By Shreeyansh, source: w3schools.net, tutorialspoint.com Prerequisite: seaborn basics

---

Random Module

- **What is a Random Number?** Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

- **Pseudo Random and True Random.** Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well. If there is a program to generate random number it can be predicted, thus it is not truly random.

Random numbers generated through a generation algorithm are called pseudo random.

- **Can we make truly random numbers?**

Yes. In order to generate a truly random number on our computers we need to get the random data from some outside source. This outside source is generally our keystrokes, mouse movements, data on network etc. We do not need truly random numbers, unless its related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

In this tutorial we will be using pseudo random numbers. Numpy offers the `random` module to work with random numbers.

In [1]:

```
import matplotlib.pyplot as plt
from numpy import random as rnd
import seaborn as sns
```

Generate random integer **from 0 to 100** by using `randint`

In [2]:

```
rnd.randint(100)
```

Out[2]: 21

To get a random float from 0 to 1 use `random`'s `rand` method

In [3]:
```
rnd.rand()
```

Out[3]:
```
0.6502925048592226
```

1. Generate a Random Array

The `randint()` method takes a `size` parameter where you can specify the shape of an array.

In [4]:
```
rnd.randint(100, size = 10)
```

Out[4]:
```
array([66, 16, 52,  8, 18, 97, 94, 14, 28,  3])
```

In [5]:
```
rnd.randint(100, size = (6,7))
```

Out[5]:
```
array([[81, 78,  6, 62, 21, 84, 77],
       [83, 67, 60, 36, 76, 76, 66],
       [80,  6, 76,  1, 74,  9, 34],
       [74, 60, 17,  7, 25, 43, 18],
       [25, 39, 54, 16,  3, 22, 11],
       [84, 33, 42, 10, 70, 27, 51]])
```

In [6]:
```
rnd.randint(100, size = (2,3,3))
```

Out[6]:
```
array([[[ 4, 59, 91],
        [37, 49, 39],
        [ 0, 58,  9]],

       [[11, 21, 41],
        [22, 69, 96],
        [89, 93,  9]]])
```

The `rand()` method also takes the argument `size` to generate random array with float values b/w 0 and 1

In [7]:
```
rnd.rand(3,4)
```

Out[7]:
```
array([[0.94540857, 0.15933393, 0.97607795, 0.2469021 ],
       [0.79462126, 0.89279348, 0.46699591, 0.17506256],
       [0.52929401, 0.78264455, 0.12153618, 0.71408397]])
```

2. Choose Random Number from Given Array

The `choice()` method allows you to generate a random value based on an array of values. It takes an array as a parameter and randomly returns one of the values.

In [8]:
```
rnd.choice(rnd.randint(100))
```

Out[8]:
```
3
```

In [9]:
```
rnd.choice([100,200,300,400,500,600,700])
```

Out[9]: 700

3. Data Distribution

Data Distribution is a list of all possible values, and how often each value occurs. Such lists are important when working with statistics and data science. The random module offer methods that returns randomly generated data distributions.

- **Random Distribution**: A random distribution is a set of random numbers that follow a certain probability density function.
- **Discrete Distribution**:The distribution is defined at separate set of events, e.g. a coin toss's result is discrete as it can be only head or tails whereas height of people is continuous as it can be 170, 170.1, 170.11 and so on.
- **Probability Density Function**: A function that describes a continuous probability. i.e. probability of all values in an array.

We can generate random numbers based on defined probabilities using the `choice()` method of the random module. The `choice()` method allows us to specify the probability for each value. The probability is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur. **The sum of all probabilities should be 1**.

You can return arrays of any shape and size by specifying the `shape` in the size parameter.

In [10]:

```
#Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7 or 9
#The probability for the value to be 3 is set to be 0.1
#The probability for the value to be 5 is set to be 0.3
#The probability for the value to be 7 is set to be 0.6
#The probability for the value to be 9 is set to be 0

x = rnd.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(10))

print(x)
```

```
[5 5 7 3 3 5 7 5 7 7]
```

In [11]:

```
rnd.choice([3,5,7,9], p = [0.1,0.3,0.6,0.0], size=(3,5))
```

Out[11]:

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

4. Random Permutation

A permutation refers to an arrangement of elements. e.g. [3, 2, 1] is a permutation of [1, 2, 3] and vice-versa. The NumPy Random module provides two methods for this: `shuffle()` and `permutation()`. The `permutation()` method generates random permutations of array (Gives a random shuffle out of all possible shuffles). **`shuffle` method makes changes to original array.**

In [12]:

```
x = rnd.randint(100, size = (10))
x
```

Out[12]: array([55, 26, 85, 85, 55, 2, 2, 71, 94, 75])

In [13]:
```python
import numpy as np
np.sort(x)
```

Out[13]: array([2, 2, 26, 55, 55, 71, 75, 85, 85, 94])

In [14]:
```python
rnd.shuffle(x)
x
```

Out[14]: array([75, 2, 94, 85, 85, 71, 26, 2, 55, 55])

In [15]:
```python
rnd.permutation(x)
```

Out[15]: array([85, 55, 94, 71, 85, 2, 2, 55, 75, 26])

Basic Data Distributions

---

1. Normal Distribution

The Normal Distribution is one of the most important distributions. It is also called the Gaussian Distribution after the German mathematician Carl Friedrich Gauss. It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc. Use the `random.normal()` method to get a Normal Data Distribution.

It has three parameters:

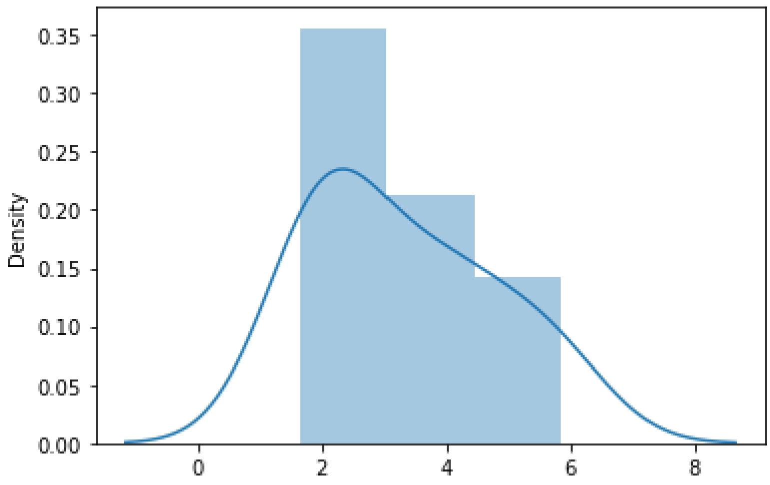
- **`loc`** - (Mean) where the peak of the bell exists.
- **`scale`** - (Standard Deviation) how flat the graph distribution should be.
- **`size`** - The shape of the returned array.

In [16]:
```python
x = rnd.normal(loc = 3, scale = (1.5), size = (2,5))
print(x)
```

```
[[2.09902951 2.09231334 1.64350852 5.31776458 2.37480643]
 [4.19154413 2.1264263  5.85638421 3.43637533 4.01912743]]
```

In [17]:
```python
sns.distplot(x, hist = True)
plt.show()
```

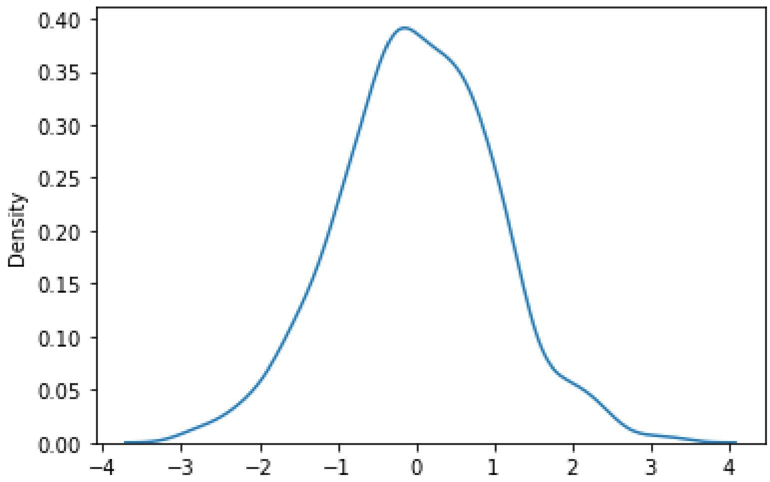
```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarn
ing: `distplot` is a deprecated function and will be removed in a future version. Pl
ease adapt your code to use either `displot` (a figure-level function with similar f
lexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



In [18]:

```
sns.distplot(rnd.normal(size = 1000), hist = False)
plt.show()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
  warnings.warn(msg, FutureWarning)
```



2. Binomial Distribution

Binomial Distribution is a Discrete Distribution. It describes the outcome of binary scenarios, e.g. toss of a coin, it will either be head or tails.

It has three parameters:

- `n` - number of trials.
- `p` - probability of occurence of each trial (e.g. for toss of a coin 0.5 each).
- `size` - The shape of the returned array.

In [19]:

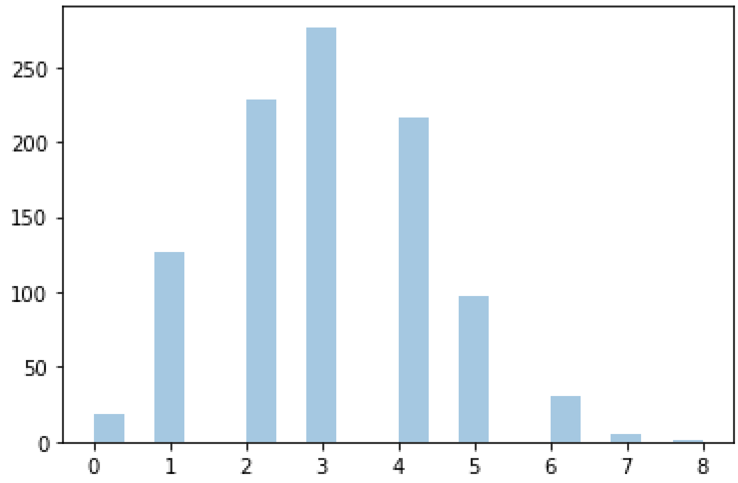
```
x = rnd.binomial(n = 100, p =0.3, size = (2,5))
x
```

Out[19]:

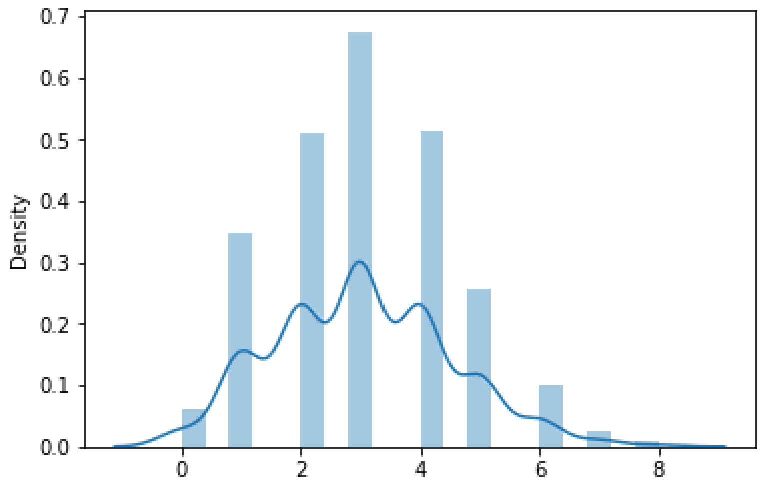
```
array([[34, 22, 33, 35, 33],
       [30, 20, 34, 33, 22]])
```

In [20]:
```
sns.distplot(rnd.binomial(n = 10, p = 0.3, size = 1000), hist = True, kde = False)
plt.show()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



In [21]:
```
sns.distplot(rnd.binomial(n = 10, p = 0.3, size = 1000), hist = True, kde = True)
plt.show()
```



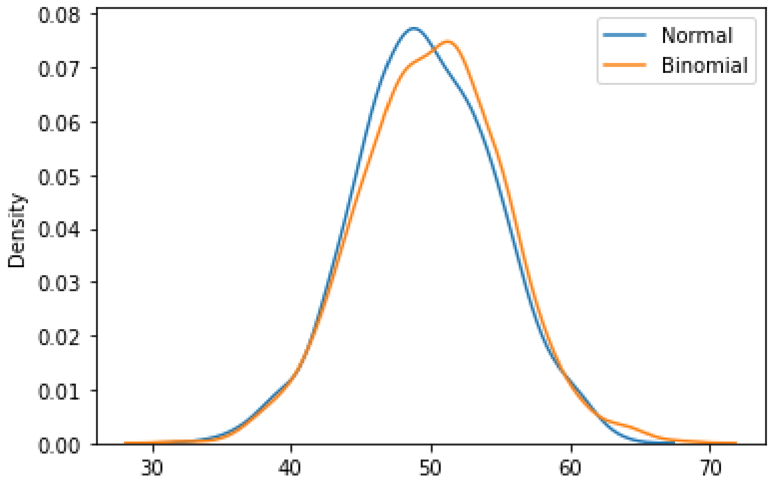
Difference b/w Normal and Binomial Distribution

The main difference is that normal distribution is continous whereas binomial is discrete, but if there are enough data points it will be quite similar to normal distribution with certain `loc` and `scale`.

In [22]:
```
sns.distplot(rnd.normal(loc=50, scale=5, size=1000), hist=False)
sns.distplot(rnd.binomial(n=100, p=0.5, size=1000), hist=False)
plt.legend(["Normal", "Binomial"])

plt.show()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
  warnings.warn(msg, FutureWarning)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarn
ing: `distplot` is a deprecated function and will be removed in a future version. Pl
ease adapt your code to use either `displot` (a figure-level function with similar f
lexibility) or `kdeplot` (an axes-level function for kernel density plots).
  warnings.warn(msg, FutureWarning)
```



3. Poisson Distribution

Poisson Distribution is a Discrete Distribution. It estimates how many times an event can happen in a specified time. e.g. If someone eats twice a day what is probability he will eat thrice?

It has two parameters:

- `lam` - rate or known number of occurences e.g. 2 for above problem.
- `size` - The shape of the returned array.

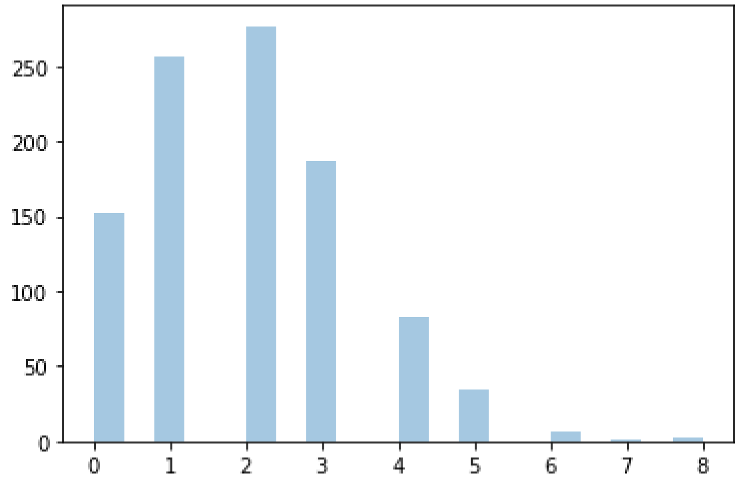
In [23]:
```
#Generate a random 1x10 distribution for occurence 2

x = rnd.poisson(lam = 2, size = 10)
x
```

Out[23]:
```
array([1, 0, 2, 2, 4, 0, 3, 2, 3, 1])
```

In [24]:
```
sns.distplot(rnd.poisson(lam = 2, size = 1000), hist = True, kde = False)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarn
ing: `distplot` is a deprecated function and will be removed in a future version. Pl
ease adapt your code to use either `displot` (a figure-level function with similar f
lexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[24]:
```
<AxesSubplot:>
```



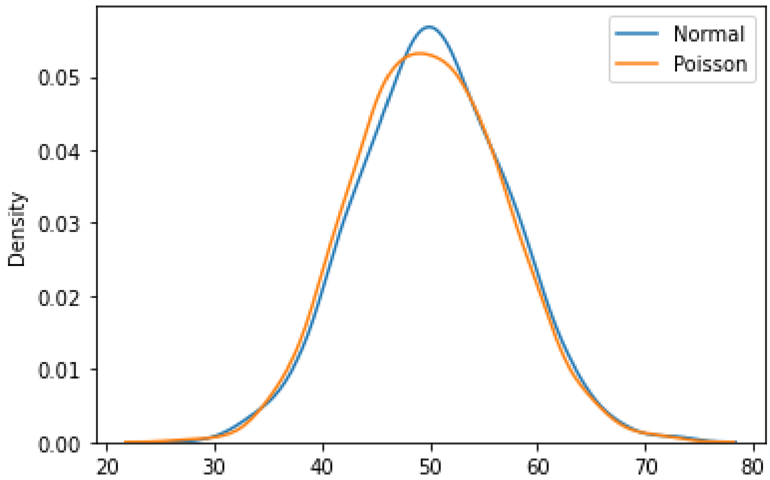
Difference b/w Normal and Poisson Distribution

Normal distribution is continous whereas poisson is discrete. But we can see that similar to binomial for a large enough poisson distribution it will become similar to normal distribution with certain std dev and mean.

In [25]:

```
sns.distplot(rnd.normal(loc = 50, scale = 7, size = 1000), hist = False)
sns.distplot(rnd.poisson(lam = 50, size = 1000), hist = False)

plt.legend(["Normal","Poisson"])
plt.show()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarn
ing: `distplot` is a deprecated function and will be removed in a future version. Pl
ease adapt your code to use either `displot` (a figure-level function with similar f
lexibility) or `kdeplot` (an axes-level function for kernel density plots).
  warnings.warn(msg, FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarn
ing: `distplot` is a deprecated function and will be removed in a future version. Pl
ease adapt your code to use either `displot` (a figure-level function with similar f
lexibility) or `kdeplot` (an axes-level function for kernel density plots).
  warnings.warn(msg, FutureWarning)
```



4. Uniform Distribution

Used to describe probability where every event has equal chances of occuring. E.g. Generation of random numbers.

It has three parameters:

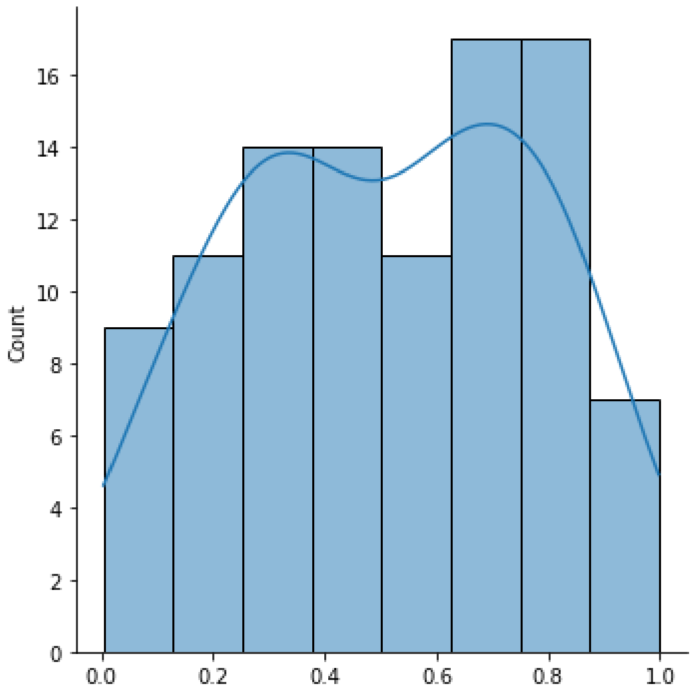
- `a` - lower bound - default 0 .0.
- `b` - upper bound - default 1.0.
- `size` - The shape of the returned array.

In [26]:
```
rnd.uniform(size = (2,3))
```

Out[26]:
```
array([[0.36609814, 0.94897095, 0.27317761],
       [0.92302136, 0.16543711, 0.10628474]])
```

In [27]:
```
#using displot instead of distplot
sns.displot(rnd.uniform(size=100), kde = True)
```

Out[27]:
```
<seaborn.axisgrid.FacetGrid at 0x1e96551a3a0>
```



5. Chi-Squared Distribution

Chi Square distribution is used as a basis to verify the hypothesis.

It has two parameters:

- `df` - (degree of freedom).

- `size` - The shape of the returned array.

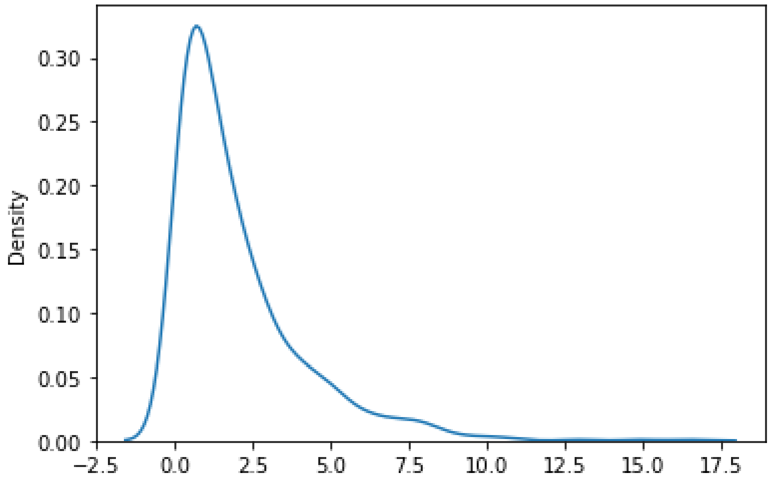
In [28]:
```
rnd.chisquare(df = 2, size = (3,5))
```

Out[28]:
```
array([[3.84001428, 0.61661746, 4.13670803, 1.01972729, 1.55056861],
       [2.30124501, 0.71041148, 0.25871092, 4.57614446, 1.24508486],
       [0.81098575, 3.2293004 , 3.61960754, 4.53723142, 1.52465053]])
```

In [29]:
```
sns.distplot(rnd.chisquare(df=2, size = 1000), hist = False)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).
  warnings.warn(msg, FutureWarning)
```

Out[29]: `<AxesSubplot:ylabel='Density'>`



Numpy uFunc

What are ufuncs?

ufuncs stands for "Universal Functions" and they are NumPy functions that operates on the ndarray object.

Why use ufuncs?

ufuncs are used to implement *vectorization in NumPy which is way faster than iterating over elements.*

They also provide broadcasting and additional methods like reduce, accumulate etc. that are very helpful for computation.

ufuncs also take additional arguments, like:

`where` boolean array or condition defining where the operations should take place.

`dtype` defining the return type of elements.

`out` output array where the return value should be copied.

In [30]:

```
x = np.matrix(rnd.randint(100, size = (4,3)))
y = np.matrix(rnd.randint(100, size = (4,3)))
a = x.reshape(-1)
b = y.reshape(-1)
```

In [31]:

```
print(a)
print(b)
```

```
[[25  1  6 88 35 62 10 99 78  1 73 81]]
[[26 45 75 89 90 39 59 88  0 65 80 55]]
```

1. Basic Arithmeric

In [32]:
```
np.add(a, b)
```

Out[32]:
```
matrix([[ 51,  46,  81, 177, 125, 101,  69, 187,  78,  66, 153, 136]])
```

In [33]:
```
np.subtract(a,b)
```

Out[33]:
```
matrix([[ -1, -44, -69,  -1, -55,  23, -49,  11,  78, -64,  -7,  26]])
```

Element-wise Multiplication

In [34]:
```
np.multiply(a,b)
```

Out[34]:
```
matrix([[ 650,   45,  450, 7832, 3150, 2418,  590, 8712,    0,   65,
         5840, 4455]])
```

Element-wise Division

In [35]:
```
np.divide(a,b)
```

```
<ipython-input-35-c364992e28ce>:1: RuntimeWarning: divide by zero encountered in tru
e_divide
  np.divide(a,b)
```

Out[35]:
```
matrix([[0.96153846, 0.02222222, 0.08      , 0.98876404, 0.38888889,
         1.58974359, 0.16949153, 1.125     ,        inf, 0.01538462,
         0.9125    , 1.47272727]])
```

Element-wise Exponentiation

$(arr_1)^{arr2}$

In [36]:
```
#Raise the valules in arr1 to the power of values in arr2
np.power(a.T, 2*np.ones_like(a.T))
```

Out[36]:
```
matrix([[ 625],
        [   1],
        [  36],
        [7744],
        [1225],
        [3844],
        [ 100],
        [9801],
        [6084],
        [   1],
        [5329],
        [6561]], dtype=int32)
```

- Both the `mod()` and the `remainder()` functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

In [37]:

```
np.mod(a,b)
```

```
<ipython-input-37-2a4695c40758>:1: RuntimeWarning: divide by zero encountered in rem
ainder
  np.mod(a,b)
```

Out[37]:
```
matrix([[25,  1,  6, 88, 35, 23, 10, 11,  0,  1, 73, 26]], dtype=int32)
```

- The `divmod()` function return both the quotient and the the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

In [38]:
```
np.divmod(a, b)
```

```
<ipython-input-38-c60d8d90d7dd>:1: RuntimeWarning: divide by zero encountered in div
mod
  np.divmod(a, b)
```

Out[38]:
```
(matrix([[0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1]], dtype=int32),
 matrix([[25,  1,  6, 88, 35, 23, 10, 11,  0,  1, 73, 26]], dtype=int32))
```

- Both the `absolute()` and the `abs()` functions do the same absolute operation element-wise but we should use `absolute()` to avoid confusion with python's inbuilt `math.abs()`

In [39]:
```
q = np.linspace(0,-100,5)
np.absolute(q)
```

Out[39]:
```
array([  0.,  25.,  50.,  75., 100.])
```

2. Rounding Decimals

There are primarily five ways of rounding off decimals in NumPy:

- truncation
- fix
- rounding
- floor
- ceil

In [40]:
```
j = rnd.normal(size = (4,3))
j
```

Out[40]:
```
array([[-0.65515175, -0.02897767, -0.30441253],
       [ 0.15224227, -0.66593262,  0.43104705],
       [-0.99217391,  0.77876444, -0.29291255],
       [ 1.70462243,  0.90092629,  0.31930816]])
```

In [41]:
```
np.trunc(j)
```

Out[41]:
```
array([[-0., -0., -0.],
       [ 0., -0.,  0.],
       [-0.,  0., -0.],
       [ 1.,  0.,  0.]])
```

In [42]:
```
np.fix(j)    #same as np.truncate(x)
```

```
array([[-0., -0., -0.],
```

Out[42]:
```
       [ 0., -0.,  0.],
       [-0.,  0., -0.],
       [ 1.,  0.,  0.]])
```

In [43]:
```
np.floor(j)
```

Out[43]:
```
array([[-1., -1., -1.],
       [ 0., -1.,  0.],
       [-1.,  0., -1.],
       [ 1.,  0.,  0.]])
```

In [44]:
```
np.ceil(j)
```

Out[44]:
```
array([[-0., -0., -0.],
       [ 1., -0.,  1.],
       [-0.,  1., -0.],
       [ 2.,  1.,  1.]])
```

The `around()` function increments preceding digit or decimal by 1 if >=5 else do nothing.

E.g. round off to 1 decimal point, 3.16666 is 3.2

In [45]:
```
np.around(3.16666,2)
```

Out[45]:
```
3.17
```

3. Logs

If $a^x = m$ then $x = log_a(m)$

In [46]:
```
l = np.abs(rnd.normal(size = (4,3)))
l
```

Out[46]:
```
array([[0.21627925, 2.29586515, 1.03868886],
       [0.78093561, 0.82785583, 1.20252987],
       [0.95994943, 1.90789986, 1.04323822],
       [1.91374691, 0.1216029 , 1.90304291]])
```

By default, `np.log(x)` takes logartim to the base e

In [47]:
```
np.log(l)
```

Out[47]:
```
array([[-1.5311849 ,  0.83110974,  0.03795921],
       [-0.24726257, -0.18891625,  0.18442756],
       [-0.04087467,  0.64600309,  0.04232955],
       [ 0.64906305, -2.10699445,  0.64345414]])
```

Logarithm to base 2 is taken by `np.log2(x)`

In [48]:
```
np.log2(l)
```

Out[48]:
```
array([[-2.20903286,  1.19903791,  0.05476357],
       [-0.35672449, -0.27254854,  0.26607273],
       [-0.05896968,  0.93198545,  0.06106862],
       [ 0.93640005, -3.03975044,  0.92830809]])
```

Logarithm to base 10 is taken by `np.log10(x)`

In [49]:

```
np.log10(l)
```

Out[49]:
```
array([[-0.66498515,  0.36094638,  0.01648548],
       [-0.10738477, -0.08204529,  0.08009587],
       [-0.01775164,  0.28055558,  0.01838349],
       [ 0.2818845 , -0.91505606,  0.27944858]])
```

Log at Any Base

NumPy does not provide any function to take log at any base, so we can use the `frompyfunc()` function along with inbuilt function `math.log()` with two input parameters and one output parameter:

In [50]:
```
from math import log
nplog = np.frompyfunc(log, 2, 1)
nplog(100,15)
```

Out[50]:
```
1.7005483074552052
```

4. Summations

Addition is done between two arguments whereas summation happens over n elements.

In [51]:
```
o = rnd.randint(100, size = (4,2))
k = rnd.randint(100, size = (4,1))
print(o)
print(k)
```

```
[[81 68]
 [29 86]
 [35 43]
 [83 87]]
[[49]
 [ 5]
 [67]
 [70]]
```

In [52]:
```
np.add(o,k) #broadcasting done automatically
```

Out[52]:
```
array([[130, 117],
       [ 34,  91],
       [102, 110],
       [153, 157]])
```

Summation Over Axis

In [53]:
```
x = rnd.randint(100, size = (4,4))
print(x)
```

```
[[ 9 34 67 78]
 [53 98  1 28]
 [20 37 31 19]
 [92 74 44 64]]
```

Add all the elements

In [54]:
```
np.sum(x)
```

Out[54]: 749

Sum along columns : Add *column* elements

$[\,[55 + 98 + 73 + 71]$

$[69 + 93 + 93 + 38]...]$

In [55]:
```
np.sum(x, axis = 1)
```

Out[55]:
```
array([188, 180, 107, 274])
```

Sum along rows : Add *row* elements

$[55 + 69 + 61 + 70\,, 98 + 93 + 31 + 59, \ldots]$

Remember that row vectors and column vector are same in numpy so you can't expect a column vector return when summing over rows

In [56]:
```
np.sum(x, axis = 0)
```

Out[56]:
```
array([174, 243, 143, 189])
```

Cumulative sum

Cummulative sum means partially adding the elements in array.

E.g. The partial sum of $[1, 2, 3, 4]$ would be $[1, 1 + 2, 1 + 2 + 3, 1 + 2 + 3 + 4] = [1, 3, 6, 10]$.

Perfom partial sum with the `cumsum()` function.

In [57]:
```
np.cumsum(x)
```

Out[57]:
```
array([  9,  43, 110, 188, 241, 339, 340, 368, 388, 425, 456, 475, 567,
       641, 685, 749], dtype=int32)
```

In [58]:
```
np.cumsum(x, axis = 0)
```

Out[58]:
```
array([[  9,  34,  67,  78],
       [ 62, 132,  68, 106],
       [ 82, 169,  99, 125],
       [174, 243, 143, 189]], dtype=int32)
```

5. Products

In [59]:
```
x
```

Out[59]:
```
array([[ 9, 34, 67, 78],
       [53, 98,  1, 28],
       [20, 37, 31, 19],
       [92, 74, 44, 64]])
```

`np.prod( )`

Product of all Elements

In [60]:
```
np.prod(x)
```
Out[60]: -993263616

Product along rows

$$[55 \times 69 \times 61 \times 70, 98 \times 93 \times 31 \times 59, \ldots]$$

In [61]:
```
np.prod(x, axis = 0)
```
Out[61]: array([877680, 9123016,   91388, 2655744])

Product along columns

$$[55 \times 98 \times 73 \times 71, 69 \times 93 \times 93 \times 38, \ldots]$$

In [62]:
```
np.prod(x, axis = 1)
```
Out[62]: array([1599156,   145432,   435860, 19171328])

`np.dot( )`

Dot product of two arrays. Specifically,

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

In [63]:
```
np.dot(12,13)
```
Out[63]: 156

In [64]:
```
print(x)
print(y)
```
```
[[ 9 34 67 78]
 [53 98  1 28]
 [20 37 31 19]
 [92 74 44 64]]
[[26 45 75]
 [89 90 39]
 [59 88  0]
 [65 80 55]]
```

Matrix Dot - Product and Cross - Product

Same as `np.matmul(x,y)` . For element-wise product use `np.multiply(x,y)` . For cross product use `np.cross(x,y`

In [65]:
```
np.dot(x,y)
```

Out[65]:
```
matrix([[12283, 15601,  6291],
        [11979, 13533,  9337],
        [ 6877,  8478,  3988],
        [15734, 19792, 13306]])
```

6. Differences

A discrete difference means subtracting two successive elements.

E.g. for $[1, 2, 3, 4]$, the discrete difference would be $[2-1, 3-2, 4-3] = [1, 1, 1]$

To find the discrete difference, use the `diff()` function. We can perform this operation repeatedly by giving parameter n.

E.g. for $[1, 2, 3, 4]$, the discrete difference with `n = 2` would be $[2-1, 3-2, 4-3]$ = $[1, 1, 1]$, then, since `n=2` , we will do it once more, with the new result: $[1-1, 1-1] = [0, 0]$

In [66]:
```
arr = np.array([10, 15, 25, 5])

np.diff(arr, n = 2)
```

Out[66]:
```
array([  5, -30])
```

7. Miscellaneous

LCM and GCD

To find the Lowest Common Multiple of all values in an array, you can use the `reduce()` method. The `reduce()` method will use the ufunc, in this case the `lcm()` function, on each element, and reduce the array by one dimension.

In [67]:
```
t = np.linspace(13,65,5).astype(int)
t
```

Out[67]:
```
array([13, 26, 39, 52, 65])
```

In [68]:
```
np.lcm.reduce(t)
```

Out[68]:
```
780
```

In [69]:
```
np.gcd.reduce(t)
```

Out[69]:
```
13
```

Trigonometry

NumPy provides the ufuncs `sin()` , `cos()` and `tan()` that *take values in radians* and produce the corresponding sin, cos and tan values.

In [70]:
```
arr = np.linspace(1,100,25).reshape(5,5)
arr
```

Out[70]:
```
array([[  1.   ,   5.125,   9.25 ,  13.375,  17.5  ],
       [ 21.625,  25.75 ,  29.875,  34.   ,  38.125],
       [ 42.25 ,  46.375,  50.5  ,  54.625,  58.75 ],
       [ 62.875,  67.   ,  71.125,  75.25 ,  79.375],
       [ 83.5  ,  87.625,  91.75 ,  95.875, 100.   ]])
```

In [71]:
```
np.sin(arr)
```

Out[71]:
```
array([[ 0.84147098, -0.91607692,  0.17388949,  0.72334146, -0.97562601],
       [ 0.35802197,  0.57880195, -0.99955393,  0.52908269,  0.41312976],
       [-0.98698706,  0.68082602,  0.23237376, -0.93838423,  0.80771166],
       [ 0.04313354, -0.85551998,  0.90510688, -0.14768153, -0.74141977],
       [ 0.96945567, -0.33310459, -0.60024952,  0.99840858, -0.50636564]])
```

In [72]:
```
np.cos(arr)
```

Out[72]:
```
array([[ 0.54030231,  0.40100259, -0.98476517,  0.6904905 ,  0.21943996],
       [-0.93371316,  0.81546815,  0.02986535, -0.84857027,  0.91067217],
       [-0.16079968, -0.73244517,  0.97262656, -0.34559374, -0.5895777 ],
       [ 0.99906932, -0.5177698 , -0.42518412,  0.98903497, -0.67104152],
       [-0.2452666 ,  0.94288988, -0.7998128 , -0.05639413,  0.86231887]])
```

`np.rad2deg` converts radian to degree and `np.deg2rad` does vice-versa.

In [73]:
```
np.rad2deg(np.arcsin(1))
```

Out[73]:
```
90.0
```

NumPy provides the `hypot(base, perp)` function that takes the base and perpendicular values and produces hypotenues based on pythagoras theorem.

In [74]:
```
np.hypot(3,4)
```

Out[74]:
```
5.0
```

Set Operations

We can use NumPy's `unique()` method to find unique elements from any array. E.g. create a set array, but remember that the set arrays should only be 1-D arrays.

- **Union**: To find the unique values of two arrays, use the `union1d()` method.
- **Intersection**: To find only the values that are present in both arrays, use the `intersect1d()` method.
- **Difference**: To find only the values in the first set that is NOT present in the seconds set, use the `setdiff1d()` method.
- **Symmetric Difference**: To find only the values that are NOT present in BOTH sets, use the `setxor1d()` method.

Note: the `setxor1d()` method takes an optional argument `assume_unique`, which if set to `True` can speed up computation. *It should always be set to True when dealing with sets.*

In [75]:
```
arr = np.array([1, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

In [76]:

```
np.unique(arr)
```

Out[76]:

```
array([1, 2, 3, 4, 5, 6, 7])
```

In [77]:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([3, 4, 5, 6])

np.union1d(arr1, arr2)
```

Out[77]:

```
array([1, 2, 3, 4, 5, 6])
```

In [78]:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([3, 4, 5, 6])

np.intersect1d(arr1, arr2, assume_unique=True)
```

Out[78]:

```
array([3, 4])
```

In [79]:

```
set1 = np.array([1, 2, 3, 4])
set2 = np.array([3, 4, 5, 6])

np.setdiff1d(set1, set2, assume_unique=True)
```

Out[79]:

```
array([1, 2])
```

In [80]:

```
set1 = np.array([1, 2, 3, 4])
set2 = np.array([3, 4, 5, 6])

np.setxor1d(set1, set2, assume_unique=True)
```

Out[80]:

```
array([1, 2, 5, 6])
```

END
