Python Code Optimization

Ayush Kumar Shah

Contents

1 General Tips

- Use real data
- Turn off anti virus
- Have a test suite ready
- Know when to measure

2 Measuring time

2.1 Using perf_counter

```
from time import perf_counter
start = perf_counter()
some_fun()
duration = perf_counter() - start
# gives results in seconds
```

2.2 Using timeit

For repeatedly executing and measuring the average.

```
from timeit import timeit
if __name__ == '__main__'
    print('catch', timeit('func_name("argument")',
          'from __main__ import func_name'))
```

2.3 In IPython

```
## Loading the functions from external file
%run -n module_name.py
```

-n means not to run main file

```
%timeit func_name('argument')
```

3 CPU Profiling

- cProfile (deterministic profiler) recommended
- Deterministic profilers record every function call, return, and exception.
- Statistical profilers record where the program is at small intervals.
- pstats to display profiler generated statistics file

3.1 Running cProfiler

3.1.1 Whole file

```
python -m cProfile module.py
```

3.1.2 Portion of file

Add the line in code:

```
if __name__ == '__main__':
    ...
    # where profiler needed
    import cProfile
    cProfile.run('fun_name(cases)')
    ## OR
    cProfile.run('fun_name(cases)', filename='prof.out or program.prof')
```

Then run the module

```
python module.py
```

In notebook

```
## Importing functions from the module
%run -n module.py
```

```
%prun fun_name(args)
## OR
%prun -D program.prof fun_name(args)

## For more info about prun
%prun?

## Sorting
%prun -s cumulative fun_name(args)
```

3.1.3 Snakeviz

```

snakeviz prof.out/program.prof
```

In notebook

```
%load_ext sankeviz
```

For single line code:

```
% sankeviz fun_name(args)
```

For multiple lines code:

```
%%snakeviz
# code here
# ....
# ...
```

Note: cProfile slows down the program. Other lightweight statistical Profilers like: vn_prof package

4 Line Profiler

- For finer granularity than just functions i.e. time taken by each line in the function
- need to install it `pip install line_profiler`
- kernprof - command line program
- Use @profile decorator

4.1 Running and viewing

```
kernprof -l module.py
## To view results
python -m line_profiler module.py module.py.lprof
```

4.2 In IPython

```
%run -n prof.py
```

```
%load_ext line_profiler
```

```
%lprun -f subfun_to_profile fun_name(args)
```

5 Tracing memory allocations

Why? - Data is stored in memory and newly created objects require storage allocation which takes time (latency) - So, memory allocation is important - Accessing memory is done in layers (CPI, Level 1 cache, Level 2 Cache, Memory, SSD / HDD(if extra memory required, memory swapped using page vault)) - Accessing Level 1 cache is fast (0.5 ns), L2 - 7 ns, Memory - 100 ns, SSD - 16,000 ns

Memory traceback can be done using `tracemalloc`

```
import tracemalloc
...
...
tracemalloc.start()

## Operations on files (encoding events using json or yaml)
```

```
...
...
snapshot = tracemalloc.take_sanpshot()
for stat in snapshot.statistics('lineno')[:10]:
  print(stat)
```

6 Memory Profiler

- used for computing memory usage statistics
- @profile decorator used
- Install using `pip install memory_profiler`

6.1 Running

```
python -m memory_profiler module.py
```

6.2 Time based memory profiling

- To see if the memory grows with time.

```
mprof run module.py
mprof plot mprofile_timestamp.dat
```

6.3 IPython

- Check documentation

```
%load_ext memory_profiler
```

```
%mprun -f subfun_to_profile function(args)
```

7 Tips for optimization

- Pick the right data structure (dequeu, heap, PriorityQueue, bisect)
- Local caching of names

  To check how python compiles a function:

```python
import dis
dis.dis(fun_name)
```

Global look up takes more time. So, save a local copy in the local function.

- Remove function calls (TANSTAAFL) since function call increases time. So, find trade off between structure and time cost. Use inline function if possible.
- Don't use property (setter and getter) unless necessary. Since propert is almost 4 times slower than normal.
- Using `__slots__`
  - Too many small objects can cause problems.
  - Object attributes are stored in `__dict__` which is 1/3 empty
  - `__slots__` removes `__dict__` (Con: attributes can't be added)
  - Use memory profiler to check memory when using `__slots__`. (Memory reduced to almost half, hence also makes faster since they can fit into CPU cache line, which is faster to access than main memory)
- using built-ins

  Eg. Sorting tasks according to priority

```python
sorted(tasks, key = lambda task: task[1])
```

Lambda function is called once per object, hence total complexity is not `n logn` due to caching.

But there are functions written in C but faster.

```python
from operator import itemgetter
sorted(tasks, key = itemgetter(1))
```

Similar built in functions which are written in C and are faster

  - `attrgetter()` - gets attribute to return
  - `sum()`
  - `max()`

- Allocate

```python
"""Fixed list allocation"""

def allocz(size):
    """Alloc zeros with range"""
    return [0 for _ in range(size)]


def allocz_fixed(size):
    """Alloc zeros with *"""
    return [0] * size
```

The second is ten times faster. When list is created and appended, python needs to reallocatr memory for the list. To avoid reallocation on every append, the list grows in fixed sizes. The sizes are 0, 4, 8, 16, 25, etc.

However, in the second approach, we know the size of the list in advanced and hence is created in one allocation.

While using this, make sure you initialize with immutable elements, since it is duplicated otherwise. Another option is use numpy's `np.zeros(size)`

- Make approximations if possible. It is okay to cheat. E.g. 1.1 * 1.1 doesn't return accurate result.

8 Caching

The act of saving computation results and reusing them instead of running the calculations again. It will speed up the execution but the memory is used to hold the results i.e. `Runtime vs memory trade-off.`

Example of fibonacci sequence program

```python
"""Cachine fibonnaci"""

def fib(n):
    """Return nth fiboacci number"""
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)


_fib_cache = {}


def fibc(n):
    """Return nth fiboacci number (cached)"""
    if n < 2:
        return 1

    val = _fib_cache.get(n)
    if val is None:
        _fib_cache[n] = val = fibc(n-1) + fibc(n-2)
    return val
```

- 1000 times faster.
- Take care of caching validation (important)

8.1 Pre-calculating results

Example: Counting number of 1s in a number (binary). Pre-calculate the results.

8.2 LRU Cache

```
from functools import lru_cache

@lru_cache(maxsize=1024)
def user_from_key(key):
    enc_key = crypt(key, salt)
    return users.get(enc_key)
```

8.3 Joblib

- Keep the computation cache between program runs
- Offers on-disk caching and parallel computing

```
time python module.py
```

9 Parallel Computing

- Parallelism vs Concurrency
  - Concurrency: dealing with lots of things at once.
  - Parallelism: doing lots of things at once.
- Amdahl's Law : formula which gives theoretical limits of speed increase or latency as a result of parallel computing.

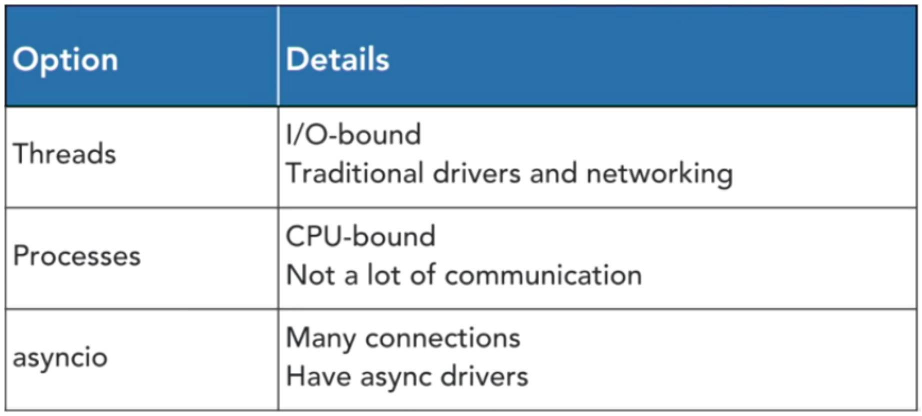
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

9.1 I/O Bound vs CPU Bound

- I/O Bound programs - spend time in input/output such as network or disk and
- CPU Bound Programs - spend time in calculations

Python provides 3 options for parallelism or concurrency: - Threads - independent units of execution that share the same memory space. Global variables are visible to all threads. Great for IO bound but not for CPU bound (due to GIL: CPython can use only one core from the CPU) - processes - independent units of execution that have different memory space. Using multiple processes, you can use multiple cores, hence great for CPU bound programs. However, no communication between processes is expensive since memory is not shared. So, data needs to be serialized, send it over socket/pipe and deserialize it on the other side. - asyncio - handle a lot of connections. uses a single thread to listen to all connections.

9.2 Summary of usage



Option	Details
Threads	I/O-bound Traditional drivers and networking
Processes	CPU-bound Not a lot of communication
asyncio	Many connections Have async drivers

Figure 1: summary

9.3 Threads

Example: Getting user info of many users in parallel from GitHub.

First, code normally and analyze using timeit and profiler. If CPU time is very less than the Wall Time and most time being spent in socket operations, then threading will improve the time.

```python
"""Thread Pool Example"""

from collections import namedtuple
from concurrent.futures import ThreadPoolExecutor
from datetime import datetime
from urllib.request import urlopen
import json

User = namedtuple('User', 'login name joined')


def user_info(login):
    """Get user information from github"""
    fp = urlopen('https://api.github.com/users/{}'.format(login))
    reply = json.load(fp)

    joined = datetime.strptime(reply['created_at'], '%Y-%m-%dT%H:%M:%SZ')
    return User(login, reply['name'], joined)


def users_info(logins):
    """Get user information for several users"""
    return [user_info(login) for login in logins]


def users_info_thr(logins):
    """Get user information for several users - using thread pool"""
    with ThreadPoolExecutor() as pool:
        return list(pool.map(user_info, logins))


if __name__ == '__main__':
    logins = [
        'gvanrossum',
        'wesm',
        'tebeka',
        'torvalds',
    ]
```

9.4 Processes

Example: Decompressing data

First, code normally and analyze using timeit and profiler. If CPU time is almost same as the Wall Time and most time being spent in computations, then processes will improve the time.

```python
"""Process pool example"""
from concurrent.futures import ProcessPoolExecutor
import bz2


def unpack(requests):
    """Unpack a list of requests compressed in bz2"""
    return [bz2.decompress(request) for request in requests]


def unpack_proc(requests):
    """Unpack a list of requests compressed in bz2 using a process pool"""
    # Default to numbers of cores
    with ProcessPoolExecutor() as pool:
        return list(pool.map(bz2.decompress, requests))


if __name__ == '__main__':
    with open('huck-finn.txt', 'rb') as fp:
        data = fp.read()

    bz2data = bz2.compress(data)
    print(len(bz2data) / len(data))  # About 27%
    print(bz2.decompress(bz2data) == data)   # Loseless

    requests = [bz2data] * 300
```

9.5 Asyncio

See example code

10 Other Tips

- Use numpy if numerical several computations required.
- Use numba

  language level optimization is difficult as Python is dynamically interpreted language. Using JIT(Just In Time) Compilation - for functions which runs multiple times.

  E.g. calculating polynomial functional value from coefficients.

```python
import numba

@numba.jit
def fun(args):
    ....
    ## no change
```

Can also run in GPU and works better if type information provided in decorator.

- Use cython
  - Superset of Python - Python + type connections to C libraries
  - C compiler required
  - Written as .pyx files
  - See example code
  - Disadvantages:
    - Build step required
    - Model dependent on architecture (OSx will not work in Windows)
- Using other implementations of Python
  - Jython (written in JAVA)
  - MicroPython (optimized for MicroController)
  - IronPython (written in .NET)
  - PyPy (written in Python, includes a JIT compiler)
- Use C Extensions - Python C API
- Design and Code Reviews - Prepare a checklist
- Use Pytest Benchmarks
- Monitoring and Alerting