Python Code Performance Measurement – Measure the right metric to optimize better!

Introduction

Performance optimization is an important concern in any data science project. Since most of the project runs on cloud platforms, there is always a cost factor associated with computational resources. We aim to write production codes in an optimized way and build machine learning models that are effective and performance-friendly.

But how do we measure the performance of a block of code or a

Performance Measurement metrics

We should measure the performance of blocks of python code in a project by recording the execution time and by finding the amount of memory being used by the block. This will help us to know the size of the system required to run the application and also get an idea of the duration of the run. In this article, we will discuss the implementation of various logics using 2 different ways and measuring the performance using time and tracemalloc libraries.

Implementation

1. Import Packages

Below are the packages that are used in this demo. The tracemalloc module used here gives the statistics on allocated memory blocks and also computes the differences between two snapshots to detect memory leaks. I have used get_traced_memory() to get the current size and peak size of memory blocks.

```
import tracemalloc
import pandas as pd
import dask.dataframe as dd
import time
```

2. Defining function to use tracemalloc

The first function is to stop the tracing, if any, and to start the fresh tracing. This ensures that the tracing of 2 code blocks doesn't interfere with each other. The second function is to find the peak size in MB of a block of code that was traced.

```
def tracing_start():
    tracemalloc.stop()
    print("nTracing Status : "
    tracemalloc.start()
    print("Tracing Status : ",
def tracing_mem():
    first_size, first_peak = t
    peak = first_peak/(1024*10
    print("Peak Size in MB - "
```

3. Comparing the peak size and time required for the run

The tracing functions created above is called to measure the peak memory size and the time required to run the commands.

A. Operations on elements of a list

The following blocks show the operation on each element of the list and store the result into a new list.

Method – 2

```
tracing_start()
start = time.time()
sq_list1 = [elem + elem**2 for
#print(sq_list1)
end = time.time()
print("time elapsed {} milli se
tracing_mem()
```

```
#Result
```

```
Tracing Status :  False
Tracing Status :  True
time elapsed 7.999420166015625
Peak Size in MB -  0.046345565
```

Method - 1

```
tracing_start()
start = time.time()
list_word = ["Quantify","perfo
s = ""
for substring in list_word:
    s += substring + " "
print(s)
end = time.time()
print("time elapsed {} milli se
tracing_mem()
```

```
#Result
```

```
Tracing Status :  False
Tracing Status :  True
time elapsed 0.0 milli seconds
Peak Size in MB -  0.015248298
```

Method – 2

```
tracing_start()
start = time.time()
list_word = ["Quantify","perfo
s = ""
" ".join(list_word)
print(s)
end = time.time()
print("time elapsed {} milli se
tracing_mem()
```

```
#Result
```

```
Tracing Status :  False
Tracing Status :  True
time elapsed 0.0 milli seconds
Peak Size in MB -  0.012733459
```

Conclusion

In this article, we discussed how to find the memory utilized while running a block of code in Python. We discussed the snippets of 4 use cases and listed down the time taken for execution and peak memory being consumed. The result is summarized below –

Use cases	Methods	Peak Size (MB)	Time of execution (ms)
Operation on elements of a list	Conventional loop	0.048	8.2
	List comprehension	**0.046**	**7.99**
String concatenation	Using plus operator	0.015	0
	Using join	**0.012**	**0**
Removing duplicate element from list	Using not in operator	0.014	0
	Using set operation	**0.012**	**0**
Reading a csv file into dataframe and group by operation	Using Pandas	4.76	136
	Using dask	**1.22**	**88**