

The matrix in Jupyter with NumPy Library

Didem B. Aykurt

Colorado State University Global

MIS542; Business Analytics

Dr. Emmanuel Tsukerman

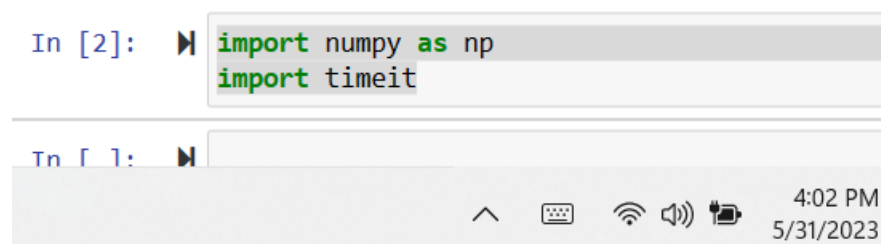
April 4, 2023

NumPy Library in Jupyter

In this project, explain the NumPy library with two different table matrices. First, a 100 x 100 table matrix is nested by three methods for loop, NumPy fromfunction, and NumPy broadcasting. Second, a 12 x 12 table matrix with the first 12 numbers to save the array file, the square root for using a unary ufunc, and mean, standard deviation example of the statistic method, and load function the saved matrix.

The NumPy library comprises matrix data and a multidimensional array of structures. The library available ndarray and an identical n-dimensional array object with methods to handle it competently. And it can perform high-level mathematical operations and systematical calculations with arrays and matrices. Let's import the NumPy and timeit libraries, create three methods, and measure their implementation time, then slice the original matrix to a 12 x 12 table matrix and save the array to a file.

Figure 1: Import the libraries.



```
In [2]: import numpy as np
import timeit
```

The screenshot shows a Jupyter Notebook interface. The top part displays a code cell with the input 'In [2]:' followed by two lines of Python code: 'import numpy as np' and 'import timeit'. The bottom part shows the start of another code cell with 'In []:' followed by a cursor. At the bottom of the notebook window, there is a status bar with icons for undo, redo, keyboard shortcuts, and a system tray showing the time '4:02 PM' and date '5/31/2023'.

1. Nested for Loop

This method uses two nested for loops to iterate over the rows and columns of the matrix and calculate the multiplication for each cell. The result is stored in a numpy array. The execution time is measured using the timeit function.

Figure 2: Create a 100 x 100 table matrix with a nested for loop.

```

def nested_for_loops(): # def is the keyword for defining a function
    matrix = np.zeros((100, 100), dtype=int) #matrix will be filled with zeros
    for i in range(100): #Outer loop range() with for loop function
        for j in range(100): #Inner loop range() with for loop function
            matrix[i, j] = (i + 1) * (j + 1)
    return matrix
nested_for_loops()

```

```

5]: array([[ 1,  2,  3, ..., 98, 99, 100],
          [ 2,  4,  6, ..., 196, 198, 200],
          [ 3,  6,  9, ..., 294, 297, 300],
          ...,
          [ 98, 196, 294, ..., 9604, 9702, 9800],
          [ 99, 198, 297, ..., 9702, 9801, 9900],
          [100, 200, 300, ..., 9800, 9900, 10000]])

```

Figure 3: Measure the nested for loop 's execution time.

```

In [28]: def nested_for_loops():
        matrix = np.zeros((100, 100), dtype=int)
        for i in range(100):
            for j in range(100):
                matrix[i, j] = (i + 1) * (j + 1)
        return matrix

        nested_time = timeit.timeit(nested_for_loops, number=1)
        print("Nested for loops execution time:", nested_time)

```

```

Nested for loops execution time: 0.003531700000166893

```

2. Numpy fromfunction

This method uses the `np.fromfunction()` function to create a matrix by applying a given function to each coordinate. In this case, the lambda function multiplies the row index by the column index plus 1. The execution time is measured using the `timeit` function.

Figure 4: Create a 100 x 100 table matrix with a NumPy fromfunction() function.

```
In [44]: def numpy_fromfunction():
        matrix = np.fromfunction(lambda i, j: (i + 1) * (j + 1), (100, 100), dtype=int)
        return matrix
        numpy_fromfunction()

Out[44]: array([[ 1,  2,  3, ..., 98, 99, 100],
                [ 2,  4,  6, ..., 196, 198, 200],
                [ 3,  6,  9, ..., 294, 297, 300],
                ...,
                [ 98, 196, 294, ..., 9604, 9702, 9800],
                [ 99, 198, 297, ..., 9702, 9801, 9900],
                [100, 200, 300, ..., 9800, 9900, 10000]])
```

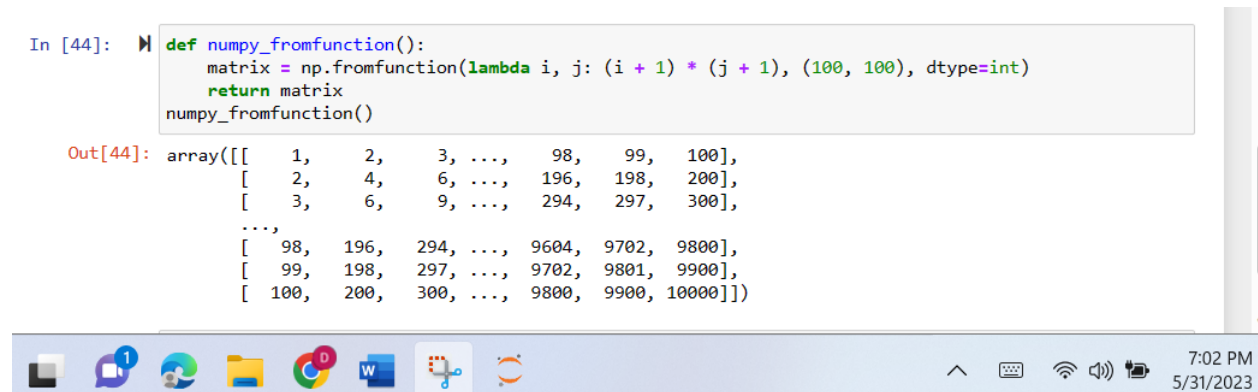
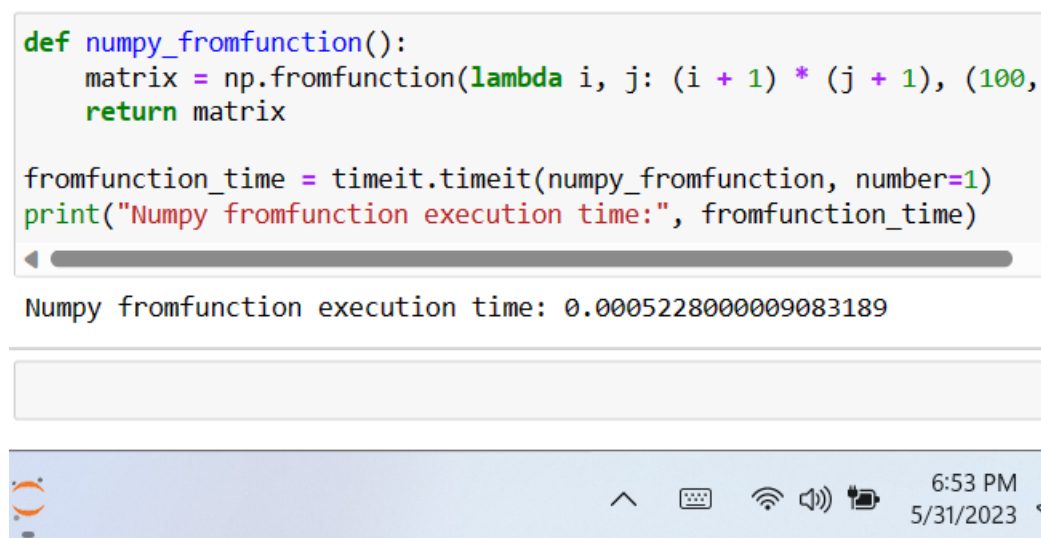


Figure 5: Measure the NumPy fromfunction 's execution time.

```
def numpy_fromfunction():
    matrix = np.fromfunction(lambda i, j: (i + 1) * (j + 1), (100,
    return matrix

fromfunction_time = timeit.timeit(numpy_fromfunction, number=1)
print("Numpy fromfunction execution time:", fromfunction_time)
```

Numpy fromfunction execution time: 0.0005228000009083189



3. Numpy broadcasting

The method uses numpy broadcasting to create the matrix. It creates two 1D arrays representing the rows and columns using np.arange(), then uses the multiplication operator * between the row and column arrays. Numpy's broadcasting rules automatically expand

the dimensions of the arrays to perform element-wise multiplication. We measure the execution time.

Figure 6: Create a 100 x 100 table matrix with a numpy broadcasting function.

```
def numpy_broadcasting():
    rows = np.arange(1, 101).reshape(100, 1)
    cols = np.arange(1, 101).reshape(1, 100)
    matrix = rows * cols
    return matrix
numpy_broadcasting()

array([[ 1,  2,  3, ..., 98, 99, 100],
       [ 2,  4,  6, ..., 196, 198, 200],
       [ 3,  6,  9, ..., 294, 297, 300],
       ...,
       [ 98, 196, 294, ..., 9604, 9702, 9800],
       [ 99, 198, 297, ..., 9702, 9801, 9900],
       [100, 200, 300, ..., 9800, 9900, 10000]])
```

9:35 AM
6/1/2023

Figure 7: Measure the NumPy broadcasting 's execution time.

```
broadcasting_time = timeit.timeit(numpy_broadcasting, number=1)
print("Numpy broadcasting execution time:", broadcasting_time)

Numpy broadcasting execution time: 0.0001292000524699688
```

9:50 AM
6/1/2023

To create a 12x12 times table matrix, a slice from the original matrix uses the `[:12, :12]` indexing, which selects the first 12 rows and the first 12 columns.

Figure 8: Create a 12x12 times table matrix using a slice of the original matrix:

```
# methods to generate the original matrix to new matrix
original_matrix = nested_for_loops()
new_matrix = original_matrix[:12, :12]
print(new_matrix)
```

```
[[ 1  2  3  4  5  6  7  8  9 10 11 12]
 [ 2  4  6  8 10 12 14 16 18 20 22 24]
 [ 3  6  9 12 15 18 21 24 27 30 33 36]
 [ 4  8 12 16 20 24 28 32 36 40 44 48]
 [ 5 10 15 20 25 30 35 40 45 50 55 60]
 [ 6 12 18 24 30 36 42 48 54 60 66 72]
 [ 7 14 21 28 35 42 49 56 63 70 77 84]
 [ 8 16 24 32 40 48 56 64 72 80 88 96]
 [ 9 18 27 36 45 54 63 72 81 90 99 108]
 [10 20 30 40 50 60 70 80 90 100 110 120]
 [11 22 33 44 55 66 77 88 99 110 121 132]
 [12 24 36 48 60 72 84 96 108 120 132 144]]
```

11:38 AM
6/1/2023

For saving the array to a file, `np.save()` function is used with the filename "times_table.npy".

This will save the matrix in numpy's binary format.

Figure 9: Save the array to a file.

```
np.save("times_table.npy", new_matrix)
```

11:54 AM
6/1/2023

Figure 10: The times_table.npy in the document list.

vIDEOS		a year ago	
<input type="checkbox"/>	MIS 542.ipynb	Running seconds ago	4.87 kB
<input type="checkbox"/>	times_table.npy	2 minutes ago	704 B

One of the unary ufunc is `np.sqrt()` function applied to the new matrix, which calculates the square root of each element in the matrix.

Figure 11: Use one of the unary ufuncs as the square root and apply the new matrix.

```
square_root = np.sqrt(new_matrix)
print(square_root)
```

```
[[ 1. 1.41421356 1.73205081 2. 2.23606798 2.44948974
 2.64575131 2.82842712 3. 3.16227766 3.31662479 3.46410162]
 [ 1.41421356 2. 2.44948974 2.82842712 3.16227766 3.46410162
 3.74165739 4. 4.24264069 4.47213595 4.69041576 4.89897949]
 [ 1.73205081 2.44948974 3. 3.46410162 3.87298335 4.24264069
 4.58257569 4.89897949 5.19615242 5.47722558 5.74456265 6. ]
 [ 2. 2.82842712 3.46410162 4. 4.47213595 4.89897949
 5.29150262 5.65685425 6. 6.32455532 6.63324958 6.92820323]
 [ 2.23606798 3.16227766 3.87298335 4.47213595 5.
 5.47722558 5.91607978 6.32455532 6.70820393 7.07106781 7.41619849 7.74596669]
 [ 2.44948974 3.46410162 4.24264069 4.89897949 5.47722558 6.
 6.4807407 6.92820323 7.34846923 7.74596669 8.1240384 8.48528137]
 [ 2.64575131 3.74165739 4.58257569 5.29150262 5.91607978 6.4807407
 7. 7.48331477 7.93725393 8.36660027 8.77496439 9.16515139]
 [ 2.82842712 4. 4.89897949 5.65685425 6.32455532 6.92820323
 7.48331477 8. 8.48528137 8.94427191 9.38083152 9.79795897]
 [ 3. 4.24264069 5.19615242 6. 6.70820393 7.34846923
 7.93725393 8.48528137 9. 9.48683298 9.94987437 10.39230485]
 [ 3.16227766 4.47213595 5.47722558 6.32455532 7.07106781 7.74596669
 8.36660027 8.94427191 9.48683298 10. 10.48808848 10.95445115]
 [ 3.31662479 4.69041576 5.74456265 6.63324958 7.41619849 8.1240384
 8.77496439 9.38083152 9.94987437 10.48808848 11. 11.48912529]
 [ 3.46410162 4.89897949 6. 6.92820323 7.74596669 8.48528137
 9.16515139 9.79795897 10.39230485 10.95445115 11.48912529 12. ]]]
```

12:01 PM
6/1/2023

The primary array statistical methods, `np.mean()` is used to calculate the mean of all elements in the matrix, and `np.std()` is used to calculate the standard deviation.

Figure 12: Use two of the basic array statistical methods.

```
mean = np.mean(new_matrix)
standard_deviation = np.std(new_matrix)
print("New matrix's mean is: ", mean)
print("New matrix's standard deviation is: ", standard_deviation)
```

```
New matrix's mean is: 42.25
New matrix's standard deviation is: 33.8963903355177
```

12:09 PM
6/1/2023

Load the two steps are unary ufunc and basic array statistical methods into “times_table.npy”, np.load() function is used with the filename "times_table.npy". This loads the saved matrix back into a numpy array.

Figure 13: Loading the array file “times_table.npy”.

```
loaded_matrix = np.load("times_table.npy")
```

A screenshot of a code editor window. The top part shows a single line of Python code: `loaded_matrix = np.load("times_table.npy")`. The code is color-coded: `loaded_matrix` is in blue, `=` is in purple, `np.load` is in red, and the string `"times_table.npy"` is in red. Below the code editor is a toolbar with several icons: an upward arrow, a circular arrow, a refresh icon, a keyboard icon, a Wi-Fi icon, and a speaker icon. On the right side of the toolbar, the time and date are displayed: "12:17 PM" and "6/1/2023".

References

Nelson, G. S. (2018). The analytics lifecycle toolkit: A practical guide for an effective analytics capability. John Wiley & Sons. ISBN-13: 9781119425069.

NumPy.com,(n.d.). Routines. <https://numpy.org/doc/stable/reference/routines.html>