University of Puerto Rico
Mayagüez Campus
Mayagüez, Puerto Rico

Department of Electrical and Computer Engineering

Is Python Fast or Slow?

by:
Manuel Alejandro Umaña Rodriguez
Dahyna Gabrielle Martínez Pérez
Jan Luis Pérez de Jesús
Christopher Hans Mayens Matías

For: Profesor José Fernando Vega Riveros
Course: ICOM 5015, Section 001D
Date: February 21,2025

Abstract:

Why use external libraries in python instead of using python code by itself? To figure out what differences a specialized external library makes in regards to regular python code, a series of experiments were made to test the functionality of both approaches and discover if using an external library was worthwhile in python. The experiment consisted of testing the effectiveness of native python libraries, specifically NumPy in this experiment, against raw python code segments in the multiplication of two types of arrays. The first type being, two 1 dimensional arrays multiplied with each other with different sets of elements ranging from 10 to 500 elements each. The second type was two 2 dimensional arrays following the same procedure. After the experiment, it showed a significant performance decline in the native python code when the size of the arrays increased. It was concluded that using external libraries to supplement the python codes is more efficient then trying to make the same functionality with python code, especially when dealing with a large amount of data being used. This was due to the fact that the NumPy libraries that were created for computational speed were written in C or C++ and its arrays were stored in a continuous place in memory.

Table of Contents:

Group Collaboration:

The tasks amongst each team member were divided by each member's experience utilizing python, as follows:

- Christopher Mayens and Jan Pérez were in charge of the development of both 1D and 2D array multiplications and time performance programs within python.

-Manuel Umaña and Dahyna Martínez were in charge of the development of the report and the creation of the plots using the data from the Jupyter Notebook and the creation of the presentation for the video with the help of Christopher Mayens.

-All members of the group were included and participated in the video. Video editing was made by Jan Pérez.

Introduction:

The performance of a program is a crucial step in determining the efficiency and the usability for current and future use of said program. No matter how legible a code is, if it's not efficient it is not useful. For this reason it is necessary to test the efficiency of code by giving it certain parameters and benchmarks. The main way this comparison was addressed during this project is by using native code using python by multiplying 2 arrays of the same dimensions and doing the same procedure using the imported numpy library. The observed limitations created by this project include, but are not limited to, hardware limitations per machine and limit of time ranges at the execution of the code.

Performance Approach and Results:

For the purpose of testing the execution speed of python code some concessions were made when designing the necessary tests.While possible to create different concise size arrays with the same values, it would deny the underlying nature for the investigation and not be comparable to real world situations where the data gathered would not be uniformly distributed. As such the elements of all arrays are filled with random integer values in the range of 1 to 100 to simulate different data acquisition in real world problems. For the purpose of conducting the experiment, an iterative loop was set up which filled the necessary arrays with values based on the amount of elements/dimensions necessary for the experiment being performed as shown below.

```
# Creates 1D Arrays
def create_arrays1D(size):
   return [random.randint(1, 100) for _ in range(size)]
# Create 2D Arrays
def create_arrays2D(size):
   return [[random.randint(1, 100) for _ in range(size)] for _ in range(size)]
```

The size range was done with 5 different values those being 10, 50, 100, 200, and 500 elements for the 1 dimensional array (1D) and with the size of elements being 10x10 ,50x50, 100x100, 200x200, and 500x500 for the 2 dimensional array(2D). Each loop created a new array with the new amount of elements based from the smaller dimensions to largest. The different sizes were chosen to be able to perceive the differences made by the calculations and the measurable efficiency of python native language and numpy library, which is compiled using lower level language of c/c++[1] for faster execution times. For native calculations for 1D array a loop iterates across the different values and multiplies both elements and stores it in a different array with the multiplied products. The same procedure is used when dealing with the 2D array; the difference being it uses a double loop for the multiplication of the necessary elements which is shown below.

```
# Multiplies 1D Arrays
def multiply_arrays1D(array1, array2):
   mult_array = []
   for i in range(len(array1)):
      mult_array.append(array1[i] * array2[i])
   return mult_array
```

```
# 1D arrays
for size in size_array:
    array1 = create_arrays1D(size)
    array2 = create_arrays1D(size)

    # Pure Python (1D) Multiplication and Time
    start = time.perf_counter()
    python_result = multiply_arrays1D(array1,array2)
    end = time.perf_counter()
    python_1D_times.append(end - start)
    print(f"\nPure Python (1D): Execution Time With {size} Elements: {end - start:.10f}")

    # NumPy (1D) Multiplication
    # Converting arrays into np_arrays
    np_array1 = np.array(array1)
    np_array2 = np.array(array2)

    start = time.perf_counter()
    numpy_result = (np.multiply(np_array1, np_array2))
    end = time.perf_counter()
    numpy_1D_times.append(end - start)
    print(f"NumPy (1D): Execution Time With {size} Elements: {end - start:.10f}")

# Multiplies 2D Arrays
def multiply_arrays2D(array1, array2, size):
    return [[array1[i][j]* array2[i][j] for j in range(size)] for i in range(size)]
# 2D arrays
for size in size_array:
    array1 = create_arrays2D(size)
    array2 = create_arrays2D(size)

    # Pure Python Multiplication
    start = time.perf_counter()
    python_result = multiply_arrays2D(array1, array2, size)
    end = time.perf_counter()
    python_2D_times.append(end - start)
    print(f"\nPure Python (2D): Execution Time With {size} Elements: {end - start:.10f}")
```

```
# NumPy (2D) Multiplication
# Converting arrays into np_arrays
np_array1 = np.array(array1)
np_array2 = np.array(array2)

start = time.perf_counter()
numpy_result = np.multiply(np_array1, np_array2)
end = time.perf_counter()
numpy_2D_times.append(end - start)
print(f"NumPy (2D): Execution Time With {size} Elements: {end - start:.10f}")
```

For every iteration of the different sizes the results are stored in an external arrays to have the necessary gathered data when formulating analysis of the results. The 1D array comparisons are the first to run, with a separate loop to run the 2D array comparisons; both results are tabulated in Table 1 and Table 2. After both comparisons are finished the results were graphed for easier comparison and are shown in Figure 1 and Figure 3. The results give a more concise response with NumPy, being more consistent and quicker when performing large operations. However due to the nature of the auto adjustment of the program the execution times may appear differently due to the focus of the graph.

Using Figure 1 as a reference, both mathematical comparisons perform similarly when dealing with a small number of elements. The difference occurs when the amount of elements increases from 10 to 50 elements the native code increases in the amount of time taken for the completion of the task. This observation is validated when higher elements are calculated where the disparity of time is further exacerbated with the increase of elements in the 1D array. A similar problem occurs with the results of Figure 3 where the native python code performance worsens with the increase in element amount for both results. This disparity can be explained with the previously given information that numpy library was created using C/C++ and as such does not require the use of the python interpreter to be able to make calculations. In addition there are several factors that make the execution time of numpy be better than the native code of python. One of the factors that influenced numpy efficiency is that the values that are stored with NumPy are stored in a continues place in memory unlike python list [2] adding to the fact that numpy arrays, or n-arrays, are homogeneous data type arrays that are broken down into smaller tasks to be done parallelly [3] for more efficient use of resources. “The way this is done is by using multithreading context via the threading module in the standard library" [5]. On the other hand the reason that the python native code takes considerably longer is because of the nature of python coding language as it is not compiled as other lower level languages, instead it is interpreted at runtime instead of being compiled to native computer code at compile time [4] as such "Python code we write must go through many, many stages of abstraction before it can become executable machine code"[4]. With that being said it should be noted that for smaller data arrays python could still optimally perform at par with NumPy meaning that for smaller calculations pure python is still efficient as is. The simplicity of writing the python, if the NumPy library were not available, its readability would still prove useful for calculations as it would be written to be precise even with the detriment of speed due to its dynamic nature.

Figures and Tables:

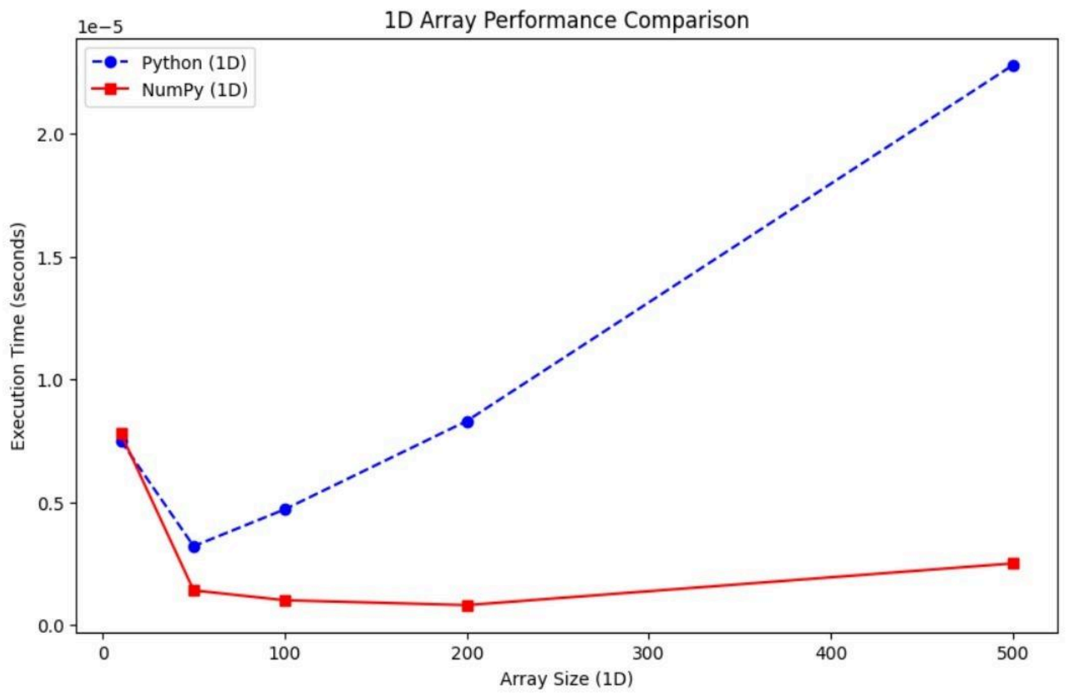

Figure 1. Comparison of 1D Array Multiplication Performances utilizing Pure Python and NumPy

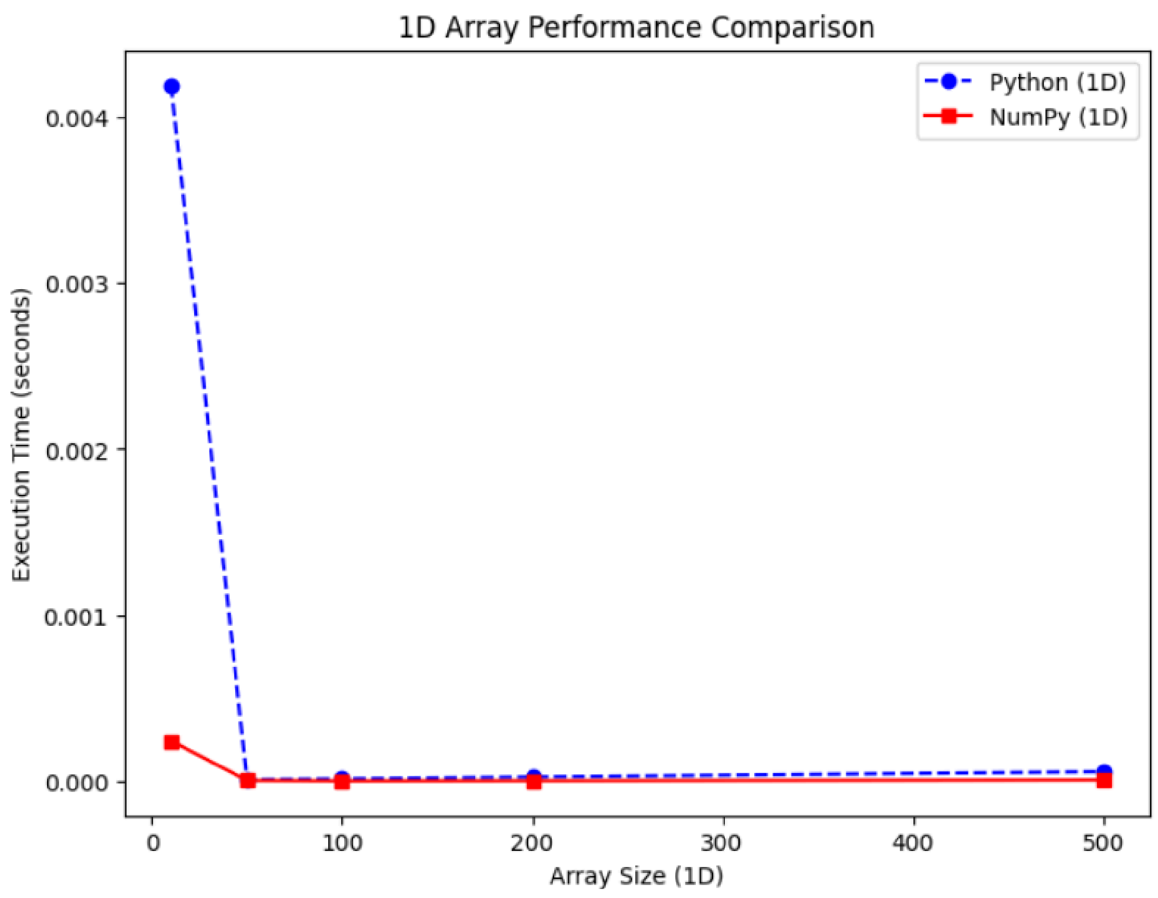

Figure 2. Comparison of 1D Array Multiplication Performances utilizing Pure Python and NumPy normalised

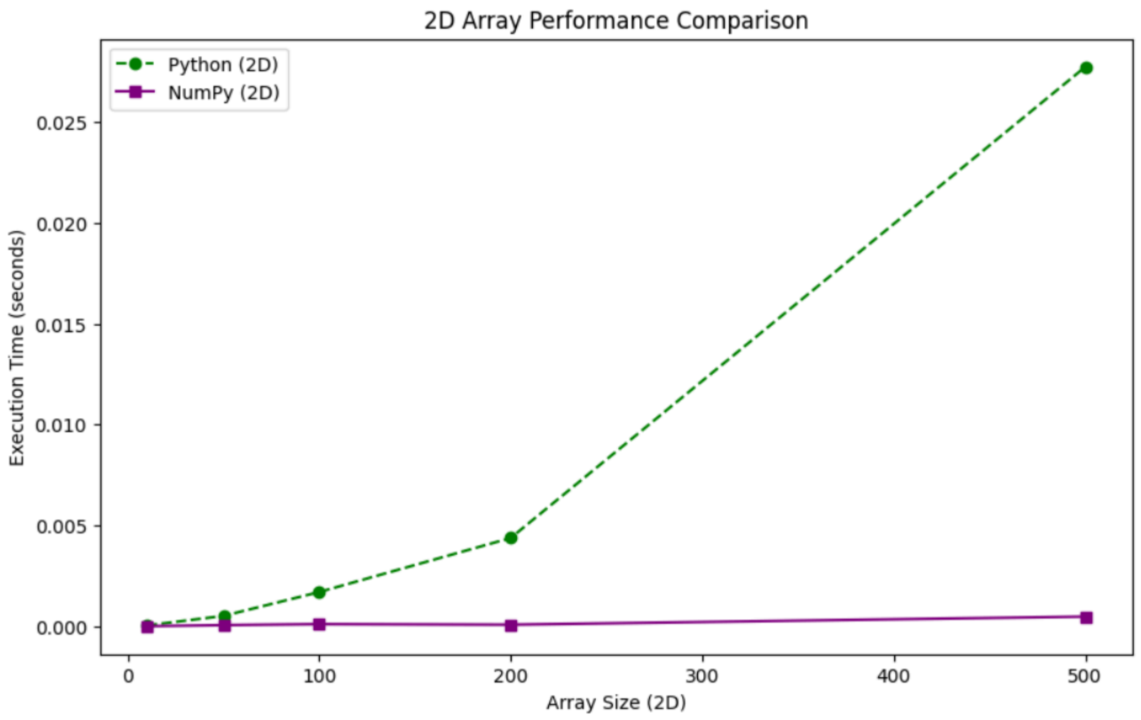

Figure 3. Comparison of 2D Array Multiplication Performances utilizing Pure Python and NumPy

Dimensions	Pure Python Execution Time (seconds)	NumPy Execution Time (seconds)
10	0.0075656000	0.0003246000
50	0.0000152000	0.0000092000
100	0.0000220000	0.0000052000
200	0.0000382000	0.0000140000
500	0.0001164000	0.0000086000

Table 1. Time Execution data of 1D Array Multiplications for Graph 1 for 1 attempt.

Dimensions	Pure Python Execution Time (seconds)	NumPy Execution Time (seconds)
10	0.0000507000	0.0000133000

50	0.0005219000	0.0000655000
100	0.0017039000	0.0001151000
200	0.0043834000	0.0000872000
500	0.0277210000	0.0004880000

Table 2. Time Execution data of 2D Array Multiplications for Graph 3 for 1 attempt.

Conclusion:

After looking at the results obtained from these experiments it becomes clear that NumPy excels handling big mathematical computations. With that as the basis it can be reasonably inferred that while Python is a great programming language when it comes to writing more simple code with great adaptability, it suffers from efficiency when handling big amounts of data and especially when handling a great number of calculations. The reason for this is that the interpreter for python gets interpreted at runtime as stated above. As such when making a great number of calculations each one of said calculations must be interpreted by changing into bycode and then interpreted by the python virtual machine for each iteration of the calculations. Making the use of external libraries is highly recommended as they are specialized for certain tasks. As python excels at adaptability on different tasks it can also perform subpar at specific high intensity tasks. This can be improved by the different specialized libraries that are constantly maintained and updated.

References:

[1] NumPy documentation, “What id NumPy,” *numpy.org*, Dec. 8, 2024. [Online]. Available: https://numpy.org/doc/2.2/user/whatisnumpy.html . [Accessed Feb.20, 2025].

[2] W3schools, “NumPy Introduction,” *w3schools.com*, 2022. [Online]. Available: https://www.w3schools.com/python/numpy/numpy_intro.asp. [Accessed Feb.20, 2025].

[3] R. Jsaha, “Why is Numpy faster in Python,” *geeksforgeeks.org*, Dec. 8, 2024. [Online]. Available: https://www.geeksforgeeks.org/why-numpy-is-faster-in-python/. [Accessed Feb.20, 2025].

[4] I. Sushant, “What makes python a slow language,” *geeksforgeeks.org*, July. 2021. [Online]. Available: https://www.geeksforgeeks.org/what-makes-python-a-slow-language/ . [Accessed Feb.20, 2025].

[5] NumPy reference, “Thread Safety,” *numpy.org*, Dec. 8, 2024. [Online]. Available: https://numpy.org/doc/2.1/reference/thread_safety.html . [Accessed Feb.20, 2025].