Assignment-10

Name-Ajay Chaudhary
Batch-Data Engineering(Batch-1)

Numpy

Before we can use NumPy we will have to import it. It has to be imported like any other module

[1]:

```
import numpy as np
```

Our first simple Numpy example deals with temperatures. Given is a list with values, e.g. temperatures in Celsius:

[2]:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]
```

We will turn our list "cvalues" into a one-dimensional numpy array:

[3]:

```
C = np.array(cvalues)
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

[4]:

```
print(C * 9 / 5 + 32)
```

```
[68.18 69.44 71.42 72.5  72.86 72.14 71.24 70.16 69.62 68.18]
```

Compared to this, the solution for our Python list looks awkward:

[5]:

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

```
[68.18, 69.44, 71.42, 72.5, 72.86, 72.14, 71.24000000000001, 70.16, 69.62, 68.18]
```

arange()

arange returns evenly spaced values within a given interval. The values are generated within the half-open interval '[start, stop)' If the function is used with integers, it is nearly equivalent to the Python built-in function range, but arange returns an ndarray rather than a list iterator as range does. If the 'start' parameter is not given, it will be set to 0. The end of the interval is determined by the parameter 'stop'.

[6]:

```
import numpy as np
a = np.arange(1, 10)
print(a)


x = range(1, 10)
print(x)    # x is an iterator
print(list(x))


# further arange examples:
x = np.arange(10.4)
print(x)
x = np.arange(0.5, 10.4, 0.8)
print(x)
```

```
[1 2 3 4 5 6 7 8 9]
range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 0.5  1.3  2.1  2.9  3.7  4.5  5.3  6.1  6.9  7.7  8.5  9.3 10.1]
```

The following usages of arange is a bit offbeat. Why should we use float values, if we want integers as result. Anyway, the result might be confusing. Before arange starts, it will round the start value, end value and the stepsize:

[7]:

```
x = np.arange(0.5, 10.4, 0.8, int)
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12]
```

linspace

The syntax of linspace:
linspace(start, stop, num=50, endpoint=True, retstep=False)
linspace returns an ndarray, consisting of 'num' equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop). If a closed or a half-open interval will be returned, depends on whether 'endpoint' is True or False. The parameter 'start' defines the start value of the sequence which will be created. 'stop' will the end value of the sequence, unless 'endpoint' is set to False. In the

latter case, the resulting sequence will consist of all but the last of 'num + 1' evenly spaced samples. This means that 'stop' is excluded. Note that the step size changes when 'endpoint' is False.

[8]:

```
import numpy as np
# 50 values between 1 and 10:
print(np.linspace(1, 10))
# 7 values between 1 and 10:
print(np.linspace(1, 10, 7))
# excluding the endpoint:
print(np.linspace(1, 10, 7, endpoint=False))
```

```
[ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
  2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
  3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12244898
  4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.2244898
  5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32653061
  6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42857143
  7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53061224
  8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
  9.81632653 10.        ]
[ 1.   2.5  4.   5.5  7.   8.5 10. ]
[1.         2.28571429 3.57142857 4.85714286 6.14285714 7.42857143
 8.71428571]
```

We haven't discussed one interesting parameter so far. If the optional parameter 'retstep' is set, the function will also return the value of the spacing between adjacent values. So, the function will return a tuple ('samples', 'step'):

[9]:

```
import numpy as np

samples, spacing = np.linspace(1, 10, retstep=True)
print(spacing)
samples, spacing = np.linspace(1, 10, 20, endpoint=True, retstep=True)
print(spacing)
samples, spacing = np.linspace(1, 10, 20, endpoint=False, retstep=True)
print(spacing)
```

```
0.1836734693877551
0.47368421052631576
0.45
```

Count Values in Pandas Dataframe

[12]:

```
# Creating dataframe with
# some missing values
NaN = np.nan
dataframe = pd.DataFrame({'Name': ['Shobhit', 'Vaibhav',
                                   'Vimal', 'Sourabh',
                                   'Rahul', 'Shobhit'],
                          'Physics': [11, 12, 13, 14, NaN, 11],
                          'Chemistry': [10, 14, NaN, 18, 20, 10],
                          'Math': [13, 10, 15, NaN, NaN, 13]})



print(dataframe.count())
print (dataframe)
```

```
Name         6
Physics      5
Chemistry    5
Math         4
dtype: int64
      Name  Physics  Chemistry  Math
0  Shobhit     11.0       10.0  13.0
1  Vaibhav     12.0       14.0  10.0
2    Vimal     13.0        NaN  15.0
3  Sourabh     14.0       18.0   NaN
4    Rahul      NaN       20.0   NaN
5  Shobhit     11.0       10.0  13.0
```

[13]:

```
# using dataframe.count()
# to count all values
dataframe.count()
```

[13]:

```
Name         6
Physics      5
Chemistry    5
Math         4
dtype: int64
```

If we want to count all the values with respect to row then we have to pass axis=1 or ‘columns’.

[14]:

```
# we can pass either axis=1 or
# axos='columns' to count with respect to row
print(dataframe.count(axis = 1))

print(dataframe.count(axis = 'columns'))
```

```
0    4
1    4
2    3
3    3
4    2
5    4
dtype: int64
0    4
1    4
2    3
3    3
4    2
5    4
dtype: int64
```

Now if we want to count null values in our dataframe.

[15]:

```
# it will give the count
# of individual columns count of null values
print(dataframe.isnull().sum())

# it will give the total null
# values present in our dataframe
print("Total Null values count: ",
      dataframe.isnull().sum().sum())
```

```
Name          0
Physics       1
Chemistry     1
Math          2
dtype: int64
Total Null values count:  4
```

Some examples to use .count()

Now we want to count no of students whose physics marks are greater than 11.

[16]:

```
# count of student with greater
# than 11 marks in physics
print("Count of students with physics marks greater than 11 is->",
      dataframe[dataframe['Physics'] > 11]['Name'].count())

# resultant of above dataframe
dataframe[dataframe['Physics']>11]
```

```
Count of students with physics marks greater than 11 is-> 3
```

[16]:

	Name	Physics	Chemistry	Math
1	Vaibhav	12.0	14.0	10.0
2	Vimal	13.0	NaN	15.0
3	Sourabh	14.0	18.0	NaN

Count of students whose physics marks are greater than 10,chemistry marks are greater than 11 and math marks are greater than 9.

[17]:

```
# Count of students whose physics marks
# are greater than 10,chemistry marks are
# greater than 11 and math marks are greater than 9.
print("Count of students ->",
      dataframe[(dataframe['Physics'] > 10) &
                (dataframe['Chemistry'] > 11) &
                (dataframe['Math'] > 9)]['Name'].count())

# dataframe of above result
dataframe[(dataframe['Physics'] > 10 ) &
          (dataframe['Chemistry'] > 11 ) &
          (dataframe['Math'] > 9 )]
```

```
Count of students -> 1
```

[17]:

	Name	Physics	Chemistry	Math
1	Vaibhav	12.0	14.0	10.0

Using read_csv() method: read_csv() is an important pandas function to read csv files and do operations on it.

```
# Python program to illustrate
# creating a data frame using CSV files

# import pandas module
import pandas as pd

# creating a data frame
df = pd.read_csv("Bibadata.csv")
print(df.head())
```

Using read_table() method: read_table() is another important pandas function to read csv files and create data frame from it.

[ ]:

```
# Python program to illustrate
# creating a data frame using CSV files

# import pandas module
import pandas as pd

# creating a data frame
df = pd.read_table("bibadata.csv", delimiter =", ")
print(df.head())
```

Using the csv module: One can directly import the csv files using the csv module and then create a data frame using that csv file.

[ ]:

```
# Python program to illustrate
# creating a data frame using CSV files

# import pandas module
import pandas as pd
# import csv module
import csv

with open("Bibadata.csv") as csv_file:
    # read the csv file
    csv_reader = csv.reader(csv_file)

    # now we can use this csv files into the pandas
    df = pd.DataFrame([csv_reader], index = None)

# iterating values of first column
for val in list(df[1]):
    print(val)
```

Pandas Inner Join

Inner join is the most common type of join you'll be working with. It returns a Dataframe with only those rows that have common characteristics. This is similar to the intersection of two sets.

[18]:

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
     'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
     'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# inner join
df = pd.merge(a, b, on='id', how='inner')

# display dataframe
df
```

[18]:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q

Pandas Left Join

With a left outer join, all the records from the first Dataframe will be displayed, irrespective of whether the keys in the first Dataframe can be found in the second Dataframe. Whereas, for the second Dataframe, only the records with the keys in the second Dataframe that can be found in the first Dataframe will be displayed.

[19]:

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
     'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
     'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# left outer join
df = pd.merge(a, b, on='id', how='left')

# display dataframe
df
```

[19]:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q
2	10	c	NaN
3	12	d	NaN

Pandas Right Outer Join

For a right join, all the records from the second Dataframe will be displayed. However, only the records with the keys in the first Dataframe that can be found in the second Dataframe will be displayed.

[20]:

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
     'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
     'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# right outer join
df = pd.merge(a, b, on='id', how='right')

# display dataframe
df
```

[20]:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q
2	9	NaN	r
3	8	NaN	s

Pandas Full Outer Join

A full outer join returns all the rows from the left Dataframe, and all the rows from the right Dataframe, and matches up rows where possible, with NaNs elsewhere. But if the Dataframe is complete then we get the same output

[21]:

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
     'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
     'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# full outer join
df = pd.merge(a, b, on='id', how='outer')

# display dataframe
df
```

[21]:

	id	val1_x	val1_y
0	1	a	p
1	2	b	q
2	8	NaN	s
3	9	NaN	r
4	10	c	NaN
5	12	d	NaN

Pandas Index Join

To merge the Dataframe on indices pass the left_index and right_index arguments as True i.e. both the Dataframes are merged on an index using default Inner Join.

[23]:

```
# importing pandas
import pandas as pd

# Creating dataframe a
a = pd.DataFrame()

# Creating Dictionary
d = {'id': [1, 2, 10, 12],
     'val1': ['a', 'b', 'c', 'd']}

a = pd.DataFrame(d)

# Creating dataframe b
b = pd.DataFrame()

# Creating dictionary
d = {'id': [1, 2, 9, 8],
     'val1': ['p', 'q', 'r', 's']}
b = pd.DataFrame(d)

# index join
df = pd.merge(a, b, left_index=True, right_index=True)
print(df)
```

```
   id_x val1_x  id_y val1_y
0     1      a     1      p
1     2      b     2      q
2    10      c     9      r
3    12      d     8      s
```