Advanced functions in numpy :

* Numpy is a huge library we will only be learning functions that will be frequently used in Data-Science.

array_equal :

* Normally when we want to check if one array is same as another array we can use comparision operator.

i.e
```
x = np.arange(0,6).reshape(3,2)
y = np.arange(0,6).reshape(3,2)

x==y
↳ array([[[True, True, True]
          [True, True, True]
          [True, True, True]]])
```

This is a element wise comparision and the result will be a boolean array with True and False.

* Instead if we want to see if arrays are same we can use np.array_equal() parameters will be arrays being compared.

```
np.array_equal(x,y)
↳ True.
```

* np.array_equal() checks if the elements are same and also if the shape is same.

ex:-

```
x = np.arange(0,6).reshape(3,2)
y = np.arange(4,10).reshape(3,2)
np.array_equal(x,y)
```

↳ False (shape-same, elements-different)

```
x = np.arange(4,10).reshape(3,2)
y = np.arange(4,10).reshape(1,6)
np.array_equal(x,y)
```

↳ False (shape-different, elements-same)

Logical operations on arrays :

* Logical operations on arrays are done element wise

```
np.logical_and (array1, array2)
np.logical_or (array1, array2)
```

* logical operators will check the boolean values and perform logical operations on it.

i.e bool(0) = False

bool (non-zero) = True (for +ve or -ve values)

bool ("") = False
↑ Empty string

bool ("...") = True
↑ non-empty string

* np.logical_and → If both the elements are True then True else False.
* np.logical_or → if either one of the elements is True then True else False.

ex:-

```
x1 = np.array([[1,2,3,4]])
x2 = np.array([[4,5,6,7]])
np.logical_and (x1,x2)
↳ array([[True, True, True, True]])
```

```
x3 = np.array([[0, 5, 0, 7]])
np.logical_and(x1, x3)
    ↳ array([[False, True, False, True]])

np.logical_or(x1, x3)
    ↳ array([[True, True, True, True]])

x4 = np.array([[0, 0, 0, 0]])
x5 = np.array([[0, 0, 2, 3]])
np.logical_or(x4, x5)
    ↳ array([[False, False, True, True]])
```

Functions considered as logical operators

np.any(): This function returns True if any one element of the array is True.

np.all(): This function returns True if all the elements of the array are True else False.

* These functions are very helpful when we are using comparision operators.

ex:- `x = np.arange(1,21).reshape(5,4)`

if we want to check elements are divisible by 2.

`x%.2 == 0`

↳ boolean array

* If we want to just check if all the elements of the array are divisible by 2

then `np.all(x%.2==0)`

↳ False

`np.any(x%.2==0)`

↳ True

* If we want to check if any elements of the array are divisible by 2.
* when we do machine learning we use matrices that are huge. These functions will be useful there.

Multiplication (*) – This is done element wise and the result array is same shape of the original array.

dot Product (np.dot()) – we use dot product when multiplying two arrays (two matrices)

Properties

1) No. of columns of first array should match no. of rows of second array.

$$a_{m \times n} \quad b_{n \times q}$$

these need to be same

2) The result array will have a shape of (m, q)

3) If the second array no. of rows does not match we can use Transpose Function to change rows to columns and columns to rows.

Transpose (.T)

ex: `x = np.arange(0,6).reshape(3,2)`

`y = np.arange(0,6).reshape(3,2)`

`x*y`

↳ `array([[0, 1]`
`[4, 9]`
`[16, 25]])`

`np.dot(x,y)`

↳ X error as shape of x = (3,2)
shape of y = (3x2)
but we need (2,3)

`y.T`

↳ `array([[0,2,4]`
`[1, 3, 5]])`

`np.dot(x, y.T)`

↳ `array([[1, 3, 5]`
`[3, 13, 23]`
`[5, 23, 41]])`

Important functionalities that we can perform on array :

* numpy has many functionalities using which we can manipulate arrays. we will learn few.
* These are called reduction functions as they reduce the number of elements in an array.
* All reduction functions can take 2 parameters.
* First parameter of a reduction function is always the array.
* second parameter is the axis.

In numpy

when axis = 0 → column wise operation is done

axis = 1 → row wise operation is done.

default axis = None.

1) np.sum(array) → sum of all elements inside the array

np.sum(array, axis=0) → sum column wise

np.sum(array, axis=1) → sum row wise

ex:-

```
x = np.arange(0,9).reshape(3,3)
```

x

↳ array([[0, 1, 2]
[3, 4, 5]
[6, 7, 8]])

np.sum(x) (i.e 0+1+2+3+4+5+6+7+8)

↳ 36

np.sum(x, axis=0)

↳ array([9, 12, 15])

(i.e [0+3+6, 1+4+7, 2+5+8])

np.sum(x, axis=1)

↳ array([3, 12, 21])

(i.e [0+1+2, 3+4+5, 6+7+8])

2) np.prod(array) → prod of all elements inside the array

np.prod(array, axis=0) → prod column wise

np.prod(array, axis=1) → prod row wise

np.prod(x) (i.e 0*1*2*3*4*5*6*7*8)

↳ 0

np.prod(x, axis=0)

↳ array([0, 28, 80])

(i.e [0*3*6, 1*4*7, 2*5*8])

np.prod(x, axis=1)

↳ array([0, 60, 336])

(i.e [0*1*2, 3*4*5, 6*7*8])

3) np.min(array) → min of all elements inside the array

np.min(array, axis=0) → min column wise

np.min(array, axis=1) → min row wise

np.min(x)

↳ 0

np.min(x, axis=0)

↳ array([0, 1, 2])

np.min(x, axis=1)

↳ array([0, 3, 6])

4) np.max(array) → max of all elements inside the array

np.max(array, axis=0) → max column wise

np.max(array, axis=1) → max row wise

```
np.max(x)
  ↳ 8

np.max(x, axis=0)
  ↳ array([6, 7, 8])

np.max(x, axis=1)
  ↳ array([2, 5, 8])
```

np.nan - This represents NULL values in array's.
i-e where there is no value
↳ missing cell in Excel or an NULL cell.

* default datatype of np.nan is float. so the array needs to be of float datatype.
* If the array is already existing and its data-type is not float we can use astype() to change the data type.

array.astype(dtype)

This temporarily converts the dtype of the array to the mentioned dtype. If we want to use the result we can assign it to a variable.

ex:-

```
x
↳ array([[0, 1, 2]
         [3, 4, 5]
         [6, 7, 8]])
```

```
x[1, 1] = np.nan
    ↳ X  error as np.nan default dtype
         is float. x is of dtype int.
```

```
x = x.astype(float)
    ↳ changing dtype and assigning
      back to x.
```

```
x
↳ array([[0.0, 1.0, 2.0]
         [3.0, 4.0, 5.0]
         [6.0, 7.0, 8.0]])
```

```
x[1, 1] = np.nan
```

x

↳ array([[0.0, 1.0, 2.0]
[3.0, nan, 5.0]
[6.0, 7.0, 8.0]])

↳ here nan is 'NULL' not a string.

* But when we are using sum() or prod() functions with arrays containing nan we will have issues.

np.sum(x)
↳ nan (as nan has no value result is also nan)

np.prod(x)
↳ nan

5) np.nansum(array) - Simillar to sum() but takes nan values as 0 for sum.

np.nansum(x, axis=0) — column wise

np.nansum(x, axis=1) — row wise

np.nansum(x)
↳ 32.0

np.nansum(x, axis=0)
↳ array([9, 12, 15])

```
np.nansum(x, axis = 1)
    ↳ array([3, 12, 21])
```

6) **np.nanprod(array)** – Similar to prod() but takes nan values as 1 for prod

np.nanprod(x, axis = 0) — column wise

np.nanprod(x, axis = 1) — row wise

```
np.nanprod(x)
    ↳ 0.0

np.nanprod(x, axis = 0)
    ↳ array([0, 28, 80])

np.nanprod(x, axis = 1)
    ↳ array([0, 60, 336])
```

np.isnan(array) function: function isnan() will check if nan is there in the array. If present it returns True else False. This is done element wise. Result is a boolean array.

ex:- x = np.arange(0,9, dtype=float).reshape(3,3)

```
x
↳ array([[0.0, 1.0, 2.0]
         [3.0, 4.0, 5.0]
         [6.0, 7.0, 8.0]])
```

```
x[0,1] = np.nan
x[1,2] = np.nan

x
↳ array([[0.0, nan, 2.0]
         [3.0, 4.0, nan]
         [6.0, 7.0, 8.0]])

np.isnan(x)
↳ array([[False, True, False]
         [False, False, True]
         [False, False, False]])
```

* If we want to replace nan with our own value. we can use boolean indexing.

```
x[np.isnan(x)] = 4.57

x
↳ array([[0.0, 4.57, 2.0]
         [3.0, 4.0, 4.57]
         [6.0, 7.0, 8.0]])
```

np.inf - This represents infinite value in arrays.

```
x[0,0] = np.inf

x
↳ array([[inf, 4.57, 2.0]
         [3.0, 4.0, 4.57]
         [6.0, 7.0, 8.0]])
```

isinf(array) function: function isinf(array) will check if inf is there in the array. If present it returns True else False.

Operation is done element wise and result is a boolean array.

```
np.isinf(x)
    ↳ array([[[True, False, False]
              [False, False, False]
              [False, False, False]]])
```

7) **np.cumsum(array)** - Returns the cummulative sum over the flattened array.

np.cumsum(x, axis=0) - Returns the cummulative sum column wise.

np.cumsum(x, axis=1) - Returns the cummulative sum row wise.

* A new array holding the result is returned.

Cummulative sum - It is a sequence of partial sums of a given sequence.

ex:- the cummulative sums of [a, b, c, ...] are [a, a+b, a+b+c, ...]

ex:-
```
x = np.array([1, 2, 3, 4])
np.cumsum(x)
    ↳ array([1, 3, 6, 10])
```

```
y = np.array([[1,2,3],[4,5,6],[7,8,9]])
np.cumsum(y)
    ↳ array([1, 3, 6, 10, 21, 28, 36, 45])
np.cumsum(y, axis=0)
    ↳ array([[1, 2, 3]
             [5, 7, 9]
             [12, 15, 18]])
np.cumsum(y, axis=1)
    ↳ array([[1, 3, 6]
             [4, 9, 15]
             [7, 15, 24]])
```

8) **np.cumprod(array)** - Returns the cummulative over the flattened array.

np.cumprod(x, axis=0) - Returns the cummulative product column wise.

np.cumprod(x, axis=1) - Returns the cummulative product row wise.

* A new array holding the result is returned.

Cummulative product It is a sequence of partial products of a given sequence.

ex:- the cummulative of [a, b, c, ...] are [a, a*b, a*b*c, ...]

ex:- x = np.array([1, 2, 3, 4])

```
np.cumprod(x)
    ↳ array([1, 2, 6, 24])

y = np.array([[1,2,3],[4,5,6],[7,8,9]])

y
 ↳ array([[1, 2, 3]
          [4, 5, 6]
          [7, 8, 9]])

np.cumprod(y)
    ↳ array([1, 2, 6, 24, 120, 720, 5040, 40320,
             362880])

np.cumprod(y, axis=0)
    ↳ array([[1, 2, 3]
             [4, 10, 18]
             [28, 80, 162]])

np.cumprod(y, axis=1)
    ↳ array([[1, 2, 6]
             [4, 20, 120]
             [7, 56, 504]])
```

9) np.nancumsum(array) - Returns the cummulative sum handling nan over the flattened array.

np.nancumsum(array, axis=0) - Returns the cummulative sum column wise handling nan.

np.nancumsum(array, axis=1) - Returns the cummulative sum row wise handling nan.

10) np.nancumprod(array) - Returns the cummulative prod handling nan over the flattened array.

np.nancumprod(array, axis=0) - Returns the cummulative prod column wise handling nan.

np.nancumprod(array, axis=1) - Returns the cummulative prod row wise handling nan.

copy() - function creates a copy of the original array.
(New array changes don't apply to old array)

view() - function creates a view of the original array.
(New array changes apply to old array)

ex:- x = np.arange(0,9).reshape(3,3)

x

↳ array([[0, 1, 2]
[3, 4, 5]
[6, 7, 8]])

x1 = x.copy()

x1

↳ array([[0, 1, 2]
[3, 4, 5]
[6, 7, 8]])

x1[1, 1] = 100

x1

↳ array([[0, 1, 2]
[3, 100, 5] (changed)
[6, 7, 8]])

x

↳ array([[0, 1, 2]
[3, 4, 5] (unchanged)
[6, 7, 8]])

x2 = x.view()

x2

↳ array([[0, 1, 2]
[3, 4, 5]
[6, 7, 8]])

x2[1, 1] = 100

x2

↳ array([[0, 1, 2]
[3, 100, 5] (changed)
[6, 7, 8]])

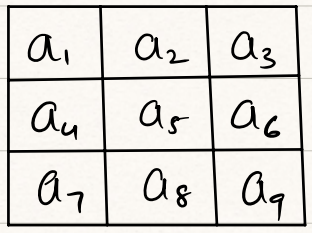
x

↳ array([[0, 1, 2]
[3, 100, 5] (changed)
[6, 7, 8]])

flatten() - This function flattens the array in to single dimension so only single row.

* we can use this when we want to iterate over sequential data in loops.
* If we want to use the result we can assign it to a variable.
* flatten will create a copy of original array. So changing after flattening will not be reflected in the original array.

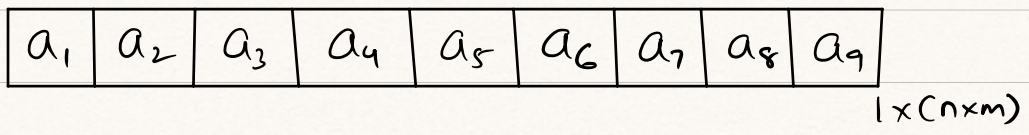
flatten in to a single dimension.



3 rows

↓

1 rows



ex:- x = np.arange(0,9).reshape(3,3)

```
x
↳ array([[[0, 1, 2]
          [3, 4, 5]
          [6, 7, 8]]])
```

```
x.flatten()
    ↳ array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```
for y in x.flatten():
    print(y)
      ↳ 0
        1
        2
        3
        4
        5
        6
        7
        8
```

Same using nested loops :-

```
for y in x:
    for z in y:
        print(z)
          ↳ 0
            1
            2
            3
            4
            5
            6
            7
            8
```

ravel(): This function does the same operation as flatten() but instead of copy the result array will be a view.

```
x.ravel()
  ↳ array([0, 1, 2, 3, 4, 5, 6, 7, 8])

y = x.ravel()  ⟶ (creates view)

y[0]
  ↳ 0

y[0] = 100

y
  ↳ array([100, 1, 2, 3, 4, 5, 6, 7, 8])
                                (changed)

x
  ↳ array([[100, 1, 2]
           [3, 4, 5]      (changed)
           [6, 7, 8]])

z = x.flatten()  ⟶ creates copy

z[2] = 120

z
  ↳ array([100, 120, 2, 3, 4, 5, 6, 7, 8])
                          (changed)

x
  ↳ array([[100, 1, 2]
           [3, 4, 5]      (unchanged)
           [6, 7, 8]])
```

np.round() function – round() is a mathematical functions that rounds float elements of array to the given number of decimals.

np.ceil() function – ceil() is a mathematical function that rounds off to the next integer value all the float elements of the given array.

np.floor() function – floor() is a mathematical function that rounds off to the lower integer value all the float elements of the given array.

ex:-

```
x = np.array([1.3435, 2.3435, 3.2333, 4.5678])

np.round(x, 2)
    ↳ precision of 2 decimal values
    ↳ array([1.34, 2.34, 3.23, 4.57])

np.round(x, 1)
    ↳ array([1.3, 2.3, 3.2, 4.6])

np.ceil(x)
    ↳ array([2.0, 3.0, 4.0, 5.0])

np.floor(x)
    ↳ array([1.0, 2.0, 3.0, 4.0])

np.round(x)
    ↳ array([1.0, 2.0, 3.0, 5.0])
```

Logarithm - The exponent or power to which a base must be raised to yeild a given number.

Expressed mathematically,

x is the logarithm of n to the base b.

if $b^x = n$, in which case one writes

$$x = \log_b n$$

ex:- $2^3 = 8$

$\therefore$ 3 is the logarithm of 8 to base 2

or $3 = \log_2 8$

Similarly $\log_{10} 100 = 2$

$\log 0$ - infinite

$\log 1$ - 0

$\log$ -ve number - undefined

np.log(array) - Calculates the natural logarithm element wise.

np.log2(array) - Calculates logarithm to base 2 element wise.

np.log10(array) - Calculates logarithm to base 10 element wise

ex:-

```
x = np.array([0, 1, 2, 3, -9, 10])
```

np.log(x)

↳ array ([-inf, 0.0, 0.69..... , nan, 2.3025..])

n.log2(x)

↳ array ([-inf, 0.0, 1.0, ... , nan, 3.3219..])

np.log10(x)

↳ array ([-inf, 0.0, 0.3013,... nan, 1.])

np.sqrt(array) - calculates the square root of the array element wise.

np.sqrt(x)

↳ array ([0.0, 1.0, 1.4142, ..., nan, 3.1622])

Assignment - snakes and Ladders (2 players)

what is happening in the Game:

1) Display the welcome message.
2) Collect the player's names.
3) Until one of the player reaches 100 do the following.
   a) roll the dice
   b) move the player forward for the value on dice
   c) if snakes head then move to tail.
   d) if ladders bottom take it to the top.
   e) else remain there and let the second player roll the dice.

Can use :-

1) for / while
2) if
3) input
4) Increment (position needs to keep incrementing)
5) comparision operator
6) Arithmetic operator.

Basics of statistics

Statistics is a field where we will deal with data.

- Collect data
- Analysis on data
- Summarize the data
- Estimate the data
- manipulation on data

Statistics common terminology

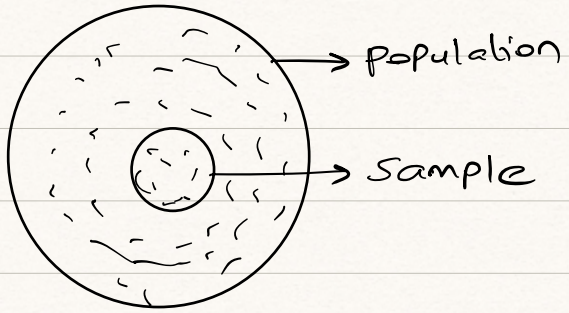
- Population (all the things)
- sample (subset of Population)

* In statistics we are conducting experiments using sample data and doing estimation on population.

ex:- Experiment : Average height of all the students at Innomatics who studied data science.

Population - height of all students from 2018-2022 (difficult to collect the data) (students ≃ 4000)

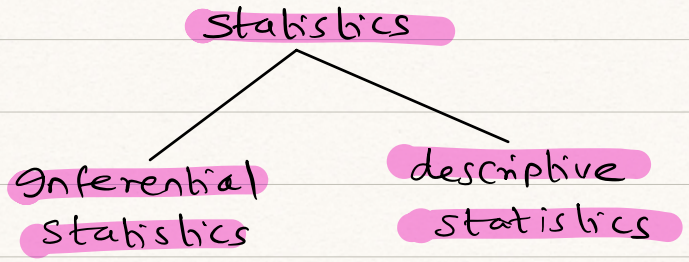
Sample - height of all the students of the current data science batches (students ≃ 500)



Sample is subset of population.

Experiment: Average height of all the students in batch 197. Here we get Population instead of Sample. But still there might be someone absent.

* we gather population or sample based on the experiment.
* Very rarely sample is equal to population.
* with the help of sample we can estimate population.



Descriptive statistics: whereever we are going to summarize (or describe) our sample.

- This deals with samples
- minimum (min)
- maximum (max)
- mean
- mode
- variance
- standard deviation

all these describe sample hence descriptive statistics.

Inferential statistics - It can be defined as a field of statistics that uses analytical tools or testing for drawing conclusions about a population by examining random samples.

* Calculate population mean with help of sample mean we will estimate using analytic tools and testing. (we will learn later)

Applying statistics concepts with the help of numpy

max - maximum, we can find the maximum value from the given set of numbers.

`np.max(array, axis)`

by default `axis = None`

`axis = 0` - column wise

`axis = 1` - row wise

min - minimum, we can find the minimum value from the given set of numbers.

`np.min(array, axis)`

by default `axis = None`

`axis = 0` - column wise

`axis = 1` - row wise

ex:-

```
x = np.array([11, 2, 3, 14, 5])
```

```
np.min(x)
  ↳ 2

np.max(x)
  ↳ 14
```

Mean - sum of all the observation in the sample by number of observations in the sample.

i.e average value.

ex:- [1, 2, 3, 4, 5]

$$\frac{1+2+3+4+5}{5} = 3 \text{ (mean)}$$

np.mean (array, axis)

by default axis = None

axis = 0 — column wise

axis = 1 — row wise

```
x = np.array([11, 2, 3, 14, 5])

np.mean(x)
  ↳ 7.0        ( (11+2+3+14+5) / 2 )
```

median - First we need to sort the given set of values in ascending order.

* For odd number of observations - medium is middle after sorting.

* For even number of observations - median is average of the two middle numbers.

ex:- 5, 4, 3, 2, 1	6, 5, 4, 3, 2, 1
Sort = 1, 2, 3, 4, 5	sort = 1, 2, 3, 4, 5, 6
median = 3	median = $\frac{3+4}{2} = 3.5$

np.median(array, axis)

by default axis = None

axis = 0 — column wise

axis = 1 — row wise

ex:- x = np.array([11, 2, 3, 14, 5])

np.median(x)

↳ 5.0

mode - This function is not present in numpy we need a different library.

This gives most number of repeating values. i.e the numbers with high frequency.

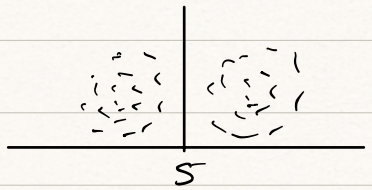
ex:- 1, 2, 3, 4, 5, 5, 4, 5, 5 (np.mode())

mode = 2

Variance - Simple definition is the spread around the mean

* First step is to calculate mean.

ex:- if we have 100 observations and they are very close to mean.



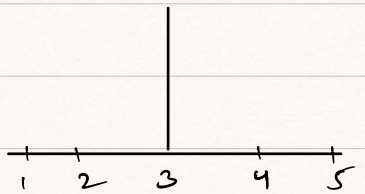
↳ variance is lower since spread is close.

If we have 100 points and they are far from mean — variance is higher since spread is far.

* If we want to calculate the spread we use statistical variance.

ex:- 1, 2, 3, 4, 5

$$\text{mean} = \frac{1+2+3+4+5}{5} = 3$$



Now the distance between

1 & 3

$1-3 = -2$ (negative distance so we use square)

$(1-3)^2 = 4$

2 & 3 — $(2-3)^2 = 1$

3 & 3 — $(3-3)^2 = 0$

4 & 3 — $(4-3)^2 = 1$

5 & 3 — $(5-3)^2 = 4$

$$\text{Variance} = \frac{\text{square of sum of distance}}{\text{total points}}$$

$$= \frac{4+1+0+1+4}{5}$$

$$\text{Variance} = 2$$

Formula for variance

$$\text{Variance} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$\bar{x}$ – sample mean

x_i – point

n – total values

in the above example

$$\text{Variance} = \frac{1}{5}\sum_{i=1}^{5}(x_i - 3)^2$$

Sample mean – $\bar{x}$

population mean – μ (myu)

* Variance is Average spread of the data points around the mean.

```
np.var(array, axis)
```

by default axis = None

axis = 0 – column wise

axis = 1 – row wise

ex:-

```
x = np.array([11, 2, 3, 14, 5])
np.var(x)
```

↳ 22.0

Standard deviation – This is a measure to show how much variation from the mean exists.

* This is equal to $\sqrt{\text{variance}}$

i.e $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2}$

`np.std(array, axis)`

by default `axis = None`

`axis = 0` – column wise

`axis = 1` – row wise

ex:-

```
x = np.array([11, 2, 3, 14, 5])
np.std(x)
    ↳ 4.69041....
```

argmin –

The min function gives the minimum value from the observation.

argmin gives the index value of the minimum value from the observations.

`np.argmin(array, axis)`

by default `axis = None`

`axis = 0` – column wise

`axis = 1` – row wise

ex:-

```
x = np.array([11, 2, 3, 14, 5])

np.argmin(x)
    ↳ 1    (index value of 2 is 1)
```

argmax: This gives the index value of the maximum value from the observations.

```
np.argmax(array, axis)
```

by default axis = None

axis = 0 — column wise

axis = 1 — row wise

ex:-

```
np.argmax(x)
    ↳ 3    (index value of 14 is 3)
```

For 2D arrays argmin and argmax is calculated on the flattened array.

```
ex:- x1 = np.array([[11,2,13,4],[20,12,4,16]])

x1.flatten()
    ↳ array([11, 12, 13, 4, 20, 12, 4, 16])

x1
↳ array([[11, 12, 13, 14]
         [20, 12, 4, 16]])
```

```
np.argmax(x1)
    ↳ 4    (index value of 20 in flattened array is 4)

np.argmin(x1)
    ↳ 1    (index value of 2 in flattened array in 2)

np.argmax(x1, axis=0)
    ↳ array([1, 1, 0, 1])

np.argmax(x1, axis=1)
    ↳ array([2, 0])

np.argmin(x1, axis=0)
    ↳ array([0, 0, 1, 0])

np.argmin(x1, axis=1)
    ↳ array([1, 2])
```

Example of using statistical numpy functions on 2D-array :

```
x1 = np.array([[1, 2, 3, 4], [15, 12, 14, 15]])
x1
↳ array([[1, 2, 3, 4]
         [15, 12, 14, 15]])
np.min(x1, axis=0)
    ↳ array([1, 2, 3, 4])
np.min(x1, axis=1)
    ↳ axis([1, 21])
np.max(x1, axis=0)
    ↳ array([15, 12, 14, 15])
np.max(x1, axis=1)
    ↳ array([4, 15])
np.mean(x1)
    ↳ 8.25
np.mean(x1, axis=0)
    ↳ array([8.0, 7.0, 8.5, 9.5])
np.mean(x1, axis=1)
    ↳ array([2.5, 14.0])
np.median(x1, axis=0)
    ↳ array([8.0, 7.0, 8.5, 9.5])
np.median(x1, axis=1)
    ↳ array([2.5, 14.5])
```

```
np.var(x1)
    ↳ 34.4375
np.var(x1, axis=0)
    ↳ array([49.0, 25.0, 30.25, 30.25])
np.var(x1, axis=1)
    ↳ array([1.25, 1.5])
np.std(x)
    ↳ 5.86834...
np.std(x1, axis=0)
    ↳ array([7.0, 5.0, 5.5, 5.5])
np.std(x1, axis=1)
    ↳ array([1.11803..., 1.2247...])
```

Random module in numpy library:

`np.random.rand()` → generates number between 0,1 not including 1.

↳ 0.78792

`np.random.rand(2,3)` → shape is given as input and creates dimensional array with random values between 0 & 1 not including 1.

↳ array([[[0.936..., 0.50..., 0.60...]
[0.50..., 0.40..., 0.20...]]])

`np.random.randint(1, 12, (12,11))`

↳ this creates a array of array of shape (12,11) filled with random integers between 1 & 12. Not including 12.

(* different from python random module where randint includes the ending value)

`np.random.choice(array, size of result)`

Choosing random elements from a array with replacement no.of times.

```
x = np.array([1, 2, 3, 4])
np.random.choice(x, size=3)
```

↳ array([3, 3, 1])

```
np.random.choice (x, size=(2,3))
  ↳ array ([[3, 3, 1]
            [1, 2, 3]])
```

↑ converts random numbers to 2D-array

np.unique – To find the unique values.

i.e values after removing duplicates.

```
x = np.array([11, 2, 3, 14, 1, 1, 1, 1, 1, 1, 1, 2, 3, 11, 5, 6, 5])

np.unique(x)
  ↳ array([1, 2, 3, 5, 6, 11, 14])
```

* If we want to get the count of each unique value (we need to give return_counts arguments as True)

```
np.unique (x, return_counts = True)
  ↳ array([1, 2, 3, 5, 6, 11, 14]) → unique value array
    array([7, 2, 2, 2, 1, 2, 1]) → count of the unique values.
```

* If we want to get the index values of the unique values in the array.
(we need to give return_index argument as True)

np.unique(x, return_index=True, return_counts=True)

↳ array order

array([1, 2, 3, 5, 6, 11, 14]) → unique value array

array([4, 1, 2, 14, 15, 0, 3]) → index array

array([7, 2, 2, 2, 1, 2, 1]) → count array

so if we want to use these arrays we can save them to variables.

unique_e, index_v, count_e = np.unique(x, return_index=True, return_counts=True)

unique_e

↳ array([1, 2, 3, 5, 6, 11, 14])

index_v

↳ array([4, 1, 2, 14, 15, 0, 3])

count_e

↳ array([7, 2, 2, 2, 1, 2, 1])

np.repeat (array, no.of times)

This repeats the elements multiple no.of times.
Input is the array and how many times to repeat.

ex:- `x = np.array([1,2,3,4])`

`np.repeat(x,3)`

↳ `array([1,1,1,2,2,2,3,3,3,4,4,4])`

np.tile (array, no.of times)

This repeats the array no.of times mentioned.
we can also use shape for no.of times to create multi dimensional array.

ex:- `np.tile(x)`

↳ `array([1,2,3,4,1,2,3,4,1,2,3,4])`

`np.tile(x, (2,3))` → shape says in rows repeat 2 times and in column repeat 3 times.

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

* if we want to repeat a particular element from the array.

`np.repeat(x[3],5)`

↳ `array([4,4,4,4,4])`

np.where() – Very Powerful function.

If we want to get the index values of the elements that satisfy the condition instead of actual values we use where() function.

ex:-

```
x = np.arange(1,21).reshape(5,4)
```

boolean masking

```
x[x%3==0]
```

↳ array([3,6,9,12,15,18])

if we want index values

```
x1 = x.flatten()
```

(Here if we don't apply flatten we will get row index & column index)

```
x1
```

↳ array([1, 2, 3, ..., 20]

```
np.where(x1%3==0)
```

↳ array([2, 5, 8, 11, 14, 17])

↳ these are the indexes of the elements satisfying the x1%3==0 condition.

if we want to get values satisfying the condition from here

```
a = np.where(x1%3==0)
x1[a]
```

↳ array([3,6,9,12,15,18])

x

↳ array ([[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 16]
[17, 18, 19, 20]])

np.where(x%3==0)

↳ (array([0, 1, 2, 2, 3, 4]), array([2, 1, 0, 3, 2, 1]))

row index — column index

This is similar to integer based indexing

b = np.where(x%3==0)

b[0] → all elements row indexes

↳ array([0, 1, 2, 2, 3, 4])

b[1] → all elements column indexes

↳ array([2, 1, 0, 3, 2, 1])

Now if we want to find values

i) using integer based indexing

x[b[0], b[1]]

↳ array([3, 6, 9, 12, 15, 18])

(or)

x[np.where(x%3==0)]

↳ array([3, 6, 9, 12, 15, 18])

Additional functionalities using where( ) function:

** Using where() function we can replace the elements to what we want based on the condition.

i.e np.where(x1%3==0, True, False)

↳ array([False, False, True, ... False])

here we are saying for values for which the condition is True we change that element to True and for values for which the condition is False we change those values to False.

This is giving us the boolean array so using this we can do masking.

x1[np.where(x1%3==0, True, False)]

↳ array([3, 6, 9, 12, 15, 18])

* These replaced elements need not be True or False, they can be anything.

ex:- values divisible by 3 - 'div by 3'

values not divisible by 3 - 'not div by 3'

np.where(x1%3==0, 'div by 3', 'not div by 3')

↳ array(['not div by 3', 'not div by 3', 'div by 3', ...])

take() function: take() function takes index values from the where function and gives the values. This is similar to what we did using integer based indexing. we do not use take() function frequently.

ex:- x1 = np.array([5, 3, 2, 1, 6, 4])

z = np.where(x1 % 2 == 0)

x1.take(z)

↳ array([2, 6, 4])

(or)

np.take(x1, z)

↳ array([2, 6, 4])

Concatenation of arrays: This is merging of multiple arrays in to a single array.

* merging can be done column wise or row wise.
* here axis = 0 - row wise merging } different from before
  axis = 1 - column wise merging }

ex:-

```
x1 = np.array([[1,2,3],[4,5,6]])
x2 = np.array([[11,12,13],[14,15,16]])
```

concatenate row wise

```
np.concatenate([x1,x2], axis=0)
↳ array([[1,2,3]
         [4,5,6]
         [11,12,13]
         [14,15,16]])
```

concatenate column wise

```
np.concatenate([x1,x2], axis=1)
↳ array([[1,2,3,11,12,13]
         [4,5,6,14,15,16]])
```

we can concatenate any number of arrays

```
np.concatenate([x1,x2,x1,x3], axis=1)
```

* default is axis=0 which is row wise

Other methods of concatenation

np.hstack() – horizontal stacking which is similar to column wise concatenation.

```
np.hstack([x1, x2, x1])
↳ array([[1, 2, 3, 11, 12, 13, 1, 2, 3]
         [4, 5, 6, 14, 15, 16, 4, 5, 6]])
```

np.vstack() – Vertical stacking which is similar to row wise concatenation

```
np.vstack([x1, x2, x1])
↳ array([1, 2, 3]
        [4, 5, 6]
        [11, 12, 13]
        [14, 15, 16]
        [1, 2, 3]
        [4, 5, 6]])
```

r_ bind – row wise binding

* This is an attribute and not a function.

```
np.r_[x1, x2]
↳ array([[1, 2, 3]
         [4, 5, 6]
         [11, 12, 13]
         [14, 15, 16]])
```

* These binds come from C-language and very fast than concatenation.

C_bind - column wise binding

```
np.c_[x1, x2]
↳ array([[1, 2, 3, 11, 12, 13]
         [4, 5, 6, 14, 15, 16]])
```

If we want to save an array or arrays and share it to others.

* Concept of save and savez.
* important in Deep learning where images are saved as numbers and shared.
* Save saves single array object.
* Savez saves multiple array objects.
* Save creates .npy (numpy python file)
* Savez creates .npz (numpy zipped file)

ex:-

```
x = np.arange(1,21).reshape(4,5)
```

To save this object x and share it

```
np.save(r"C:.....\array1", x)
```

↑ raw format — ↑ filepath — ↑ array object to be saved

↳ array1.npy

if we want to load a .npy file.

```
np.load(r"C:\....\array1.npy")
```

↳ array([[[1, 2, ...]
[.....]
[.....]
[....., 20]]])

```
x1 = np.arange(1, 11).reshape(2, 5)
x2 = np.arange(1,31).reshape(6,5)
```

* To save multiple array objects x, x1, x2 and share it we use np.savez which creates as .npz file.

```
np.savez(r"C:\....\array2", x, x1, x2)
```

raw format — file path & file name — array objects to be saved

↳ array1.npy

when we load this since multiple arrays are stored iterable object is created.

```
q = np.load(r"C:\....\array2.npz")
q.files        (files gets list of files)
```

↳ ['arr_0', 'arr_1', 'arr_2']

```
q['arr_0']
```

↳ array([[[1, 2, 3, ...]
[.....]
[.....]
[....., 20]]])

```
q['arr_1']
```

↳

```
array ([[1, 2, 3, ...]
        [------ ]
        [------ ]
        [------ ]
        [---- 30]])
```

Sorting the array in increasing order or decreasing order :

* This can be done row wise or column wise
* axis = 0 column wise
  axis = 1 row wise
  default is row wise

* All the elements in the array are shuffled when sorting.

ex:-

```
q1 = np.array([[3, 2, 6, 9, 1], [1, 2, 9, 3, 5], [16, 13, 14, 12, 9]])

q1
↳ array([[3, 2, 6, 9, 1]
         [1, 2, 9, 3, 5]
         [16, 13, 14, 12, 9]])

np.sort(q1, axis=0)      (column wise)
↳ array([[1, 2, 6, 3, 1]
         [3, 2, 9, 9, 5]
         [16, 13, 14, 12, 9]])
```

```
np.sort(q1, axis=1)        (row wise, if axis is not mentioned then this is default)
↳ array([[1,2,3,6,9]
         [1,2,3,5,9]
         [9,12,13,14,16]])
```

* If we want to get the index values (original) of the sorted and shuffled elements.

argsort():

displays the original index of the sorted values.

```
ex:- q2 = np.array([3,2,6,9,1])
q2
↳ array([3,2,6,9,1])
np.argsort(q2)                    (Sorted array([1,2,3,6,9])
↳ array([4,1,0,2,3]) ← original index values of sorted array
```

using these sorted indexes we can get sorted values

```
q2
↳ array([3,2,6,9,1])
q2[np.argsort(q2)]
↳ array([1,2,3,6,9])
```

* This method useful during conditional sorting based on indexes.
  - used in the next assignment.

np.asarray( ) – Convert sequential data to array.

* Not used commonly instead we use np.array( )

```
l = [1,2,3,4,5]
np.asarray(l)
```

(or)

```
t = ([1,2,3],[4,5,6]) or t = (1,2,3,4,5)
np.asarray(t)
```

Convert array to list

```
np.array(l).tolist()
```

→ [1,2,3,4,5]

(or)

```
x = np.arange(1,21).reshape(5,4)
x.tolist()
```

→ [[1,2,3,4],[5,6,7,8],[9,10,11,12],...[ ]]

(returned nested lists)

Common elements between two arrays: This is similar to sets.

```
np.intersect1d(array1, array2)
```

ex:-

```
a = np.array([1,2,3,4,5,6])
b = np.array([1,3,5,7])
```

np.intersect1d(a,b)

↳ array([1,3,5])

Trignometry: All trignometry functions are supported in numpy.

ex:- np.sin(array), np.cos(array), np.tan(array),....

result will be an array of angles.

ex:-

```
a = np.array([1,2,3,4,5,6])
np.sin(a)
↳ array([0.8414, 0.909, ....., -0.27...])
```

If we want to preserve data integrity by keeping the data in original data type:

* we can use data type called 'object' i.e dtype='O'

ex:- np.array([1, 'a', 2.2, 'b', 3+2j], dtype='O')

↳ array([1, 'a', 2.2, 'b', 3+2j], dtype=object)

without converting to dtype='O'.

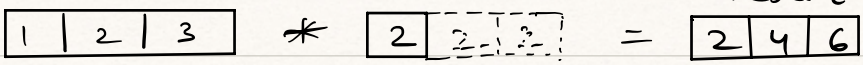
```
np.array([1, 'a', 2.2, 'b', 3+2j])
↳ array(['1', 'a', '2.2', 'b', '3+2j'], dtype='<U64')
```

↳ everything is converted to string data type.

Broadcasting: The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.

* Subject to certain constraints the smaller array is 'broadcast' across the larger array so that they have compatible shapes.
* Operations on arrays is done element wise so the shape of arrays needs to be same when shape is different broadcasting comes in to picture.

ex:- a(3,) * b(1,) = result(3,)



i.e 'b' is streched to match the shape of 'a' so arithmetic operation can be performed.

General Broadcasting Rules:

When operating on two arrays, numpy compares their shape element-wise. It starts with the trailing (i.e right most) dimension and works its way left.

Two dimensions are compatible when

1. they are equal or
2. one of them is 1.

If these conditions are not met we will get Value Error.

ex:- Broadcastable arrays

A (2d array) : 5 × 4

B (1d array) : 1

result (2d array) : 5 × 4

a (4×3)

0	0	0
10	10	10
20	20	20
30	30	30

b (3)

1	2	3

A (3d array) : 15 × 3 × 5

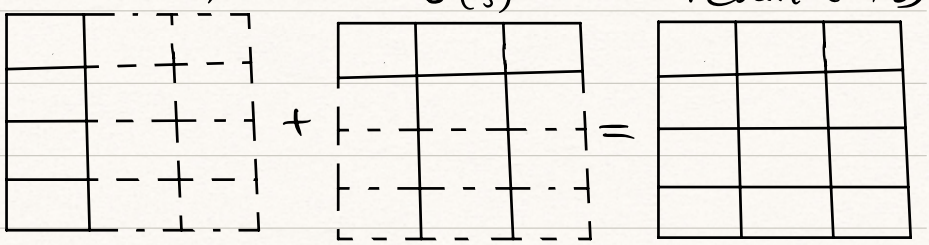
B (3d array) : 15 × 1 × 5

result (3d array) : 15 × 3 × 5

result (4×3)

1	2	3
11	12	13
21	22	23
31	32	33

a (4×1) + b (3) = result (4×3)



A (3d array) : 15 × 3 × 5

B (2d array) : 3 × 1

result (3d array) : 15 × 3 × 5

Non-Broadcastable arrays

A (1d array) : 3

B (1d array) : 4 # trailing dimensions do not match

A (2d array) : 2 × 1

B (3d array) : 8 × 4 × 3

ex:-

a(4x3)

0	0	0
10	10	10
20	20	20
30	30	30

b(4)

1	2	3	4

✗ mismatch of 3 & 4.

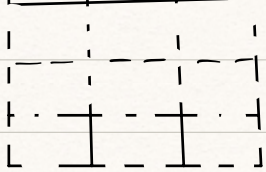
```
a = np.arange(1,10).reshape(3,3)
b = np.array([-1,0,1])  → (3,)  → matches
```

a * b

```
array([[[-1, 0, 3]
        [-4, 0, 6]
        [-7, 0, 9]]])
```

b(3)

-1	0	1



a(3,3)

1	2	3
4	5	6
7	8	9

a/b

```
array([[[-1.0, inf, 3.0]
        [-4.0, inf, 6.0]
        [-7.0, inf, 9.0]]])
```

a*b (3,3)

-1	0	3
-4	0	6
-7	0	9

Jupyter Notebook

when we press shift + tab

it gives information of

functions or attributes

Assignment

Normal sorting shuffles all elements

- sort such a way that depending upon the first column elements sorting, the entire row is moved up or down.

ex:-

$$x = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}$$

if $a_4 > a_7 > a_1$

$$\text{output} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_7 & a_8 & a_9 \\ a_4 & a_5 & a_6 \end{bmatrix}$$ } the whole rows are shuffled

Hint: use argsort( ) and slicing.

ex:- `x = np.array([[5, 6, 2, 7]`
`[1, 7, 3, 8]`
`[2, 5, 8, 9]`
`[0, 5, 7, 6]])`

\# first column elements

`x[:, 0]` ← index of first column elements

↳ array([5, 1, 2, 0])

\# sorting these numbers and returning indexes

```
np.argsort(x[:,0])
↳ array([3,1,2,0])
```

now using integer based indexing to retrieve the values.

```
x[np.argsort(x[:,0])]
↳ array([[0,5,7,6]
         [1,7,3,8]
         [2,5,8,9]
         [5,6,2,7]])
```