Numpy commands

July 18, 2019

0.1 Numpy Commands

This file consists of useful Numpy commands from Random,linear algebra libraries and also some other functions

[132]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.Random library

[133]:
```
np.random.rand(3,2) #create a 3x2 array of random values between 0 and 1
```

[133]:
```
array([[0.38734262, 0.65746878],
       [0.8160697 , 0.90042571],
       [0.65609903, 0.62370866]])
```

[134]:
```
np.random.randn(3,3) #creates a random 3x3 array of values from standard normal
 ↪distribution
```

[134]:
```
array([[ 0.10549219,  0.79660127,  0.12826328],
       [ 1.79133473,  1.5245318 ,  0.81386083],
       [ 0.59409159,  1.51377016, -0.49635222]])
```

[135]:
```
np.random.randint(1,5,size=(2,2)) #create array of size 10 with values between
 ↪1(inclusive) to 5(exclusive)
```

[135]:
```
array([[1, 2],
       [3, 2]])
```

[136]:
```
np.random.random() #returns a random sample between 0.0(exclusive) and 1.
 ↪0(inclusive)
np.random.sample() #returns a random sample between 0.0(exclusive) and 1.
 ↪0(inclusive)
```

[136]:
```
0.31876138609234017
```

[137]:
```
df = np.array([22,34,56])
np.random.choice(df,1,p=[0.1,0.6,0.3])  #choose 1 number from df where p is
 ↪probabilites of there occurences
```

[137]:
```
array([56])
```

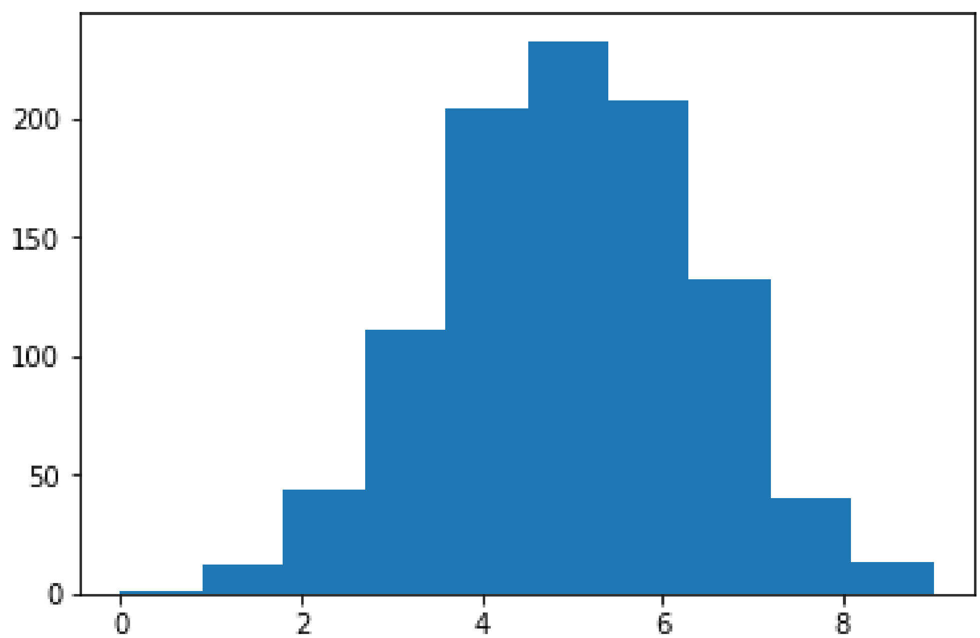
[138]:
```
np.random.choice(4,3,replace = True) #choose 3 number between 0-4(exclusive)␣
 ↪where repition is allowed
```

[138]:
```
array([1, 1, 3])
```

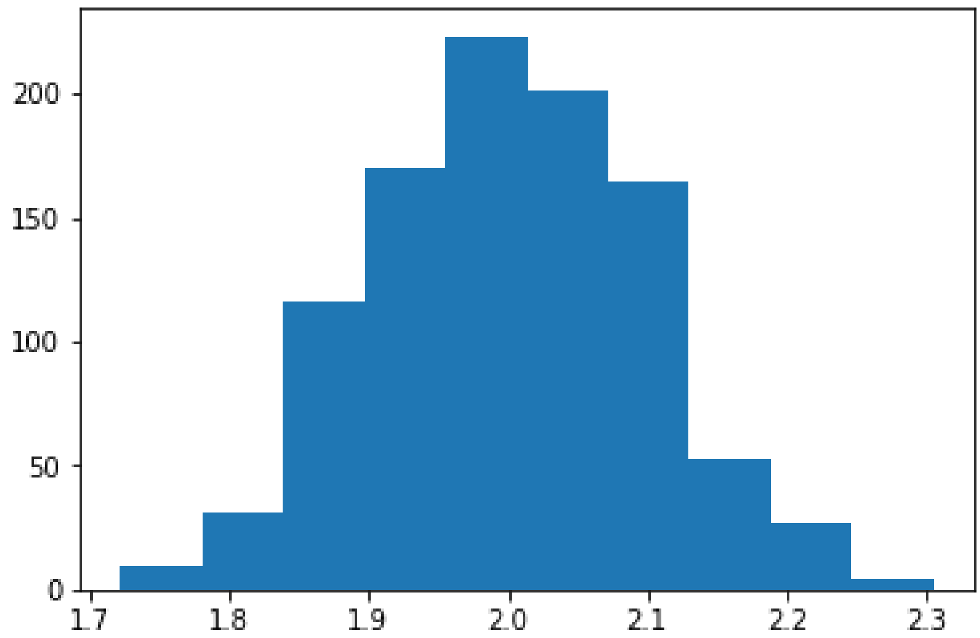
[139]:
```
np.random.choice(4,3,replace = False) #choose 3 number between 0-4(exclusive)␣
 ↪where repition is not allowed
```

[139]:
```
array([3, 0, 1])
```

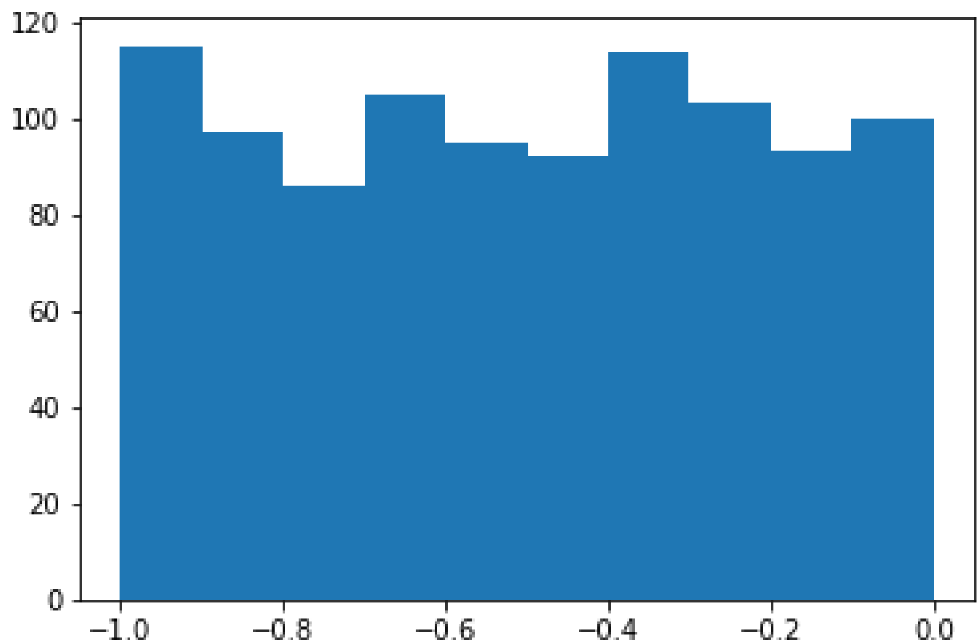
[140]:
```
n, p = 10, 0.5  # number of trials, probability of each trial
s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
plt.hist(s);
```



[141]:
```
mean, std = 2, 0.1 # mean and standard deviation
s = np.random.normal(mean, std, 1000)
plt.hist(s);#binomial distribution is discrete and normal idstribution is␣
 ↪continous.
```



[142]:
```
s = np.random.uniform(-1,0,1000)
plt.hist(s);#uniform distribution of between -1 and 0 form 1000 times
```



[143]:
```
np.random.permutation(10) #randomly arrange array of numbers between 0-9.
```

[143]: array([3, 2, 9, 0, 1, 7, 8, 4, 6, 5])

[144]:
```
np.random.permutation([1, 4, 9, 12, 15]) #random arrange the mentioned numbers␣
↪in that array.
```

[144]: array([9, 1, 15, 4, 12])

2. Linear Algebra

[145]:
```
np.dot(3,5) #dot product of two values
```

[145]: 15

[146]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[4,5],[6,7]])
np.dot(a,b) #returns dot product of two matrices
```

[146]:
```
array([[16, 19],
       [36, 43]])
```

[147]:
```
np.dot([2j, 3j], [2j, 3j]) #dotproduct of conjugates
```

[147]: (-13+0j)

[148]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[4,5],[6,7]])
c=np.array([[2,3],[3,4]])    #the dot product of two or more arrays in a single␣
↪function call,
np.linalg.multi_dot([a,b,c]) #while automatically selecting the fastest␣
↪evaluation order.
```

[148]:
```
array([[ 89, 124],
       [201, 280]])
```

[149]:
```
a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
np.matmul(a, b)   #matrix multiplication of two arrays.
```

[149]:
```
array([[4, 1],
       [2, 2]])
```

[150]:
```
b = [[4, 1], [2, 2]]
np.linalg.matrix_power(b,2) #Raise a square matrix to the (integer) power 2.
```

[150]:
```
array([[18,  6],
       [12,  6]])
```

[151]:
```
A = np.array([[1,-2j],[2j,5]])
L = np.linalg.cholesky(A) #Cholesky decomposition of a matrix.
L
```

[151]:
```
array([[1.+0.j, 0.+0.j],
       [0.+2.j, 1.+0.j]])
```

[152]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
np.linalg.eigvals(a) #returns eigen values of matrix 'a'
```

[152]:
```
array([ 1.61168440e+01, -1.11684397e+00, -9.75918483e-16])
```

[153]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
w,v = np.linalg.eig(a)   # computes eigen values and right eigen vectors of␣
 ↪square array. w is eigen values, v is eigen vector
print(w)
print(' ')
print(v)
```

```
[ 1.61168440e+01 -1.11684397e+00 -9.75918483e-16]

[[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

[154]:
```
b = [[4, 1], [2, 2]]
np.linalg.matrix_rank(b)  #find the rank of a matrix.
```

[154]:
```
2
```

[155]:
```
b = [[4, 1], [2, 2]]
np.linalg.det(b) #find the determinant of an array
```

[155]:
```
6.0
```

[156]:
```
b = [[4, 1], [2, 2]]
np.linalg.inv(b)  #finds the multiplicative inverse of a matrix.
```

[156]:
```
array([[ 0.33333333, -0.16666667],
       [-0.33333333,  0.66666667]])
```

3. other useful commands

[157]:
```
#1. numpy where for conditional search
a = np.array([6,6,7,8,4,5,3,6,7,1,2,2,2])
index_gt_5 = np.where(a > 5) #index where a > 5
a.take(index_gt_5) #get values in those indexes
```

[157]:
```
array([[6, 6, 7, 8, 6, 7]])
```

[158]:
```
#2.Position of maximum and minumum element in array
a = np.array([61,43,555,6,7,8,99])
print(np.argmax(a))  #index of maximum element(index starts from 0)
np.argmin(a) #indexof minimum element
```

```
2
```

[158]:
```
3
```

[159]:
```python
#3.concatination of two arrays
a = np.zeros([4, 4])
b = np.ones([4, 4])
#vertical concatintion
print(np.concatenate([a, b], axis=0))
print(' ')
print(np.vstack([a,b]))
np.r_[a,b] #all the three do same work
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[159]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

[160]:
```python
#horizontal concatination
print(np.concatenate([a, b], axis=1))
print(' ')
print(np.hstack([a,b]))
np.c_[a,b] #all the three do same work
```

```
[[0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]]
```

```
[[0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]]
```

[160]:
```
array([[0., 0., 0., 0., 1., 1., 1., 1.],
       [0., 0., 0., 0., 1., 1., 1., 1.],
       [0., 0., 0., 0., 1., 1., 1., 1.],
       [0., 0., 0., 0., 1., 1., 1., 1.]])
```

[161]:
```
#4.sorting a array in different ways
arr = np.random.randint(1,6, size=[8, 4])
print('arr')
print(arr)
print('each column sort')
print(np.sort(arr,axis =0)) #sort each column
print('each row sort')
np.sort(arr,axis =1) #sort each row
```

```
arr
[[4 3 4 5]
 [4 3 3 1]
 [1 2 5 5]
 [1 3 2 3]
 [5 4 1 1]
 [2 3 3 1]
 [1 3 2 5]
 [4 3 3 2]]
each column sort
[[1 2 1 1]
 [1 3 2 1]
 [1 3 2 1]
 [2 3 3 2]
 [4 3 3 3]
 [4 3 3 5]
 [4 3 4 5]
 [5 4 5 5]]
each row sort
```

[161]:
```
array([[3, 4, 4, 5],
       [1, 3, 3, 4],
       [1, 2, 5, 5],
       [1, 2, 3, 3],
       [1, 1, 4, 5],
       [1, 2, 3, 3],
       [1, 2, 3, 5],
       [2, 3, 3, 4]])
```

[163]:
```python
sort_acc_1stcol = arr[:, 0].argsort()
# Sort 'arr' by *first column* without disturbing the content in rows
arr[sort_acc_1stcol]
```

[163]:
```
array([[1, 2, 5, 5],
       [1, 3, 2, 3],
       [1, 3, 2, 5],
       [2, 3, 3, 1],
       [4, 3, 4, 5],
       [4, 3, 3, 1],
       [4, 3, 3, 2],
       [5, 4, 1, 1]])
```