Turbo Python Performance To Achieve 100x Faster

June 2023

Challenges and Perceptions:
-Python GIL design is slow
-Python is a slow at runtime
-Python performance is slower comparing to c++ and java

Can it be improved?
-Let’s find it out

Techniques used in this performance testing

-Step up and use built-in functions
-Use vectorization
-Use math functions
-Use multi-processing, concurrency
-Memoization and caching
-Use different language at server backend
-Engineering thoughts

Note:
Often many things can impact python runtime performance
From hardware, cpu, memory, latency in addition to code
Ideas presented here focus on python source code only

In [1]:
```python
# Use built-in functions and libraries, they are tested and optimzied
import string
def upper_basic(n):
    newList = []
    for w in string.ascii_lowercase*n:
        newList.append(w.upper())
```

In [2]:
```python
%timeit upper_basic(1000)
```

```
4.12 ms ± 23.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [3]:
```python
def upper_o2(n):
    newList = map(str.upper, string.ascii_lowercase*n)
```

In [72]:
```python
%timeit upper_o2(1000)
```

```
1.22 µs ± 18.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

60% better when using map() which does elementwise operation

Use vectorization

apply operations to all elements of an array in one go
"for" loop manipulates one row at a time

In [20]:
```
def find_sum(n):
    total = 0
    for i in range(n):
        total += i
```

In [21]:
```
%timeit find_sum(1_000_000)
```

```
74.9 ms ± 1.22 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [22]:
```
import numpy as np
def find_sum_vector(n):
    total = 0
    total = np.sum(np.arange(n))
```

In [23]:
```
%timeit find_sum_vector(1_000_000)
```

```
2.08 ms ± 78.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [113]:
```
(74.9-2.08)/74.9
```

Out[113]:
```
0.9722296395193591
```

97% better when using vectorization in numpy

Deep learning multi-linear regression calculations

$$y = m_1x_1 + m_2x_2 + m_3x_3 + m_4x_4 + m_5x_1 + c$$

Use loop for million of rows of calculations is slow
Vectorization is the optimal solution

In [38]:
```
# create random data
import numpy as np
m = np.random.rand(1,5)
n = np.random.rand(100000,5)
m.shape, n.shape

```

Out[38]:
```
((1, 5), (100000, 5))
```

In [39]:
```
# use loop for calculations
import numpy as np

def loop_reg_sum(col, row):
    m = np.random.rand(1,col)
    n = np.random.rand(row,col)
    result = []
    for i in range(row):
        total = 0
        for j in range(col):
            total += n[j][j]*m[0][j]
#         print(i, total)
        result.append(total)

```

In [40]:
```
%timeit loop_reg_sum(5, 100_000)
```

```
407 ms ± 7.12 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [41]:
```
# use vectorization
def vec_reg_sum(col, row):
    m = np.random.rand(1,col)
    n = np.random.rand(row,col)
    result = np.dot(n, m.T)

```

In [42]:
```
%timeit vec_reg_sum(5, 100_000)
```

```
6.45 ms ± 270 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

407 vs 6.45 Regression using python loop and numpy vectorization

In [49]:

```python
# use built-in sum
def sum_range(n=1_000_000):
    return sum(range(n))
```

In [50]:

```python
# use numpy (implemented in c, faster)
import numpy
def sum_numpy(n=1_000_000):
    return numpy.sum(numpy.arange(n))    # this is a one c call, but a whole array is created in memory

```

In [51]:

```python
# use math knowledge
def sum_math(n=1_000_000):
    return (n * (n-1)) // 2
```

In [52]:

```python
import timeit

print('while loop\t\t', timeit.timeit(while_loop, number = 1))
print('for loop\t\t', timeit.timeit(for_loop, number = 1))
print('for_loop_with_increment\t\t', timeit.timeit(for_loop_with_increment, number = 1))
print('for_loop_with_test\t\t', timeit.timeit(for_loop_with_test, number = 1))
print('for_loop_with_increment_and_test\t\t', timeit.timeit(for_loop_with_increment_and_test, number = 1))
print('sum_range\t\t', timeit.timeit(sum_range, number = 1))
print('sum_numpy\t\t', timeit.timeit(sum_numpy, number = 1))
print('sum_math\t\t', timeit.timeit(sum_math, number = 1))

# python programming consideration
# use math formula
# use c implementation
# use built-in function, sum, map ... which loops for you
# for or while loop


```

```
while loop		0.1154504309999993
for loop		3.713000069183181e-06
for_loop_with_increment		0.11514549399998941
for_loop_with_test		0.0975002409999206
for_loop_with_increment_and_test		0.1432448260000001
sum_range		0.022303372999999738
sum_numpy		0.005475752999927863
sum_math		2.3169999394667684e-06
```

Different loops and their performances

In [53]:
```python
## memoization or cache to optimize
# useful for recursive functions, or operations used over and over again
# you don't want to repeat to calculate values again
```

In [115]:
```python
# use cache dict
from time import perf_counter
from functools import wraps

def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]
    return wrapper

```

In [116]:
```python
# fibonacci using memoize
def fibonacci_plain(n=100) -> int:
    if n < 2:
        return n
    return fibonacci_plain(n-1) + fibonacci_plain(n-2)
```

In [*]:
```python
# no memoization call, very slow, cpu humming, 20 mins still running, killed this cell
start = perf_counter()
fibonacci_plain()
end = perf_counter()
print(end-start)
```

In [57]:
```python
# fibonacci using memoize
@memoize
def fibonacci(n=1000) -> int:
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

In [58]:
```python
# get result instantly
print('fibonacci with memorize\t\t', timeit.timeit(fibonacci, number = 1))
```

```
fibonacci with memorize		0.0036856550000265997
```

Memoization and caching reducing intermediate operations

Screen Shot 2023

```python
    outputs = []
    for url in urls.values():
        print(url)
        outputs = outputs + [requests.get(url).text
        #print(outputs)

    count_https = []
    count_http = []
    for output in outputs:
        count_https += re.findall("https://", outpu
        count_http += re.findall("http://", output)

    print(len(count_https), len(count_http))

# index = 0
# while count_https[index]:
#     if index >= len(count_https):
#         break
#     index +=  1

start = time.perf_counter()
count_words_in_web_page()
elapsed = time.perf_counter() - start
print(f'{elapsed:.2f} seconds')
```

```
https://google.com
https://yahoo.com
https://microsoft.com
https://google.com
https://apple.com
https://ibm.com
https://amazon.com
https://twitter.com
https://tiktok.com
https://oracle.com
https://intel.com
https://tesla.com
https://nasa.com
https://ebay.com
https://wikipedia.com
3071 732
10.50 seconds
```

```python
    "2": "https://yahoo.com",
    "3": "https://microsoft.com",
    "4": "https://google.com",
    "5": "https://apple.com",
    "6": "https://ibm.com",
    "7": "https://amazon.com",
    "8": "https://twitter.com",
    "9": "https://tiktok.com",
    "10": "https://oracle.com",
    "11": "https://intel.com",
    "12": "https://tesla.com",
    "13": "https://nasa.com",
    "14": "https://ebay.com",
    "15": "https://wikipedia.com"
}

# mark as async
async def count_words_in_web_page_async():
    outputs = []

    async with httpx.AsyncClient() as client:
        tasks = (client.get(url) for url in urls.values())
        reqs = await asyncio.gather(*tasks)    # waits for task, but await till all donee

        outputs = [req.text for req in reqs]
        #print(outputs)

    count_https, count_http =[], []
    for output in outputs:
        count_https += re.findall("https://", output)    # text processing, not use pre-compiled re
        count_http += re.findall("http://", output)
#     print(count_https)
#     print(count_http)


start = time.perf_counter()
await (count_words_in_web_page_async())   # schedule func to run
# asyncio.run(count_words_in_web_page_async())   # for python>3.7 and ipython < 7.0
elapsed = time.perf_counter() - start
print(f'{elapsed:.2f} seconds')
```

```
1.03 seconds
```

10.50 vs 1.03 - Use async for web text scraping

```
(base) user-2:bin user$ cat main.rs
extern crate webserver;

use webserver::ThreadPool;
use std::net::TcpListener;
use std::io::prelude::*;
use std::net::TcpStream;
use std::fs::File;
use std::thread;
use std::time::Duration;

fn main() {

        let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
        let pool = ThreadPool::new(8);

        for stream in listener.incoming() {

                let stream = stream.unwrap();

                pool.execute(|| {
                        handle_connection(stream);
                });

        }

}

fn handle_connection(mut stream: TcpStream) {

        let mut buffer = [0; 512];

        stream.read(&mut buffer).unwrap();

        let get = b"GET / HTTP/1.1\r\n";
        let sleep = b"GET /sleep HTTP/1.1\r\n";

        let (status_line, filename) = if buffer.starts_with(get) {
                ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
        } else if buffer.starts_with(sleep) {
                thread::sleep(Duration::from_secs(5));
                ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
        } else {
                ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
        };

    let mut file = File::open(filename).unwrap();
    let mut contents = String::new();

    file.read_to_string(&mut contents).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Rust multi thread server

```
url = 'http://localhost:7878/'

def  fetch(session, url):
    with session.get(url) as response:
        #print(response)
        pass

@timer(1, 1)
def main():
    with requests.Session() as session:
        for _ in range(5000):
            fetch(session, url)


import requests
import timeit
from multiprocessing.pool import Pool

url = 'http://localhost:7878/'

def fetch(session, url):
    with session.get(url) as response:
        #print(response)
        pass

def timer(number, repeat):
    def wrapper(func):
        runs = timeit.repeat(func, number=number, repeat=repeat)
        print(sum(runs) / len(runs))
    return wrapper

if __name__ ==  "__main__":

    @timer(1, 1)
    def task():
        with Pool() as pool:
            with requests.Session() as session:
                pool.starmap(fetch, [(session, url) for _ in range(5000)])
```

```
(base) user-2:client user$
(base) user-2:client user$ cat ../readme.txt
# start server
(base) user-2:rust_web_server_concurrent user$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/main`

# start 01_simple-http.py
# synchronous calls
(base) user-2:client user$ python 01_simple_http_sync.py
11.093317224


# multiprocessing using multi-cores to run multi processeso
(base) user-2:client user$ python 02_multi_processing_http.py
2.7904895300000003
```

Python web api call

Python web api call

```
url = 'http://localhost:7878/'

def  fetch(session, url):
    with session.get(url) as response:
        #print(response)
        pass

@timer(1, 1)
def main():
    with requests.Session() as session:
        for _ in range(5000):
            fetch(session, url)


import requests
import timeit
from multiprocessing.pool import Pool

url = 'http://localhost:7878/'

def fetch(session, url):
    with session.get(url) as response:
        #print(response)
        pass

def timer(number, repeat):
    def wrapper(func):
        runs = timeit.repeat(func, number=number, repeat=repeat)
        print(sum(runs) / len(runs))
    return wrapper

if __name__ ==  "__main__":

    @timer(1, 1)
    def task():
        with Pool() as pool:
            with requests.Session() as session:
                pool.starmap(fetch, [(session, url) for _ in range(5000)])
```

```
(base) user-2:client user$
(base) user-2:client user$ cat ../readme.txt
# start server
(base) user-2:rust_web_server_concurrent user$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/main`

# start 01_simple-http.py
# synchronous calls
(base) user-2:client user$ python 01_simple_http_sync.py
11.093317224


# multiprocessing using multi-cores to run multi processeso
(base) user-2:client user$ python 02_multi_processing_http.py
2.7904895300000003
```

In [2]:
```
!python -V
```
```
Python 3.10.11
```

In [1]:
```
import math
import numpy as np
```

Factorial goes faster

In [39]:
```
# basic factorial
%timeit math.prod(range(1, 150))
```
```
10.6 µs ± 32.3 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [33]:
```
math.prod(range(1,6))
```
Out[33]:
```
120
```

In [34]:
```
def f(x):
    return x * f(x-1) if x > 1 else 1
```

In [28]:
```
f(5)
```
Out[28]:
```
120
```

In [40]:
```
%timeit math.factorial(150)
```
```
1.8 µs ± 3.93 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

In [44]:
```
(1.8/10.6)
```
Out[44]:
```
0.169811320754717
```

Use new versions

compute factorials like a boss

Winning ideas:
Left shift is cheaper than multiplying by two
Pulling out events leaves recurring odd factories
Dynamic programming reuses previously computed odd factorials

In [75]:
```
s = ""
so = ""
se = ""
for i in range(1,20):
    s += str(i) + '*'
    if i%2 == 0:
        se += str(i) + '*'
    else:
        so += str(i) + '*'
print(s, so, se)
```

```
1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19* 1*3*5*7*9*11*13*15*17*19* 2*4*6*8*10*12*14*16*18*
```

In [71]:
```
1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

Out[71]: 121645100408832000

In [79]:
```
# collect even terms, next divide each even by 2
1*3*5*7*9*11*13*15*17*19 *2*4*6*8*10*12*14*16*18
```

Out[79]: 121645100408832000

In [81]:
```
# divide terms by two and replace with left shift
1*3*5*7*9*11*13*15*17*19 *1*2*3*4*5*6*7*8*9 << 9
```

Out[81]: 121645100408832000

In [84]:
```
1*3*5*7*9*11*13*15*17*19 *1*3*5*7*9 * 1*2*3*4 <<13
```

Out[84]: 121645100408832000

In [86]:
```
1*3*5*7*9*11*13*15*17*19 *1*3*5*7*9 * 1*3 *1*2 <<15
```

Out[86]: 121645100408832000

In [88]:
```
# replace even term with left shift
1*3*5*7*9*11*13*15*17*19 *1*3*5*7*9 * 1*3 <<16
```

Out[88]: 121645100408832000

In [93]:
```
# factor-out common subsequences and replace with powers
(1*3)**3 *(5*7*9)**2 *(11*13*15*17*19)**1 <<16
```

Out[93]: 121645100408832000

How python 3 implement factorial() performance

Conclusions

-Python is the programming language to solve all problems
-Python is a popular language for many use cases
-Performance can be improved
-Python is improving, by itself and community

-In the era of ML/AI, no doubt python will become popular
-To learn, experiment, build quickly MVP in less time

Q & A