Peak Performance in python?

- @dishantsethi

$whoami

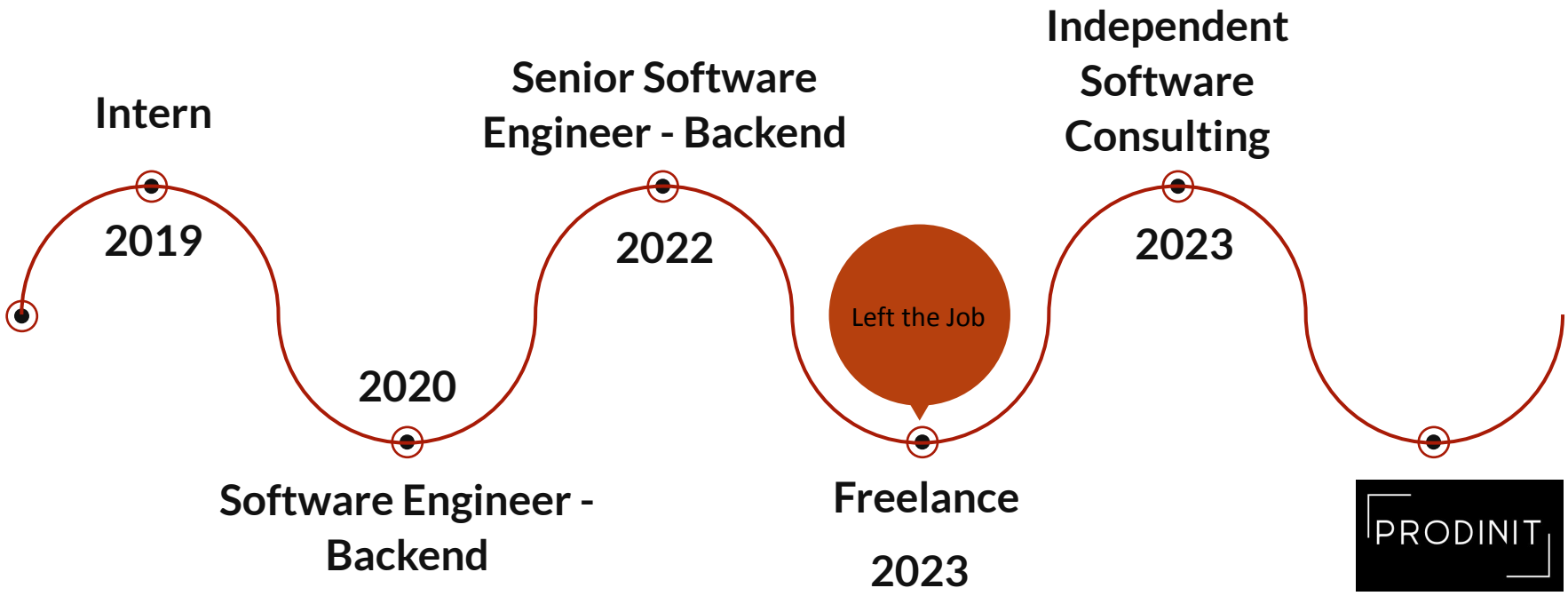
I'm a software engineer

I turn '**I have no idea what I'm doing**' moments into '**I can't believe it works**' victories.

I'm the one who turns

- Coffee into code
- Pizza into programs, and
- Bugs into successful builds.

my code is so clean, it makes my laundry jealous


Intern
2019
2020
Software Engineer - Backend
Senior Software Engineer - Backend
2022
Left the Job
Freelance
2023
Independent Software Consulting
2023
PRODINIT

Things I have worked on

Web Applications

Dev/ML/LLMOps Pipelines

GenAI Applications

Enough about me

Let's start!

Agenda?

~~Agenda?~~ Motivation

Be a better dev

Write optimised code

Ensure minimal resources usage

But, Motivation to?

Achieve peak performance

Build fast executable systems

Go beyond functionality

Before optimising Set A benchmark

What is your performance today?

Are you making things better or worse?

How to find that baseline?

Use profiler

Use built-in profilers like ‘cProfile’ to know time taken by each function to execute

Use ‘snakeviz’ library for better visualisation of cProfile output

Use external libraries like ‘austin’ for line level profiling.

Now that we know how to set a baseline before optimising anything

Here are some of my learnings for code optimisation..

Don’t refactor the whole codebase. Start with small programs. Keep it small and atomic. You would not want to see improvements in one area and slowdown at the other. Hard to debug what cause the slowdown.

Try to reproduce the changes and test the optimisation in different CPUs. CPUs can have multiple processes running at a time, so you cannot be sure about improved performance just by running it once on one single CPU.

Don’t assume that the impact of the changes will be same across Python versions.

If you’re performance improvement is less than 10%, improve your infra instead of code.

Don’t test the optimisation on random dummy data. Try to test it with as realistic data as possible thinking of it a production app.

Let's start with the basics

Simple scenarios

```
import os

for _ in range(100):
    os.path.exists("/")
```

```
from os.path import exists

for _ in range(100):
    exists("/")
```

```
for _ in range(100):
    try:
        do_something()
    except Exception:
        pass
```

```
try:
    for _ in range(100):
        do_something()
except Exception:
    pass
```

Simple scenarios

```python
def with_kwargs(**kwargs):
    return kwargs.get('a') + kwargs.get('b')

with_kwargs(a=1, b=2)
```

```python
def without_kwargs(a: int, b: int):
    return a + b

without_kwargs(a=1, b=2)
```

```python
res = []
for i in range(100_000):
    if i%2 == 0:
        res.append(i)
```

```python
res = [i for i in range(100_000) if i%2 == 0]
```

Simple scenarios

```
def add(x: int, y: int):
    return x+y
x = 1
for i in range(100_000):
    add(i, x)
```

```
x = 1
for i in range(100_000):
    i + x
```

```
x = ["Pycon", "de", "&", "Pydata", "berlin"]
for i in range(100_000):
  len(x) + 1
```

```
x = ["Pycon", "de", "&", "Pydata", "berlin"]
j = len(x)
for i in range(100_000):
  j + 1
```

Simple scenarios

Use __slots__ for
faster data access

```
import timeit

# faster data access
class Slotted(object):
  foo: str
  __slots__ = 'foo'

slotted = Slotted()

min(timeit.repeat(get_set_delete_fn(slotted)))
>> 0.08520045899990691
```

```
import timeit

class NotSlotted(object):
  foo: str

not_slotted = NotSlotted()

min(timeit.repeat(get_set_delete_fn(not_slotted)))
>> 0.10123675000022558
```

Simple scenarios

Don't over optimise the data structure for simple use cases.

```python
from dataclasses import dataclass

@dataclass
class myClass:
  foo: str
  bar: str

for _ in range(100000):
  obj = myClass("foo", "bar")
  str(obj)

>> 00:00:00:114
```

```python
from collections import namedtuple

myClass = namedtuple("myClass", ["foo",
"bar"])

for _ in range(100000):
  obj = myClass("foo", "bar")
  str(obj)

>> 00:00:00:59
```

dict >> class >> namedtuple >> dataclass

Readability

Not related to optimisation by really important when you're writing a good quality code.

```
def mysteryOrbitLauncher(nuclearFusionGenerator):
    antimatterReactorOutput = 1
    for quantumFluctuation in range(1, nuclearFusionGenerator + 1):
        antimatterReactorOutput *= quantumFluctuation
    return antimatterReactorOutput

interstellarTravelSpeed = 5
warpDriveCapacity = mysteryOrbitLauncher(interstellarTravelSpeed)
print("Warp Drive Capacity at Speed", interstellarTravelSpeed, ":", warpDriveCapacity)
```

This is just a simple factorial function, but the naming conventions make it look like a complex function.

Enough of basics

We will talk about applications where python is used

WEB APPLICATIONS

DATA APPLICATIONS

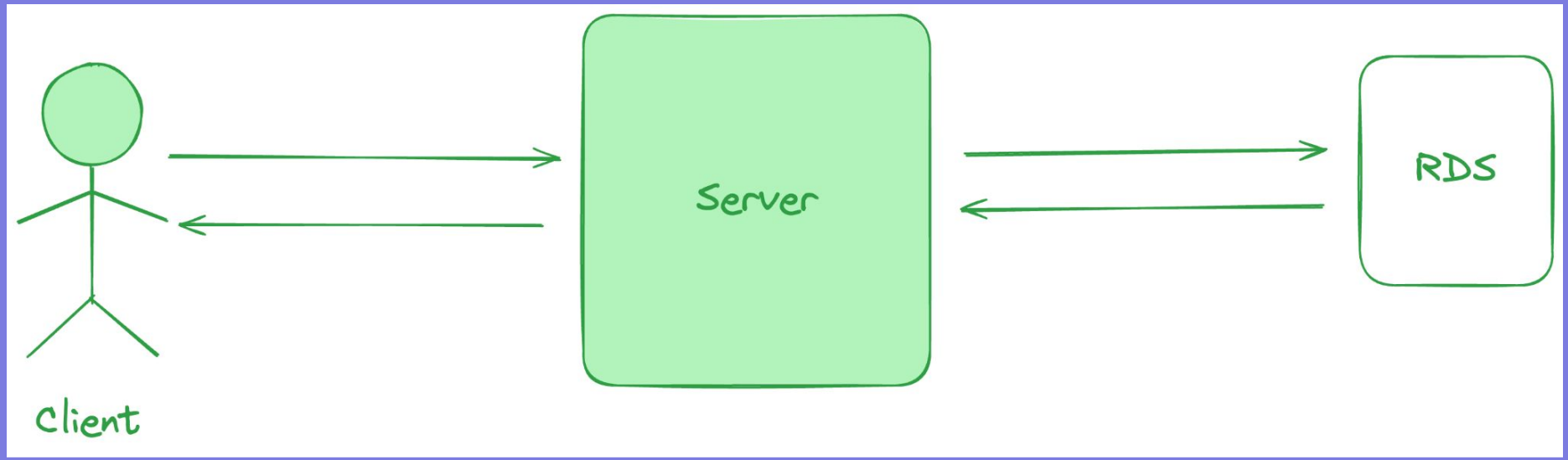
GENAI APPLICATIONS

Web applications

Build APIs

Write Business Logic

Interact with Databases


Client
Server
RDS

How to optimise python Web applications


Async
Whatt?

Why is this a myth?

Bottlenecks decide the area of optimisation

3 ways to optimise a web application

Optimise architecture

Optimise infrastructure

Last one would be to Optimise code

Optimising code will not help if you have a bad architecture.

Here are some of my learnings for optimising code of python web applications..

Use profiling to analyze software bottlenecks

Use Context managers to hide or unhide function code. This will not block the memory.

Use built-in functions - You can write high-quality, efficient code, but it’s hard to beat the underlying libraries

Avoid global variables to keep better track of scopes and unnecessary memory usage.

Use serializers instead of if-else

Use built-in create/update/destroy views from available libraries to build simple and modular APIs

Exit Early: Try to leave a function as soon as you know it can do no more meaningful work.

Eventually use multithreading and caching to further improve performance. Not parallelizing tasks that can be executed concurrently limits performance and underutilizes computational resources

Make sure database connection pooling in on point. You're not draining connections

Use proper indexing techniques

You don't need sharding for a long time.

Though we are discussing python optimisation but if the bottleneck is database

And the most important, architect the application in such a way that the database cannot be a bottleneck for a long time.

Make sure you have separate read/write replicas if required.

Data applications

Involves large dataframes

Involves large CSV files

Memory Efficient way of loading CSV file

When dealing with smaller datasets, the entire file can often be loaded into memory without exceeding available resources. However, it is better to use low_memory=True for improving performance and preventing memory issues, especially if running other processes concurrently.

```
pd.read_csv('./downloads/laptop/data.csv',low_memory=True)
```

Chunk large csv files

Use pandas.read_csv() with chunking to read a large CSV file effectively in chunks to avoid memory limitations. Chunking avoids loading the entire file into memory at once, making it more efficient and less prone to memory limitations.

```
chunk_size = 100000

df=pd.read_csv('./downloads/laptop/data.csv',
chunksize=chunk_size,low_memory=True)
```

Dask for parallel processing

Use dask.dataframe for parallel processing and efficient handling of large datasets. It reads the CSV file in parallel and can leverage multiple cores on your system for faster processing.

```
import dask.dataframe as dd

data=dd.read_csv('./downloads/laptop/data.csv')
```

Looping Techniques for dataframe

Instead of iterating through each row in the DataFrame using itertuples(). Use apply() method with a lambda function. This is a versatile tool for applying custom logic to DataFrames.

```
s = pd.read_csv("stock.csv")


# adding 5 to each value

new = s.apply(lambda num : num + 5)
```

TIME CHECK! PHEW

```
> import this

The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.
```

```
> import this

The Zen of Python, by Tim Peters

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one, and preferably only one, obvious way to do it.

Although that way may not be obvious at first.
```

```
> import this

The Zen of Python, by Tim Peters
```

```
Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good
idea.

Namespaces are one honking great idea -- let's do more of
those!
```

Time check again! Phew

GenAI based LLM optimisation?

Hit me up after this talk

Attention Attention

Most important part of this talk

3

2

1

0

BECAUSE WE ARE ENGINEERS

FEEDBACK

Hire me?

Linkedin - @dishantsethi
Twitter - @Dishantsethi14
Email - dishantsethi14@gmail.com

Question?

https://github.com/dishantsethi/pyconde-2024