Introduction to Computing

Efficient Programming in Python

Malay Bhattacharyya

Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

October, 2024

1 Time Analysis

2 Efficient Indexing

3 Operators

4 Conditioning

5 Efficient Loops

6 Avoiding library functions

7 Others

How to measure the elapsed time?

```
import time
start = time.time()
# INCLUDE YOUR CODE HERE
end = time.time()
print(end - start)
```

How to measure the elapsed time?

```
import time
start = time.time()
# INCLUDE YOUR CODE HERE
end = time.time()
print(end - start)
```

Note: This gives the execution time in seconds.

Let's perform a comparative analysis!!!

```
import statistics as st
import time
start = time.time()
ls = [5]*10000
st.mean(ls)
end = time.time()
print(end - start)
```

```
import statistics as st
import time
start = time.time()
ls = [5]*10000
st.fmean(ls)
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
import statistics as st
import time
start = time.time()
ls = [5]*10000
st.mean(ls)
end = time.time()
print(end - start)
```

Output:
0.008999824523925781

```
import statistics as st
import time
start = time.time()
ls = [5]*10000
st.fmean(ls)
end = time.time()
print(end - start)
```

Output:
0.0009999275207519531

Efficient use of negative indexing

Negative indexing allows to index into a list, tuple or other indexable container relative to the end of the container, rather than the start.

Note: If the last index would have been -0, then it would be the same as the first index because -0 = 0.

Let's perform a comparative analysis!!!

```
ls = [range(1,100)]
import time
start = time.time()
for i in range(1,10000):
    ls[-1]
end = time.time()
print(end - start)
```

```
ls = [range(1,100)]
import time
start = time.time()
for i in range(1,10000):
    ls[n-1]
end = time.time()
print(end - start)
```

```
ls = [range(1,100)]
import time
start = time.time()
for i in range(1,10000):
    ls[len(ls)-1]
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
ls = [range(1,100)]
import time
start = time.time()
for i in range(1,10000):
    ls[-1]
end = time.time()
print(end - start)
```

Output:
0.0019998550415039062

```
ls = [range(1,100)]
import time
start = time.time()
for i in range(1,10000):
    ls[n-1]
end = time.time()
print(end - start)
```

Output:
0.0025999485015869140

```
ls = [range(1,100)]
import time
start = time.time()
for i in range(1,10000):
    ls[len(ls)-1]
end = time.time()
print(end - start)
```

Output:
0.0039999485015869140

Optimizing the use of operators

For basic data types, the operators $+$, $-$, $*$, $/$ and $+=$, $-=$, $*=$, $/=$, performs respectively the same.

However, for derived data types, $+=$, $-=$, $*=$, $/=$ might be faster.

Let's perform a comparative analysis!!!

```
import time
start = time.time()
sum = 0
for i in range(1,10000):
    sum = sum + i
end = time.time()
print(end - start)
```

```
import time
start = time.time()
sum = 0
for i in range(1,10000):
    sum += i
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
sum = 0
for i in range(1,10000):
    sum = sum + i
end = time.time()
print(end - start)
```

Output:
0.0030002593994140625

```
import time
start = time.time()
sum = 0
for i in range(1,10000):
    sum += i
end = time.time()
print(end - start)
```

Output:
0.0029997825622558594

Efficient use of bitwise operators – Increment

Note that, $x + (2\text{'s complement of } x) = 0$
$\implies x + ((1\text{'s complement of } x) + 1) = 0$
$\implies x + 1 = -\ (1\text{'s complement of } x)$.

Hence, We can use the following representation to increment the value of a variable x by 1:

```
-~x
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
x = 0
for i in range(1,10000):
    x + 1
end = time.time()
print(end - start)
```

```
import time
start = time.time()
x = 0
for i in range(1,10000):
    -~x
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
x = 0
for i in range(1,10000):
    x + 1
end = time.time()
print(end - start)
```

Output:
0.000700002098083496

```
import time
start = time.time()
x = 0
for i in range(1,10000):
    -~x
end = time.time()
print(end - start)
```

Output:
0.0019998550415039062

Efficient use of bitwise operators – Multiplication

Try to avoid arithmetic multiplication as and when possible.

```
m = 123
val = m * 65
```

Efficient use of bitwise operators – Multiplication

Try to avoid arithmetic multiplication as and when possible.

```
m = 123
val = m * 65
```

It should be written using *bitwise left shift* as follows.

```
m = 123
val = (m << 6) + m
```

Note: `m * n` will return the same result as that of `(m << p) + m`, where `n` can be represented as $2^p + 1$.

Efficient use of bitwise operators – Modular division

Never perform modular division with a power of 2.

```
m = 97
val = m % 8
```

Efficient use of bitwise operators – Modular division

Never perform modular division with a power of 2.

```
m = 97
val = m % 8
```

It should be written using *bitwise AND* as follows.

```
m = 97
val = m & 7
```

Note: `m % n` will return the same result as that of `m & (n-1)`, where `n` is a power of 2.

Efficient use of bitwise operators – Checking alphabets

Never perform case insensitive alphabet checking as follows.

```
if ch == 'a' or ch == 'A':
    vowel += 1
if ch == 'e' or ch == 'E':
    vowel += 1
if ch == 'i' or ch == 'I':
    vowel += 1
if ch == 'o' or ch == 'O':
    vowel += 1
if ch == 'u' or ch == 'U':
    vowel += 1
```

Efficient use of bitwise operators – Checking alphabets

Never perform case insensitive alphabet checking as follows.

```
if ch == 'a' or ch == 'A':
    vowel += 1
if ch == 'e' or ch == 'E':
    vowel += 1
if ch == 'i' or ch == 'I':
    vowel += 1
if ch == 'o' or ch == 'O':
    vowel += 1
if ch == 'u' or ch == 'U':
    vowel += 1
```

It can be made faster using *bitwise OR* as follows.

```
ch = ch | 0x20
if ch == 'a':
    vowel += 1
if ch == 'e':
    vowel += 1
if ch == 'i':
    vowel += 1
if ch == 'o':
    vowel += 1
if ch == 'u':
    vowel += 1
```

Lazy evaluation

Conditionals/Boolean expressions in C are evaluated from left to right. Evaluation stops as soon as the value of the expression is known. Remaining sub-expressions are not evaluated.

Lazy evaluation

Conditionals/Boolean expressions in C are evaluated from left to right. Evaluation stops as soon as the value of the expression is known. Remaining sub-expressions are not evaluated.

Examples:

1. `(x > y) and (a != b)`: If x is less than y, then the expression is False, irrespective of the value of the second sub-expression
2. `(n > 0) or (i == j)`: If n is greater than 0, the expression is True, irrespective of the value of the second sub-expression

Lazy evaluation

Typical usage:

```
while i < N and ls[i] >= 0:
    ...
```

- If $i >= N$, $ls[i]$ is not checked.
- This is useful because checking $ls[i] >= 0$ when $i >= N$ may lead to memory faults.
- In such expressions, $i >= N$ serves as a guard condition.

Loop jamming

Never use two loops where one will suffice.

```
for i in range(0, 10):
    <Statement 1>
for i in range(0,10):
    <Statement 2>
```

Loop jamming

Never use two loops where one will suffice.

```
for i in range(0, 10):
    <Statement 1>
for i in range(0,10):
    <Statement 2>
```

It should be written as follows:

```
for i in range(0,10):
    <Statement 1>
    <Statement 2>
```

Note: If a single loop contains a lot of operations (it might not fit into the processor's instruction cache), then two separate loops may be faster than a single combined one.

Loop unrolling

The loop overhead can be reduced by decreasing the number of iterations and replicating the body of the loop.

```
for i in range(0,100):
    <Statement>
```

Loop unrolling

The loop overhead can be reduced by decreasing the number of iterations and replicating the body of the loop.

```
for i in range(0,100):
    <Statement>
```

The above code can be written as follows:

```
for i in range(0,100,2):
    <Statement>
    <Statement>
```

Note: The need for condition check adds some overhead to the program.

Loop inversion

We should always write count-down-to-zero loops and use simple termination conditions for a better efficiency.

Loop inversion

We should always write count-down-to-zero loops and use simple termination conditions for a better efficiency.

```
fact = 1
i = 1
while i <= n:
    fact *= i
    i += 1
```

Loop inversion

We should always write count-down-to-zero loops and use simple termination conditions for a better efficiency.

```
fact = 1
i = 1
while i <= n:
    fact *= i
    i += 1
```

The above code can be made efficient by writing as follows:

```
fact = 1
i = n
while i:
    fact *= i
    i -= 1
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
for i in range(1,10000):
    fact = 1
    i = 1
    while i <= 10:
        fact *= i
        i += 1
end = time.time()
print(end - start)
```

```
import time
start = time.time()
for i in range(1,10000):
    fact = 1
    i = 10
    while i:
        fact *= i
        i -= 1
end = time.time()
print(end - start)
```

```
import time
start = time.time()
for i in range(1,10000):
    fact = 1
    for i in range(1,11):
        fact *= i
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
for i in range(1,10000):
    fact = 1
    i = 1
    while i <= 10:
        fact *= i
        i += 1
end = time.time()
print(end - start)
```

Output:
0.03900003433227539

```
import time
start = time.time()
for i in range(1,10000):
    fact = 1
    i = 10
    while i:
        fact *= i
        i -= 1
end = time.time()
print(end - start)
```

Output:
0.03899979591369629

```
import time
start = time.time()
for i in range(1,10000):
    fact = 1
    for i in range(1,11):
        fact *= i
end = time.time()
print(end - start)
```

Output:
0.004299993133544922

Loops versus list comprehensions

List comprehensions are faster than for loops only to create lists. This is because we are creating a list by appending new elements to it at each iteration.

However, if we want to perform some computations (or call an independent function multiple times) and do not want to create a list, then for loops are faster than list comprehensions.

Let's perform a comparative analysis!!!

```
import time
start = time.time()
ls = ['x', 'xyxy', 'xyxyx']
print([l for l in ls if l == l[::-1]])
end = time.time()
print(end - start)
```

```
import time
start = time.time()
ls = ['x', 'xyxy', 'xyxyx']
lsNew = []
for l in ls:
    if l == l[::-1]:
        lsNew.append(l)
print(lsNew)
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
ls = ['x', 'xyxy', 'xyxyx']
print([l for l in ls if l == l[::-1]])
end = time.time()
print(end - start)
```

Output:
0.00002598762512207031 2

```
import time
start = time.time()
ls = ['x', 'xyxy', 'xyxyx']
lsNew = []
for l in ls:
    if l == l[::-1]:
        lsNew.append(l)
print(lsNew)
end = time.time()
print(end - start)
```

Output:
0.00002074241638183594

List comprehensions versus generator expressions

List comprehensions are usually faster than generator expressions as generator expressions create another layer of overhead to store references for the iterator. However, list comprehensions allocate more memory than generator expressions.

Let's perform a comparative analysis!!!

```
import time
import sys
start = time.time()
lc = [num for num in range(10000)]
sum(lc)
end = time.time()
print(end - start)
print(sys.getsizeof(lc))
```

```
import time
import sys
start = time.time()
ge = (num for num in range(10000))
sum(ge)
end = time.time()
print(end - start)
print(sys.getsizeof(ge))
```

Let's perform a comparative analysis!!!

```
import time
import sys
start = time.time()
lc = [num for num in range(10000)]
sum(lc)
end = time.time()
print(end - start)
print(sys.getsizeof(lc))
```

Output:
0.0012502670288085938
85176

```
import time
import sys
start = time.time()
ge = (num for num in range(10000))
sum(ge)
end = time.time()
print(end - start)
print(sys.getsizeof(ge))
```

Output:
0.0050668716430664062
104

Optimizing mathematical operations – Avoiding `sqrt()`

The function `sqrt()` can often be avoided, especially in comparisons where comparing the value squared gives the same result.

In many equations, terms cancel out, either always or in some special cases. Work on them before writing the program.

Optimizing mathematical operations – Avoiding `pow()`

Avoid the use of `pow()` for computing small integer powers.

```
m = pow(2,4)
print(m)
```

Optimizing mathematical operations – Avoiding pow()

Avoid the use of pow() for computing small integer powers.

```
m = pow(2,4)
print(m)
```

It should be written as follows.

```
m = 2**4
print(m)
```

OR

```
m = 2*2*2*2
print(m)
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
for i in range(1,10000):
    m = pow(2,4)
end = time.time()
print(end - start)
```

```
import time
start = time.time()
for i in range(1,10000):
    m = 2**4
end = time.time()
print(end - start)
```

```
import time
start = time.time()
for i in range(1,10000):
    m = 2*2*2*2
end = time.time()
print(end - start)
```

Let's perform a comparative analysis!!!

```
import time
start = time.time()
for i in range(1,10000):
    m = pow(2,4)
end = time.time()
print(end - start)
```

Output:
0.006999969482421875

```
import time
start = time.time()
for i in range(1,10000):
    m = 2**4
end = time.time()
print(end - start)
```

Output:
0.0019998550415039062

```
import time
start = time.time()
for i in range(1,10000):
    m = 2*2*2*2
end = time.time()
print(end - start)
```

Output:
0.0009999275207519531

Optimizing mathematical operations – Avoiding division

Instead of repeatedly dividing by x, compute 1/x and multiply accordingly. It is really beneficial if you do more than 3 divides.

```
n = 10
x = 0.5
result = 1.0
for i in range(1,n):
    result /= x
```

This can be efficiently done in the following alternative way:

```
n = 10
x = 0.5
result = 1.0
x = 1 / x
for i in range(1,n):
    result *= x
```

Avoiding division

In standard processors, divisions are time-consuming because they take a constant time plus a time for each bit to divide.

```
if (a / b) > c:
    print('OK')
```

Avoiding division

In standard processors, divisions are time-consuming because they take a constant time plus a time for each bit to divide.

```
if (a / b) > c:
    print('OK')
```

It can be efficiently written as follows:

```
if a > (b * c):
    print('OK')
```

Here, the only assumptions are b is non-negative and $b * c$ fits into an integer. The latter one is also safe if $b = 0$.

Type casting

Avoid `type casting` wherever possible. Integer and floating point instructions often operate on different registers, so a casting requires a copy.

Example:

```
i = 2.7
print(int(i)) # Prints i = 2.0
print(i) # Prints i = 2.7
```

Type casting

Avoid `type casting` wherever possible. Integer and floating point instructions often operate on different registers, so a casting requires a copy.

Example:

```
i = 2.7
print(int(i)) # Prints i = 2.0
print(i) # Prints i = 2.7
```

Shorter integer types (char and short) still require the use of a full-sized register, and they need to be padded to 32/64-bits and then converted back to the smaller size before storing back in memory. However, this cost must be weighed against the additional memory cost of a larger data type.

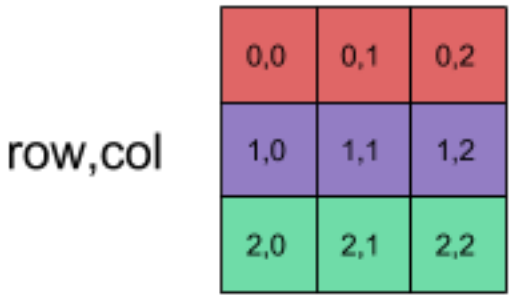
Memory organization in lists

Two and higher dimensional arrays are still stored in one dimensional memory. This means `ls[i][j]` and `ls[i][j+1]` are adjacent to each other, whereas `ls[i][j]` and `ls[i+1][j]` may be arbitrarily far apart.

When modern CPUs load data from main memory into processor cache, they fetch more than a single value. Instead they fetch a block of memory containing the requested data and adjacent data (a cache line). This means after `ls[i][j]` is in the CPU cache, `ls[i][j+1]` has a good chance of already being in cache, whereas `ls[i+1][j]` is still likely to be in the main memory.

Memory organization in arrays

Accessing data in a more-or-less sequential fashion, as stored in physical memory (in row major fashion), can dramatically speed up the code.



Function calls

Move loops inside function calls. Replace the following code

```
Function():
    ...
for i in range(1,n):
    Function()
```

with this code

```
Function():
    for i in range(1,n):
        ...
Function()
```

Note: Jumps/branches are expensive and hence should be avoided whenever possible. Note that, function calls require two jumps, in addition to stack memory manipulation.