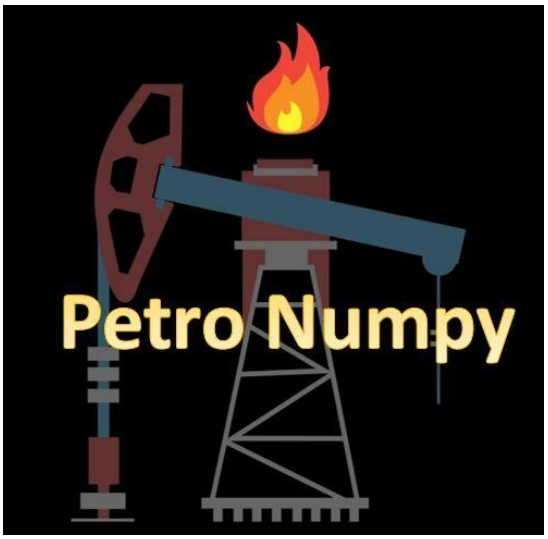
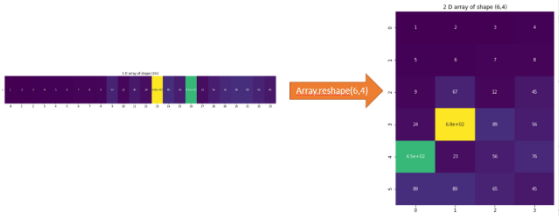
PETROLEUM FROM SCRATCH

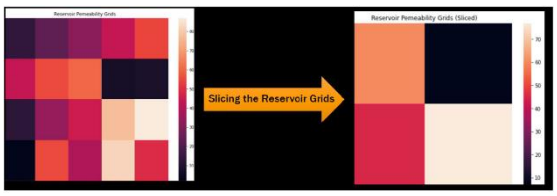
Petro Numpy

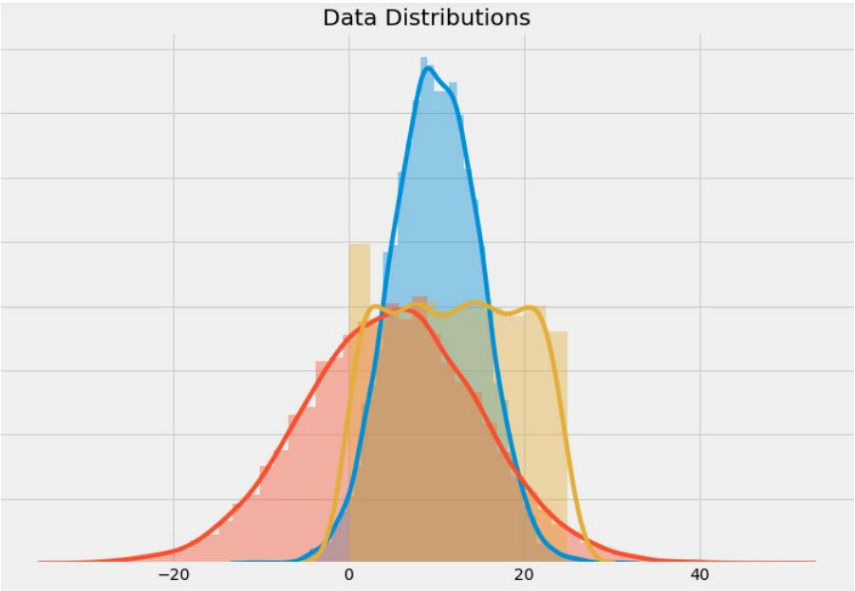
By

Jaiyesh Chahar









Resources

Playlist to follow: https://youtube.com/playlist?list=PLLwtZopJNyqZLJSbXM7k5Z7GYjORn-fyI

All code available at: https://github.com/jaiyesh/15Petro-Numpy-Days

Contents

PETROLEUM FROM SCRATCH

Image Source: https://numpy.org/

Numpy Introduction

> Numpy Introduction:
- Numpy stands for Numerical Python.
- Numpy is the fundamental package for scientific computing in python
- Numpy has functions for working in the domain of linear algebra, Fourier transform, and matrices.
- Numpy is partially written in python, but most of the parts are written in C or C++ that's why Numpy operations are fast and computationally efficient.
- Numpy API is extensively used in Pandas, Scipy, Matplotlib, sklearn and most other data snd scientific python packages.
- Numpy provides an array object 'ndarray' that is 50X faster than python lists.

> Installing NumPy:
You can install NumPy by running the following command:
>>> conda install numpy
or
>>> pip install numpy

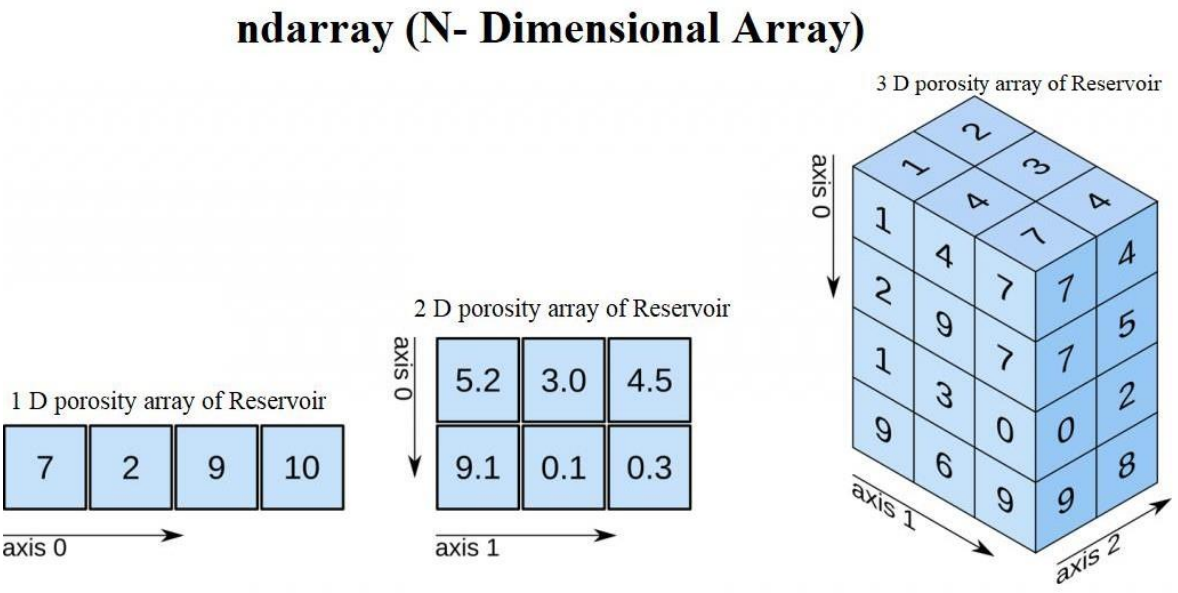
> Importing Numpy in jupyter notebook:
>>> import numpy as np
- NumPy is usually imported under the np alias.

- The source code for NumPy at this github repository: https://github.com/numpy/numpy

- Numpy Introduction jupyter Notebook: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%201.ipynb

- Video "Introduction to Numpy" by Petroleum From Scratch: https://youtu.be/XObOb0deymk

---



Numpy Arrays

> Numpy Arrays:

- The array object in NumPy is called ndarray (n dimensional array).

- An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways.

- NumPy arrays are faster and more compact than Python lists.

- An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. This behaviour is called the locality of reference in computer science. This is the main reason why NumPy is faster than lists. Also, it is optimized to work with the latest CPU architectures.

> How to create a basic array from other data structures:

- To create a NumPy array, use the function np.array().

- All you need to do to create a simple array is pass a list to it, or you can also pass a tuple.

- In jupyter notebook: tuples and lists are converted to arrays, and also converted back. List of lists is used to make high dimensional arrays.

- Numpy Array jupyter Notebook: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%202.ipynb

- Video "Numpy arrays" by Petroleum From Scratch: https://youtu.be/XObOb0deymk

---

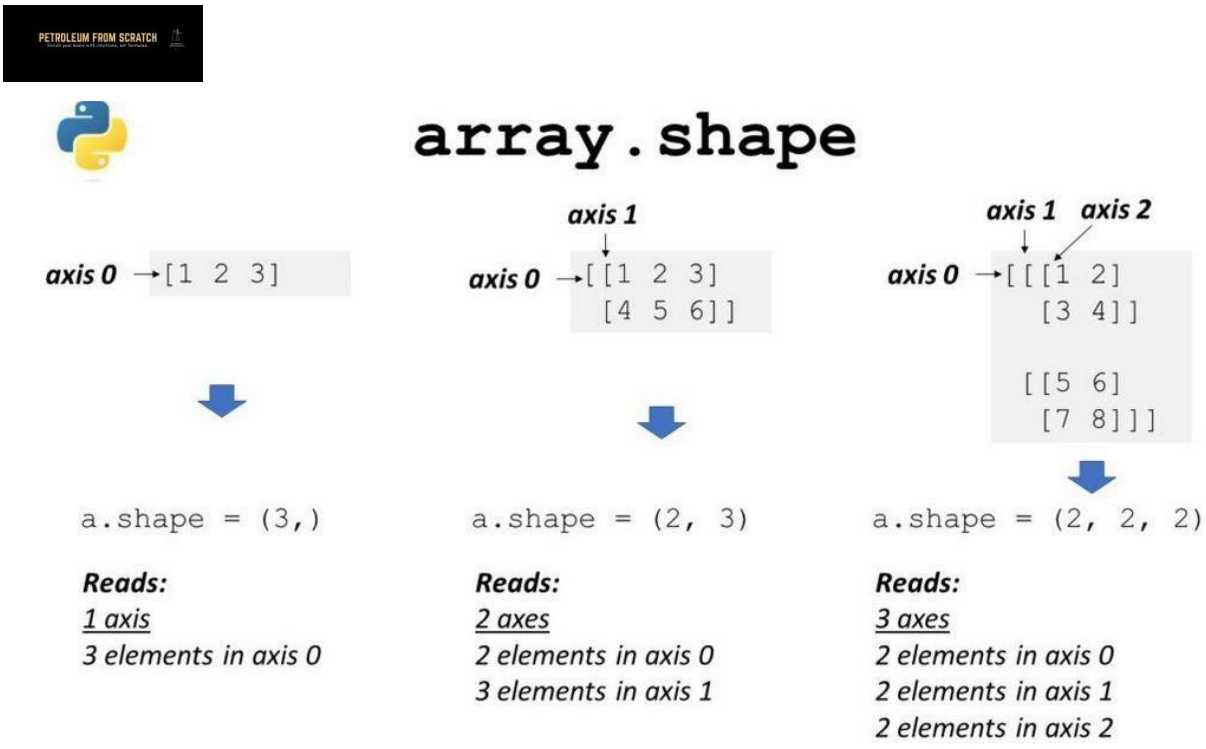


Image Source: https://blog.finxter.com/

Arrays Dimension and Shape

- The dimension of array defines the number of axes. 0 D array will be a single point, 1 D arrays will be like a vector such as [1, 2, 3], a 2D array is a matrix, and so forth.

- While comparing with Coordinate geometry 0D array can be seen as a point, 1 D array as a line, 2D array as rectangle, square etc. 3 D array as cuboid, cube etc.

- 1 D array: Array having 0-D arrays as its elements

- 2 D array: 1D array as its element, a grayscale image pixel values is an example of 2D array.

- 3 D array: 2 D array as its elements, a colour image having 3 channels of Red, green and blue is an example of 3 D array. Or creating batches of time series data.

- High dimensional array: An array can have any number of dimensions.

- When the array is created, you can define the number of dimensions by using the ndmin argument.

- Shape: The shape of the array is a tuple of integers giving the size of the array along each dimension.

>>> Important commands:

- Array.ndim : NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

- ndmin : When the array is created, you can define the number of dimensions by using the ndmin

argument.
Eg. arr = np.array([1, 2, 3, 4], ndmin=5) ==> 5 Dimensional array

- array.shape: >for 2D array it will return a tuple of (number of rows,number of columns)
> for 3D array => (number of 2D arrays present , number of rows in 2D arrays, number of columns in 2D arrays)

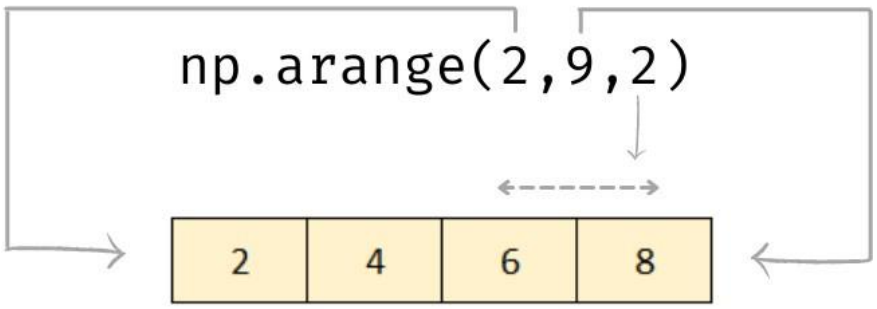
=> 1 D arrays: store reservoir properties when we are considering a linear reservoir, heterogeneity in 1 direction.

=> 2D arrays: Storing reservoir properties considering areal heterogeneity.

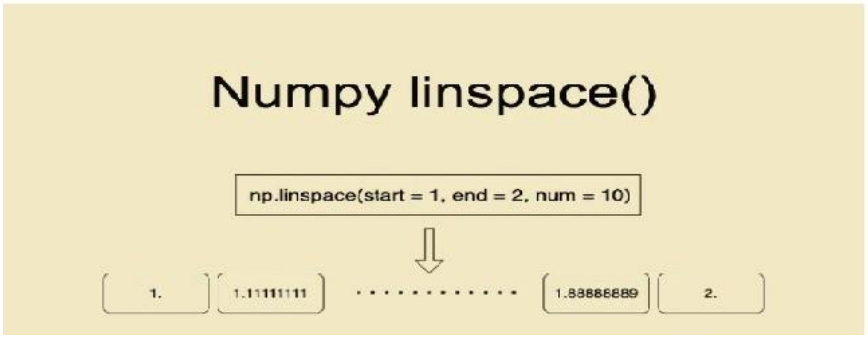
=> 3D arrays: Storing reservoir properties considering volumetric heterogeneity.

- Arrays Dimension and Shape jupyter Notebook: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%203.ipynb

- Discussed the concept in Video "Array's Dimensions and Shape
" by Petroleum From Scratch: https://youtu.be/ElkJgkswraw



Equally spaced values



Arange and Linspace

- Arange and linspace are inbuilt methods for generating 1D array.

- Arange: Creates arrays with regularly incrementing values.

>>> Syntax: numpy.arange(start, stop, step) = > starting point of array, ending point of array and step size.

- Example: Make an array of pressures ranging from 0 to 5000 psi with a difference of 500 psi
>>>> pressures = np.arange(0,5500,500)
>>>> pressures
Output => array([0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000])

- Linspace: Linspace will create arrays with a specified number of elements, and spaced equally between the specified beginning and end values. Creating n data points between two points

>>> Syntax: numpy.linspace(start, stop, number of points) = > starting point of array, ending point of array and number of data points you wants.

- Example: Create saturation array from 0 to 1 having 100 points

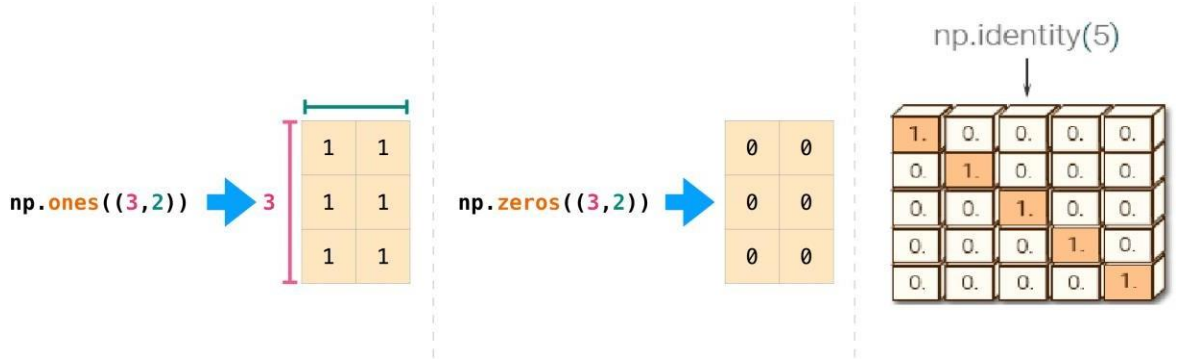
>>>>saturations = np.linspace(0,1,100)
>>>>saturations
Output => array([0., 0.01010101 , ……100 points……, 0.98989899,1.])

-The advantage of linspace function is that you guarantee the number of elements and the starting and end point.

- The arange(start, stop, step) will not include the value stop while in linspace end point is also included.

- Using inbuilt methods for generating arrays jupyter Notebook: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%204.ipynb

- Discussed the concept in Video "Inbuilt Methods for generating Arrays" by Petroleum From Scratch: https://youtu.be/rKtLCXlWegU



Zeroes, Ones, and Identity

> Zeroes, ones and identity:

> Zeroes: For creating an array full of zeros. This zeroes array can be used as a place holder for storing reservoir properties while performing some operation in reservoir simulation
>>> syntax: numpy.zeroes((shape of array))

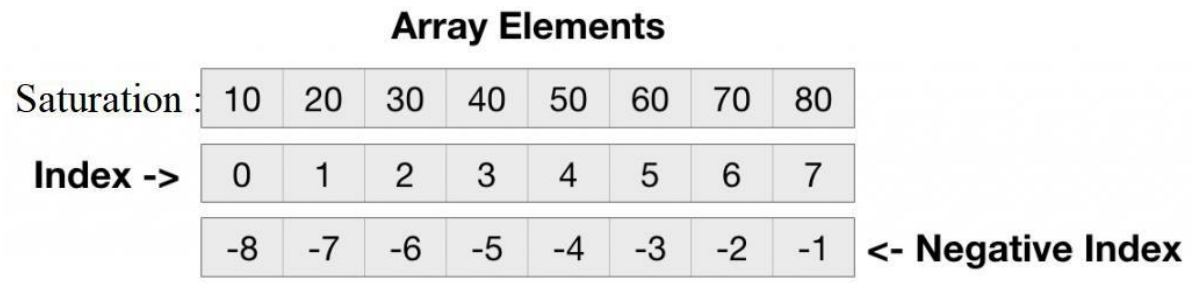
> Ones: Return a new array of given shape filled with ones. This can be used for performing some arithmetic operations on reservoir grid array.
>>> syntax: numpy.ones((shape))

> Identity: Returns an identity matrix, The identity array is a square array with ones on the main diagonal.
>>> syntax: numpy.identity(Number of rows (and columns) in n x n output)

- All these inbuilt array creation techniques are useful in performing various operations or data manipulations in reservoir simulation with python.

- Zeroes, ones and identity matrixes jupyter Notebook: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%205.ipynb

- Discussed the concept in Video " Inbuilt Methods for generating Arrays" by Petroleum From Scratch: https://youtu.be/rKtLCXlWegU



Array Indexing (1-D) arrays

- Accessing array element

- You can access an array element by referring to its index number

- Just like lists, the indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1.

>>>Syntax = array[index]

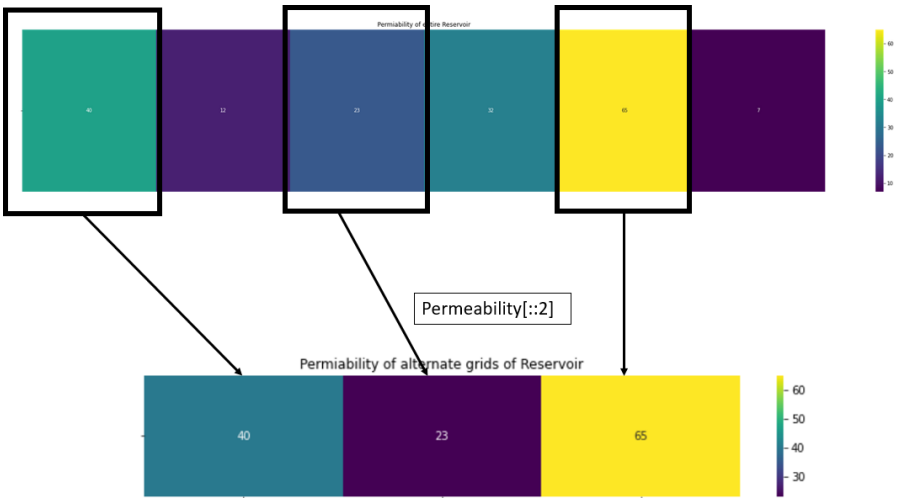
- Negative indexes can be used to access an array from the end.
Last element index = -1

>>>example:
>>>> import numpy as np
>>>> porosity = np.array([0.2,0.4,0.55,0.23,0.877,0.54,0.55])
>>>> print(f'Value of porosity for last index is {porosity[-1]}')
Output : Value of porosity for last index is 0.55

And we will get same output if porosity[6] is used instead of porosity[-1].

- Array Indexing (1 D Arrays) jupyter Notebook: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%206.ipynb

- Discussed the concept in Video "Array Indexing (1-D arrays)"
by Petroleum From Scratch: https://youtu.be/MBVg8L-0Rp0



Array Slicing (1-D) arrays

- Multiple elements can be accessed by using double square brackets.

>>>> Syntax: Array[[indexes]]

- Slicing Array: Slice a subset of array from a superset

>>>> Syntax: Array[start:stop:step size]

- By default starting index is 0, stop index is last element's index and step size is considered as 1

> Example: Consider you have linear reservoir, and we are modelling its heterogeneity as a linear model, so permeability of each grid is:

>>>>> permeability = np.array([10,12,23,12,5,7])

Now I want to use alternate grid's permeability for some calculation, so I can get that using:

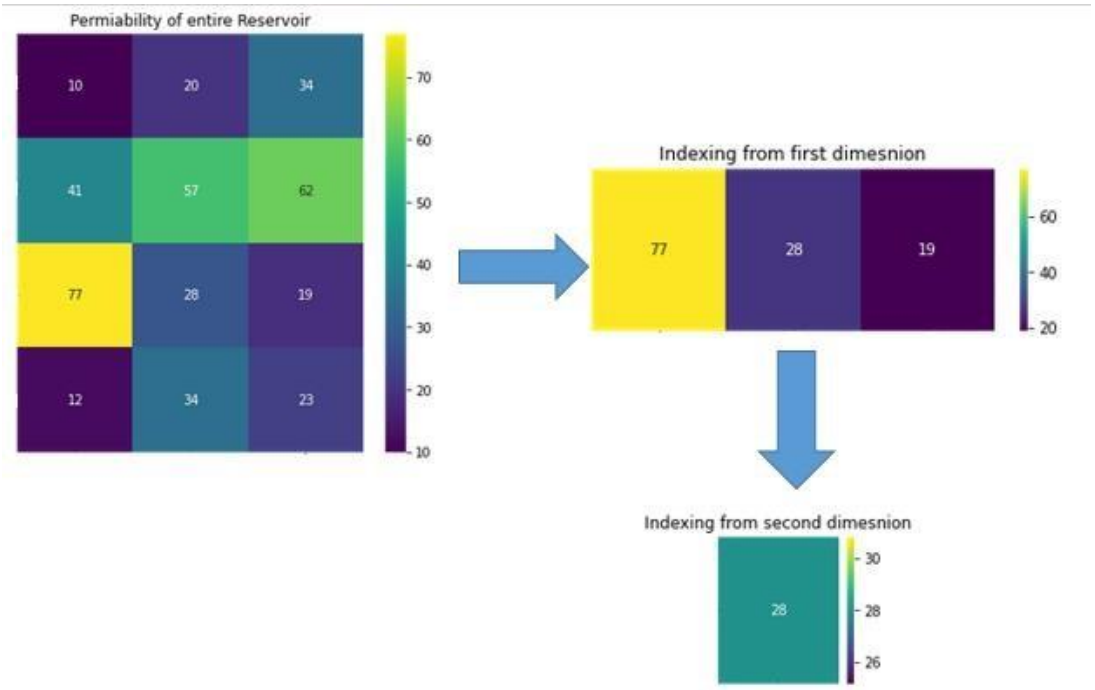
>>>>> permeability[::2]

Output => array([10,23,5])

- More detailed Examples are discussed in video and jupyter notebook.

- Array Slicing (1-D) arrays jupyter Notebook code: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%207.ipynb

- Discussed the concept in Video " Array Slicing (1-D arrays) " by Petroleum From Scratch: https://youtu.be/zfIz5ft1Vg4



Indexing Higher Dimensional arrays

- Indexing in 1D array is discussed earlier.

- Accessing element from higher dimensional array: If indexing is done with fewer indices than dimensions the output will be a subdimensional array

>>> For Example:

>>>> permeability = np.array([[1,2,3],[4,5,6],[7,8,9],[12,34,23]])

>>>> permeability

Output -> array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

[12, 34, 23]])

>>>> permeability[2] => This will give the output of element present in 2nd index of this 2D array and that is => 3r row of array

Output -> array([7, 8, 9])

- Now, if a certain element is need to be accessed so indices in all the dimensions need to be given.

>>>> Example: From permeability array we need to access permeability of grid present in 3rd row, 2nd column, so it can be done in two ways:

1.permeability[2][1] = Indexes 2D array first and get a 1D array and after that indexes this 1D array element

Output = > 8

2. permeability[2,1] = Provide the indices in different dimensions in one square bracket with commas.

Output => 8

- Note that permeability[2][1] == permeability[2,1] though the first case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 1.

For higher dimensional array same thing can be done

>>>> Example:

>>>> perm = np.array([[[1,2,3],[2,3,4]],[[34,45,56],[34,78,23]]])

>>>> perm

Output: array([[[1, 2, 3],

[2, 3, 4]],

[[34, 45, 56],

[34, 78, 23]]])

- For accessing 23 from this, indices of 45 are = 1(for the first dimension, i.e. batches of 2D array), 2(for second dimension, i.e. rows of 2D array), 2(for third diemension, i.e. element of 1D array)

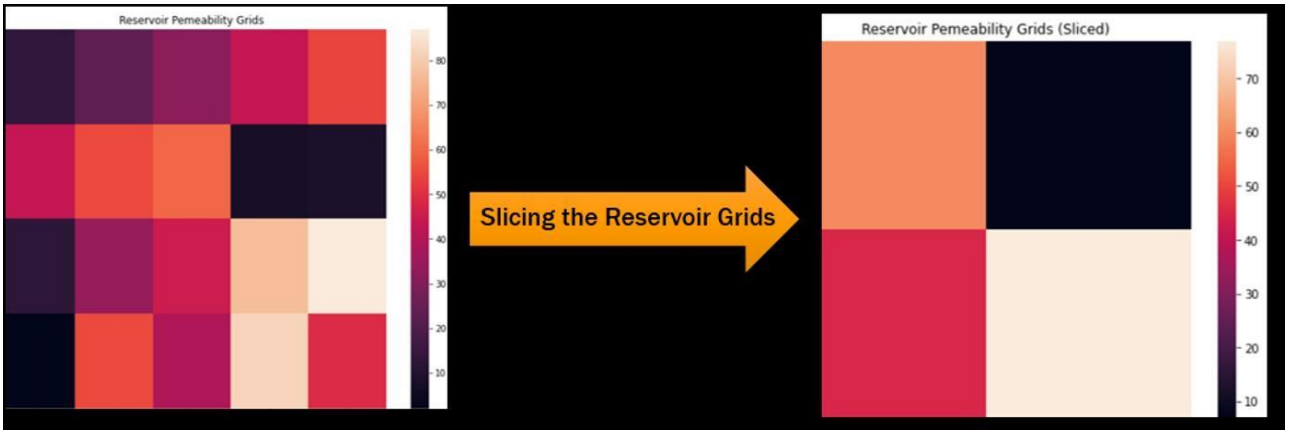
>>>> perm[1,2,2]

Output => 23

- More detailed Examples of high dimensional array indexing are discussed in video and jupyter notebook.

- Indexing for Higher dimensional Arrays jupyter Notebook
code: https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%208.ipynb

- Discussed the concept in Video " Indexing for Higher dimensional Arrays" by Petroleum From Scratch : https://youtu.be/MuOl_rcBrPI



Slicing Higher Dimensional arrays

- Slicing in 1D array is discussed earlier.

- Slicing an array from a higher dimensional array: Slicing will be done in a similar way, but as done in indexing slicing indices in all the dimension can be used.

-> Syntax = array[start index for 1st Dimension: stop index for 1st dim : step size for 1st dim, start 2nd dim: stop 2nd dim : step 2nd dim,… ..... , start nth dim: stop nth dim: step nth dim]

>>> For example:

>>>> perm = np.array([[13,23,32,43,54],[43,55,60,7,8],[12,34,45,77,87],[2,55,39,82,49]])

>>>> perm

Output -> array([[13, 23, 32, 43, 54],

[43, 55, 60, 7, 8],

[12, 34, 45, 77, 87],

[2, 55, 39, 82, 49]])

- Now from the above grid of reservoir, if only 3rd and 4th point from 2nd and 3rd row is needed for certain calculation, then by using array slicing, this subset can be sliced from the given superset.

>>>> perm_sliced = perm[1:3,2:4]

>>>> perm_sliced

Output -> array([[60, 7],

[45, 77]])

- Note: Element at the stop index is not included in sliced array.

Application of array slicing in Machine Learning

1. Split Input and Output Features:

- It is common to split data into input variables (X) and the output variable (y).

- This can be done by slicing all rows and all columns up to, but before the last column, then separately indexing the last column.

- For the input features, all rows and all columns except the last one(y) can be selected by specifying ':' for in the rows index, and :-1 in the columns index.

>>>> X = data[:,:-1] => It takes all rows and all columns except last

- For the output column, all rows again using ':' and index just the last column by specifying the -1 index.

>>>> y = data[:,-1] => It takes all rows from the last column

2. Split Train and Test Rows:

It is common to split a loaded dataset into separate train and test sets.

- Slicing can be used for splitting training and testing data with shuffle of without using scikit learn train test split function.

- This is a splitting of rows where some portion will be used to train the model and the remaining portion will be used to estimate the skill of the trained model.

- This would involve slicing all columns by specifying ':' in the second dimension index. The training dataset would be all rows from the beginning to the split point.

>>>> train = data[:split, :] => all columns data till split row

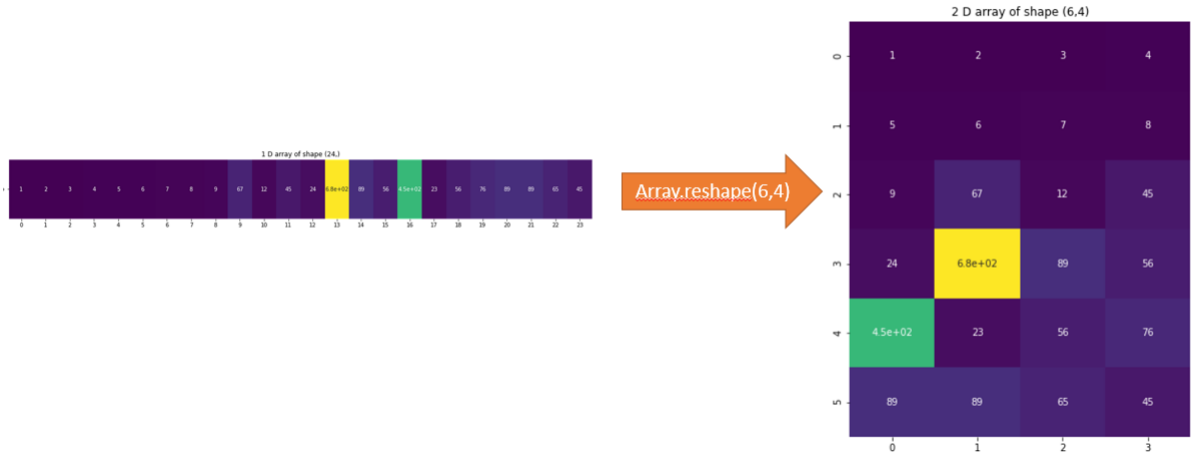
- The test dataset would be all rows starting from the split point to the end of the dimension.

>>>> test = data[split:, :] => all columns data after split row

- More detailed Examples of high dimensional array slicing are discussed in video and jupyter notebook.

- Slicing Higher dimensional Arrays jupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%209.ipynb

- Discussed the concept in Video " Slicing for Higher dimensional Arrays" by Petroleum From Scratch : https://youtu.be/RXznCpJXX9I



Array Reshape

- Shape: The shape of an array is the number of elements in each dimension.

>>> Example:

>>>> perm = arr = np.array([[[1,2,3,4],[4,5,6,5]],[[3,4,5,8],[5,6,7,9]],[[23,34,56,78],[45,67,78,56]]])

- perm is a 3D array containing information about the permeability for each grid in reservoir

>>>> perm.shape

Output -> (3, 2, 4) => 3 elements in 1st dimension (number of 2D arrays batch), 2 elements in 2nd dimension (number of rows in 2D arrays), 4 elements in 3rd dimension (number of columns in 2D arrays)

- Reshaping means changing the shape of an array.

- By reshaping we can add or remove dimensions or change number of elements in each dimension.

- Total number of elements in array remains same.

Reshaping 1D array to a 2D array:

- Lets take an array :

>>> perm =np.array([1,2,3,4,5,6,7,8,9,67,12,45,24,678,89,56,454,23,56,76,89,89,65,45])

Shape => (24,)

- Now if this array can be reshaped in a 2D array, the only condition is that number of elements after reshaping needs to be same as that of initial array.

- So this can be reshaped into 6 * 4 array (Total elements remains 24)

>>> perm2d = perm.reshape(6,4)

>>> perm2d

```
Output -> array([[ 1,  2,  3,  4],
                 [ 5,  6,  7,  8],
                 [ 9, 67, 12, 45],
                 [24, 678, 89, 56],
                 [454, 23, 56, 76],
                 [89, 89, 65, 45]])
```

- In a similar way any n-d array can be reshaped into any n-d array, number of elements remains need to be unchanged.

- Cannot reshape array of size 24 into shape (7,3)

Unknown Dimension

- You are allowed to have one "unknown" dimension.

- Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

- Pass -1 as the value, and NumPy will calculate this number for you.

Example : Lets take the above perm array only.

>>>> perm.reshape(3,-1) => Just number of rows needed is known, numpy will automatically calculate the other dimension (in this case number of columns)

```
Output -> array([[ 1,  2,  3, 4,  5,  6,  7,  8],
                 [ 9, 67, 12, 45, 24, 678, 89, 56],
                 [454, 23, 56, 76, 89, 89, 65, 45]])
```

In a similar way for any dimension:

>>>> perm.reshape(-1,2,3) = > Here rows and columns of 2d array is known, but total number of @d array batches is not known i.e. 1st dimension.

```
Output - >   array([[[ 1,  2,  3],
                     [ 4,  5,  6]],
                    [[ 7,  8,  9],
                     [ 67, 12, 45]],
                    [[ 24, 678, 89],
                     [ 56, 454, 23]],
                    [[ 56, 76, 89],
                     [ 89, 65, 45]]])
```

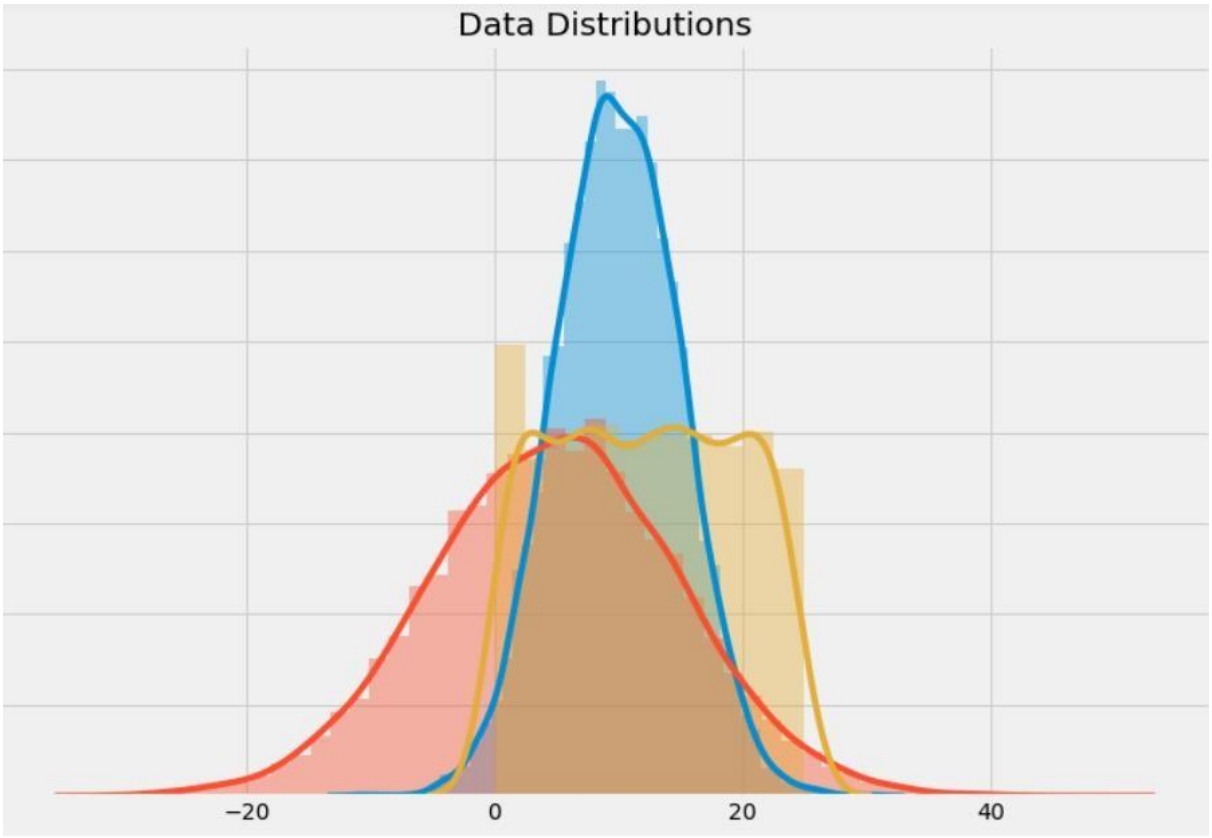
- Here it is seen that numpy automatically handled the unknown dimension.

More detailed Examples are discussed in video and jupyter notebook.

- Reshape jupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%2010.ipynb

- Discussed the concept in Video " Array Reshape" by Petroleum From Scratch : https://youtu.be/JhzA-hNzNQs

---



Data Distribution

- Before going to random number generation, brief knowledge of some data distributions is important.

- Probability Distributions

- A sample of data will form a distribution.

- Distributions are used to solve real life problems using data analysis. For Data Scientist, Distribution is a must known concept.

- Probability will give us mathematical calculations and distributions will help us to visualize data.

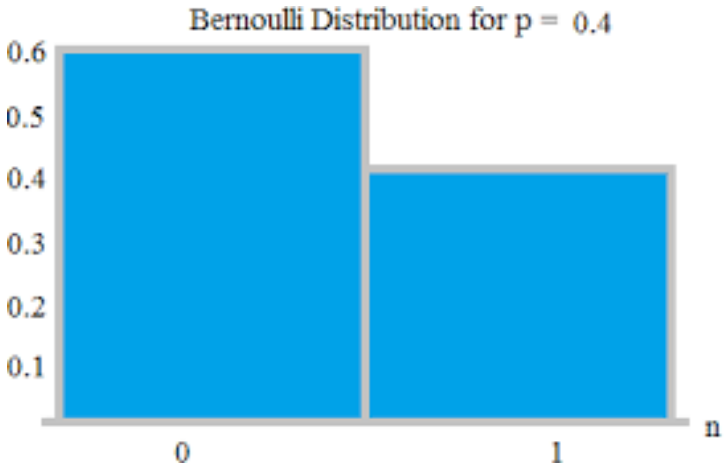
- Some important Probability distributions are:

1. Bernoulli Distribution
2. Uniform Distribution
3. Binomial Distribution
4. Normal Distribution
5. Poisson Distribution
6. Exponential Distribution

In the given document, brief about these distributions is discussed, with probability function, expected value(mean), variance and example for each distribution.

- Brief about these distributions is discussed, with probability function, expected value(mean), variance and example for each distribution.

1. Bernoulli Distribution

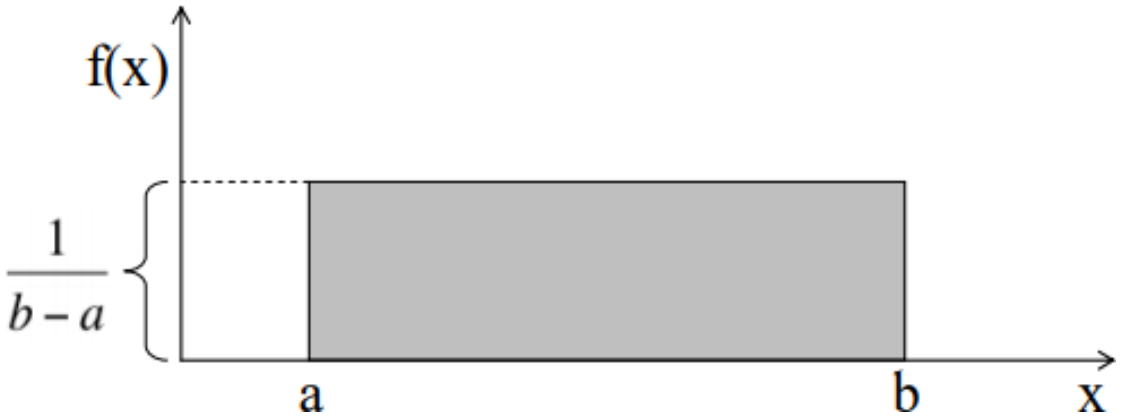


- Two Possible Outcomes: 1(success) and 0(failure)
- Random Variable X which has Bernoulli distribution can take value 1 with probability of success as 'p' and the value 0 with probability of failure as 'q' = (1-p).
- Probability density function:

$$P(x) = \begin{cases} 1-p, & x = 0 \\ p, & x = 1 \end{cases}$$

- Expected Value : E(X) = 1*p + 0*(1-p) = p
- Variance: E(X^2) – [E(X)]^2 = p(1-p)
- Eg. – Tossing Coin, whether a certain workover operation will result in success or failure

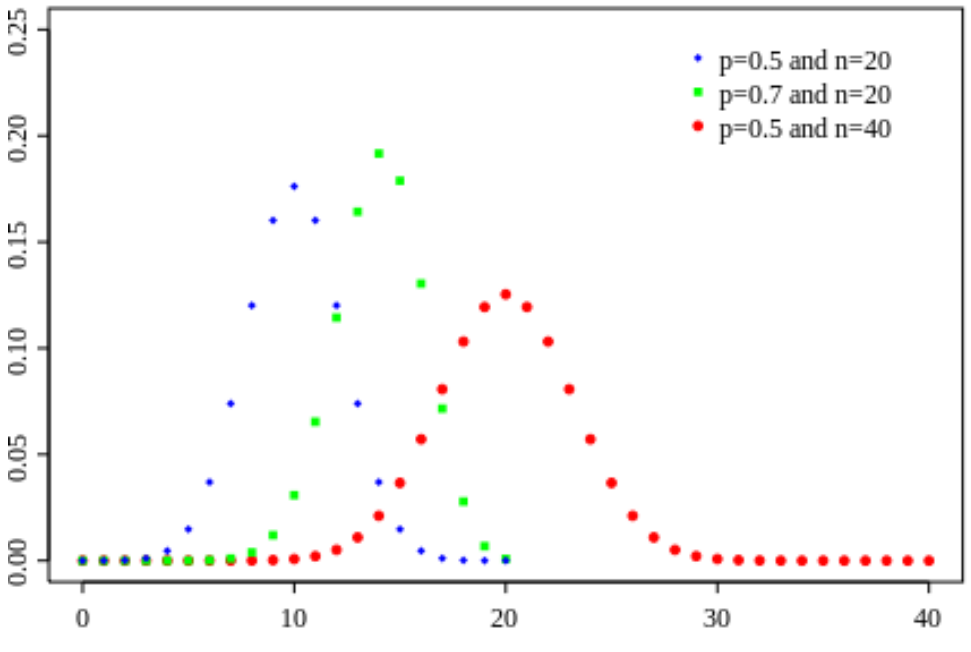
2. Uniform Distribution



- Probability of every outcome is equally likely.
- All the n number of possible outcomes are equally likely.
- Probability density function:

- $f(x) = \frac{1}{(b-a)}$ $\quad -\infty < a \leq x \leq b < -\infty$
- Shape of uniform distribution curve is rectangular.
- Expected Value: E(X) = (a+b)/2
- Variance: V(X) = (b-a)^2/12
- Like all probability distributions for continuous random variables, the area under the graph of a random variable is always equal to 1.
- Drawing a crad from a card deck, rolling a dice are examples of uniform distribution.

3. Binomial Distribution



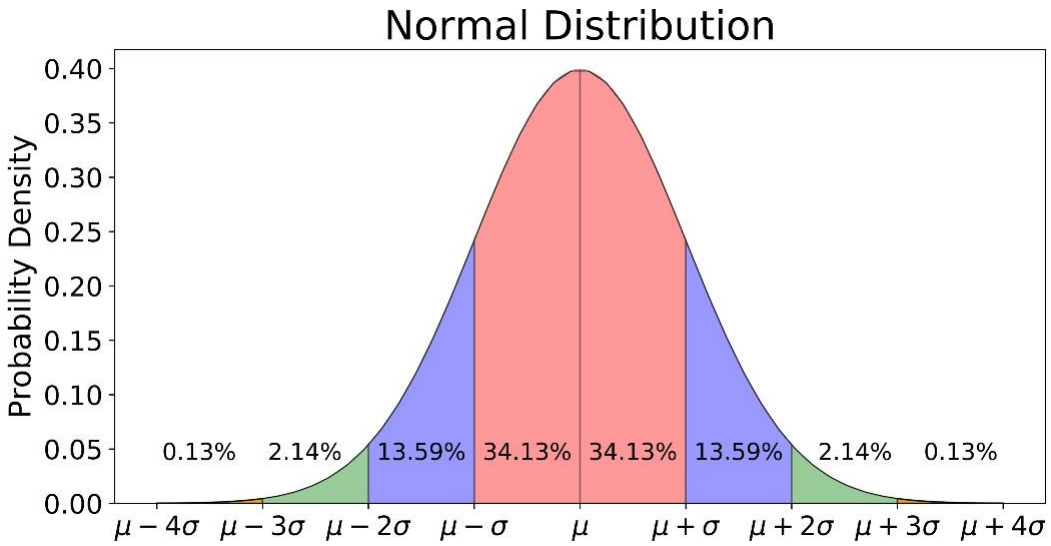
- Only two outcomes are possible, such as success or failure, gain or loss, win or lose in an experiment that is repeated multiple times.
- Each trial or observation is independent.
- Probability of success(p) and failure (1-p) is same for all the trials.
- An experiment with only two possible outcomes repeated n number of times is called binomial.
- Mathematical Representation for binomial:

$$P(x) = \frac{n!}{(n-x)!\,x!} p^x q^{n-x}$$

- Expected Value: E(X) = n*p
- Variance: V(X) = n*p*q

- Example: Tossing coin n times.

4. Normal Distribution

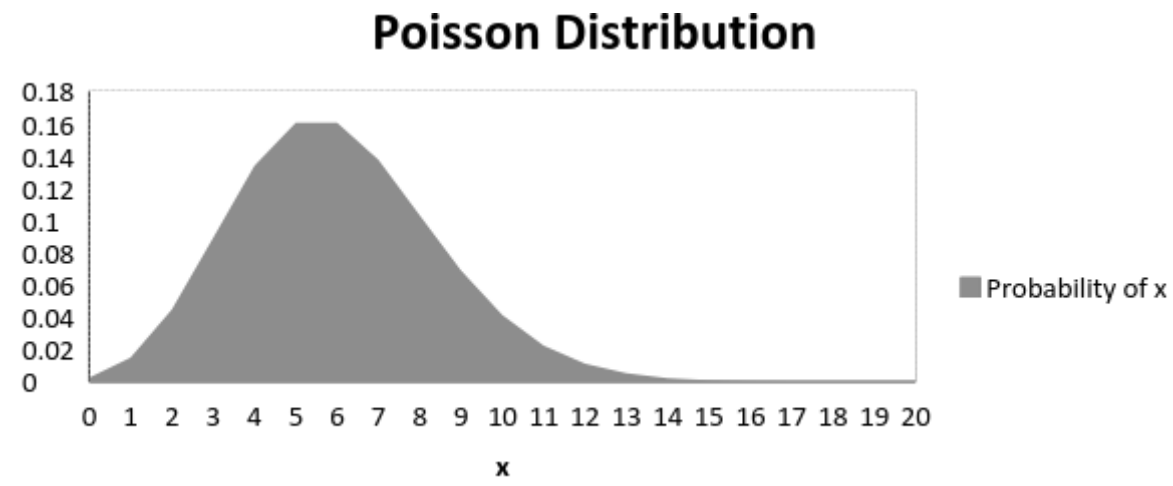


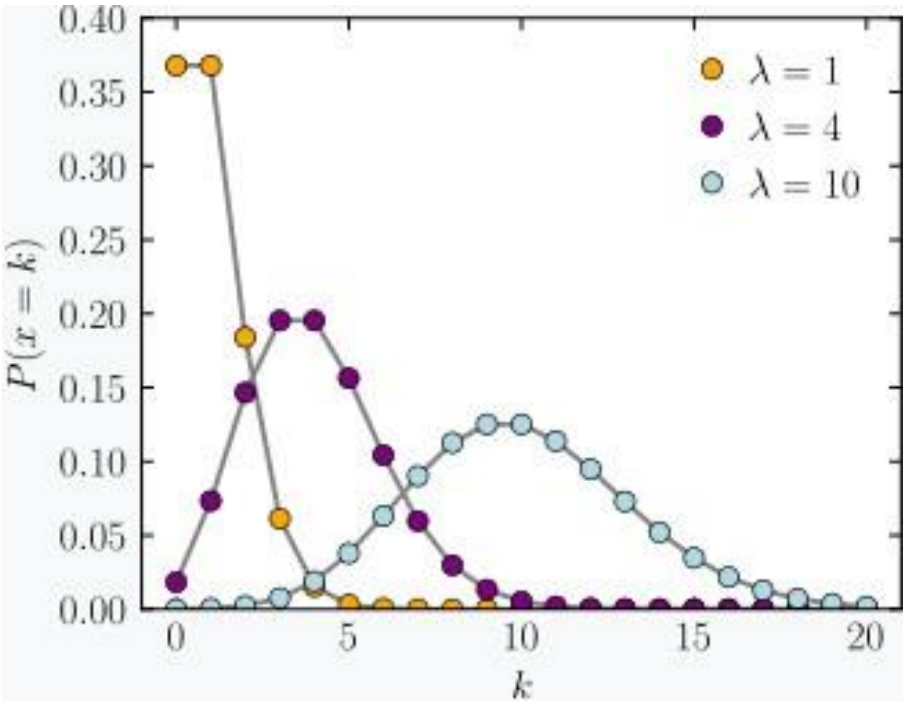
- Most situations in Universe represents Normal Distribution.
- Characteristics of Normal Distribution:
    1. Mean, median and mode coincides
    2. Bell Shaped Curve
    3. Total area under curve is 1
    4. The curve is symmetrical around mean.
- Probability density function is given by:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2\}}$$

- Expected Value: E(X) = μ
- Variance = (Standard Deviation)^2
- The Standard Deviation is a measure of how spread out numbers are.
- About 68% of values drawn from a normal distribution are within one standard deviation σ away from the mean; about 95% of the values lie within two standard deviations; and about 99.7% are within three standard deviations
- E.g.: Heights of people, porosities, permeability, oil saturation, size of things produced by machines, errors in measurements, blood pressure, marks in test etc.
- About 68% of values drawn from a normal distribution are within one standard deviation σ away from the mean; about 95% of the values lie within two standard deviations; and about 99.7% are within three standard deviations

5. Poisson Distribution





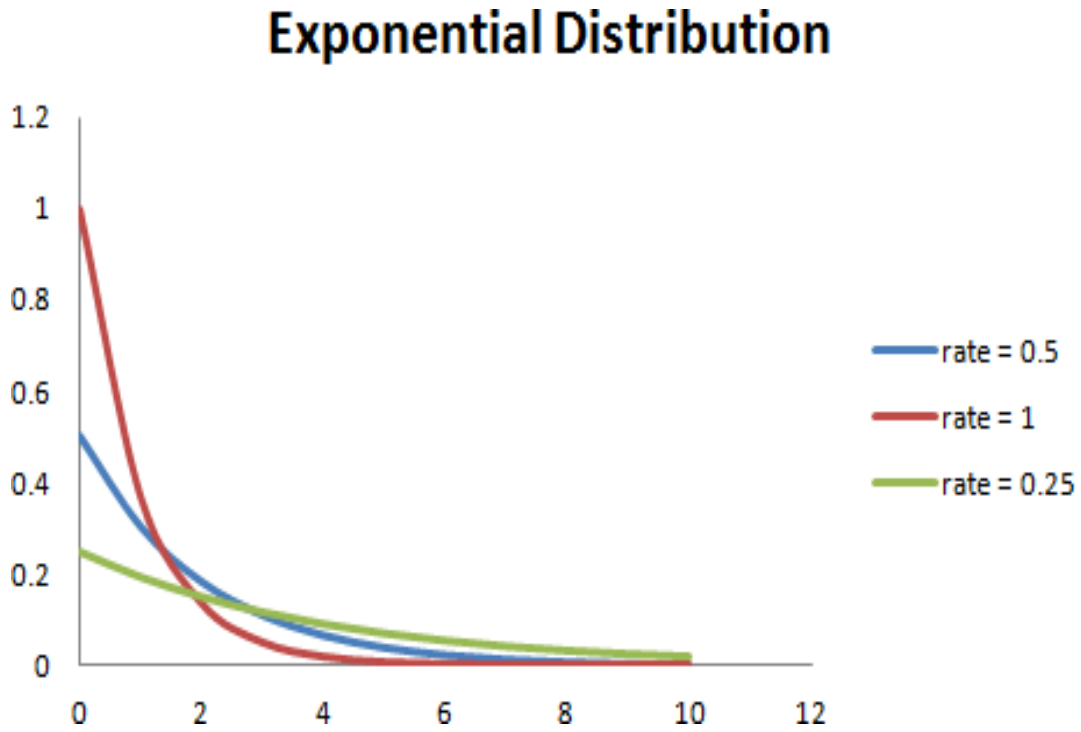
- Applicable in situations where events occurs at random points of time and space but our interest lies only in number of occurrence of events within specified period of time.
- Probability Mass function:

$$P(X = x) = e^{-\mu}\frac{\mu^x}{x!}$$

Where, λ =rate at which an event occurs,
t = length of a time interval,
X = number of events in that time interval
μ = λ times length of time interval t (λ*t)

- Mean = $\mu = (\lambda * t)$
- Variance = $\mu = (\lambda * t)$
- Eg.- If it rains average 15 days a month of August, The number of days it rains in August will have a Poisson Distribution. Most Likely numbers will be 12,13,14,15,16,17,18 days. But 1 and 30 will have probability as low as zero.

6. Exponential Distribution



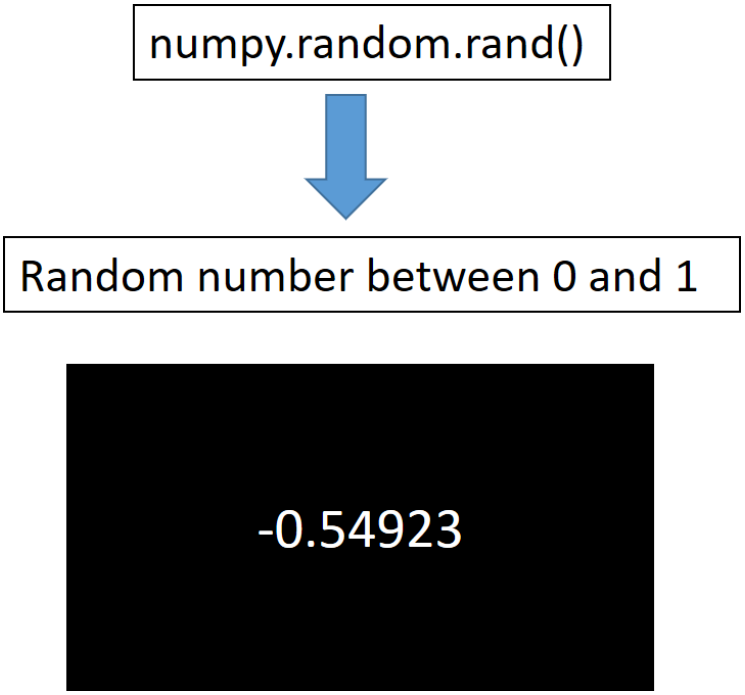
- Describes time between events in a Poisson process.
- E.g. – From last example the time between two rainy days can be modeled with an exponential distribution.
- Models waiting times
- Exponential distribution is widely used for survival analysis. From the expected life of a machine to the expected life of a human, exponential distribution successfully delivers the result.
- Probability Density Function:

$$f(x) = \{ \lambda e^{-\lambda x}, \; x \geq 0$$, $\lambda > 0$ which is the rate of event happening.

- Greater the rate, faster curve drops and if lower the rate, flatter the curve is.
- Expected Value: $E(X) = 1/\lambda$
- Variance: $Var(X) = (1/\lambda)^2$
- Widely used for determining the time elapsed between events.

- Data Distributions jupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%2011.ipynb

---



Random Number Generation Part 1

- Numpy provides various modules for generating random numbers.

- These can be used for generating synthetic data.

- In oil and gas industry there are not many open source data on which a student can practice data science skills. So, it is always good to generate your own data for practicing purpose.

- For example you want to train a model of lithology prediction based on various parameters (porosity, perm. etc). So by using literature you can generate data for these parameters and train your data (All these things are for practice purpose only)

- In the first part, topic of discussion will be Generating totally random numbers, that doesn't follow any probability distributions.

1. Randint

- For generating Random Integers

- Syntax => numpy.random.randint(minimum number, maximum number, shape)

- Example =>

```
>>>> perm = np.random.randint(250,350,(3,4,3))

>>>> perm
```

Output -> array([[[282, 260, 348],

[304, 295, 324],

[290, 273, 314],

[280, 292, 313]],

[[256, 334, 289],

[316, 330, 280],

[258, 299, 253],

[266, 313, 325]],

[[325, 254, 297],

[312, 325, 312],

[337, 308, 311],

[280, 304, 268]]])

2. Rand

- Generating Random Float between 0 and 1

- Syntax => numpy.random.rand(size)

For example:

>>>> saturation = np.random.rand(3,4,2)

>>>> saturation

Output -> array([[[0.12846978, 0.45411542],

[0.93118591, 0.92026651],

[0.94816721, 0.4838314],

[0.98266435, 0.65996996]],

[[0.99711212, 0.67790134],

[0.19576955, 0.09826027],

[0.87453202, 0.92221878],

[0.7664368 , 0.63849025]],

[[0.55540053, 0.30079785],

[0.40914499, 0.39044007],

[0.48366523, 0.58018903],

[0.67436538, 0.57560239]]])

3. Choice

- Generate a random value based on an array of values

>>>> np.random.choice([43,56,76,89],size = (3,5))

Output -> array([[89, 89, 76, 76, 56],

[43, 76, 76, 89, 56],

[56, 76, 76, 43, 43]])

- By using choice, probability for each value can also be specified

- The sum of all probability numbers should be 1.

Example:

>>>> x = np.random.choice([3, 5, 7, 9], p=[1, 0, 0, 0], size=(5,6))

>>>> x

Output-> array([[3, 3, 3, 3, 3, 3],

[3, 3, 3, 3, 3, 3],

[3, 3, 3, 3, 3, 3],

[3, 3, 3, 3, 3, 3],

[3, 3, 3, 3, 3, 3]])

More detailed Examples are discussed in video and jupyter notebook.

- Random Number Generation 1 jupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%2012.ipynb

- Discussed the concept in Video "Random Numbers Generation Part 1" by Petroleum From Scratch : https://youtu.be/Q5bvbBoeSbU

PETROLEUM FROM SCRATCH

Random Number Generation Part 2

Random Number Generation Part 2

Numbers following a certain probability distribution can also be generated using numpy.

- Some of the important data distributions are discussed earlier.

- Data Distributions: List of all possible values, and how often a value occurs, Important in statistics

- Random Distribution : A random distribution is a set of random numbers that follow a certain probability density function.

- Probability Density Function: A function that describes a continuous probability. i.e. probability of all values in an array.

Some of the numpy functions used to generate data following some specific distribution:

1. Normal Distribution

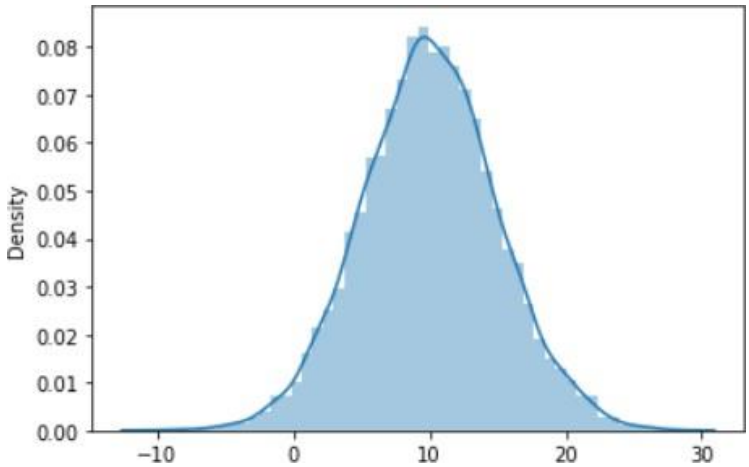
- Syntax: np.random.normal(loc = mean,scale = standard deviation, size)

Example:

>>>> permeability: np.random.normal(loc = 10,scale = 5, size =(10000)) => mean 10, std 5

>>>> sns.distplot(permeability)

Output ->



2. Standard Normal Distribution

- mean = 0

- STD = 1

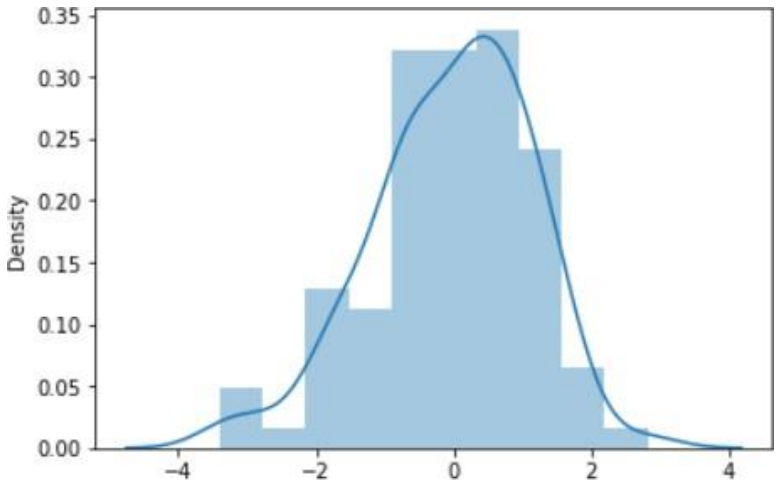
- Syntax-> np.random.normal(size)

Example:

>>>> x = np.random.normal(size = 100)

>>>> sns.distplot(x)

Output->



- np.random.rand(size) can also be used for generating normal distribution

3. Uniform Distribution

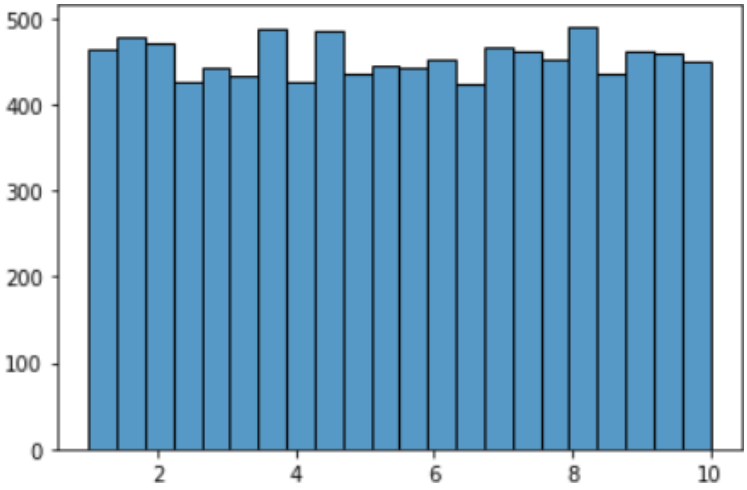
- Syntax: np.random.uniform(minimum, maximum, number of points)

Example:

>>>> perm = np.random.uniform(1,10,10000)

>>>> sns.heatmap(perm)

Output ->



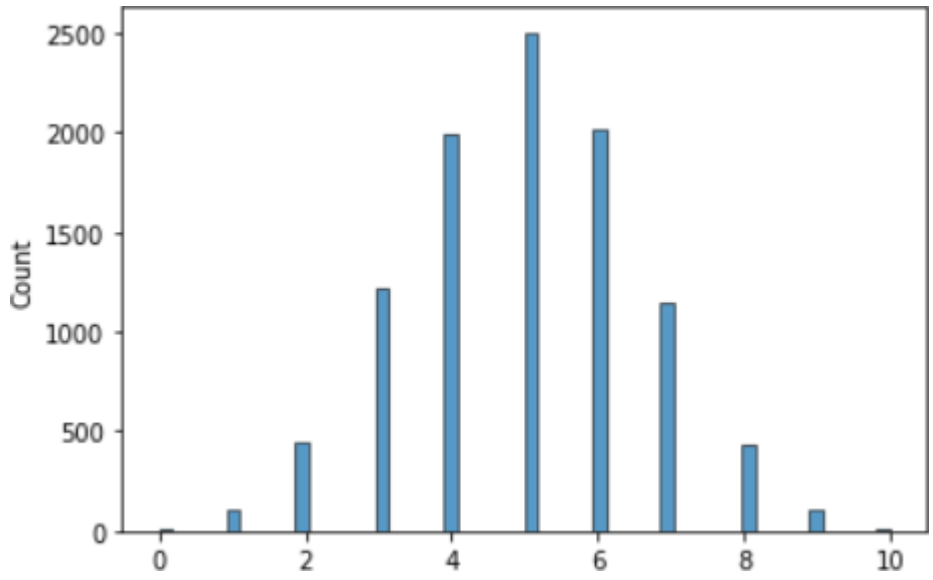
4. Binomial Distribution

- Example :

>>>> x = np.random.binomial(n = 10, p = 0.5,size = 10000)

>>>> sns.distplot(x)

Output->

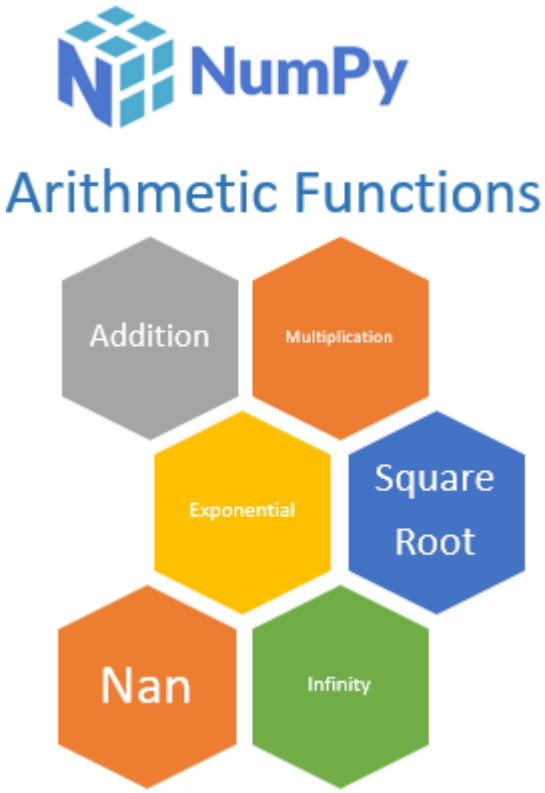


More detailed Examples are discussed in video and jupyter notebook.

- Random Number Generation 2: Random Data Distributionsjupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%2013.ipynb

- Discussed the concept in Video "Random Numbers Generation Part 2" by Petroleum From Scratch : https://youtu.be/74Jt2gK9ETs

---



Numpy Arithmetic Functions

Numpy contains a large number of various mathematical functions:

1. Trignometric
2. Arithmetic
3. Exponential, complex number handling etc.

Addition

- One of benefits of using numpy arrays over lists is that numpy arrays offers arithmetic operations that can not be performed using lists.

Example: Additions of two lists is not possible, using + sign will result in concatenation of lists

>>>> a = [1,2,3,44,5]

>>>> b = [4,5,6,7,8]

>>>> c = a+b

>>>> c

Output -> [1,2,3,44,5,4,5,6,7,8] => Concatenation of lists

For addition of elements => numpy arrays will be helpful.

>>>> ara = np.array(a)

>>>> arb = np.array(b)

>>>> arc = ara + arb

>>>> arc

Output -> array([5, 7, 9, 51, 13])

Multiplication/Division

- Multiplication/Division of 2 lists is also not possible, will result in error.

>>>> a*b

Output -> TypeError: can't multiply sequence by non-int of type 'list'

- But elementwise multiplication can be done using numpy arrays.

>>>> ara*arb

Output-> array([4, 10, 18, 308, 40])

Power

>>>> ara**2 => Square of elements of ara.

Output-> array([1,   4,   9, 1936,   25], dtype=int32)

>>>> ara**arb => element wise exponential power

Output-> array([1,   32,   729, 1450229760,   390625], dtype=int32)

Notes: Operations between arrays of different shapes will result in error

>>>> arrd = np.array([1,23,4])

>>>> ara * arrd => multiplying n array having 5 elements with the array having 3 elemets, will result in an error.

Output-> ValueError: operands could not be broadcast together with shapes (5,) (3,)

Absolute

- Calculate the absolute value element-wise

>>>> ar = np.array([-1,-5,-10,234,45,3])

>>>> np.abs(ar)

Output-> array([1, 5, 10, 234, 45,   3]) => Returns all positive values.

Square Root

- Return the non-negative square-root of an array, element-wise.

>>>> np.sqrt(ara)

Output-> array([1. , 1.41421356, 1.73205081, 6.63324958, 2.23606798])

Infinity and Nan

- Numpy also provides infinite and nan elements so that while performing some calculations there is no error when some number is divided by zero, in core python calculations that results in an error

Example->

>>>> a/0

Output-> TypeError: unsupported operand type(s) for /: 'list' and 'int'- But

- But when numpy is used, there will be an infinity element.

>>>> np.array([1,2,3,4])/0

Output-> array([inf, inf, inf, inf])

>>>> np.inf

Output-> inf

- Similarly an nan element is also present in numpy.

>>>> np.nan

Output-> nan

More detailed Examples are discussed in video and jupyter notebook.

- Numpy Arithmetic Functions jupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%2014.ipynb

- Discussed the concept in Video " Numpy Functions 1: Arithmetic" by Petroleum From Scratch : https://youtu.be/L08CX-XQqmU

Numpy Functions Part 2

Arithmetic functions provided by numpy has been discussed earlier. Apart from arithmetic functions, numpy also provides some other mathematical function. So, following functions will be discussed:

1. Statistical Functions (Mean, Standard Deviation, min, max values)

2. Rounding Functions

3. Trigonometric Functions

4. Logarithmic and Exponential Functions

5. Dot Product

6. Matrix Multiplication

1. Statistical Functions

- For calculating average, standard deviation and other statistical quantities of numpy arrays, numpy provides functions.

Example:-

- minimum value and index corresponding to minimum value element

>>>> ar = np.array([1,2,4,6,7,8,9,-12,-45,-110,556,342])

>>>> ar.min()

Output => -110

- Sometimes index is also used for further calculations, so:

>>>> ar.argmin()

Output => 9 => The element having minimum value is present at 9^{th} index in the array.

- Same operations for maximum value, using .max() and .argmax()

- Sorting: Array can be sorted.

Syntax => array.sort()

>>>> ar.sort()

>>>> ar

Output=> array([-110, -45, -12, 1, 2, 4, 6, 7, 8, 9, 342, 556])

2. Rounding Function

- Numpy provides many ways for rounding decimals:

Example=>

- np.round(array, decimal upto which rounding need to be done)

>>>> ar = np.array([0.4,0.89,0.000035343,1.565677,3.5,88.97567463423,-1.54656778])

>>>> np.round(ar,4)

Output=> array([0.4 , 0.89 , 0. , 1.5657, 3.5 , 88.9757, -1.5466])

- np.floor(array): Rounding to nearest lower integer

>>>> np.floor(ar)

Output => array([0., 0., 0., 1., 3., 88., -2.])

- np.ceil(array): Rounds off decimal to nearest upper integer.

>>>> np.ceil(ar)

Output => array([1., 1., 1., 2., 4., 89., -1.])

3. Trigonometric Functions

- NumPy provides the trigonometric functions sin(), cos() and tan() that take values in radians and produce the corresponding sin, cos and tan values.

Example=>

>>>> ara = np.array([1, 2, 3, 44, 5])

>>>> np.sin(ara)

Output-> array([0.84147098, 0.90929743, 0.14112001, 0.01770193, -0.95892427])

- By default all of the trigonometric functions take radians as parameters but we can convert radians to degrees and vice versa as well in NumPy.

>>>> angles = np.array([90, 180, 270, 360])

>>>> np.deg2rad(angles)

Output-> array([1.57079633, 3.14159265, 4.71238898, 6.28318531])

- Inverse trigonometric functions are also available

>>>> np.rad2deg(np.arcsin([1,-1,0]))

Output-> array([90., -90., 0.])

- Hypot function: That takes the base and perpendicular values and produces hypotenuse

>>>> np.hypot(3,4)

Output-> 5.0

4. Logarithmic and Exponential Functions

- Exp function for euler number:

>>>> np.exp(1)

Output-> 2.718281828459045

>>>> np.exp([1,2,4,6])

Output -> array([2.71828183, 7.3890561 , 54.59815003, 403.42879349])

- Functions for calculating log are also present in numpy:

- Natural Log: log base e

>>>> np.log(np.exp(1))

Output - > 1.0

- Log base 10:

>>>> np.log10(100)

Output -> 2.0

- Log base 2:

>>>> np.log2(8)

Output -> 3

5. Dot Product

- Dot product of two numpy arrays can be done using .dot()

>>>> ara = np.array([1, 2, 3, 44, 5])

>>>> arb = np.array([4, 5, 6, 7, 8])

>>>> ara.dot(arb) = dot product

Output -> 380

- If both arrays a and b are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both arrays a and b are 2-D arrays, it is matrix multiplication, but using matmul or a @ b is preferred.
- If either array a or array b is 0-D (scalar), it is equivalent to multiply and using numpy.multiply(a, b) or a * b is preferred.
- If array a is an N-D array and array b is a 1-D array, it is a sum product over the last axis of a and b.

6. Matrix Multiplication

- When moving to deep learning, matrix multiplication plays a major role, and tensorflow and numpy has a lot of similarities.

- a*b : If a is of m X n and b is of n(this need to be n) X j, then matrix multiplication is of m X j

>>>> ar = np.array([[2,3,4,5],[4,5,6,7]])

>>>> ar.shape

Output -> (2, 4)

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

>>>> art.shape

Output -> (4,2)

>>>> ar@art => Condition satisfies, hence product is possible and matrix of shape 2X2 will be expected as output

Output-> array([[78, 92],

[116, 138]])

More detailed Examples are discussed in video and jupyter notebook.

-Numpy Functions Part 2jupyter Notebook code : https://github.com/jaiyesh/15Petro-Numpy-Days/blob/main/Day%2015.ipynb

- Discussed the concept in Video " Numpy Functions 2: All Functions" by Petroleum From Scratch : https://youtu.be/q3FyV3SJfWU
