Profiling and Performance Basics

Mikel Solabarrieta – 2025-06-24

 hi@mikel.xyz

 @mikel@cyberplace.social

 mikelsr

 mikelsr

Sources

https://github.com/mikelsr/perfbasics

About me

Deusto

IEIA + II

DeustoTech

MORElab – IoT

UPC

Masters – Research on High Performance Computing

Barcelona Supercomputing Center

Predictable Parallel Computing – Hardware Acceleration

StreamSets / IBM

Real-time ETL tools

Openfort

Transaction infrastructure

Wetware

Distributed systems platform

Profiling is...

Analyzing the execution of a program to understand its runtime behavior and performance characteristics.

There are multiple types of profiling, depending on what is being analyzed:

- **CPU**
- Memory
- IO

Profiling a program affects its behaviour!

Benchmarking is...

Measuring and evaluating the performance of software.

Benchmarking: big picture. Time it takes to run the program, total memory usage...

Profiling: detailed view. Time it takes for each function to run, calls of each function...

Performance is important because it…

Lets you do more with less:
Attend more requests, run more services or save hardware costs.

Improves power efficiency:
Every instruction a computer executes costs energy.

Provides a better user experience:
By making things happen and respond faster.

Opens new possibilities:
A slower program might have more limited functionality.

Performance isn’t everything!

Most of the times, correctness, readability, maintainability… should come before performance.

Other people will likely need to work with your code in the future.

You will likely need to work with your code in the future.

Evaluate the priorities of each project and balance them accordingly.

Performance isn’t everything!

Wednesday, April 22, 2009

Tail Recursion Elimination

I recently posted an entry in my Python History blog on the origins of Python's functional features. A side remark about not supporting tail recursion elimination (TRE) immediately sparked several comments about what a pity it is that Python doesn't do this, including links to recent blog entries by others trying to "prove" that TRE can be added to Python easily. So let me defend my position (which is that I don't *want* TRE in the language). If you want a short answer, it's simply unpythonic. Here's the long answer:

About Me

Guido van Rossum

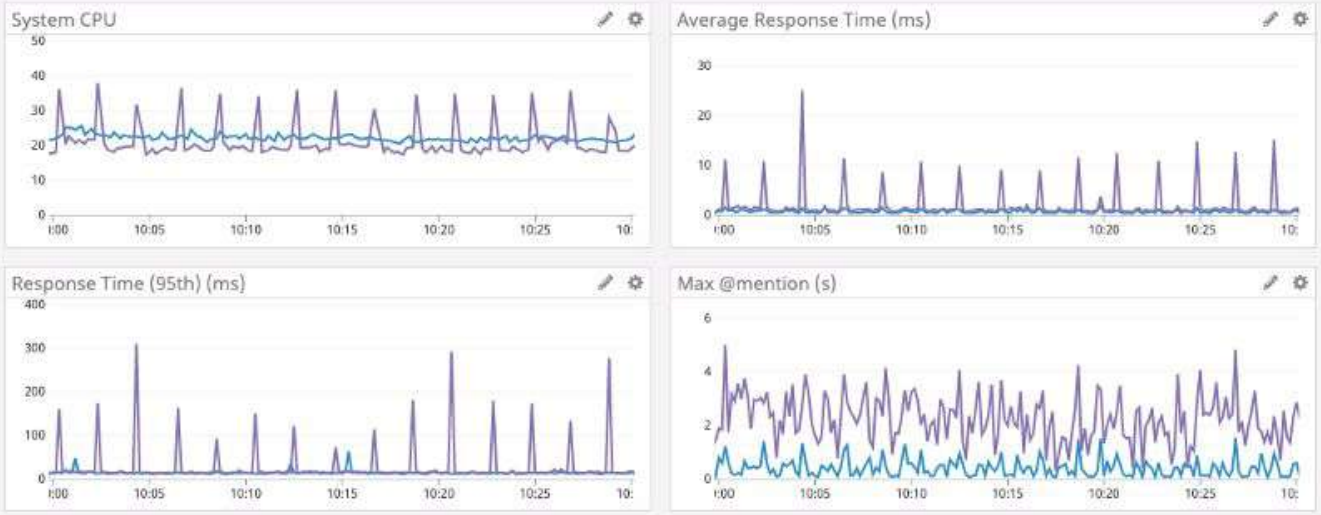
Python's BDFL

View my complete profile

Blog Archive

https://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html

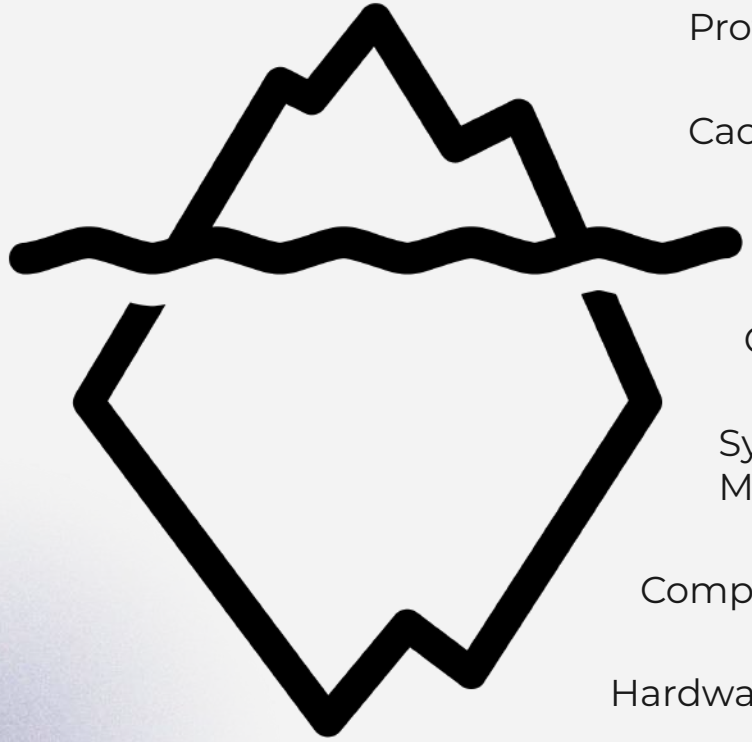
But it is important.



Purple (spiky) - Go ; Blue (plane) Rust

https://discord.com/blog/why-discord-is-switching-from-go-to-rust

The performance improvement iceberg



Profiling

Algorithms

Data Structures

Caching

Libraries

Data Locality

Runtimes

Concurrency

Parallelism

Synchronization Mechanisms

Inter-Process Communication

Compiler Directives

SIMD

Hardware Acceleration

And so many more...

Things we'll talk about

Basic tips

Big O

Memory layout and hierarchy

Concurrency and parallelism

Hardware acceleration

Garbage Collection

Data locality

Profiling

Basic Tips

1. **Don’t reinvent the wheel**
   Someone has likely faced this problem before, and created an optimized way of solving it. The best example are native types and methods.
2. **Re-use**
   Existing objects instead of creating new objects
3. **Cache when possible (re-use!)**
   @lru_cache, network requests...
4. **Moving things around is slow**
   Over the network, from one process to another, from one place in memory to another, from memory to cache...
5. **Pre-allocate when possible**
   So things are moved around less.
6. **Learn data structures.**
   Guaranteed to speed up and improve almost any program you write.
7. **Idle time is wasted time**
   Sending network requests, waiting for events... can free the CPU to do other work.

Basic Tips

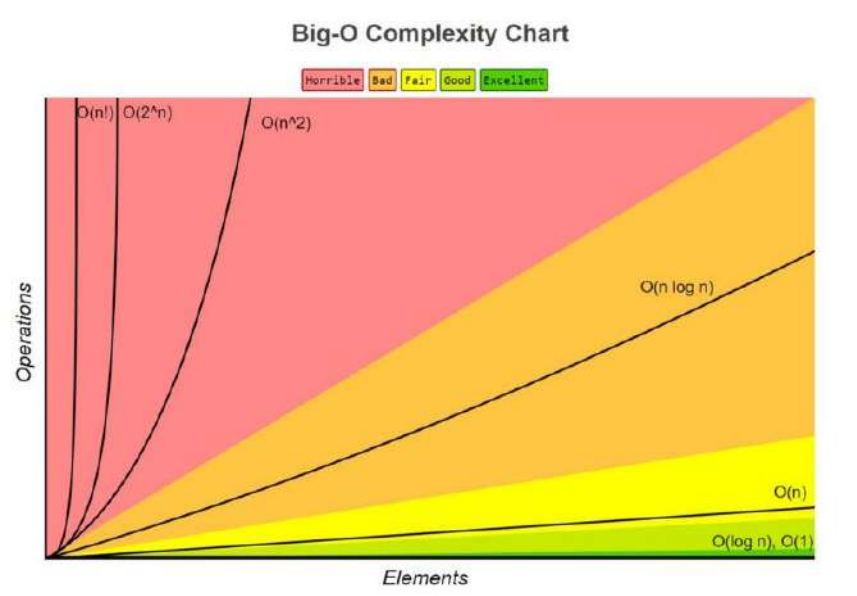
8. **Profile**

   You know what your code does, now learn how it does it. Use what you learn to improve upon it.

Less Basic Tips

1. **Remember your software runs on hardware**
   Each hardware has its own characteristics. There are also some general ones: cache is faster than memory, which is faster than storage. Division is slower than multiplication. Networks can be unreliable...
2. **Learn your language**
   Each programming language has its own best practices. E.g. working with pointers might be faster in C, but slower in Go.
3. **Reflection is expensive**
   Knowing things at compile time is faster.
4. **Concurrency and parallelism**
   Write code that can do more than one thing at once.
5. **Hardware acceleration**
   CPUs are not the only ones capable of running software! You likely have GPUs, TPUs, FPGAs..
6. **Heap escape analysis**
   Stack is faster than heap, and some languages let you see when variables escape the stack and go to the heap.

Big O



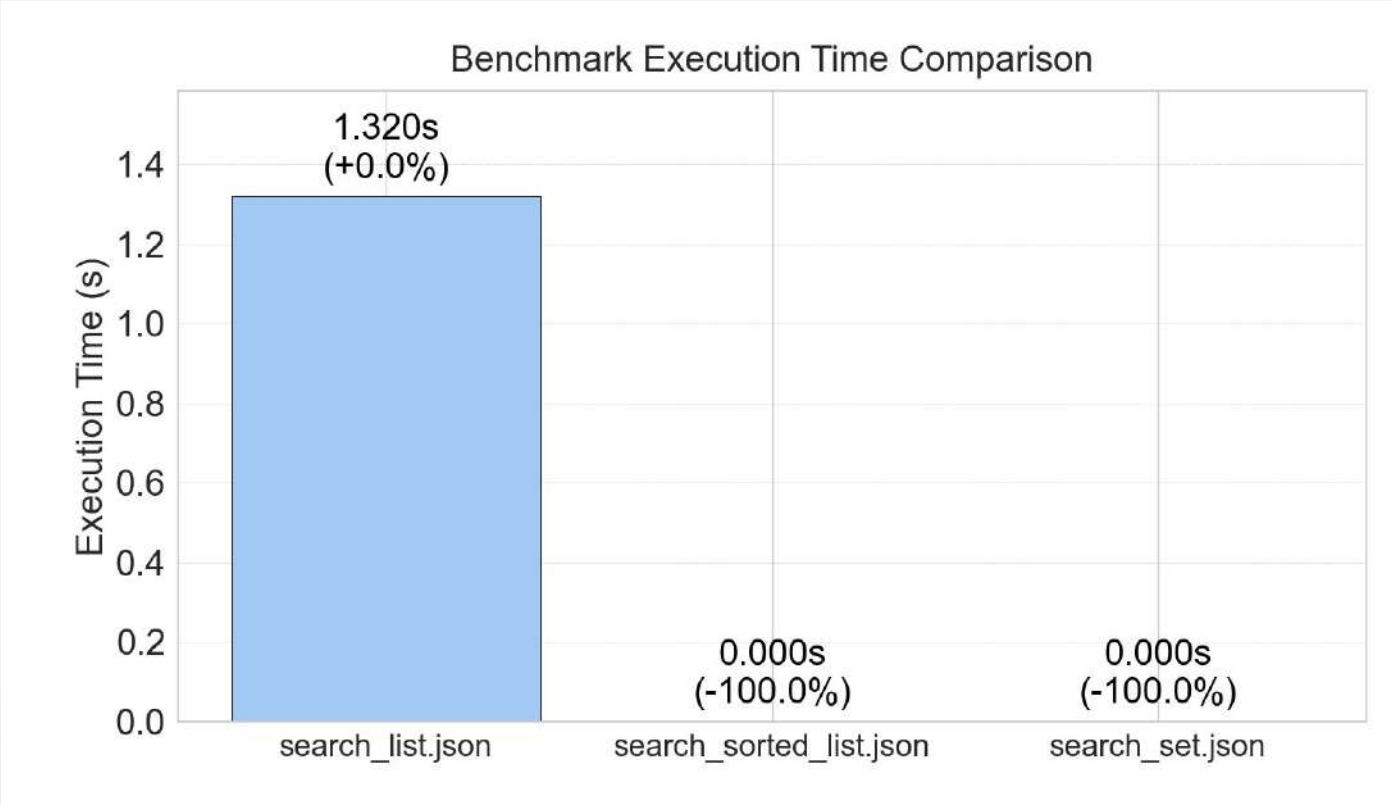
Source:
https://www.freecodecamp.org/news/all-you-need-to-know-about-big-o-notation-to-crack-your-next-coding-interview-9d575e7eec4/

Big O: Learn algorithms and data structures

Useful in practically any situation. Simple example, check if a set of numbers contains a value.

```python
import bisect

LEN = 100_000_000
ITEMS = [i for i in range(LEN)]
LOOKUPS = [421, 540_354, 29_233_421, 99_999_999, 1, 50_000_000, 63_312_512]
LIST = [i for i in range(LEN)]
SET = {i for i in range(LEN)}

def search_list():
    for item in LOOKUPS:
        assert item in LIST

def search_sorted_list():
    for item in LOOKUPS:
        i = bisect.bisect_left(LIST, item)
        assert i < LEN and LIST[i] == item

def search_set():
    for item in LOOKUPS:
        assert item in SET
```

Big O


Benchmark Execution Time Comparison
Execution Time (s)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1.320s
(+0.0%)
0.000s
(-100.0%)
0.000s
(-100.0%)
search_list.json
search_sorted_list.json
search_set.json

Pre-allocate

```
def with_initial_size(n):
    l = [0] * n
    for i in range(n):
        l[i] = i
    return l


def without_initial_size(n):
    l = []
    for i in range(n):
        l.append(i)
    return l
```

Pre-allocate

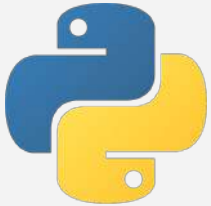
Pre-allocate

```
def without_initial_size(n):
    l = [i for i in range(n)]
    return l
```

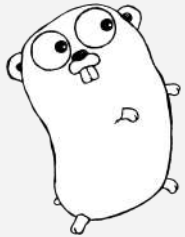
Garbage Collection

Many garbage-collected languages allow some control over when and how the garbage collector runs. But does **have to** run.

```
import gc
gc.enable()
gc.disable()
gc.collect()
```



```
import "runtime"

runtime.GC()
```



```
System.gc()
```

https://sematext.com/java-garbage-collection-tuning/

Garbage Collection

It can have a great impact on your application performance:

How We Saved 70K Cores Across 30 Mission-Critical Services (Large-Scale, Semi-Automated Go GC Tuning @Uber)
https://www.uber.com/en-ES/blog/how-we-saved-70k-cores-across-30-mission-critical-services/

Careful!

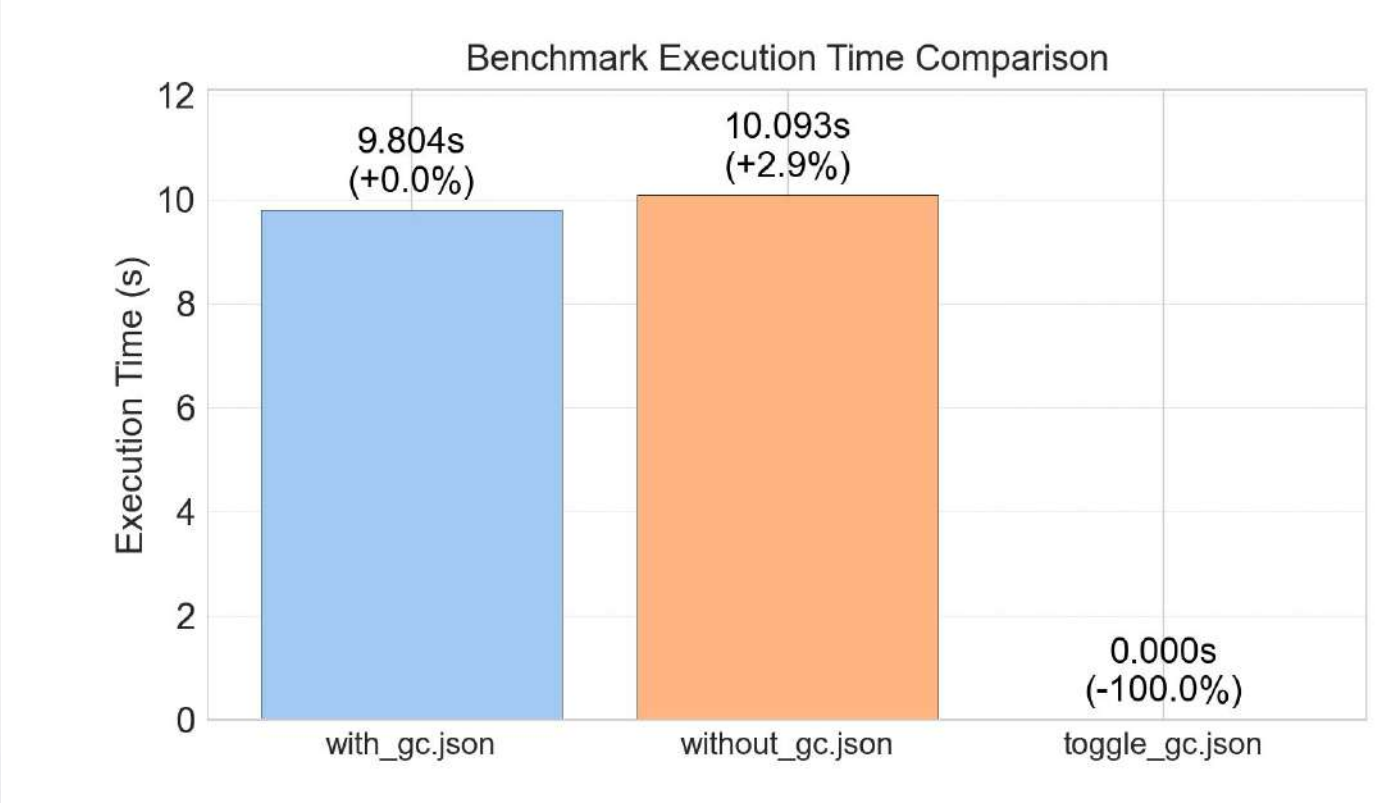
Garbage Collectors, interpreters, compilers... are usually greatly designed and know what they are doing.

GC example

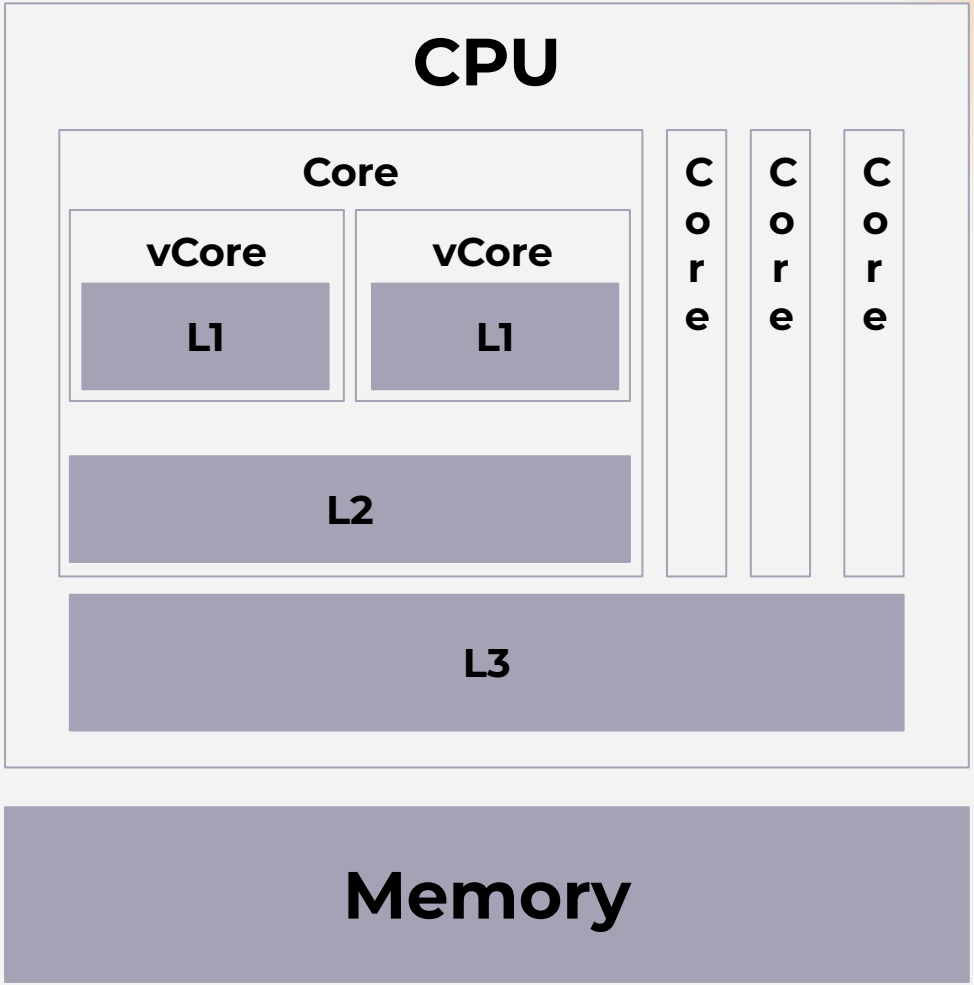
```python
import gc

def make_objects():
    objects = []
    for _ in range(200_000_000):
        objects.append(object())
    return objects

def toggle_gc():
    try:
        gc.disable()
    finally:
        gc.enable()
```

```python
def with_gc():
    return make_objects()


def without_gc():
    try:
        gc.disable()
        return make_objects()
    finally:
        gc.enable()
```

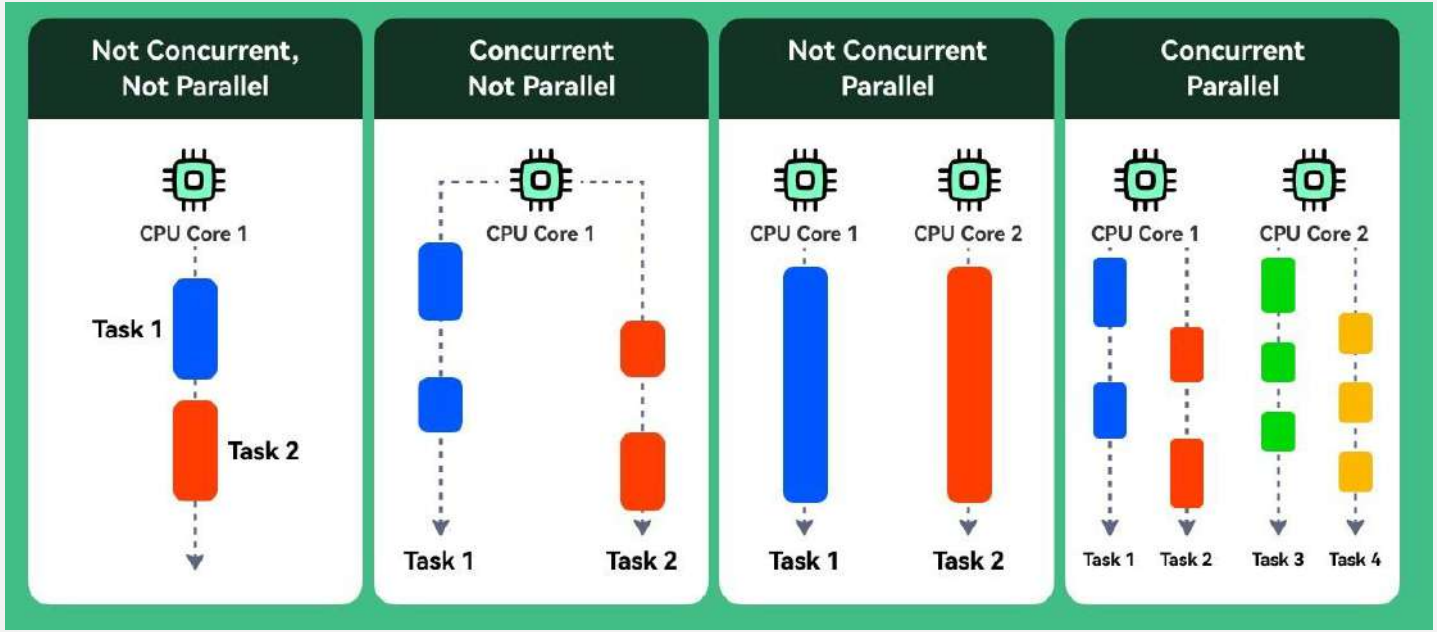
GC example


Benchmark Execution Time Comparison
Execution Time (s)
12
10
8
6
4
2
0
9.804s
(+0.0%)
10.093s
(+2.9%)
0.000s
(-100.0%)
with_gc.json
without_gc.json
toggle_gc.json

Memory Layout and Hierarchy



Concurrency vs Parallelism



Source: ByteByteGo https://www.youtube.com/watch?v=RlM9AfWf1WU

Concurrency vs Parallelism

Concurrency: take advantage of your IDLE time

Parallelism: take advantage of your multiple cores

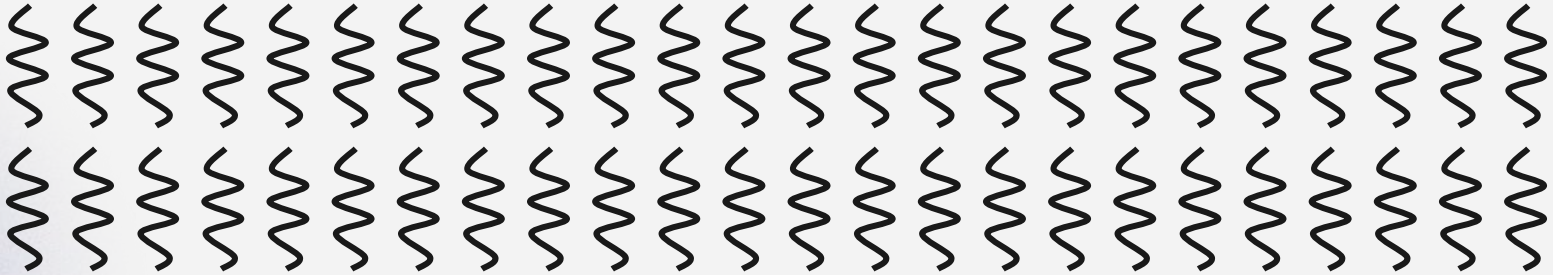
Hardware Acceleration

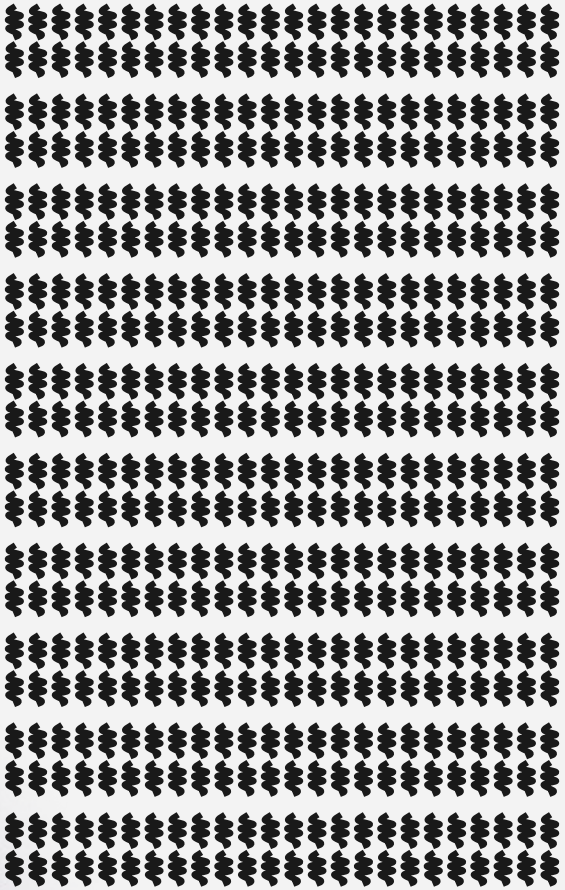
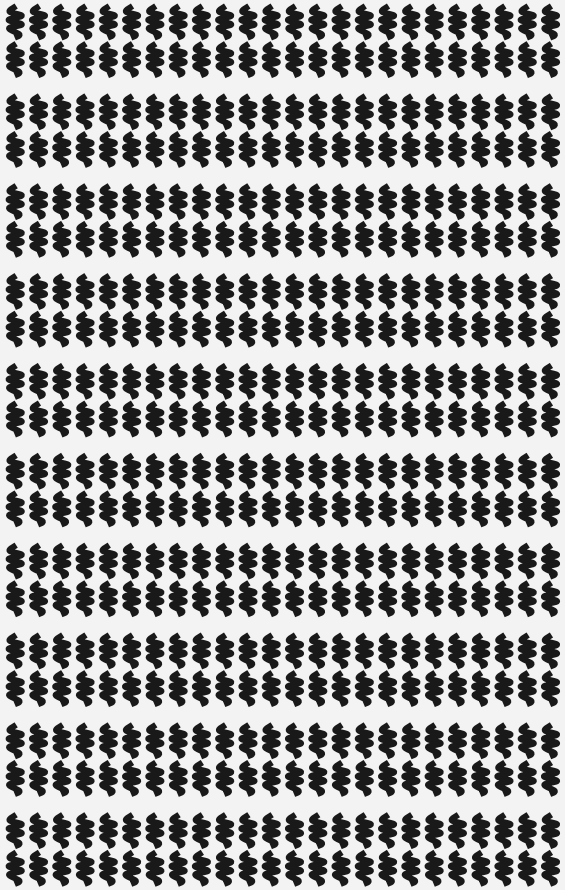
CPUs are **good** at everything, bad at nothing.

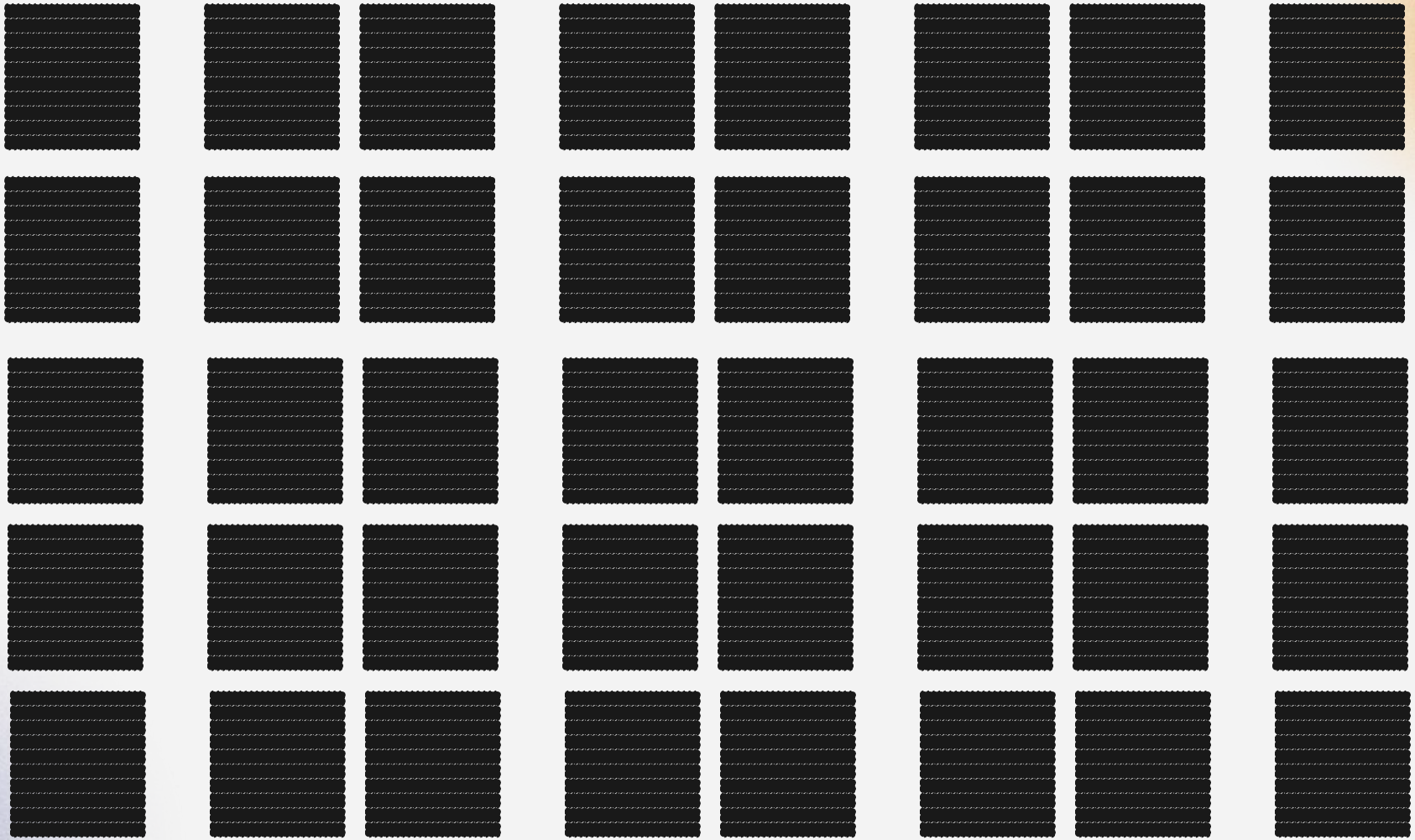
Other processor types are **great** at something, bad at everything.

For example, a CPU usually has 10s of simultaneous threads.

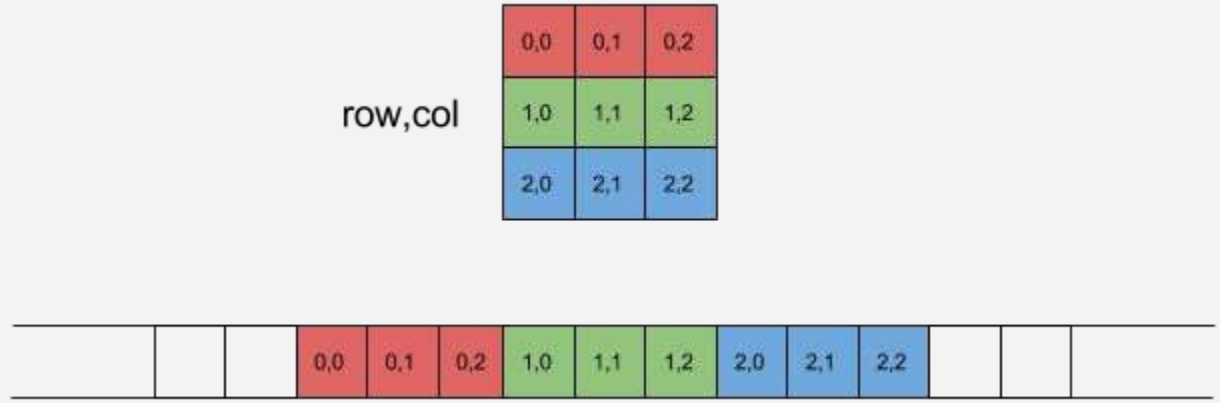
Do you know how many a GPU has?





Data Locality



Source: https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Data Locality

```
def iterate_by_row():
    total = 0
    size = len(MATRIX)
    for col in range(size):
        for row in range(size):
            total += MATRIX[col][row]
    return total
```

```
def iterate_by_column():
    total = 0
    size = len(MATRIX)
    for row in range(size):
        for col in range(size):
            total += MATRIX[col][row]
    return total
```

```
MATRIX = [[randint(0, 100) for _ in range(_ROWS)] for _ in range(_COLS)]
```

Data Locality

Matrix: 3000x3000 items. Each item 24 bytes (Python is weird). 216MB.

Data Locality

Matrix: 10000x10000 items. 2.4GB.

What Other Ways Can You Think Of?

What Other Ways Can You Think Of?

- Parallelize
- Use external libraries
- Use another interpreter

  (pypy.org, great performance in nested loops)
- ...

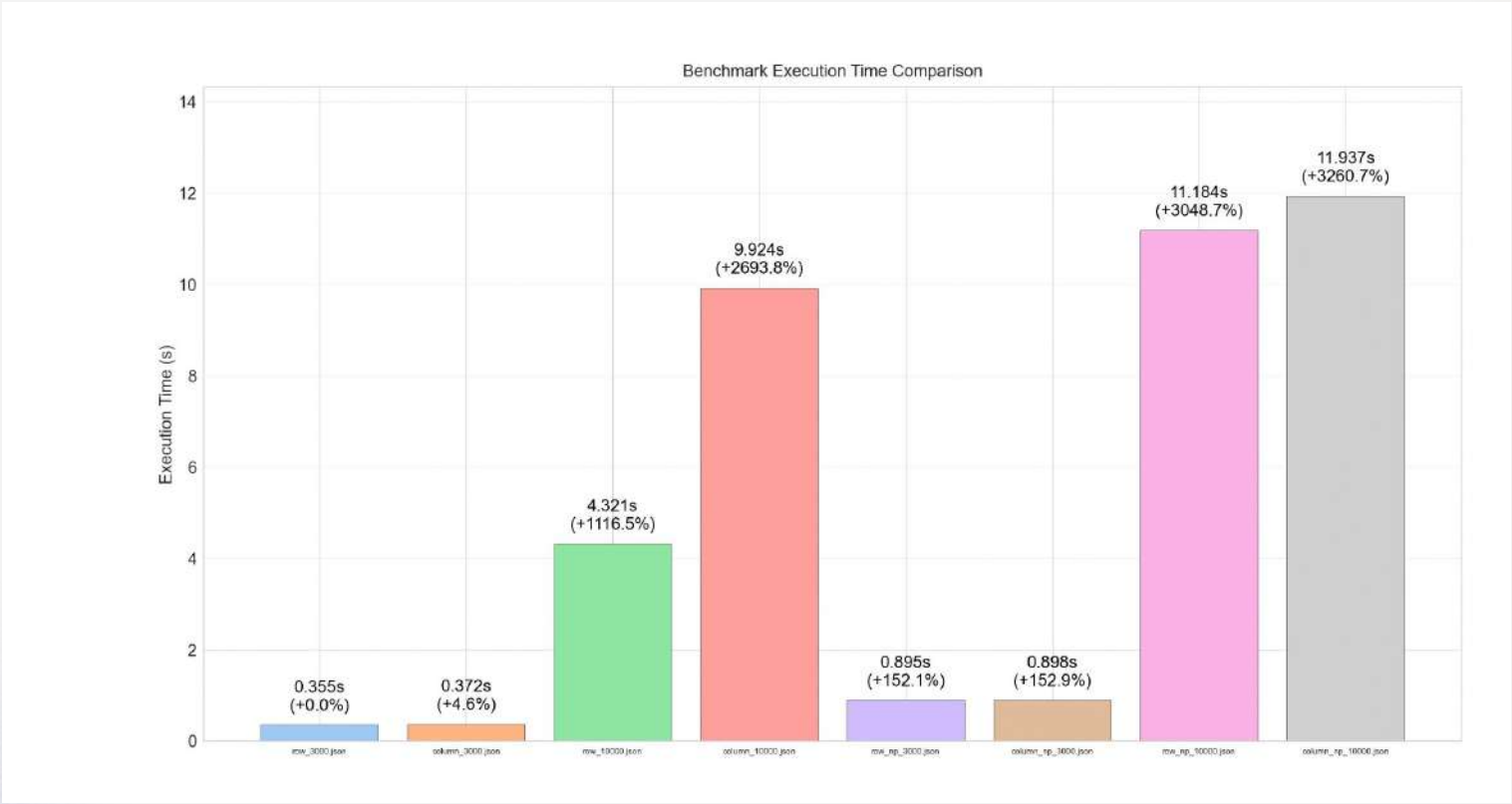
Introducing Numpy

```
def iterate_by_row():
    total = 0
    for col in range(MATRIX.shape[0]):
        for row in range(MATRIX.shape[1]):
            total += MATRIX[col, row]
    return total
```

```
def iterate_by_column():
    total = 0
    for row in range(MATRIX.shape[1]):
        for col in range(MATRIX.shape[0]):
            total += MATRIX[col, row]
    return total
```

```
MATRIX = np.random.randint(0, 101, size=(_COLS, _ROWS))
```

Surprisingly slower!

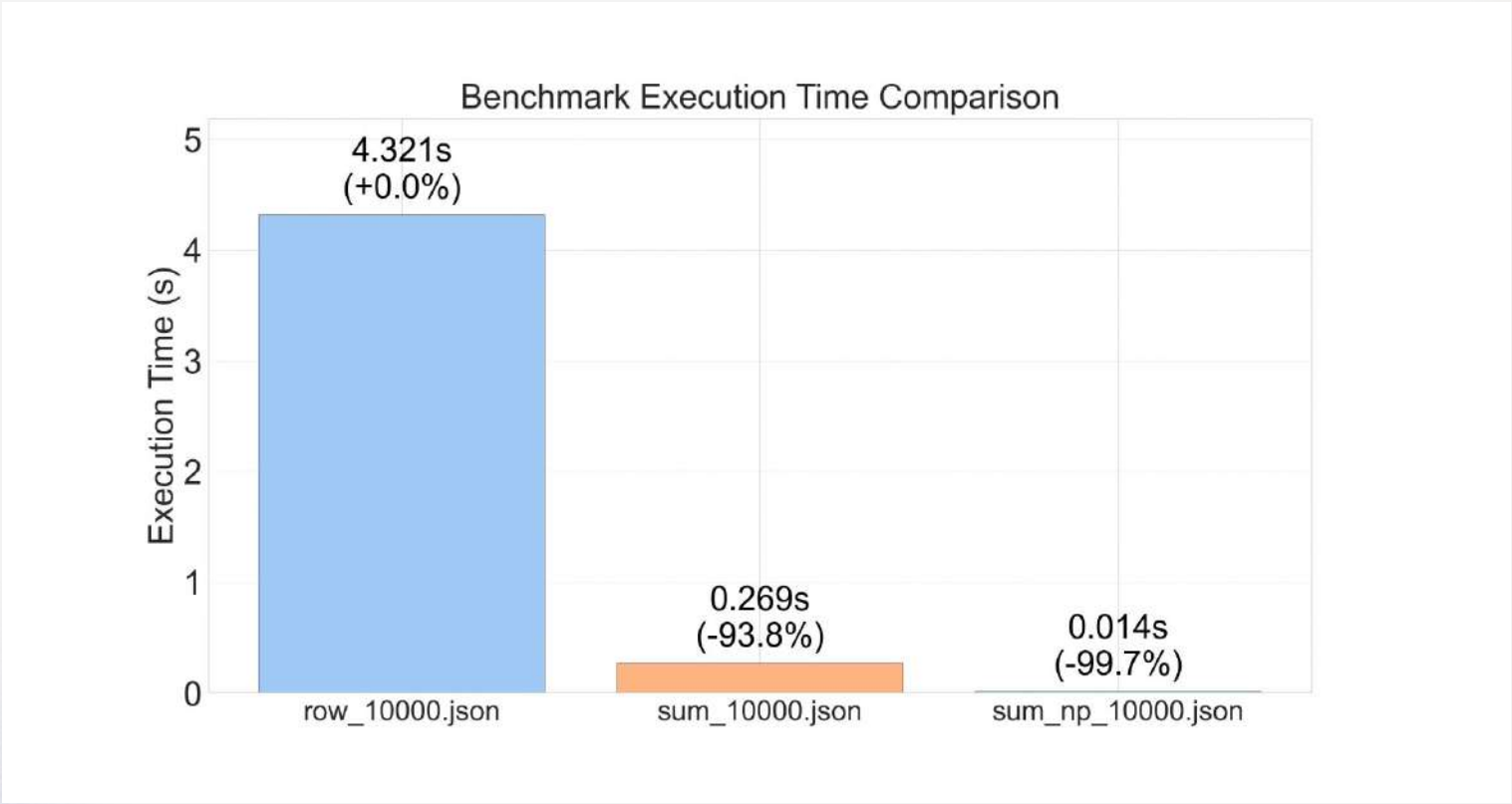


Choose the Right Tool and Method

```python
# native python
def matrix_sum():
    return sum(sum(row) for row in MATRIX)
```

```python
# numpy
def matrix_sum():
    return np.sum(MATRIX)
```

By Row vs Native Sum vs Numpy Sum



Data Locality

Think about how your data is structure in memory, and take advantage of things that are close to each other.

```
class MyClass:
    def __init__(self):
        self.a = 100
        self.b = 100
        self.c = 100
        self.d = 100
        ...

objects = [MyClass(), MyClass(), MyClass()]


# In memory:
objects = [a, b, c, d, ..., a, b, c, d, ..., a, b, c, d, ...]
```

Data Locality

```
objects = [MyClass() for _ in range(1_000_000)]


def inc():
    for obj in objects:
        obj.a += 1
        obj.b += 1
        obj.c += 1
```

Data Locality

```
def per_object():
    for obj in objects:
        for attr in ('a', 'b', 'c', 'd', ...):
            setattr(obj, attr, getattr(obj, attr) + 1)

def per_attr():
    for attr in ('a', 'b', 'c', 'd', ...):
        for obj in objects:
            setattr(obj, attr, getattr(obj, attr) + 1)
```

Data Locality

```
def per_object():
    for obj in objects:
        for attr in ('a', 'b', 'c', 'd', ...):
            setattr(obj, attr, getattr(obj, attr) + 1)

def per_attr():
    for attr in ('a', 'b', 'c', 'd', ...):
        for obj in objects:
            setattr(obj, attr, getattr(obj, attr) + 1)
```

Profiling demo

Lab

1. Download https://github.com/mikelsr/perfbasics
2. Install the Python dependencies from requirements.txt
3. Take a look at lab/wordcount.txt
4. Before starting to improve it:
    a. Profile it: to get an idea of what to improve first.
    b. Benchmark it: so you can measure your improvements.
5. Comment and share your progress! Discussion is good.

Thanks!

You can reach me through:


hi@mikel.xyz


@mikel@cyberplace.social


mikelsr


mikelsr