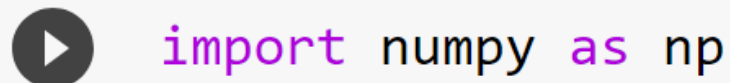


Chapter 13

Knowing the numpy library

numpy i.e. Numerical Python is largely used for numerical and scientific computing. It provides support for large, multi-dimensional arrays and matrices, along with an extensive collection of mathematical functions to operate on them efficiently.

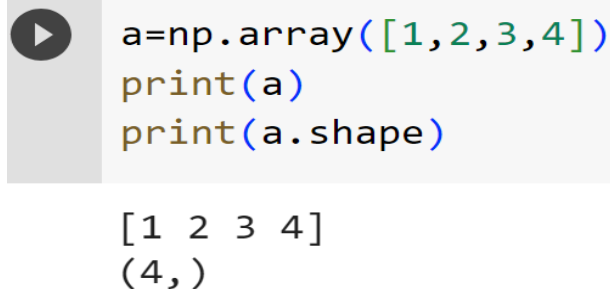
As python supports aliasing i.e giving a name to an existing module, hence we will import the *numpy* module as *np* just for simplicity.

A code snippet showing the import statement for the numpy module. It features a play button icon on the left and the text `import numpy as np` in a purple font on a light gray background.

```
import numpy as np
```

Figure 13.1: Aliasing the numpy module.

13.1 Initialise a 1-D array using numpy


A code snippet showing the initialization of a 1-D array and printing its shape. It features a play button icon on the left and the code `a=np.array([1,2,3,4])`, `print(a)`, and `print(a.shape)` on a light gray background. Below the code, the output is shown: `[1 2 3 4]` and `(4,)`.

```
a=np.array([1,2,3,4])
print(a)
print(a.shape)
```

`[1 2 3 4]`
`(4,)`

Figure 13.2: Initialise a 1-D array using numpy.

13.2 Initialise a 2-D array using numpy



```
a=np.array([[1,2,3],[3,4,5]])  
print(a)  
print(a.shape)
```

```
[[1 2 3]  
 [3 4 5]]  
(2, 3)
```

Figure 13.3: Initialise a 2-D array using numpy.


13.3 Different library functions of numpy

Different functions for numpy are-


13.3.1 ones()

This function creates an array of specified shape filled with only ones.

In the below diagram fig.13.4, we can see that an array of length 3 is created by the ones() function. It should be noted that, **by default**, the data type of the array is **float**.



```
a=np.ones(3)  
print(a)
```



```
[1. 1. 1.]
```

Figure 13.4: using np.ones().

However, we can cast the data type to a type of our wish. As example, in fig. 13.5 we have cast the array as boolean and in fig. 13.6 we have cast the matrix of zeroes as string.

```
[ ] a=np.ones(3,dtype=bool)
    print(a)

[ True  True  True]
```

Figure 13.5: using `np.ones()` and casting the array as boolean type.

```
▶ a=np.ones((3,2),dtype=str)
  print(a)

[[ '1'  '1' ]
 [ '1'  '1' ]
 [ '1'  '1' ]]
```

Figure 13.6: using `np.ones()` creating a matrix and casting it as string type.

13.3.2 zeros()

This function creates an array of specified shape filled with only zeroes.

In the below diagram fig.13.7, we can see that an array of length 3 is created by the `zeros()` function. It should be noted that, **by default**, the data type of the array is ***float***.

```
▶ a=np.zeros(3)
  print(a)

[0. 0. 0.]
```

Figure 13.7: using `np.zeros()`.

However, we can cast the data type to a type of our wish. As example, in fig. 13.8 we have cast the array as boolean and in fig. 13.9 we have cast the matrix of zeroes as string.

```
▶ a=np.zeros(3,dtype=bool)
  print(a)
```

👤 [False False False]

Figure 13.8: using `np.zeros()` and casting the array as boolean type.

```
▶ a=np.zeros((3,2),dtype=str)
  print(a)
```

👤 `[['' '']`
`[['' '']`
`[['' '']`

Figure 13.9: using `np.zeros()` creating a matrix and casting it as string type.

13.3.3 `identity()`

A Square matrix with diagonal entries as 1 and rest as 0's is called as identity matrix.


In the below diagram fig.13.10, we can see that a matrix of dimension 3×3 is created by the `identity()` function. It should be noted that, **by default**, the data type of the array is ***float***.

```
▶ a=np.identity(2)
  print(a)
```

👤 `[[1. 0.]`
`[0. 1.]]`

Figure 13.10: using `np.identity()`

However, the type can be converted to a data we wish.

```
 a=np.identity(3,dtype=int)  
print(a)
```



```
 [[1 0 0]  
 [0 1 0]  
 [0 0 1]]
```

Figure 13.11: using `np.identity()` creating an identity matrix and casting it as integer type.


13.3.4 concatenation()

This function concatenates 2 arrays.

```
 a=np.array([[1,2],[3,4]])  
print(a)
```

```
[[1 2]  
 [3 4]]
```

Figure 13.12: creating a 2-D numpy array named as **a**

```
 b=np.array([[4,5],[6,7]])  
print(b)
```



```
 [[4 5]  
 [6 7]]
```

Figure 13.13: creating a 2-D numpy array named as **b**.

```
 c=np.concatenate((a,b),axis=0) #rowwise concatenation  
print(c)
```



```
  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$ 
```

Figure 13.14: Doing row wise concatenation

```
 c=np.concatenate((a,b),axis=1) #columnwise concatenation  
print(c)
```




```
  $\begin{bmatrix} 1 & 2 & 4 & 5 \\ 3 & 4 & 6 & 7 \end{bmatrix}$ 
```

Figure 13.15: Doing column wise concatenation.


13.3.5 Row-wise concatenation using vstack()

```
 c=np.vstack((a,b)) #rowwise concatenation  
print(c)
```

```
  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$ 
```

*Figure 13.16: Performing row wise concatenation on the previously created 2-D arrays **a** and **b**.*

13.3.6 column-wise concatenation using `hstack()`

```
 c=np.hstack((a,b)) #columnwise concatenation  
print(c)
```



```
 [[1 2 4 5]  
 [3 4 6 7]]
```

Figure 13.17: Performing column wise concatenation on the previously created 2-D arrays **a** and **b**.

13.4 Linear algebraic transformations using numpy module

13.4.1 Dot product

Dot product is basically the sum of the multiplication of corresponding elements of same size.

```
 a=np.array([1,2,3])  
b=np.array([4,5,6])  
c=np.dot(a,b)  
print(c)
```

```
 32
```

Figure 13.18: using `np.dot()`

13.4.2 Matrix multiplication

```
▶ x=np.array([[1,2,3],[4,5,6]])  
y=np.array([[7,8,9],[10,11,12],[13,14,15]])  
z=np.matmul(x,y)  
print(z)  
  
➞ [[ 66  72  78]  
    [156 171 186]]
```

Figure 13.19: using np.matmul() to multiply two matrices.

```
▶ x=np.array([[1,2,3],[4,5,6]])  
y=np.array([[7,8,9],[10,11,12],[13,14,15]])  
z=x@y  
print(z)  
  
➞ [[ 66  72  78]  
    [156 171 186]]
```

Figure 13.20: using @ operator to multiply two matrices.

13.4.3 Determinant

```
▶ x=np.array([[1,2],[4,5]])  
y=np.linalg.det(x)  
print(y)  
  
➞ -2.9999999999999996
```

Figure 13.21: calculating determinant of a matrix.

13.4.4 Inverse of matrix

```
▶ x=np.array([[1,2],[4,5]])  
  y=np.linalg.inv(x)  
  print(y)  
  
↳ [[-1.66666667  0.66666667]  
    [ 1.33333333 -0.33333333]]
```

Figure 13.22: calculating inverse of a matrix.

13.4.5 Transpose of matrix

```
▶ x=np.array([[1,2,3],[4,5,6]])  
  print(x)  
  
↳ [[1 2 3]  
    [4 5 6]]
```

Figure 13.23: Creating a 2-D array using numpy

```
▶ z=np.transpose(x)  
  print(z)  
  
↳ [[1 4]  
    [2 5]  
    [3 6]]
```

Figure 13.24: calculating transpose of a matrix using np.transpose().

13.5 Statistical operations

13.5.1 mean()

This function basically calculates the average from an array of data.

13.5.2 median()

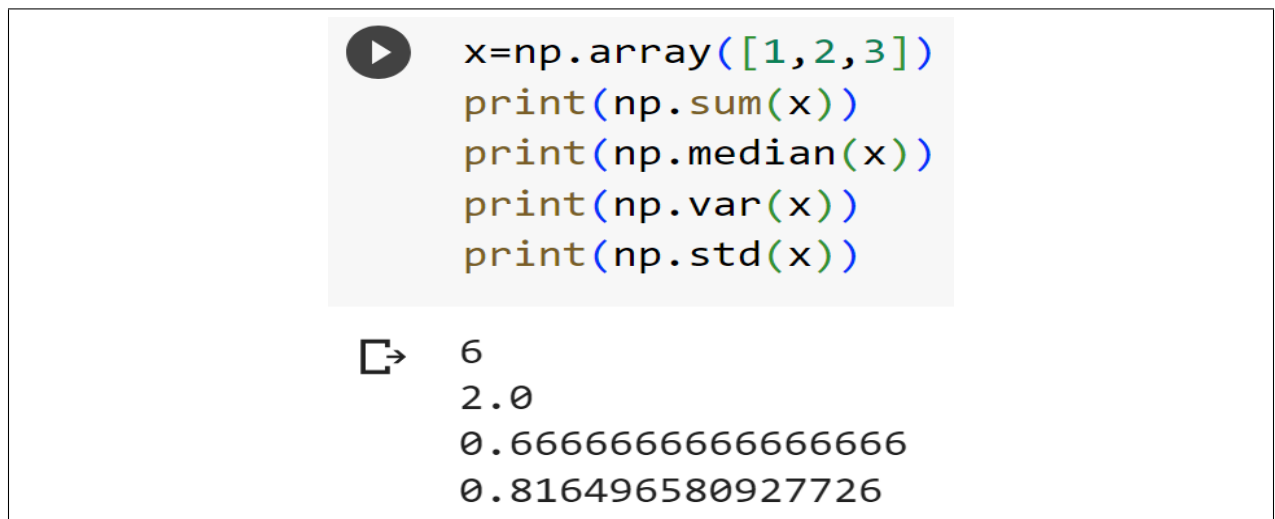
This function calculates the statistical median of the elements of the array given.

13.5.3 var()

This function calculates the statistical variance of the elements of the array given.

13.5.4 std()

This function calculates the statistical standard deviation of the elements of the array given.



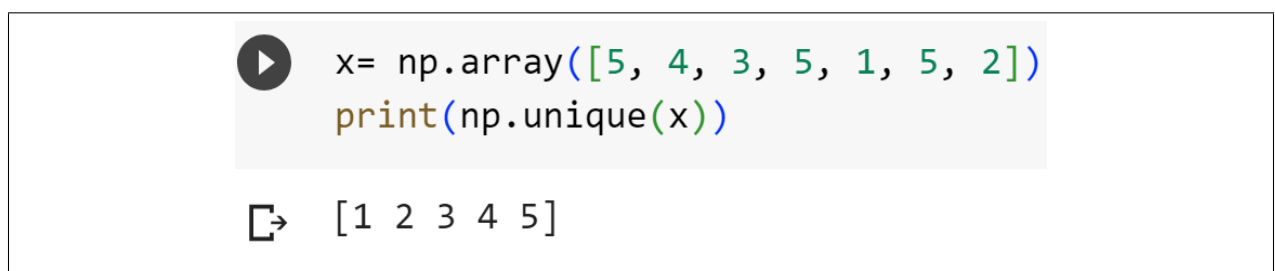
```
x=np.array([1,2,3])
print(np.sum(x))
print(np.median(x))
print(np.var(x))
print(np.std(x))
```

6
2.0
0.6666666666666666
0.816496580927726

Figure 13.25: calculating the statistical functions using numpy.

13.5.5 unique()

Fetches the distinct values from an array.



```
x= np.array([5, 4, 3, 5, 1, 5, 2])
print(np.unique(x))
```

[1 2 3 4 5]

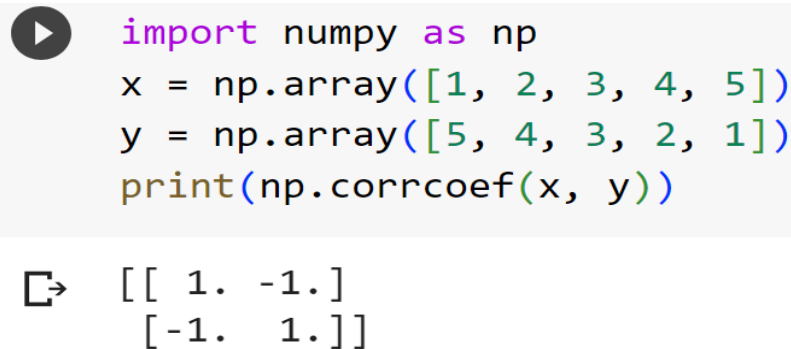
Figure 13.26: using np.unique() to extract unique elements of an array.

13.5.6 cov()

This function calculates the statistical co variance between 2 matrices or arrays of same dimension.

13.5.7 corrcoef()

This function calculates the statistical co variance between 2 matrices or arrays of same dimension.

A code execution snippet showing the import of numpy as np, creation of two arrays x and y, and the use of np.corrcoef to calculate the correlation coefficient. The output is a 2x2 matrix.

```
import numpy as np
x = np.array([1, 2, 3, 4, 5])
y = np.array([5, 4, 3, 2, 1])
print(np.corrcoef(x, y))
```

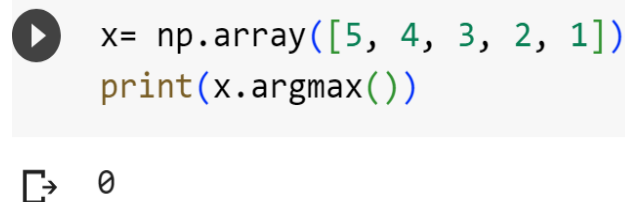
➡ $\begin{bmatrix} 1. & -1. \\ -1. & 1. \end{bmatrix}$

Figure 13.27: calculating correlation coefficient using numpy module.

13.5.8 argmax()

This function return the index of the maximum element in the array.

For example- In the given example in fig.13.28 the sequence given was [5, 4, 3, 2, 1] where the maximum element 5 resides in index 0. So the output is 0 here.

A code execution snippet showing the creation of an array x and the use of x.argmax() to find the index of the maximum element. The output is 0.

```
x= np.array([5, 4, 3, 2, 1])
print(x.argmax())
```

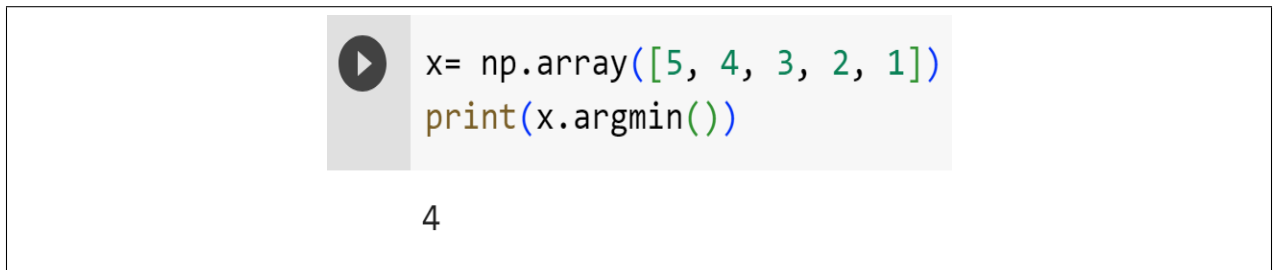
➡ 0

Figure 13.28

13.5.9 argmin()

This function return the index of the minimum element in the array.

For example- In the given example in fig.13.29 the sequence given was [5, 4, 3, 2, 1] where the minimum element 1 resides in index 4. So the output is 4 here.



```
x= np.array([5, 4, 3, 2, 1])
print(x.argmin())
```

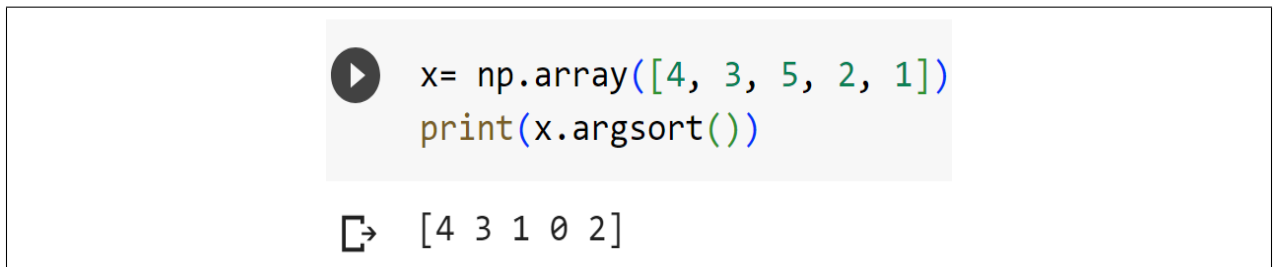
4

Figure 13.29

13.5.10 argsort()

This function return the indices that would sort the array.

For example- In the given example in fig.13.30 the sequence given was [4, 3, 5, 2, 1]. If we want to sort this sequence in ascending order, then the sequence will be [1, 2, 3, 4, 5]. The current index of 1,2,3,4,5 in the unsorted array are 4,3,1,0,2 respectively. So the output is the same.



```
x= np.array([4, 3, 5, 2, 1])
print(x.argsort())
```

[4 3 1 0 2]

Figure 13.30

Apart from these functions numpy also uses different trigonometric functions (*sin()*, *cos()*, *tan()*), inverse circular functions (*arcsin()*, *arc-cos()*, *arctan()*) , arithmetic functions (*add()*, *subtract()*, *multiply()*, *divide()*, *power()*) etc. However only the mostly used relevant functions are discussed.