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.

```
import random, numpy as np

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

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

```
[6.32617217e-02 8.99412243e-01 9.68521077e-04 9.68173707e-01
 9.99777108e-01 5.16934586e-01 3.01408888e-01 3.65351179e-01
 6.21882503e-01 2.26786662e-01]
[[0.91040801 0.68845525 0.67057286]
 [0.50112476 0.85140007 0.78679905]
 [0.5177447  0.61279656 0.07217415]
 [0.52880316 0.49137718 0.6646854 ]
 [0.97022214 0.59268689 0.05259058]]
```

> **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.

```
a = np.random.ranf(5)
print(a)
```

```
[0.50326041 0.97257709 0.22953519 0.65575433 0.4971661 ]
```

> **4. randint():**
>
> - Generate random int numbers.

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

```
a = np.random.randint(2,10,10,dtype='i')
print(a)

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

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

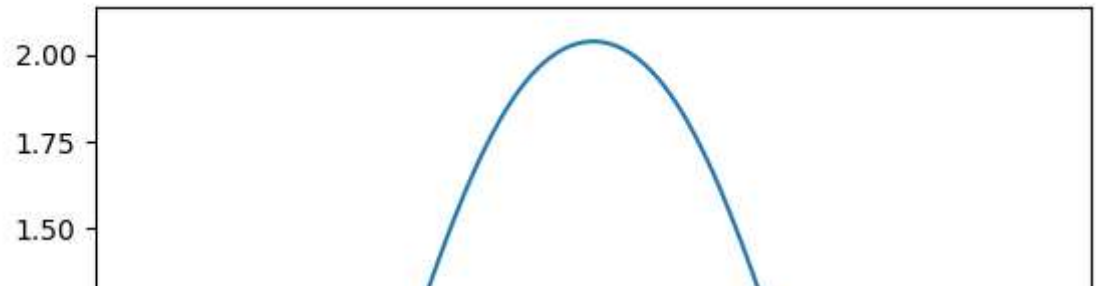
> **5. randn():**
>
> - Generate the normally distributed numbers around (0,0) i.e. Origin co-ordinates.

- **Syntax -** np.random.randn(num)
- Return 1-D array.

```
a = np.random.randn(10)
print(a)

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

```
[-1.6220184   0.19871897 -0.45786781 -0.19958471 -0.77432159 -1.28661937
  0.21046636  0.92348633 -1.2843993  -0.40684533]
<Axes: ylabel='Density'>
```



Sorting Functions:

> **numpy.sort() :**

- This function returns a sorted copy of an array.


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

> **numpy.argsort() :**

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

```
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.

```
# 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.

```
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.

```
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.

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

print(np.partition(arr,2))
```

```
[2 3 4 5 6 8 7 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.

```
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.

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

```
Max element :  1
```

```
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]))
```

> **numpy.argmin() :**

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

```
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.

```
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.

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

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

> **numpy.flatnonzero():**

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

```
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.

```
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]), array([0, 1, 2, 0, 1]))
(array([0, 1, 2, 6, 7]),)
```

numpy.extract():

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

```
x = np.arange(9.).reshape(3, 3)
condition = np.mod(x,2) == 0

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

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

✓ 0s completed at 20:05