Tasca4_Sprint3_Numerical_Programming

January 18, 2021

1 *Tasca 4, Sprint 3 Numeric Programming*

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

1.1 Exercise 1

Create a function that, given an array of one dimension, gives you a basic statistical summary of the data. If it detects that the array has more than one dimension, it should display an error message.

Solution:

The following steps are taken for this exercise:

```
1. Creating arrays of different dimensions using "random" from "numpy".
2. A function called "stat_summary" is created with an array as an input.
3. This functions checks the dimension of the array, and returns an error message if it is not
4. If the functions detects a one-dimensional array, then it will go on to print basic statist
                              data-type
                              Range
                              Minimum value
                              Maximum value
                              Weighted Average
                              Arithmetic Mean
                              Median
                              Standard deviation
                              Variance
                              Correlation coefficient
```

```
In [2]: # creating four arrays of different dimensions using random numbers of range (1, 100)
        ar1 = random.randint(100, size=(2, 3))
        ar2 = random.randint(100, size=(20))
        ar3 = random.randint(100, size=(5))
        ar4 = random.randint(100, size=(3, 6))

        def stat_summary(array):
            try:
                assert(array.ndim == 1), "Error 2020: More than can be handeled, the dimension
```

```python
            print('\n')
            print('Basic statisctical summary of the data: ')
            print('Data = ', array, '| data-type:', array.dtype)
            print('Range:', np.ptp(array))
            print('Minimum value:', np.min(array))
            print('Maximum value:', np.max(array))
            print('Weighted average:', np.average(array))
            print('Arithmetic Mean:', np.mean(array))
            print('Median:', np.median(array))
            print('Standard deviation: {:.4f}'.format(np.std(array)))
            print('Variance: {:.4f}'.format(np.var(array)))
            print('Correlation coef.: ', (np.corrcoef(array)))
            print('\n')
        except AssertionError as msg:
            print(msg)

    stat_summary(ar1)
    stat_summary(ar2)
    stat_summary(ar3)
    stat_summary(ar4)
```

```
Error 2020: More than can be handeled, the dimension I mean!


Basic statisctical summary of the data:
Data =  [36 40 68 48 41 41 17 91 24 76 30 70 57 66 75 64 14 70 50 36] | data-type: int64
Range: 77
Minimum value: 14
Maximum value: 91
Weighted average: 50.7
Arithmetic Mean: 50.7
Median: 49.0
Standard deviation: 20.8761
Variance: 435.8100
Correlation coef.:  1.0




Basic statisctical summary of the data:
Data =  [78 89 14 10 35] | data-type: int64
Range: 79
Minimum value: 10
Maximum value: 89
Weighted average: 45.2
Arithmetic Mean: 45.2
Median: 35.0
Standard deviation: 32.5908
```

```
Variance: 1062.1600
Correlation coef.:  1.0


Error 2020: More than can be handeled, the dimension I mean!
```

1.2 Exercise 2

Create a function that generates an NxN square of random numbers between 0 and 100.

Solution:

To achieve this, the following steps are taken.

```
1. The function created for this exercise is called "generate_square".
2. Inside, I use the in-built 'random.randint' function of numpy library. With this, you can sp
3. I am using a user-input prompt to enter an integer value, pass it to the random number gene
```

```
In [3]: def generate_square():
            n = int(input("What size of square do you want? "))
            square = random.randint(100, size=(n, n))
            print('A ' + str(n) + 'x' + str(n) + ' square of randome numbers between 0 and 100
            print(square)

        generate_square()
```

```
What size of square do you want? 7
A 7x7 square of randome numbers between 0 and 100 is:
[[34 75 55  0 48 11 77]
 [25 65 51  4 51 71 87]
 [43 72  5  0 67 23 67]
 [95 11 15 37 39  7 66]
 [66 33 26 30 12  5 90]
 [72 56 93 96  8  0 97]
 [80 36 10  7 99 57 46]]
```

1.3 Exercise 3

Create a function that given a two-dimensional table, calculates the totals per row and the totals per column.

Solution:

```
1. The function created is called "total_row_column", which takes a table as an input.
2. Inside, I am using in-built functios of numpy library to calculate the sums over rows and co
3. In numpy arrays, axis 1 refers to the rows and axis 2 refers to the columns.
4. The total sums per row and column are then printed in an array each for as many rows and col
5. The test table for the created function is generated by using 'random.randint' function. I h
6. The function "total_row_column" works for any two-dimensional table.
```

```
In [4]: def total_row_column(table):
            #table = random.randint(lim, size=(r, c))
            (r, c) = table.shape
            print('A two-dimensional table of size ' + str(r) + 'x' + str(c) + ' is: ')
            print(table)
            print('\n')
            print('Total per ' + str(r) + ' rows:', table.sum(axis = 1))
            print('\n')
            print('Total per ' + str(c) + ' columns:', table.sum(axis = 0))

        lim = 10 # range for random number generator, I kept 10 to make it easy to verify the
        r = 9 # number of rows
        c = 3 # number of columns
        table = random.randint(lim, size=(r, c))

        total_row_column(table)
```

```
A two-dimensional table of size 9x3 is:
[[7 8 3]
 [7 1 4]
 [5 0 6]
 [1 7 7]
 [6 0 4]
 [7 6 2]
 [1 0 7]
 [3 4 7]
 [9 4 6]]


Total per 9 rows: [18 12 11 15 10 15  8 14 19]


Total per 3 columns: [46 30 46]
```

1.4 Exercise 4

Manually implements a function that calculates the correlation coefficient. Learn about its uses and interpretation.

Solution:

To achieve this, the following steps are taken:

```
1. The function created to calculate the correlation coefficient is called "corr_coeff", which
2. First, the two datasets are stacked together to form a matrix.
3. Second, the covariance is calculated as the average of the product between the values from
4. Finally, the correlation co-efficient is calculated as the covariance divided by the produc
5. At the end of the function "corr_coeff", the correlation coefficient, manually calculated a
6. To demonstrate the use of the function, I use two datasets generated with random numbers fr
```

```
        data 1 = Random numbers with Gaussian distribution with a mean of 100 and a standard de
        data 2 = Gaussian noise added with a mean of a 50 and a standard deviation of 10 to dat
7. These two datasets are given to the function "corr_coeff" and the two correlation coefficien
```

```
In [5]: from numpy import random
        from numpy.random import randn
        from numpy.random import seed
        from numpy import cov
        import numpy as np

        def corr_coeff(d1, d2):
            n = d1.size
            # calculate covariance
            x = np.column_stack([d1, d2])
            x = x - x.mean(axis = 0)
            covar = np.dot(x.T, x.conj()) / (n - 1)
            #print('Covariance is {:.2f}'.format(covar[0][1]))

            # correlation coefficient from covariance
            corr = covar[0][1] / (np.std(d1) * np.std(d2))
            print('Correlation coefficient = {:.4f}'.format(corr))
            print('With in-built function = {:.4f}'.format(np.corrcoef(d1, d2)[0][1]))
            return corr

        # Generating data
        N = 1000
        #seed(130)
        data1 = 20 * randn(N) + 100
        data2 = data1 + (10 * randn(N) + 50)

        corr = corr_coeff(data1, data2)
        #print(corr)
```

```
Correlation coefficient = 0.9012
With in-built function = 0.9003
```

Notice that, for the datasets that I have generated are with Gaussian distribution and are separated by a Gaussian noise. In this case, the two datasets are expected to be highly correlated positively, and the correlation coefficient evidently has a high value (0.89).

Next, I generate two datasets with 1000 random numbers using *randint* of size 10000 with random values between 0 and 100. These two datasets are completely random, and the correlation coefficient shows the value close to 0, indicating that the two are, indeed, independent of each other.

```
In [6]: # checking the function with two datasets of 10000 random numbers, unseeded.
        a1 = random.randint(100, size=(1000))
        a2 = random.randint(100, size=(1000))
        corr = corr_coeff(a1, a2)
```

```
Correlation coefficient = 0.0513
With in-built function = 0.0512
```

Notes:

An important thing to remember that, the this correlation coefficient is a measure of the **linear relationship** between two random variables. This is important because the built-in function *corrcoef* of numpy calculates the Pearson correlation coefficient, and getting the value of it to be zero is sometimes misinterpreted as the two datasets not being correlated at all. The distinction to be made is that the two datasets (or variables or data samples) are *not linearly related* but can have non-linear dependency between them.