Statistics Function in Numpy by Mrittika Megaraj

Random Generation Function

random mudule:

- Python random module is a built-in module for random numbers in Python.
- These are sort of fake random numbers which do not possess True randomness.

1. rand():

- Return 1-D array with random values between 0 to 1, can provide (x,y) args to shape the array to form multi-dim array.

In [1]:
```
import random, numpy as np

b = np.random.rand(10)
print(b)

a = np.random.rand(5, 3)
print(a)
```

```
[0.27440777 0.95168032 0.89432766 0.76926979 0.73419359 0.3841947
 0.72743353 0.03001044 0.2308077  0.21702153]
[[0.92052836 0.29667304 0.90877154]
 [0.65645662 0.67756915 0.54664712]
 [0.29797975 0.30479065 0.45901084]
 [0.95911592 0.75059262 0.64985947]
 [0.17566988 0.50843891 0.52691204]]
```

2. random():

- Random numbers from 0 to 1 range.
- Takes only 1 arg, to return only 1-D array.
- Have to use reshape() method, to form multi-dim array.

a = np.random.random(10) print(a) a.reshape(2,5) print(a)

3. ranf():

- Generate random nums between 0 to 1.
- Takes only 1 arg.

In [2]:
```
a = np.random.ranf(5)
print(a)
```

```
[0.75464426 0.37973494 0.93951934 0.17206266 0.09794126]
```

4. randint():

- Generate random int numbers.
- **Syntax -** np.random.randint(low, high[None by default], size[None by default], dtype='i')

In [3]:
```
a = np.random.randint(2,10,10,dtype='i')
print(a)

b = np.random.randint(2,10)
print(b)
```

```
[9 8 5 8 6 3 5 3 3 9]
8
```

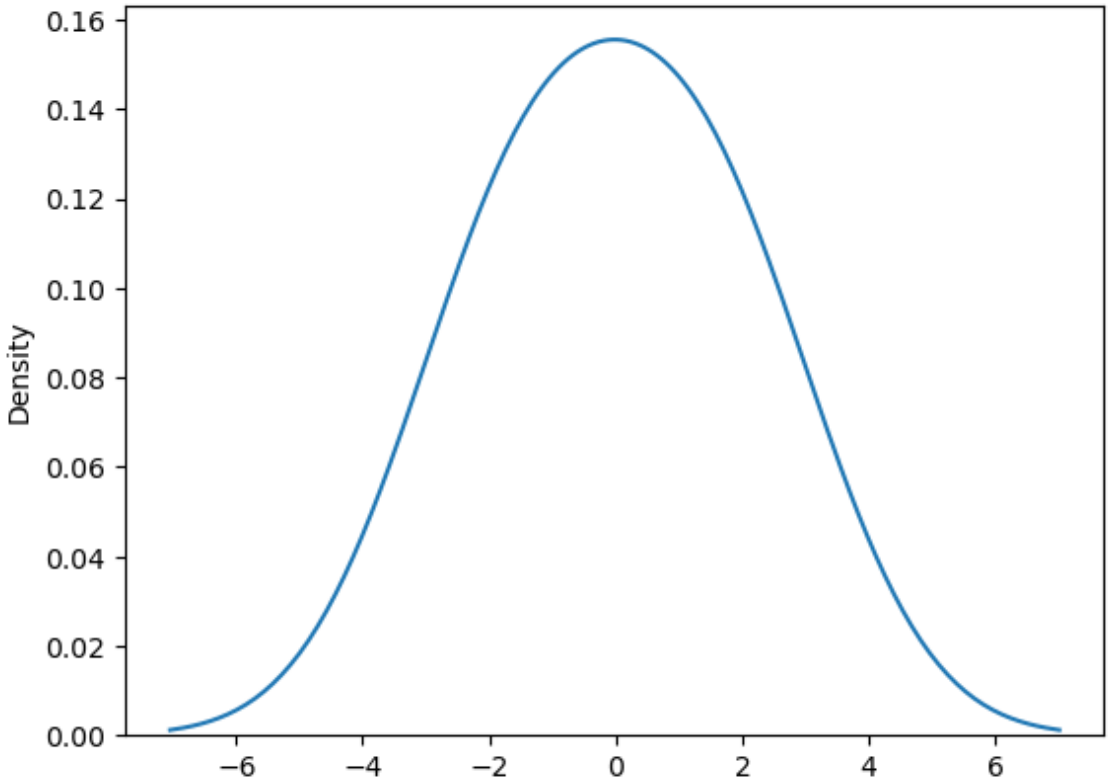
5. randn():

- Generate the normally distributed numbers around (0,0) i.e. Origin co-ordinates.
- **Syntax -** np.random.randn(num)
- Return 1-D array.

In [4]:
```
a = np.random.randn(10)
print(a)

# Plot normally distributed graph
import seaborn as sns
a = np.random.randn(2)
sns.kdeplot(a)
```

```
[-0.21517243 -0.43670005  0.502274    0.34239743  0.14009762  1.88264593
 -0.83057011 -0.69571428 -2.19611707  0.06735687]
```

Out[4]:
```
<Axes: ylabel='Density'>
```



Normal (Gaussian) Distribution

In [5]:
```
from numpy import random

y1 = random.normal(loc=1, scale=2, size=(2, 3))
y2 = random.binomial(n=10, p=0.5, size=10)
y3 = random.poisson(lam=2, size=10)
y4 = random.uniform(size=(2, 3))
y5 = random.logistic(loc=1, scale=2, size=(2, 3))

print(y1)
print("-------")
print(y2)
print("-------")
print(y3)
print("-------")
print(y4)
print("-------")
print(y5)
```

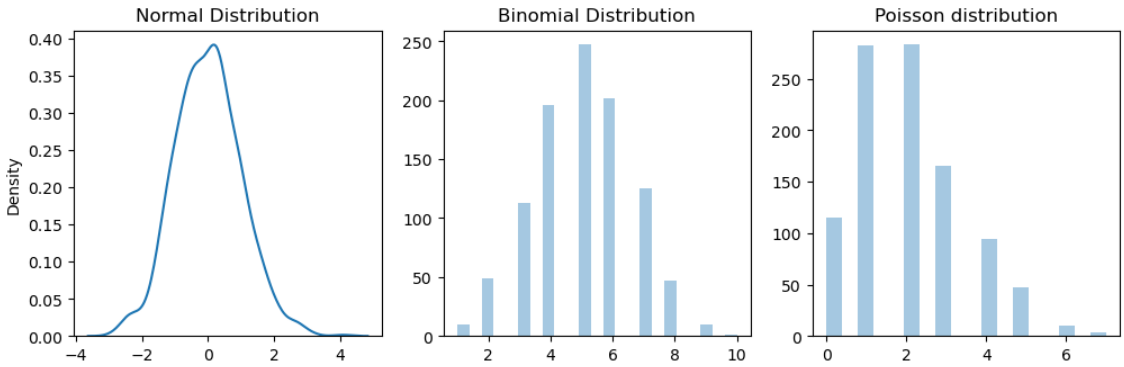
```
[[-1.35639951  2.51663696  1.86624539]
 [-0.51552833 -1.15084862  4.23080859]]
-------
[4 4 2 6 6 4 4 7 4 4]
-------
[1 4 1 0 5 3 2 1 4 3]
-------
[[0.44494749 0.51389724 0.438791  ]
 [0.44485319 0.14351008 0.90352906]]
-------
[[ 1.02292489 -4.84581465  1.12422688]
 [ 3.55466298 -2.01357969 -1.88858759]]
```

Visulaization of Distribution

In [6]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Create subplots to display the distributions using Seaborn
plt.figure(figsize=(12,12))

# Normal distribution
plt.subplot(331)
sns.distplot(random.normal(size=1000), hist=False)
plt.title('Normal Distribution')

# Binomial distribution
plt.subplot(332)
sns.distplot(random.binomial(n=10, p=0.5, size=1000), hist=True, kde=False)
plt.title('Binomial Distribution')

# Poisson distribution
plt.subplot(333)
sns.distplot(random.poisson(lam=2, size=1000), kde=False)
plt.title('Poisson distribution')
```

```
C:\Users\Mrittika\AppData\Local\Temp\ipykernel_24220\1820010395.py:9: Use
rWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function w
ith
similar flexibility) or `kdeplot` (an axes-level function for kernel dens
ity plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751 (http
s://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751)

  sns.distplot(random.normal(size=1000), hist=False)
C:\Users\Mrittika\AppData\Local\Temp\ipykernel_24220\1820010395.py:14: Us
erWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function w
ith
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751 (http
s://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751)

  sns.distplot(random.binomial(n=10, p=0.5, size=1000), hist=True, kde=Fa
lse)
C:\Users\Mrittika\AppData\Local\Temp\ipykernel_24220\1820010395.py:19: Us
erWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function w
ith
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751 (http
s://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751)

  sns.distplot(random.poisson(lam=2, size=1000), kde=False)
```

Out[6]:
```
Text(0.5, 1.0, 'Poisson distribution')
```



Alias: An alias refers to multiple variables that all point to the same underlying NumPy array object. They share the same data in memory. Changes in alias array will affect the original array.

View: The .view() method creates a new array object that looks at the same data as the original array but does not share the same identity. It provides a way to view the data differently or with different data types, but it still operates on the same underlying data.

Copy: A copy is a completely independent duplicate of a NumPy array. It has its own data in memory, and changes made to the copy will not affect the original array, and vice versa.

In [7]:
```
original_arr = np.array([1, 2, 3])

# alias of original array
alias_arr = original_arr

# chaing a value in alias array
alias_arr[0]=10

# you can observe that it will also change the original array
original_arr
```

Out[7]:
```
array([10,  2,  3])
```

In [8]:
```
original_arr = np.array([1, 2, 3])
# Changes to view_arr will affect the original array
view_arr = original_arr.view()

# Modify an element in the view
view_arr[0] = 99

# Check the original array
print(original_arr)
```

```
[99  2  3]
```

In [9]:
```
original_arr = np.array([1, 2, 3])
# Changes to copy_arr won't affect the original array
copy_arr = original_arr.copy()

copy_arr[0] = 100

# Copy doesn't change the original array
print(original_arr)
```

```
[1 2 3]
```

Sorting Functions:

numpy.sort() :

- This function returns a sorted copy of an array.

In [10]:
```
a = np.array([[12, 15], [10, 1]])
```

numpy.argsort() :

- This function returns the indices that would sort an array.

In [11]:
```
a = np.array([9, 3, 1, 7, 4, 3, 6])
b = np.argsort(a)
print('Sorted indices of original array->', b)
```

```
Sorted indices of original array-> [2 1 5 4 6 3 0]
```

numpy.lexsort() :

- This function returns an indirect stable sort using a sequence of keys.

In [12]:
```
# Numpy array created
# First column
a = np.array([9, 3, 1, 3, 4, 3, 6])

# Second column
b = np.array([4, 6, 9, 2, 1, 8, 7])
print('column a, column b')
for (i, j) in zip(a, b):
    print(i, ' ', j)

# Sort by a then by b
ind = np.lexsort((b, a))
print('Sorted indices->', ind)
```

```
column a, column b
9   4
3   6
1   9
3   2
4   1
3   8
6   7
Sorted indices-> [2 3 1 5 4 6 0]
```

numpy.ndarray.sort():

- Sort an array, in-place.

In [13]:
```
arr = np.array([9, 3, 1, 7, 4, 3, 6])
np.ndarray.sort(arr)

print(arr)
```

```
[1 3 3 4 6 7 9]
```

numpy.sort_complex():

- Sort a complex array using the real part first, then the imaginary part.

In [14]:
```
arr = np.array([12+4j, 5+6j, 5+4j, 5+2j, 8+6j, 6+5j])

print(np.sort_complex(arr))
```

```
[ 5.+2.j  5.+4.j  5.+6.j  6.+5.j  8.+6.j 12.+4.j]
```

numpy.partition():

- Return a partitioned copy of an array.

In [15]:
```
arr = np.array([4,3,5,2,6,8,7,9])
kth = 3
print(np.partition(arr,kth ))
```

```
[3 2 4 5 6 7 8 9]
```

Searching

- Searching is an operation or a technique that helps finds the place of a given element or value in the list.
- Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.

numpy.argmax() :

- This function returns indices of the max element of the array in a particular axis.

In [16]:
```
arr = np.arange(12).reshape(3, 4)
print("\nMax element : ", np.argmax(arr))
```

```
Max element :  11
```

numpy.nanargmax() :

- This function returns indices of the max element of the array in a particular axis ignoring NaNs.
- The results cannot be trusted if a slice contains only NaNs and Infs.

In [17]:
```
array = [np.nan, 4, 2, 3, 1]
print("\nMax element : ", np.nanargmax(array))
```

```
Max element :  1
```

In [18]:
```
arr2 = np.array([[np.nan, 4], [1, 3]])
print(("\nIndices of max in array2 : ", np.nanargmax(arr2)))

print(("\nIndices at axis 1 of array2 : ", np.nanargmax(arr2, axis = 1)))
```

```
('\nIndices of max in array2 : ', 1)
('\nIndices at axis 1 of array2 : ', array([1, 1], dtype=int64))
```

numpy.argmin() :

- This function returns the indices of the minimum values along an axis.

In [19]:
```
array = np.arange(8)

print("\nIndices of min element : ", np.argmin(array, axis=0))
```

```
Indices of min element :  0
```

Counting

numpy.count_nonzero() :

- Counts the number of non-zero values in the array.

In [20]:
```
a = np.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]])
b = np.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]], axis=0)

print("Number of nonzero values is :",a)
print("Number of nonzero values is :",b)
```

```
Number of nonzero values is : 5
Number of nonzero values is : [1 1 1 1 1]
```

numpy.nonzero():

- Return the indices of the elements that are non-zero.

In [21]:
```
a = np.array([[0,1,7,0,0],[3,0,0,2,19]])
print(np.nonzero(a))
```

```
(array([0, 0, 1, 1, 1], dtype=int64), array([1, 2, 0, 3, 4], dtype=int6
4))
```

numpy.flatnonzero():

- Return indices that are non-zero in the flattened version of a.

In [22]:
```
a = np.array([[0,1,7,0,0],[3,0,0,2,19]])
print(np.flatnonzero(a))
```

```
[1 2 5 8 9]
```

numpy.where() function:

- This function is used to return the indices of all the elements which satisfies a particular condition.

In [23]:
```
arr = np.array([[34, 12, 56], [5, 8, 3], [33, 77, 5]])
print(np.where(arr>10))

arr = np.array([34, 12, 56, 5, 8, 3, 33, 77, 5])
print(np.where(arr>10))
```

```
(array([0, 0, 0, 2, 2], dtype=int64), array([0, 1, 2, 0, 1], dtype=int6
4))
(array([0, 1, 2, 6, 7], dtype=int64),)
```

numpy.extract():

- The extract() function returns the elements satisfying any condition.

In [24]:
```
x = np.arange(9.).reshape(3, 3)
condition = np.mod(x,2) == 0

print(np.extract(condition, x))
```

```
[0. 2. 4. 6. 8.]
```

Numpy for Linear Algebra

Complex Matrix Operations

In [25]:
```
A = np.array([[1, 2], [3, 4]])

# Calculate the inverse of A
A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

In [26]:
```
A = np.array([[2, -1], [1, 1]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
print("eigenvalues:",eigenvalues)
print("eigenvectors:",eigenvectors)
```

```
eigenvalues: [1.5+0.8660254j 1.5-0.8660254j]
eigenvectors: [[0.70710678+0.j         0.70710678-0.j        ]
 [0.35355339-0.61237244j 0.35355339+0.61237244j]]
```

Advanced Numpy Techniques

Masked Arrays

Masked arrays in NumPy allow you to work with data where certain elements are invalid or missing. A mask is a Boolean array that indicates which elements should be considered valid and which should be masked (invalid or missing).

Masked arrays enable you to perform operations on valid data while ignoring the masked element

In [27]:
```
import numpy.ma as ma

# Temperature dataset with missing values (-999 represents missing values)
temperatures = np.array([22.5, 23.0, -999, 24.5, -999, 26.0, 27.2, -999, 28

# Calculate the mean temperature without handling missing values
mean_temperature = np.mean(temperatures)

# Print the result = -316.14
print("Mean Temperature (without handling missing values):", mean_temperatu


# Create a mask for missing values (-999)
mask = (temperatures == -999)

# Create a masked array
masked_temperatures = ma.masked_array(temperatures, mask=mask)

# Calculate the mean temperature (excluding missing values)
mean_temperature = ma.mean(masked_temperatures)

# Print the result = 25.28
print("Mean Temperature (excluding missing values):", mean_temperature)
```

```
Mean Temperature (without handling missing values): -316.14444444444445
Mean Temperature (excluding missing values): 25.28333333333333
```

Structured Arrays

In [28]:
```
# Define data types for fields
dt = np.dtype([('name', 'S20'), ('age', int), ('salary', float)])

# Create a structured array
employees = np.array([('Alice', 30, 50000.0), ('Bob', 25, 60000.0)], dtype=

# Access the 'name' field of the first employee
print(employees['name'][0])

# Access the 'age' field of all employees
print(employees['age'])
```

```
b'Alice'
[30 25]
```

In [ ]: