2b.Timing

November 2, 2024

1 Timing and optimizing code

Q: Are computers so fast that optimizing code for speed no longer matters?

1.1 The Context

- It depends
- Often optimizing isn't important
- Computers are faster
- It still can matter
- Try to learn & use best-practices

1.2 Video

Modern Data Warehousing with BigQuery - Watch from 5:00 to 7:20.

1.2.1 Take-aways

- You might care about speed when dealing with big data
- Some of the cutting edge problems to be solved *still* have to do with optimization

Considerations - Size of dataset - How often a task needs to be done - Time it takes to make process faster/time it takes to run it as is. - How much are you going to learn from the process of making your code faster?
- Is anybody else going to see your code?

1.3 How to speed up your code?

Below are a few easy ways.

1.3.1 1. List Comprehensions vs For Loop

[ ]:
```
import numpy as np
```

[ ]:
```
r = 1_000_000
```

[3]:
```
# For Loop
# %% will turn this into a cell magic
# It will work on multi-line code
%%timeit
```

```
squares = []

for n in range(r):
  squares.append(n**2)
```

```
507 ms ± 113 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[4]:
```
# List Comprehension
# % indicates a magic command
# %timeit will apply to one line
result = %timeit -r3 -n10 -o squares = [n**2 for n in range(r)]
result
```

```
380 ms ± 63.5 ms per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

[4]:
```
<TimeitResult : 380 ms ± 63.5 ms per loop (mean ± std. dev. of 3 runs, 10 loops
each)>
```

[5]:
```
result.all_runs
```

[5]:
```
[4.687220814, 3.4598238140000035, 3.2474136390000012]
```

1.3.2 2. Numpy Arrays

[6]:
```
%timeit squares = (np.arange(r))**2
```

```
1.8 ms ± 172 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[7]:
```
%timeit squares = list((np.arange(r))**2)
```

```
63.6 ms ± 11.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[8]:
```
%timeit?
```

1.3.3 3. Multiple Assignments

[9]:
```
%%timeit
a = 5
b = 10
c = 20
d = 25
```

```
34.7 ns ± 0.611 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[10]:
```
%timeit a,b,c,d = 5,10,20,25
```

```
34.3 ns ± 0.798 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

1.3.4 4. Use F Strings

[11]:
```
name = 'Dwight Schrute'
```

[12]:
```
%timeit name + ' loves beets'
```

```
97 ns ± 28.6 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[13]:
```
%timeit '%s loves beets' % name
```

```
168 ns ± 42.2 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[14]:
```
%timeit  '{} loves beets'.format(name)
```

```
227 ns ± 8.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

[15]:
```
%timeit f'{name} loves beets'
```

```
101 ns ± 27.4 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

1.3.5 5. Use Built-In Functions

Built-in functions such as `len()`, `abs()`, `min()`, and `max()` are written in C and are therefore very efficient.

1.3.6 6. Use Enumerate

[16]:
```
our_list = [i**2 for i in range(5)]
our_list
```

[16]:
```
[0, 1, 4, 9, 16]
```

[17]:
```
# Without enumerate
# note using time only runs the script once
%%time
output = list(range(len(our_list)))
for i in range(len(our_list)):
  output[i] = (f'{i}^2 = {our_list[i]}')

print("\n".join(output))
```

```
0^2 = 0
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
CPU times: user 128 µs, sys: 4 µs, total: 132 µs
Wall time: 391 µs
```

[18]:
```
%%time
for i, item in enumerate(our_list):
```

```
  print(f'{i}^2 = {item}')
```

```
0^2 = 0
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
CPU times: user 126 µs, sys: 3 µs, total: 129 µs
Wall time: 134 µs
```

[19]:
```
%%time
print("\n".join( f'{i}^2 = {item}' for i, item in enumerate(our_list) ) )
```

```
0^2 = 0
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
CPU times: user 1.85 ms, sys: 0 ns, total: 1.85 ms
Wall time: 1.76 ms
```

There are many other ways to speed up your code, you just have to decide if it is needed for whatever application you are working on.

1.4 Metric Prefixes & Scientific Notation

Wikipedia Metric Prefixes & Symbols

288.0 ms 10E-3 - decimal place goes over three -> 0.288 s

3.59 µs 10E-6 - decimal place goes over six places -> 0.00000359 s

[ ]:
```

```