивания `@profile` на все нужные функции.
- ▶ `memory_profiler` – аналог `line_profiler` для замера количества расходуемой памяти. Запускается через `python -m memory_profiler my_file.py`

# JIT-компиляция

Как писать на Python циклы, за которые не стыдно

# В этом разделе мы узнаем

- ▶ Что такое `JIT`-компиляция и почему это полезно для Python-разработки.

Или “почему не стоит пытаться всё на свете пытаться векторизовать через `numpy`.” :)

- ▶ Как пользоваться `numba` для ускорения вычислений на pure Python + `numpy`.
- ▶ Какие у `numba` есть ограничения и в каких случаях нет смысла её использовать.

# JIT-компиляция

- ▶ **Компиляция** — процесс преобразования кода программы в набор машинных инструкций.
- ▶ **JIT-компиляция** — компиляция **во время работы** программы, что особенно актуально для интерпретируемых языков.

Продвинутые JIT-компиляторы умеют сами находить высоконагруженные участки кода и оптимизировать их.



# Плюсы JIT-компиляции

- ▶ Некоторые JIT-компиляторы имеют встроенный профайлер и могут эффективно оптимизировать узкие места в коде.
- ▶ JIT-компиляция удобна для Python, где поля и методы объектов могут динамически изменяться.
- ▶ Можно реализовать JIT-компиляцию под конкретные нужды: скажем, под оптимизацию вычислений.

# Минусы JIT-компиляции

- ▶ Запуск JIT-компилятора может занимать много времени.
- ▶ В случае Python JIT-компиляция происходит только после первого вызова функции (потому в ней нет смысла, если вы запускаете функцию всего один раз).
- ▶ JIT-компиляторы поддерживают какое-то подмножество языка. В какой-то момент вам может перестать его хватать.
- ▶ Байт-код, генерируемый JIT-компилятором, может не быть кросс-платформенным: будет работать на вашей машине, но не на серверах на работе.



- ▶ `numba` — JIT-компилятор вычислительных процедур в Python-коде.
- ▶ `numba` поддерживает **не все** возможности Python, а только базовые конструкции языка, некоторые стандартные библиотеки и `numpy`.
- ▶ `numba` позволяет писать обычный код с `for`- и `while`-циклами там, где сложно (или нецелесообразно) написать векторизованный код на `numpy`.



Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

[Learn More](#)[Try Numba »](#)

## Accelerate Python Functions

Numba translates Python functions to optimized machine code at runtime using the industry-standard [LLVM](#) compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

[Learn More »](#)[Try Now »](#)

```
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

## Built for Scientific Computing

Numba is designed to be used with NumPy arrays and functions.

Numba generates specialized code for different array data types and layouts to optimize performance. Special decorators can create [universal functions](#) that broadcast over NumPy arrays just like NumPy functions do.

Numba also works great with Jupyter notebooks for interactive computing, and with distributed execution frameworks, like Dask and Spark.

[Learn More »](#)[Try Now »](#)

```
@numba.jit(nopython=True, parallel=True)
def logistic_regression(Y, X, w, iterations):
    for i in range(iterations):
        w -= np.dot(((1.0 /
                      (1.0 + np.exp(-Y * np.dot(X, w))
                      - 1.0) * Y), X)

    return w
```

# Parallelize Your Algorithms

Numba offers a range of options for parallelizing your code for CPUs and GPUs, often with only minor code changes.

## Simplified Threading

```
@jit(nopython=True, parallel=True)
def simulator(out):
    # iterate loop in parallel
    for i in prange(out.shape[0]):
        out[i] = run_sim()
```

Numba can automatically execute NumPy array expressions on multiple CPU cores and makes it easy to write parallel loops.

[Learn More »](#)[Try Now »](#)

## SIMD Vectorization

```
LBB0_8:
    vmovups (%rax,%rdx,4), %ymm0
    vmovups (%rcx,%rdx,4), %ymm1
    vsubps  %ymm1, %ymm0, %ymm2
    vaddps  %ymm2, %ymm2, %ymm2
```

Numba can automatically translate some loops into vector instructions for 2-4x speed improvements. Numba adapts to your CPU capabilities, whether your CPU supports SSE, AVX, or AVX-512.

[Learn More »](#)[Try Now »](#)

## GPU Acceleration



With support for both NVIDIA's CUDA and AMD's ROCm drivers, Numba lets you write parallel GPU algorithms entirely from Python.

[Numba CUDA »](#)[Numba ROCm »](#)

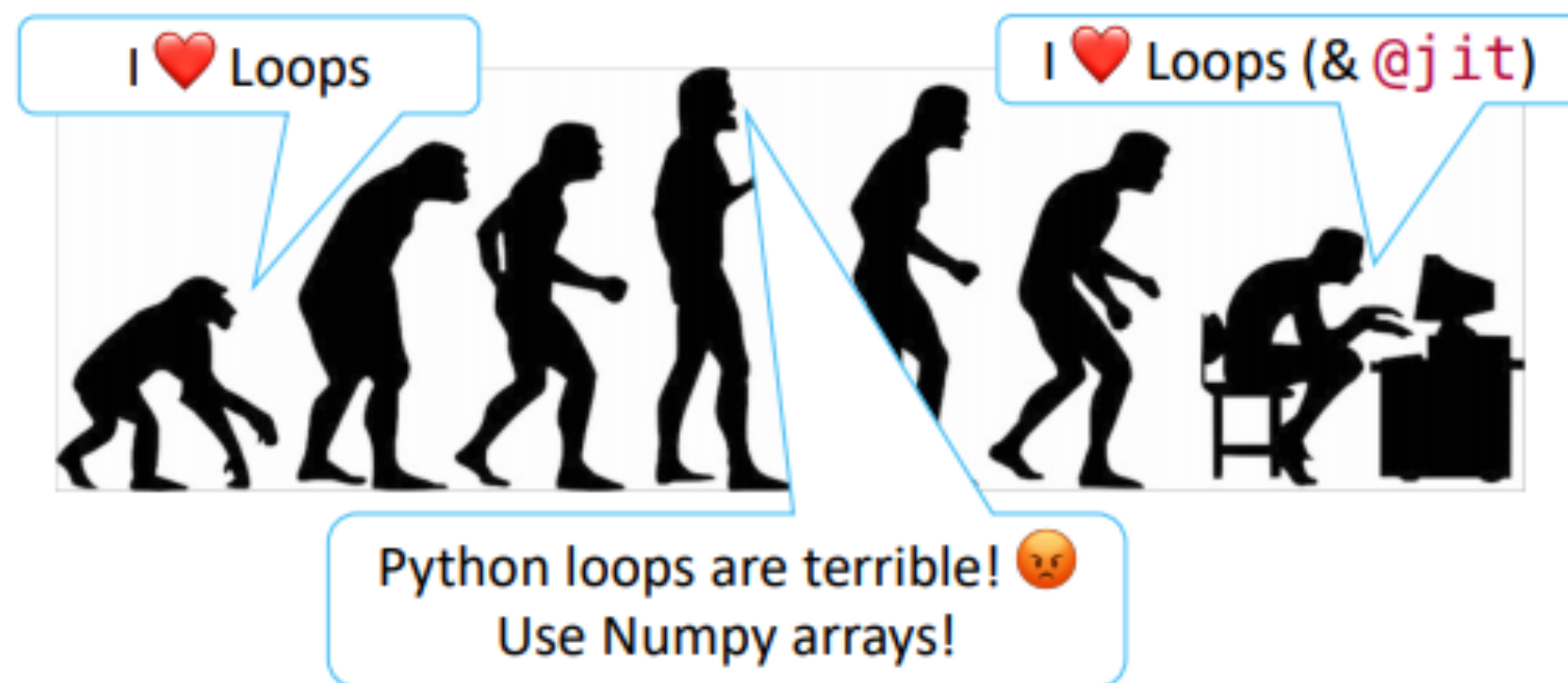
**Смотри второй ноутбук занятия  
(раздел про JIT-компиляцию)**

# Плюсы numba

- ▶ Не нужно переписывать код: навесили один декоратор и всё, вы восхитительны.
- ▶ Даёт ощутимый прирост скорости в вычислительных задачах.
- ▶ Можно писать циклы и не бояться.
- ▶ Если очень хочется, то можно автоматически переносить код на GPU (но это не так просто)



## Evolution of a Programmer





# Минусы numba

- ▶ Не поддерживает `scipy`
- ▶ Не поможет в случае Python-кода общего назначения (где работа с файлами, сетью и так далее)
- ▶ Первый запуск функции не быстрее обычного
- ▶ Непонятные ошибки компиляции

# Recap

► Ускорение кода: `@numba.jit(nopython=True)`

► Распараллеливание циклов:

`@numba.jit(nopython=True, parallel=True)`  
`numba.prange` вместо `range`

► Эффективный запуск функции вдоль нулевой размерности тензора:

`@numba.vectorize` (выход — скаляр)  
`@numba.guvectorize` (выход — вектор)

# Распараллеливание вычислений в CPU-bound задачах

`multiprocessing` и `joblib`

# В этом разделе мы узнаем

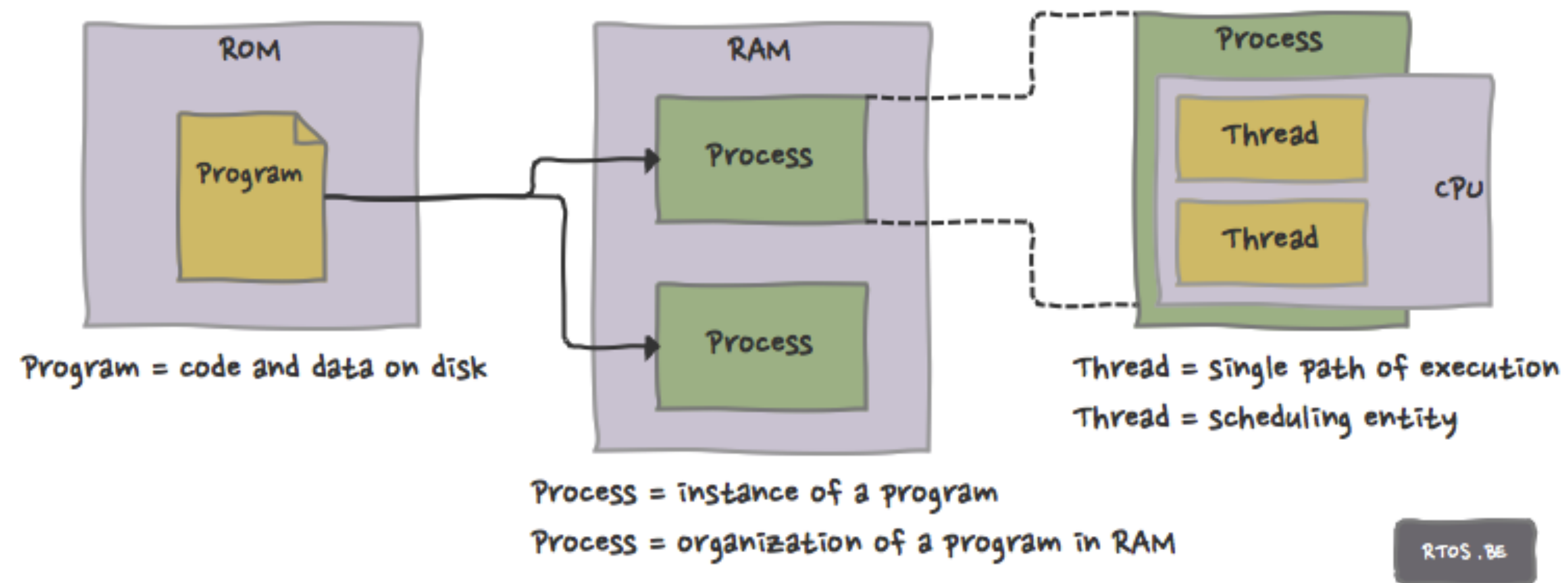
- ▶ Какими способами можно распараллеливать вычисления на Python.
- ▶ Как пользоваться библиотеками `multiprocessing` и `joblib`.
- ▶ Почему распараллеливание не серебряная пуля и что делать, если ваш параллельный код работает медленнее обычного.

# Распараллеливание вычислений

- ▶ До сих пор мы везде писали однопоточный **код**, который никак не использует факт наличия **нескольких** процессорных ядер.
- ▶ Многоядерные системы — уже давно **норма**, нам нужно научиться учитывать эти реалии.

# Распараллеливание вычислений

- ▶ Есть два основных подхода к распараллеливанию кода: `thread-based` и `process-based`.
- ▶ **Потоки** (`threads`) легковесны, имеют общую память и вычисляются в пределах того процесса, где запущена ваша программа.
- ▶ **Процессы** же создают копию родительского окружения и могут ничего не знать друг о друге.

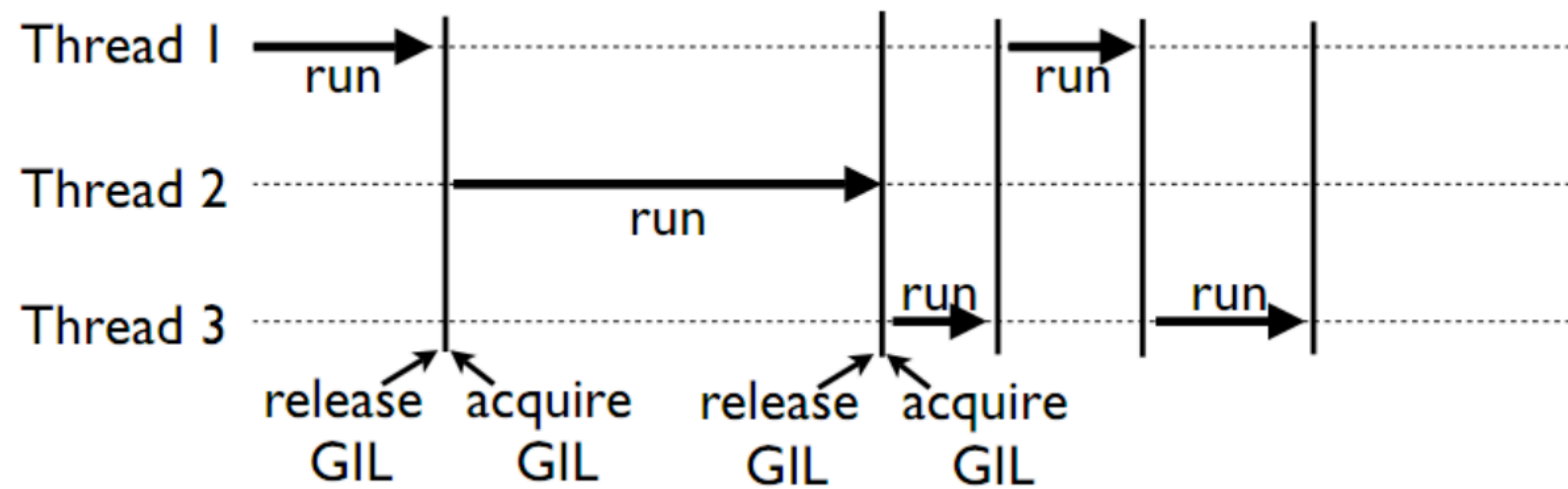


Источник: <http://www.rtos.be/2013/08/programs-processes-and-threads-part-2/>

# GIL

- ▶ **Многопоточные вычисления** в Python **невозможны**, т.к. его механизм сборки мусора запрещает интерпретатору работать в таком режиме.
- ▶ За этим следит **GIL** — **Global Interpreter Lock**. Были попытки создать интерпретатор без GIL, все они закончились **ничем**.





Источник: [http://cs.mipt.ru/advanced\\_python/lessons/lab10.html](http://cs.mipt.ru/advanced_python/lessons/lab10.html)

# Параллелизация в Python

- ▶ Поэтому в Python возможна **только** `process-based` параллелизация.
- ▶ Она представлена такими библиотеками как `multiprocessing` и `joblib`.
- ▶ DS-задачи часто разбиваются на **независимые подзадачи**, которые можно вычислять одновременно. Обычно при этом на части бьются и **данные**, потому отсутствие разделяемой памяти не играет роли.

# Закон Амдала

- В общем случае прирост производительности от распараллеливания можно оценить сверху по **закону Амдала**.

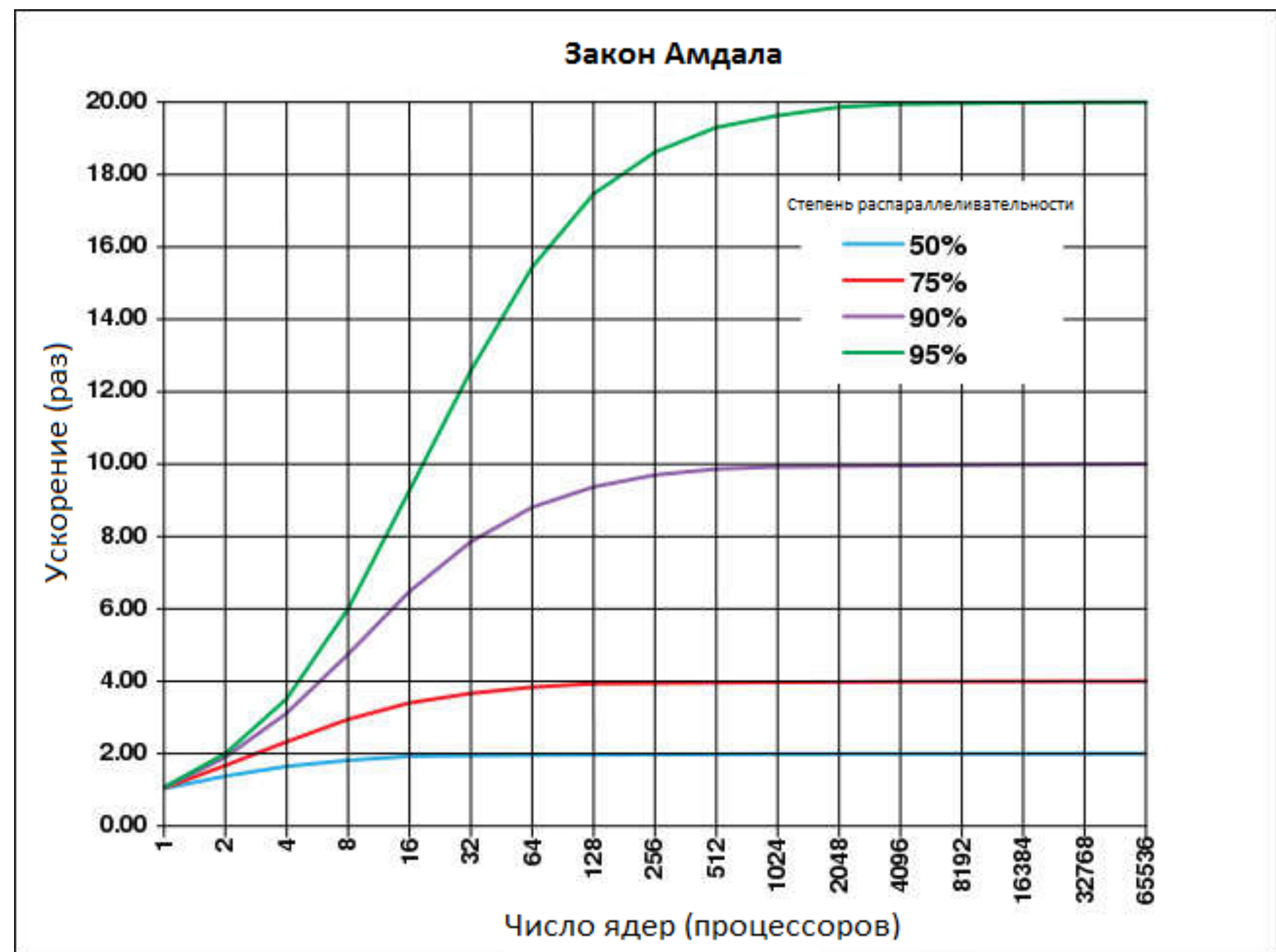
$$S_n = \frac{p \cdot n + s}{p + s} = \frac{n(p + s)}{p + s} + \frac{s - s \cdot n}{p + s} = n + (1 - n) \frac{s}{p + s} = n + (1 - n)s, \text{ где}$$

$s$  — доля последовательных расчётов в программе,

$p$  — доля параллельных расчётов в программе,

$n$  — количество процессоров.

Источник: <https://medium.com/german-gorelkin/amdahls-law-79a8edb040e2>



**Смотри второй ноутбук занятия  
(раздел про **параллелизацию**)**

# Плюсы параллельных вычислений в Python

- ▶ Большую часть DS-задач можно разбить на независимые подзадачи, каждую из которых можно исполнять параллельно.
- ▶ Прирост скорости при этом пропорционален числу процессорных ядер.
- ▶ Обычно распараллеливание не требует особых усилий от разработчика.

# Минусы параллельных вычислений в Python

- ▶ Параллелизация в Python основана на **процессах**. У них **нет** разделяемой памяти, вам придётся создавать копии объектов в RAM.
- ▶ Создание и уничтожение процесса **занимает время**. Если неграмотно подойти к вопросу, ваш параллельный код может быть **медленнее** исходного.
- ▶ Более сложный код не всегда параллелится так легко, как в учебных примерах. Вам придётся понимать, что такое pickle-протоколы и как под них подстроиться.



# Recap

- ▶ В DS большую часть задач можно свести к парадигме `MapReduce`, а потому можно эффективно распараллелить.
- ▶ `joblib.Parallel` — объект, который позволяет указать, сколько процессов создавать.
- ▶ `joblib.delayed` — обёртка над функциями, которая позволяет передавать их в `joblib.Parallel`



# Сегодня мы узнали..

- ▶ ...почему Python тормозит и почему в большей части случаев это не имеет значения.
- ▶ ...что делать, если производительность всё же критична. Как найти узкие места в коде.
- ▶ ...что такое JIT-компиляция и как без труда использовать её в Python.
- ▶ ...почему для DS-задач актуально распараллеливание кода и в чём нюансы этого подхода в случае Python.

**Спасибо за  
Внимание!**

**Вопросы?**

**Оставьте отзыв! Мы их  
все читаем, нам очень  
важно знать ваше  
мнение!**