

Basics of Python II

by Subhrasankar Chatterjee 

NUMPY

If you're using python for data science, either you have used NumPy or must have heard about it. Most of the statistical analysis which needs data to be stored in memory uses Numpy.

INSTALLATION

```
In [1]: !pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\subhrasankar\appdata\local\programs\python\python37\lib\site-packages (1.20.1)
```

```
[notice] A new release of pip available: 22.2 -> 22.2.2  
[notice] To update, run: python.exe -m pip install --upgrade pip
```

IMPORTING PACKAGE

```
In [3]: import numpy as np
```

BUT WHY NUMPY?

NumPy provides efficient storage and better ways of handling data for Mathematical Operations

```
In [ ]: ## NumPy is meant for creating homogeneous n-dimensional arrays (n = 1..n).  
## Unlike Python lists, all elements of a NumPy array should be of same type.  
  
py_arr = [1,2,"Hello",3,"World"] # Valid Code  
  
# numpy_arr = np.array([1,2,"Hello",3,"World"], dtype=np.int32) # Error
```

NumPy uses much less memory to store data

- The NumPy arrays takes significantly less amount of memory as compared to python lists.
- It also provides a mechanism of specifying the data types of the contents, which allows further optimisation of the code.

```
In [7]: import sys  
  
py_arr = [1,2,3,4,5,6]  
numpy_arr = np.array([1,2,3,4,5,6])  
  
sizeof_py_arr = sys.getsizeof(1) * len(py_arr)           # Size = 168  
sizeof_numpy_arr = numpy_arr.itemsize * numpy_arr.size  # Size = 48  
  
print(sizeof_py_arr)  
print(sizeof_numpy_arr)
```

```
168  
24
```

Optimizing Further

```
In [19]: # For NumPy arrays elements limited to 1 Byte / 8 Bits
numpy_arr = np.array([1,2,3,4,5,6], dtype = np.int8)
sizeof_numpy_arr = numpy_arr.itemsize * numpy_arr.size # Size = 6
print(sizeof_numpy_arr)

# For NumPy arrays elements limited to 2 Bytes / 16 Bits

numpy_arr = np.array([1,2,3,4,5,6], dtype = np.int16)
sizeof_numpy_arr = numpy_arr.itemsize * numpy_arr.size # Size = 12
print(sizeof_numpy_arr)
```

```
6
12
```

Using NumPy for creating n-dimension arrays

- An n-dimension array is generally used for creating a matrix or tensors, again mainly for the mathematical calculation purpose.
- Compare to python list base n-dimension arrays, NumPy not only saves the memory usage, it provide a significant number of additional benefits which makes it easy to mathematical calculations

Creating a numpy array

```
In [11]: ls = [10,12,14,16,20,22]
ls_array = np.array(ls)
print(ls_array)

array_0 = np.zeros((3,3))
print(array_0)
```

```
[10 12 14 16 20 22]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

How do we know the size of the array?

```
In [13]: np.shape(ls_array)
```

```
Out[13]: (6,)
```

The dimensions of the array can be changed at runtime as long as the multiplicity factor produces the same number of elements.

For example, a $2 * 5$ matrix can be converted into $5 * 2$ and a $1 * 4$ into $2 * 2$.

```
In [15]: ls_array = ls_array.reshape((3,2))
print(ls_array)

ls_array = ls_array.reshape((2,3))
print(ls_array)
```

```
[[10 12]
 [14 16]
 [20 22]]
[[10 12 14]
 [16 20 22]]
```

NumPy can also generate a predefined set of number for an array.

The output of this function will always be a single dimension set of numbers. However, we can use reshape on this output to generate dimension of our choice.

```
In [17]: np_nd_arr = np.arange(0,100)
print(np_nd_arr)
np_nd_arr = np_nd_arr.reshape((10,10))
print(np_nd_arr)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```

As **reshape(x,y)** can convert an array into multi dimensional array, similarly, its possible to create a single dimensional array from any N-D array

```
In [18]: f_arr = np_nd_arr.ravel()
print(f_arr)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
```

Slicing an array

```
In [33]: np_nd_arr = np.arange(0,100)
np_nd_arr = np_nd_arr.reshape((10,10))
print(np_nd_arr.shape)
print(np_nd_arr)

np_nd_arr[3:,3:]
```

```
(10, 10)
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```

```
Out[33]: array([[33, 34, 35, 36, 37, 38, 39],
               [43, 44, 45, 46, 47, 48, 49],
               [53, 54, 55, 56, 57, 58, 59],
               [63, 64, 65, 66, 67, 68, 69],
               [73, 74, 75, 76, 77, 78, 79],
               [83, 84, 85, 86, 87, 88, 89],
               [93, 94, 95, 96, 97, 98, 99]])
```

Mathematical operations on NumPy n-Dimension Arrays

- NumPy is not only about efficient storing the data, it also makes it extremely easy to perform mathematical operations on it.
- Any actions on n-dimension arrays behaves exactly similar to mathematical operations.

NumPy n-dimensional arrays makes it extremely easy to perform mathematical operations on it

Multiplication with a scalar

```
In [20]: py_arr = [1,2,3] * 2
print(py_arr)

np_arr = np.array([1,2,3]) * 2
print(np_arr)
```

```
[1, 2, 3, 1, 2, 3]
[2 4 6]
```

Elementwise multiplication/addition of a matrix

```
In [21]: np_arr1 = np.array([[1,2,3],[4,5,6]])
np_arr2 = np.array([[1,2,3],[4,5,6]])
np_arr3 = np_arr1 * np_arr2

print(np_arr3)

np_arr3 = np_arr1 + np_arr2

print(np_arr3)
```

```
[[ 1  4  9]
 [16 25 36]]
[[ 2  4  6]
 [ 8 10 12]]
```

Mathematical Functions

```
In [22]: np_arr1 = np.array([[1,2,3],[4,5,6]])
print(np.sqrt(np_arr1))
```

```
[[1.          1.41421356 1.73205081]
 [2.          2.23606798 2.44948974]]
```

```
In [27]: print(np_arr1.sum()) # 21
print(np_arr1.min()) # 1
print(np_arr1.max()) # 6
```

```
21
1
6
```

The concept of axis

```
In [23]: print(np.mean(np_arr1))
```

```
3.5
```

```
In [24]: print(np.mean(np_arr1,axis=0))
```

```
[2.5 3.5 4.5]
```

```
In [25]: print(np.mean(np_arr1,axis=1))
```

```
[2. 5.]
```

Finding Elements in NumPy array

```
In [28]: np_arr = np.array([1,2,0,4,5])
find = np.where(np_arr > 2)
print(find)
```

```
(array([3, 4], dtype=int64),)
```

```
In [29]: np_arr = np.array([1,2,0,4,5])
find = np.nonzero(np_arr)
print(find)
```

```
(array([0, 1, 3, 4], dtype=int64),)
```

```
In [30]: n_arr = np.array([1,2,3,0,3,0,2,0,0,2])
np.count_nonzero(n_arr)
```

```
Out[30]: 6
```

Vectorization

Time it up!!!

```
In [36]: import time

tic = time.time()

arr = np.arange(100)

toc = time.time()

print("Time Elapsed: ", toc - tic)
```

```
Time Elapsed: 0.0
```

A Good researcher always **cross validates** his findings

```
In [35]: import timeit

setup = '''
import numpy as np
'''

snippet = 'arr = np.arange(100)'

num_runs = 10000

time_elapsed = timeit.timeit(setup = setup, stmt = snippet, number = num_runs)

print("Time Elapsed: ", time_elapsed / num_runs)
```

Time Elapsed: 5.911199999900418e-07

Alternate 1

```
In [37]: import timeit

setup = '''
import numpy as np
'''

def fn():
    return np.arange(100)

num_runs = 10000

time_elapsed = timeit.timeit(setup = setup, stmt = fn, number = num_runs)

print("Time Elapsed: ", time_elapsed / num_runs)
```

Time Elapsed: 6.881799999973737e-07

Alternate 2

```
In [38]: %timeit arr = np.arange(100)
```

551 ns \pm 17 ns per loop (mean \pm std. dev. of 7 runs, 1000000 loops each)

Compare Timings

```
In [49]: import numpy as np
import random

# Create 2 Lists of 1000 random float numbers between -1 to 1
a = [random.uniform(-1.0, 1.0) for _ in range(100000)]
b = [random.uniform(-1.0, 1.0) for _ in range(100000)]

# 3rd List to store the results
c = [0.0]*100000

# Traditional C-style Loopy-addition
def sum_using_loop(a,b,c):
    for i in range(len(a)):
        c[i] = a[i] + b[i]
    return c

# Element-wise-Addition in Numpy Land..
def sum_using_numpy_builtin(a,b):
    c = np.add(a,b) # or simply use 'c = a + b'
    return c
```

```
In [56]: %timeit sum_using_loop(a,b,c)
# Average run time: 10.5 ms ± 15.9 μs
```

```
a = np.array(a)
b = np.array(b)
c = np.array(c)
```

```
%timeit sum_using_numpy_builtin(a, b)
# Average run time: 79.6 μs ± 835 ns
```

23.9 ms ± 1.44 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
41.4 μs ± 2.59 μs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Loops in Python

Python first goes line-by-line through the code, compiles the code into bytecode, which is then executed to run the program. Let's say the code contains a section where we loop over a list. Python is dynamically typed, which means it has no idea what type of objects are present in the list (whether it's an integer, a string or a float). In fact, this information is basically stored in every object itself, and Python can not know this in advance before actually going through the list. Therefore, at each iteration python has to perform a bunch of checks every iteration like determining the type of variable, resolving it's scope, checking for any invalid operations etc.

Contrast this with C, where arrays are allowed to be consisting of only one data type, which the compiler knows well ahead of time. This opens up possibility of many optimizations which are not possible in Python. For this reason, we see loops in python are often much slower than in C, and nested loops is where things can really get slow

OK! So loops can slow your code. So what to do now?

What if we can restrict our lists to have only one data type that we can let Python know in advance?

Can we then skip some of the per-iteration type checking Python does to speed up our code.

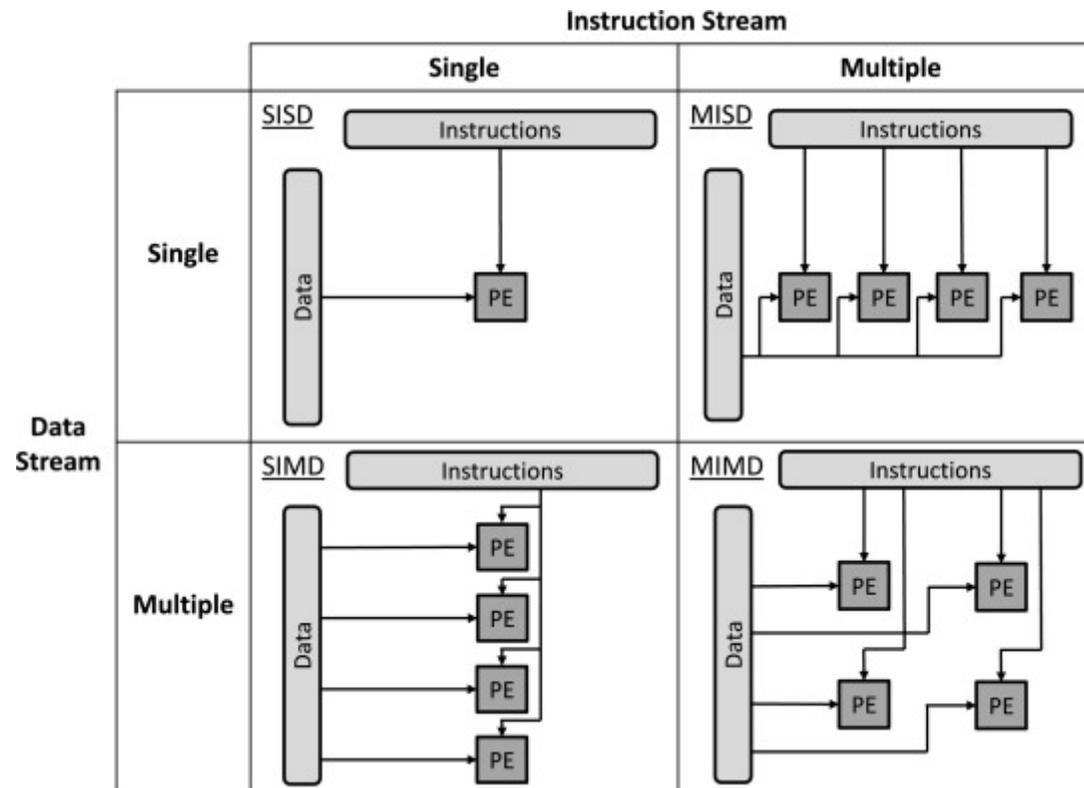
How is Numpy managing such a ridiculous speedup??

- Some of the obvious answers include:

- Numpy is primarily written in C, which is faster than Python. Numpy arrays are homogeneous (all array elements have fixed data-type- np.float32, np.uint8, etc. compared to python lists that have no such restriction), thus allowing numbers to be stored in contiguous memory locations for faster access (exploiting locality of reference)

BUT IS THAT ALL THERE IS TO IT????

Spoiler Alert: Numpy uses Vector Instructions (SIMD) for speeding up 'ufunc's , you can skip to the next section, or read on if you're interested in finding out how.



Modern CPUs, have dedicated hardware registers called Vector-Registers, that are capable of operating on multiple operands (of the same type/size) simultaneously, introducing a parallelism referred to as 'Single Instruction Multiple Data' (or SIMD) mechanism

The wider capacity implies, a **256 bit vector register** can hold & operate on **'8' 32-bit floats** in parallel within a single clock cycle! — **Nearly 4x speed-up**, compared to the biggest general purpose register

How does this newly acquired knowledge make me a smarter Data-Scientist?

So now that you know the insider secret, it's in your best interest to :

- Avoid using for-loops as much as possible and utilise numpy-builtin operations & methods.
- A quick way to vectorize your code is using **ufunc**.
- For complex operations, the trick is to try and convert the loop into a sequence of matrix operations (dot-products/matrix-mul/add, etc.).

UFUNC in Numpy

```
In [51]: def myadd(x, y):  
         return x+y  
  
myadd = np.frompyfunc(myadd, 2, 1)  
  
print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
```

```
[6 8 10 12]
```

```
In [54]: print(a.shape)  
         print(b.shape)  
  
(100000,)  
[-0.55284135  0.79334463 -0.60956541 ... -0.76483981 -0.2603647  
 0.07815007]
```

```
In [53]: print(myadd(a,b))  
  
[-0.16703697964252373 -0.08339548681112108 -0.2679210471824449 ...  
-0.202015904438696 -0.2311442743211538 -0.4964110327007738]
```

```
In [57]: %timeit myadd(a,b)
```

```
10.1 ms ± 139 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```