Aumkar Gadekar
Rno-21
Batch B

Experiment 1

Aim : Using Numpy Library to perform and understand the following operations :

- Indexing
- Handling non-existing values
- Comparing runtime behaviours

Theory :

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Indexing :
There are two types of indexing :

Basic Slicing and indexing : Consider the syntax x[obj] where x is the array and obj is the index. Slice object is the index in case of basic slicing. Basic slicing occurs when obj is :

1. a slice object that is of the form start : stop : step
2. an integer
3. or a tuple of slice objects and integers

All arrays generated by basic slicing are always view of the original array.

```
In [52]: a = np.array([[7 ,2,5 ,9 ],[3 ,4,11,13 ],[5 ,6,7.5,9 ],[15 ,8,5.2,9.3]])

In [53]: a
Out[53]: array([[ 7. ,  2. ,  5. ,  9. ],
                [ 3. ,  4. , 11. , 13. ],
                [ 5. ,  6. ,  7.5,  9. ],
                [15. ,  8. ,  5.2,  9.3]])

In [54]: a[1]  # One dimensional sub array
Out[54]: array([ 3.,  4., 11., 13.])

In [55]: a[:,3]
Out[55]: array([ 9. , 13. ,  9. ,  9.3])

In [56]: a[1,2] #Accessing single element
Out[56]: 11.0

In [57]: a[1:,2:] #slicing across multiple dimensions
Out[57]: array([[11. , 13. ],
                [ 7.5,  9. ],
                [ 5.2,  9.3]])
```

Advanced indexing : Advanced indexing is triggered when obj is :

1. an ndarray of type integer or Boolean

2. or a tuple with at least one sequence object
3. is a non tuple sequence object

Advanced indexing returns a copy of data rather than a view of it. Advanced indexing is of two types integer and Boolean.

Handling non-existing values :

Users interested in dealing with missing data within NumPy are generally pointed to the masked array subclass of the ndarray, known as ‘numpy.ma’. This class has a number of users who depend strongly on its capabilities, but people who are accustomed to the deep integration of the missing data placeholder “NA” in the R project and others who find the programming interface challenging or inconsistent tend not to use it.

This NEP proposes to integrate a mask-based missing data solution into NumPy, with an additional bit pattern-based missing data solution that can be implemented concurrently or later integrating seamlessly with the mask-based solution.

The mask-based solution and the bit pattern-based solutions in this proposal offer the exact same missing value abstraction, with several differences in performance, memory overhead, and flexibility.

The mask-based solution is more flexible, supporting all behaviors of the bit pattern-based solution, but leaving the hidden values untouched whenever an element is masked.

The bit pattern-based solution requires less memory, is bit-level compatible with the 64-bit floating point representation used in R, but does not preserve the hidden values and in fact requires stealing at least one bit pattern from the underlying dtype to represent the missing value NA.

Both solutions are generic in the sense that they can be used with custom data types very easily, with no effort in the case of the masked solution, and with the requirement that a bit pattern to sacrifice be chosen in the case of the bit pattern solution.

```
In [66]: print('Handling non-existing values')
         h=[23,64,12,78,34,68,2,85,21,48,57,89]
         h.append(np.nan)
         h

         Handling non-existing values

Out[66]: [23, 64, 12, 78, 34, 68, 2, 85, 21, 48, 57, 89, nan]
```

```
In [67]: h=[np.nan,21,32,45,68,78,np.nan,42,6,np.nan,98,22,84,np.nan,1,34,75,np.nan,np.nan]
         np.nan_to_num(h)

Out[67]: array([ 0., 21., 32., 45., 68., 78.,  0., 42.,  6.,  0., 98., 22., 84.,
                 0.,  1., 34., 75.,  0.,  0.])
```

Comparing runtime behaviours

As it is visible by the time taken, we can see that an array created in Numpy is more optimal than an array created in python. It is almost 20 times more efficient to use Numpy. Numpy also has a library of functions to use for dataset cleaning, manipulation, etc

Compare runtime behavior

```
In [70]: print('COMPARING RUNTIME BEHAVIOUR')
         a=[]
         %time for i in range(0,10000) : a.append(i)

         %time a=np.arange(0,10000)

         COMPARING RUNTIME BEHAVIOUR
         Wall time: 998 µs
         Wall time: 0 ns
```

Conclusion : Thus, we have learned how to use NumPy arrays for array indexing, math operations, and loading and saving data. We have also learned how to handle non existent data and how numpy arrays are better than python arrays