University of Puerto Rico
Recinto de Mayagüez

Is Python Fast or Slow?

- Group A -
Daniel Mestres Piñero
Natanael Santiago Morales
Leonel Osoria Toledo
ICOM5015 - 001D
February 20, 2023

Introduction:

Python is an interpreted, object-oriented, high-level programming language [1]. Its simple and easy to learn syntax make it an excellent choice for beginner programmers and its extensive libraries make it suitable for practically any purpose, be it desktop applications, data science, database development and more. Python is widely considered to be slow compared to other languages such as C or C++, being preferable to use these languages for programs which require high amounts of calculations [2]. This is true if Python is utilized without any included or implemented libraries, which are the reasons why Python is so versatile and used today. Libraries such as Numpy, a math oriented library, are implemented in highly optimized C code [3]. This combines the ease of use of Python, with the speed and performance of a low-level language like C or C++. In this report we'll analyze the difference in performance when calculating the dot product of two matrices using both an iterative method and a Numpy implementation. Using the data obtained, we will answer the question: "Is Python Fast or Slow?".

A matrix is a set of numbers arranged in rows and columns to form a rectangular array [4]. If there are m rows and n columns, it is said to be an "m x n" matrix. The following figure shows an example of a 2 x 3 matrix:

$$\begin{bmatrix} 1 & 3 & 8 \\ 2 & -4 & 5 \end{bmatrix}$$

Figure 1: 2 x 3 Matrix [4]

The multiplication, or the dot product, of two matrices is only defined when the number of columns of the first matrix A equals the number of rows of the second matrix B. The resultant matrix C has as many rows as A and as many columns as B [4]. The element Cij is given by the following formula:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj}.$$

Figure 2: Cij Element Formula [4]

Procedure:

The experiment consists of implementing the dot product of two matrices utilizing both an iterative approach and assisted by the Numpy library. The design of the experiment will take into consideration the compatibility of the two matrices to be multiplied, by first checking if the number of columns of the first matrix equals the number of rows of the second matrix. Furthermore, the experiment is divided into two parts, finding the dot product of two 1-D

matrices, or vectors, and finding the dot product of two 2-D matrices. The Python Time library will be used to find the execution time with an extremely high precision. An additional constraint of only using integer elements was imposed. The purely iterative approach, utilizing only for loops to calculate each element in the resultant matrix can be found in the next figure below:

```
# [4x1][4x4]
r_a1 = len(M1x4)
c_a1 = len(M1x4[0])
r_a2 = len(M4x4)
c_a2 = len(M4x4[0])

# Check for compatibility
if(c_a1 != r_a2):
    print("Arrays not product compatible!")
else:
    # Declare product array with correct dimensions
    result = [[0 for x in range(c_a2)] for y in range(r_a1)]

    start_time = time.time()

    # Calculate product
    for i in range(r_a1):
        for j in range(c_a2):
            for k in range(c_a1):
                result[i][j] += M1x4[i][k] * M4x4[k][j]

    end_time = time.time()

    #print(result)
    print("Execution time[4x1][4x4]: ", end_time - start_time, " seconds")
```

Figure 3: Iterative Dot Product Implementation

The next figure shows code performing the same function but this time utilizing the Numpy library:

```python
import time
import numpy as np

# Define 2D arrays
M1x4 = np.array([1, 2, 3, 4])
M4x4 = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])

# Add exec time measurement for 1x4 @ 4x4
start_time = time.time()
C = np.dot(M1x4, M4x4)
end_time = time.time()
print("Execution time for multiplying M1x4 and M4x4: ", end_time - start_time, " seconds")
```

Figure 4: Numpy Dot Product Implementation

Both implementations were used to perform the dot product on a wide variety of both 1-D and 2-D matrices, recording the execution time taken by each in the tables and graphs below:

TABLE I
1-D Dot Product

Operation [rows x cols]	Iterative Time (us)	Numpy Time (us)
[1x1][1x1]	2.86	7.15
[1x2][2x1]	3.34	8.58
[1x4][4x1]	3.82	8.82
[1x8][8x1]	5.01	8.82
[1x16][16x1]	6.20	8.82
[1x32][32x1]	8.58	9.54
[1x64][64x1]	13.11	10.25
[1x128][128x1]	21.94	10.01

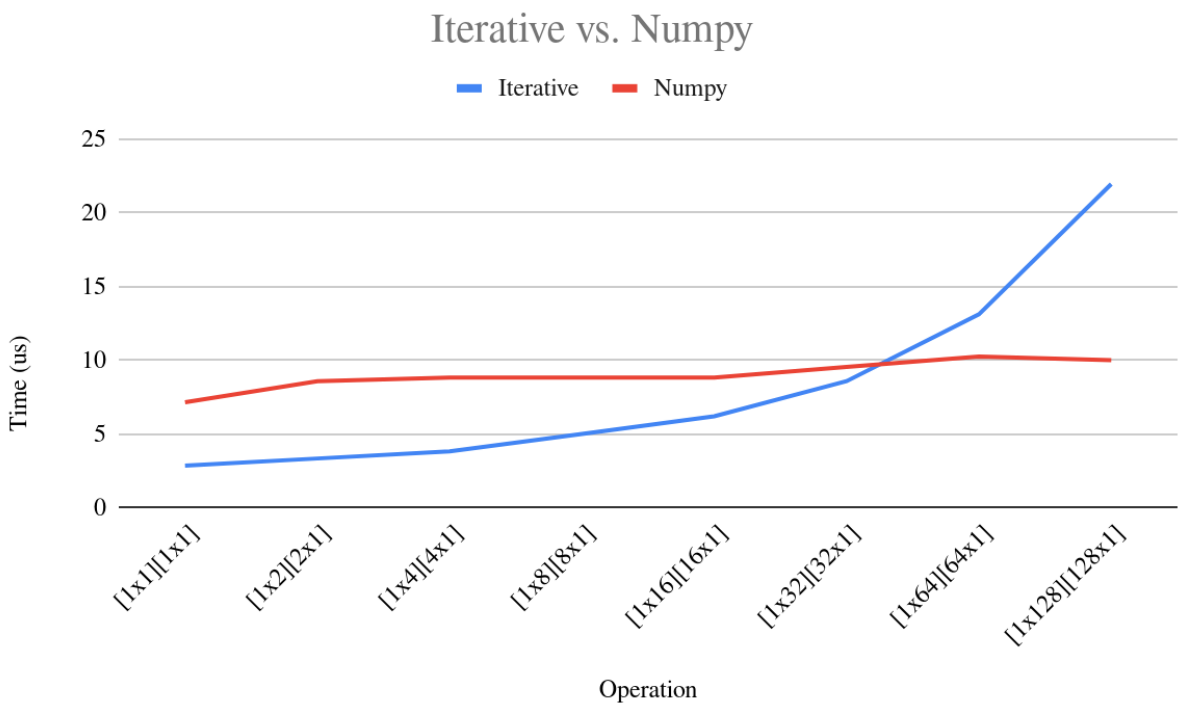

Figure 5: Iterative vs. Numpy 1-D Time

TABLE II
2-D Dot Product

Operation [rows x cols]	Iterative Time (us)	Numpy Time (us)
[3x4][4x4]	14.07	5.72
[3x3][3x5]	15.50	3.82
[7x9][9x9]	140.67	4.77
[3x196][196x3]	286.10	4.29
[20x5][5x12]	386.71	3.58
[10x12][12x18]	532.87	5.01
[10x128][128x3]	592.47	4.77
[128x3][3x196]	14,306.07	110.86

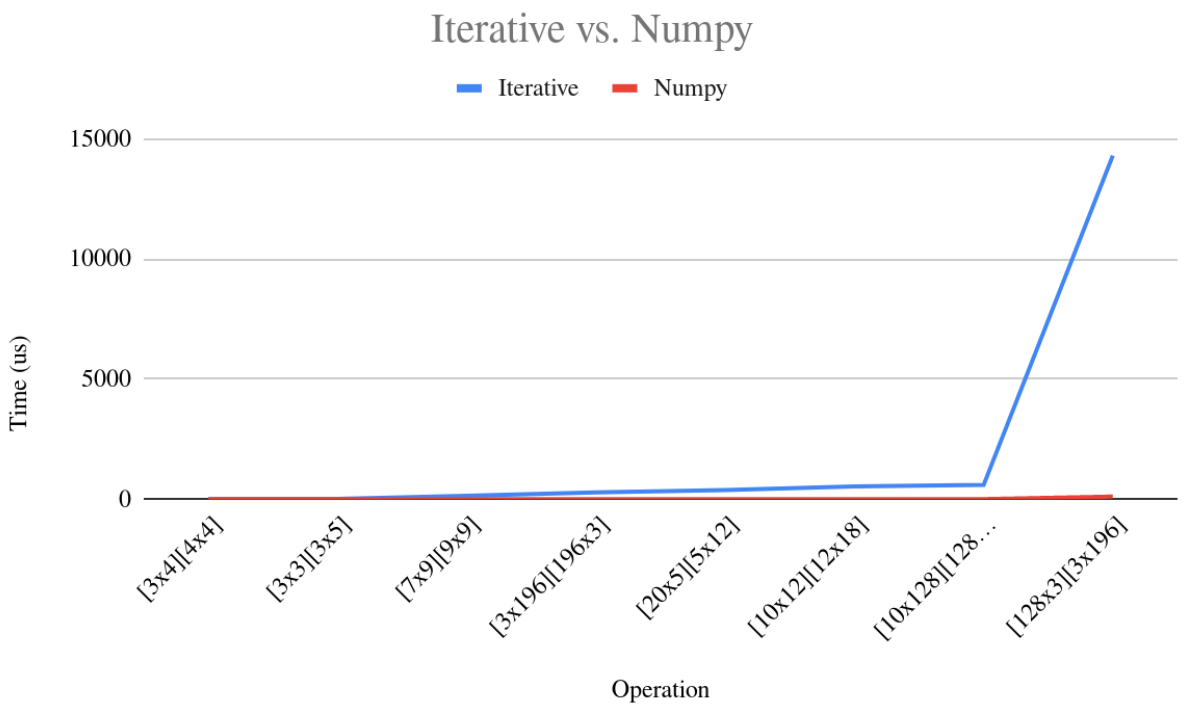

Figure 6: Iterative vs. Numpy 2-D Time

Analysis:

Analyzing the 1-D dot product cases and the graph in figure 5, it clearly shows that the iterative method does in fact outperform the Numpy method for arrays with dimensionalities in the 30's. After this point however, the iterative method shows a linear growth in time while the Numpy method shows a logarithmic growth in time. These results were expected as looking at our implementation of an iterative method, it uses 3 nested for loops. This results in an estimated time complexity of O(m * n * k), where m = rows of the first matrix, n = columns of the second matrix and k = columns of the first matrix. The linear aspect comes from the fact that m and n are fixed at a value of 1. The Numpy method on the other hand, calculates the inner product of two vectors without performing complex conjugation as both matrices are 1-D [5].

Moving on to the 2-D dot product cases, we can clearly observe in table II and figure 6 how the iterative method performs considerably worse than the Numpy method. For the iterative process the time of execution is observed to be longer in every single test case and showing a highly exponential growth as the dimensionality increases. These results were expected as the rows of the first matrix and columns of the second matrix are no longer fixed at a value of 1.

Conclusion:

In conclusion, we can confirm to be able to obtain fast execution times while having the easy readability of Python if libraries implemented in a highly optimized low-level language are used. While the iterative method can be suitable and more than enough for small data sets, it is evident that a dedicated and highly optimized library is a necessity for big data sets. Acknowledging this, we can answer that Python is in fact fast, but only when appropriate methods are utilized or the data sets are sufficiently small. This experiment taught us the importance of first researching methods and implementations for a given task, as the first idea that comes to mind may not necessarily be the most efficient or easily readable solution.

References:

[1] "What is python? executive summary," *Python.org*. https://www.python.org/doc/essays/blurb/. (accessed: Feb 18 2023).

[2] M. Żołądkiewicz, "Is python slow? use cases and comparison to other languages," We develop and design Web & mobile apps · Monterail, 07-Dec-2022. https://www.monterail.com/blog/is-python-slow. (accessed: Feb 18 2023).

[3] NumPy. https://numpy.org/. (accessed: Feb 18 2023).

[4] "Matrix," Encyclopædia Britannica, 05-Jan-2023. https://www.britannica.com/science/matrix-mathematics. (accessed: Feb 18 2023).

[5] "Numpy.dot" numpy.dot - NumPy v1.24 Manual. https://numpy.org/doc/stable/reference/generated/numpy.dot.html. (accessed: Feb 19 2023).