Practical 10: Numerical Computing with Python and Numpy

Aim: To perform Various numerical computing with python and Numpy

Objectives: The objective of this practical is to work with numerical data in Python and going from Python lists to Numpy arrays and learn about Multi-dimensional Numpy arrays and their benefits.

Problem Statement

1] To compare dot products performance using Python loops vs. Numpy arrays on two vectors with a million elements each.
2] Perform Various operation with a Multi-dimensional Numpy arrays

Attach code and screen shots of output

[1]:
```
import numpy as np
```

[2]:
```
#python lists
arr1 = list(range(100000))
arr2 = list(range(100000, 200000))

#Numsy arrays
arr1_np = np.array(arr1)
arr2_np = np.array(arr2)
```

[3]:
```
%%time
result = 0
for x1, x2 in zip(arr1, arr2):
    result += x1*x2
result
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 53 ms
```

[3]: 833323333350000

[4]:
```
%%time
np.dot(arr1_np, arr2_np)
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 0 ns
```

[4]: 893678192

[5]:
```
marks = np.array([[77, 67, 43],
                  [91, 88, 64],
                  [87, 79, 58],
                  [95, 43, 87],
                  [69, 96, 70]])
```

[6]:
```
print(marks)
```

```
[[77 67 43]
 [91 88 64]
 [87 79 58]
 [95 43 87]
 [69 96 70]]
```

[7]:
```
#2D array martrix
marks.shape
```

[7]:
```
(5, 3)
```

[8]:
```
r1, r2, r3 = 0.6, 0.9, 0.8
```

[10]:
```
randoms = np.array([r1, r2, r3])
```

[11]:
```
print(randoms)
```

```
[0.6 0.9 0.8]
```

[12]:
```
#1D array (vector)
randoms.shape
```

[12]:
```
(3,)
```

[13]:
```
# 3D array
numbers = np.array([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.5]]])
```

[14]:
```
numbers.shape
```

[14]:
```
(2, 2, 3)
```

[15]:
```
randoms.dtype
```

[15]:
```
dtype('float64')
```

[16]:
```
marks.dtype
```
[16]:
```
dtype('int32')
```
[17]:
```
numbers.dtype
```
[17]:
```
dtype('float64')
```
[19]:
```
randoms.dtype
```
[19]:
```
dtype('float64')
```
[20]:
```
marks.dtype
```
[20]:
```
dtype('int32')
```
[22]:
```
numbers.dtype
```
[22]:
```
dtype('float64')
```
[23]:
```
np.matmul(marks, randoms)
```
[23]:
```
array([140.9, 185. , 169.7, 165.3, 183.8])
```
[24]:
```
marks @ randoms
```
[24]:
```
array([140.9, 185. , 169.7, 165.3, 183.8])
```

Conclusion:

In this practical, we explored the Numpy library for numerical computing in Python.
We also learnt that Numpy operations and functions are implemented internally in C++, which makes them much faster than using Python statements & loops that are interpreted at runtime.
As we studied, using np.dot is 100 times faster than using a for loop.
This makes Numpy especially useful while working with really large datasets with tens of thousands or millions of data points.