Advanced Python Programming(2005409)

IV Sem CSE

Skill Oriented Course

List of Experiments:

Week-1: Study and implementation of various Basic Slicing and Advanced Indexing operations of NumPy arrays using Python over example data series?

Week-2: Implement the program using python Aggregations like Min, Max, and etc.?
Example: Consider the heights of all US presidents and find the Average Height of prime ministers of America? This data is available in the file "*president_heights.csv*".

Week-3: Write a python Program using Numpy Comparisons, Masks, and Boolean Logic? Example: Consider the series of data that represents the amount of precipitation each day for a year in a given city and count the Rainy Days.

Week-4: Write a python Program using Numpy Fancy Indexing in single and multiple dimensions by selecting Random Points?

Week-5: Study and implementation of various Pandas operations on

i) Data sets ii) Data Frames iii) Crosstab iv) Group by
v) Filter vi) Missing values

Week-6: Implement the python program using pandas

i) Program to Combining Datasets using Merge.
ii) Program to Combining Datasets using joins.

Week-7: Implement the python program using pandas

i) Program using Pandas on Pivot Tables.
ii) Program using Pandas to Vectorized String Operations.

Week-8: Program using Pandas to Working with Time Series
Example: Visualizing Seattle Bicycle Counts data set.

Week-9: Implement the python program for the following matplotlib features

i) Color bars.
ii) Annotation
iii) Matplotlib to Text.
iv) Histograms
v) Scatter Plots
vi) Box plot

Week 10: Write the python program to implement various sub packages of Scipy.

Week11: Write a Python program to create a parent class and child class along with their own methods. Access parent class members in child class to implement the following sceneries.

a) Constructors & destructors
b) Polymorphism

Example:
Create a class ATM and define ATM operations to create account, deposit, check_balance, withdraw and delete account. Use constructor to initialize members.

Week-12: Implement the various data cleaning steps of example data sets using python nympy and pandas

Week13: Implement the feature selection of data set using appropriate sklearn libraries.

Week-1: Study and implementation of various Basic Slicing and Advanced Indexing operations of NumPy arrays using Python over example data series?

BASIC SLICING OPERATIONS

A Python slice object is constructed by giving **start, stop**, and **step** parameters to the built-in **slice** function.

```
import numpy as np
```

```
a = np.arange(10)
s = slice(2,7,2)
print a[s]
# slice single item
import numpy as np
a = np.arange(10)
b = a[5]
print b
```
Its output is as follows −
```
5
# slice items starting from index
import numpy as np
a = np.arange(10)
print a[2:]
```
Now, the output would be −
```
[2  3  4  5  6  7  8  9]
# slice items between indexes
import numpy as np
a = np.arange(10)
print a[2:5]
```
Here, the output would be −
```
[2  3  4]
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print a
# slice items starting from index
print 'Now we will slice the array from the index a[1:]'
print a[1:]
```
Slicing can also include ellipsis (…) to make a selection tuple of the same length as the dimension of an array. If ellipsis is used at the row position, it will return an ndarray comprising of items in rows.
```
# array to begin with
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print ('Our array is:' )
print(a)
print('\n')
# this returns array of items in the second column
print ('The items in the second column are:'  )
print (a[...,1])
print('\n')
# Now we will slice all items from the second row
print ('The items in the second row are:' )
print (a[1,...])
print('\n')

# Now we will slice all items from column 1 onwards
print ('The items column 1 onwards are:' )
print (a[...,1:])
```
NumPy - Advanced Indexing

There are two types of advanced indexing − Integer and Boolean.
Advanced indexing always returns a copy of the data. As against this, the slicing only presents a view

Integer Indexing

In the following example, one element of specified column from each row of ndarray object is selected. Hence, the row index contains all row numbers, and the column index specifies the element to be selected.
```
import numpy as np
x = np.array([[1, 2], [3, 4], [5, 6]])
y = x[[0,1,2], [0,1,0]]
```

```
print y
import numpy as np
x = np.array([[ 0,  1,  2],[ 3,  4,  5],[ 6,  7,  8],[ 9, 10, 11]])
print ('Our array is:' )
print (x )
print ('\n')
rows = np.array([[0,0],[3,3]])
cols = np.array([[0,2],[0,2]])
y = x[rows,cols]
print ('The corner elements of this array are:' )
print (y)
```

The output of this program is as follows −

```
Our array is:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
The corner elements of this array are:
[[ 0  2]
 [ 9 11]]
```

Advanced and basic indexing can be combined by using one slice (:) or ellipsis (…) with an index array. The following example uses slice for row and advanced index for column. The result is the same when slice is used for both. But advanced index results in copy and may have different memory layout.

```
import numpy as np
x = np.array([[ 0,  1,  2],[ 3,  4,  5],[ 6,  7,  8],[ 9, 10, 11]])
print 'Our array is:'
print x
print '\n'
# slicing
z = x[1:4,1:3]
print 'After slicing, our array becomes:'
print z
print '\n'
# using advanced index for column
y = x[1:4,[1,2]]
print 'Slicing using advanced index for column:'
print y
```

The output of this program would be as follows

```
Our array is:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
After slicing, our array becomes:
[[ 4  5]
 [ 7  8]
 [10 11]]
Slicing using advanced index for column:
[[ 4  5]
 [ 7  8]
 [10 11]]
```

Boolean Array Indexing

This type of advanced indexing is used when the resultant object is meant to be the result of Boolean operations, such as comparison operators.

In this example, items greater than 5 are returned as a result of Boolean indexing.

```
import numpy as np
x = np.array([[ 0,  1,  2],[ 3,  4,  5],[ 6,  7,  8],[ 9, 10, 11]])
print 'Our array is:'
print x
print '\n'
# Now we will print the items greater than 5
print 'The items greater than 5 are:'
print x[x > 5]
```

The output of this program would be −

```
Our array is:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
The items greater than 5 are:
[ 6  7  8  9 10 11]
```

In this example, NaN (Not a Number) elements are omitted by using ~ (complement operator).

```
import numpy as np
a = np.array([np.nan, 1,2,np.nan,3,4,5])
print a[~np.isnan(a)]
```

Its output would be −

```
[ 1.   2.   3.   4.   5.]
```

The following example shows how to filter out the non-complex elements from an array.

```
import numpy as np
a = np.array([1, 2+6j, 5, 3.5+5j])
print a[np.iscomplex(a)]
```

Here, the output is as follows –

```
[2.0+6.j  3.5+5.j]
```

Week-2: Implement the program using python Aggregations like Min, Max, and etc.?

Example: Consider the heights of all US presidents and find the Average Height of prime ministers of America? This data is available in the file "*president_heights.csv*".

Description: usage of different aggregate functions of numpy over randomly generated list of 100 elements like mean, median, std,var,sum,min,max.

```
import numpy as np
array1 = np.random.randint(1,1000,size = (100))
print(array1)
print("Mean: ", np.mean(array1))
print("median: ", np.median(array1))
print("Std: ", np.std(array1))
 print("Var: ", np.var(array1))
 print("Sum: ", np.sum(array1))
 print("Prod: ", np.prod(array1))
print("min: ", np.min(array1))
print("max: ", np.max(array1))
print("argmin: ", np.argmin(array1))
print("argmax: ", np.argmax(array1))
```

Example: What is the Average Height of US Presidents?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

```
import pandas as pd
data = pd.read_csv('president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)
print("Mean height:       ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:    ", heights.min())
print("Maximum height:    ", heights.max())
```

```
Mean height:       179.738095238
Standard deviation: 6.93184344275
Minimum height:     163
 Maximum height:     193
```

Week-3: Write a python Program using Numpy Comparisons, Masks, and Boolean Logic? Example: Consider the series of data that represents the amount of precipitation each day for a year in a given city and count the Rainy Days.

Description: numpy mask is a condition to select subset of qualified arrays elements, outcome of every element is Boolean type.

```
import numpy as np
x = np.array([1, 2, 3, 4, 5])
print('x<3 : \n', x < 3)
print('x>3 : \n', x > 3)
print('x<=3 : \n', x <= 3)
print('x>=3 : \n', x >= 3)
print('x==3 : \n', x == 3)
print('x!=3 : \n', x != 3)
print('2 * x == x ** 2:  \n', (2 * x) == (x ** 2))
# Here is a two-dimensional example:
x = np.random.randint(10, size=(3, 4))
print(x)
print(x<6)
# how many values less than 6?
count=np.count_nonzero(x < 6)
print(count)
count=np.sum(x < 6)
print(count)
# how many values less than 3 or greater than 5?
count=np.sum((x > 5) | (x < 3))
print("count=", count)
# how many values less than 6 in each row?
count=np.sum(x < 6, axis=1)
print(count)
# are there any values greater than 8?
y=np.any(x > 8)
print(y)
# are all values less than 10?
y=np.all(x < 10)
print(y)
# are all values in each row less than 8?
y=np.all(x < 8, axis=1)
print(y)
```

```
import numpy as np
import pandas as pd

# use pandas to extract rainfall inches as a NumPy array
rainfall = pd.read_csv('data/Seattle2014.csv')['PRCP'].values
inches = rainfall / 254.0  # 1/10mm -> inches
inches.shape
# construct a mask of all rainy days
rainy = (inches > 0)

# construct a mask of all summer days (June 21st is the 172nd day)
days = np.arange(365)
summer = (days > 172) & (days < 262)

print("Median precip on rainy days in 2014 (inches):   ",
      np.median(inches[rainy]))
print("Median precip on summer days in 2014 (inches):  ",
      np.median(inches[summer]))
print("Maximum precip on summer days in 2014 (inches): ",
      np.max(inches[summer]))
print("Median precip on non-summer rainy days (inches):",
      np.median(inches[rainy & ~summer]))
```

Week-4: Write a python Program using Numpy Fancy Indexing in single and multiple dimensions by selecting Random Points?

Description : Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)
print([x[3], x[7], x[2]])
#Alternatively, we can pass a single list or array of indices to obtain the same result
ind = [3, 7, 2]
print(x[ind])
#When using fancy indexing, the shape of the result reflects the shape of the index arrays rather than the
shape of the array being indexed:
ind = np.array([[3, 7],
            [4, 5]])
print(x[ind])
#Fancy indexing also works in multiple dimensions. Consider the following array:
X = np.arange(12).reshape((3, 4))
print(X)
#Like with standard indexing, the first index refers to the row, and the second to the column:
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
print(X[row, col])
```

Week-5: Study and implementation of various Pandas operations on

i) Data sets ii) Data Frames iii) Crosstab iv) Group by v) Filter vi) Missing values

Description

Data Frames

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

A pandas DataFrame can be created using the following constructor −

pandas.DataFrame(data, index, columns, dtype, copy

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print (df)
#Rows can be selected by passing row label to a loc function.
print(df.loc['rank1'])
#Rows can be selected by passing integer location to an iloc function.
print (df.iloc[2])
#addition of  columm
df['gender']= ['male','female','male','male']
print(df)
# using del function
print ("Deleting the first column using DEL function:")
del df['Name']
print (df)
# using pop function
print ("Deleting another column using POP function:")
df.pop('Age')
print (df)
```

pandas-Crosstab

```
import pandas as pd
df = pd.read_csv("survey.csv")
```

```
print(df)
```

	Name	Nationality	Gender	Age	Handedness
0	Kathy	USA	Female	23	Right
1	Linda	USA	Female	18	Right
2	Peter	USA	Male	19	Right
3	John	USA	Male	22	Left
4	Fatima	Bangadesh	Female	31	Left
5	Kadir	Bangadesh	Male	25	Left
6	Dhaval	India	Male	35	Left
7	Sudhir	India	Male	31	Left
8	Parvir	India	Male	37	Right
9	Yan	China	Female	52	Right
10	Juan	China	Female	58	Left
11	Liang	China	Male	43	Left

```
X=pd.crosstab(df.Nationality,df.Handedness)
print(X)
Out:
```

Handedness	Left	Right
Nationality		
Bangadesh	2	0
China	2	1
India	2	1
USA	1	3

```
X=pd.crosstab(df.Gender,df.Handedness)
print(X)
Out[18]:
```

Handedness	Left	Right
Gender		
Female	2	3
Male	5	2

<u>Margins</u>

```
X=pd.crosstab(df.Gender,df.Handedness, margins=True)
Print(X)
Out[19]:
```

Handedness	Left	Right	All
Gender			
Female	2	3	5
Male	5	2	7
All	7	5	12

Multi Index Column and Rows

```
X=pd.crosstab(df.Gender, [df.Handedness,df.Nationality], margins=True)
Print(X)
```

Handedness	Left				Right			All
Nationality	Bangadesh	China	India	USA	China	India	USA	
Gender								
Female	1	1	0	0	1	0	2	5
Male	1	1	2	1	0	1	1	7
All	2	2	2	1	1	1	3	12

X=pd.crosstab([df.Nationality, df.Gender], [df.Handedness], margins=True)
Print(X)

	Handedness	Left	Right	All
Nationality	Gender			
Bangadesh	Female	1	0	1
	Male	1	0	1
China	Female	1	1	2
	Male	1	0	1
India	Male	2	1	3
USA	Female	0	2	2
	Male	1	1	2
All		7	5	12

Normalize: Dividing all values by sum of all values
X=pd.crosstab(df.Gender, df.Handedness, normalize='index')
Print(X)
Out[22]:

Handedness	Left	Right
Gender		
Female	0.400000	0.600000
Male	0.714286	0.285714

Group By

weather_by_cities.csv

	day	**city**	**temperature**	**windspeed**	**event**
0	1/1/2017	new york	32	6	Rain
1	1/2/2017	new york	36	7	Sunny
2	1/3/2017	new york	28	12	Snow
3	1/4/2017	new york	33	7	Sunny
4	1/1/2017	mumbai	90	5	Sunny
5	1/2/2017	mumbai	85	12	Fog
6	1/3/2017	mumbai	87	15	Fog
7	1/4/2017	mumbai	92	5	Rain

	day	city	temperature	windspeed	event
8	1/1/2017	paris	45	20	Sunny
9	1/2/2017	paris	50	13	Cloudy
10	1/3/2017	paris	54	8	Cloudy
11	1/4/2017	paris	42	10	Cloudy

1. What was the maximum temperature in each of these 3 cities?
2. What was the average windspeed in each of these 3 cities?

```
import pandas as pd
df = pd.read_csv("weather_by_cities.csv")
g = df.groupby("city")
for city, data in g:
   print("city:",city)
   print("\n")
   print("data:",data)
g.get_group('mumbai')
# Maximum temperature in each of these 3 cities?
g.max()
# mean temperature in each of these 3 cities?
g.mean()
# Minimum temperature in each of these 3 cities?
g.min()
# describe g
g.describe()
#city wise data count
g.size()
#city wise data count
g.count()
#city wise data plot
%matplotlib inline
g.plot()
```

Group data using custom function: Let's say you want to group your data using custom function. Here the requirement is to create three groups

1. Days when temperature was between 80 and 90
2. Days when it was between 50 and 60
3. Days when it was anything else

For this you need to write custom grouping function and pass that to groupby

```
def grouper(df, idx, col):
   if 80 <= df[col].loc[idx] <= 90:
      return '80-90'
   elif 50 <= df[col].loc[idx] <= 60:
      return '50-60'
   else:
      return 'others'
g = df.groupby(lambda x: grouper(df, x, 'temperature'))
for key, d in g:
   print("Group by Key: {}\n".format(key))
   print(d)
```

Missing Values

```
#read data set
import pandas as pd
df = pd.read_csv("weather_data.csv")
print (df)
```

```
# fill not available fields with '0'
new_df = df.fillna(0)
print(new_df)
# fill not available fields with '0'
new_df = df.fillna({  'temperature': 0,  'windspeed': 0,    'event': 'no event' })
print(new_df)
#fill NaN values with last valid value
new_df = df.bfill(axis=0)
print(new_df)
#fill NaN values with next valid value
new_df = df.bfill(axis=0)
print(new_df)
# fill not available fields using interpolate
new_df = df.interpolate(method="linear")
print(new_df)
# drop the rows with NaN values
new_df = df.dropna(axis=0,thresh=5)
print(new_df)
```

Filters

description: A common operation in data analysis is to filter values based on a condition or multiple conditions. Pandas provides a variety of ways to filter data points (i.e. rows).

```
import numpy as np
import pandas as pd
df = pd.DataFrame({
'name':['Jane','John','Ashley','Mike','Emily','Jack','Catlin'],
'ctg':['A','A','C','B','B','C','B'],
'val':np.random.random(7).round(2),
'val2':np.random.randint(1,10, size=7)
})

print(df)
print("\n")

#We can use the logical operators on column values to filter rows.
#ows in which the value in "val" column is greater than 0.5.
print(df[df.val > 0.5], "\n")

#The "&" signs stands for "and" , the "|" stands for "or".
print(df[(df.val < 0.5) | (df.val2 == 7)])
print("\n")
#isin method is another way of applying multiple condition for filtering.
names = ['John','Catlin','Mike']
print(df[df.name.isin(names)])
print("\n")
#str accessor provide flexible ways to filter rows based on strings
print(df[df.name.str.startswith('J')])
print("\n")
#The tilde operator is used for "not" logic in filtering.
print(df[~df.name.str.startswith('J')])
print("\n")
#The nlargest and nsmallest functions allow for selecting rows that have the largest or smallest values in a column,
print(df.nlargest(3, 'val'))
print("\n")
print(df.nsmallest(2, 'val2'))
print("\n")
#loc: select rows or columns using labels
#iloc: select rows or columns using indices
#rows 3 and 4, all columns
print(df.iloc[3:5, :] )
print("\n")
#prints the rows labelled with b,c,d
df.index = ['a','b','c','d','e','f','g']
print(df.loc['b':'d', :])
```

Week-6: Implement the python program using pandas

i) Program to Combining Datasets using Merge.

ii) Program to Combining Datasets using joins.

Description: Both join and merge can be used to combines two dataframes but the join method combines two dataframes on the basis of their indexes whereas the merge method is more versatile and allows us to specify columns beside the index to join on for both dataframes.

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=True)
# import the pandas library
import pandas as pd
left = pd.DataFrame({
  'id':[1,2,3,4,5],
  'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
  'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right = pd.DataFrame(
  {'id':[1,2,3,4,5],
  'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
  'subject_id':['sub2','sub4','sub3','sub6','sub5']})
print(left)
print (right)
# merge to two data sets based on 'id'
df= pd.merge(left,right,on='id')
print(df)
# merge to two data sets based on 'id' and 'subject_id'
df= pd.merge(left,right,on=['id','subject_id'])
print(df)
# left outer join based on 'subject_id'
df= pd.merge(left, right, on='subject_id', how='left')
print(df)
# right outer join based on 'subject_id'
df=pd.merge(left, right, on='subject_id', how='right')
print(df)
# full outer join based on 'subject_id'
df= pd.merge(left, right, how='outer', on='subject_id')
print(df)
# Inner join based on 'subject_id'
df= pd.merge(left, right, on='subject_id', how='inner')
print(df)
```

Description: In order to join dataframe, we use `.join()` function this function is used for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

```
# importing pandas module
import pandas as pd
# Define a dictionary containing employee data
data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
      'Age':[27, 24, 22, 32]}
# Define a dictionary containing employee data
data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
      'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}
# Convert the dictionary into DataFrame
df = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])
```

```
# Convert the dictionary into DataFrame
df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])
print(df, "\n\n", df1)
# joining dataframe
res = df.join(df1)
print(res)
# getting union
res1 = df.join(df1, how='outer')
print(res1)
# using on argument in join
df=df.assign(Key=['K0', 'K1', 'K2', 'K3'])
print(df)
res2 = df.join(df1, on='Key')
print(res2)
```

Week-7: Implement the python program using pandas

i) Program using Pandas on Pivot Tables.

ii) Program using Pandas to Vectorized String Operations.

Pivot table is a statistical table that summarizes a substantial table like big datasets. It is part of data processing. This summary in pivot tables may include mean, median, sum, or other statistical terms. we can create a pivot table in Python using Pandas using the dataframe.pivot() method.

Syntax : dataframe.pivot(self, index=None, columns=None, values=None, aggfunc)

Parameters –

index: Column for making new frame's index.

columns: Column for new frame's columns.

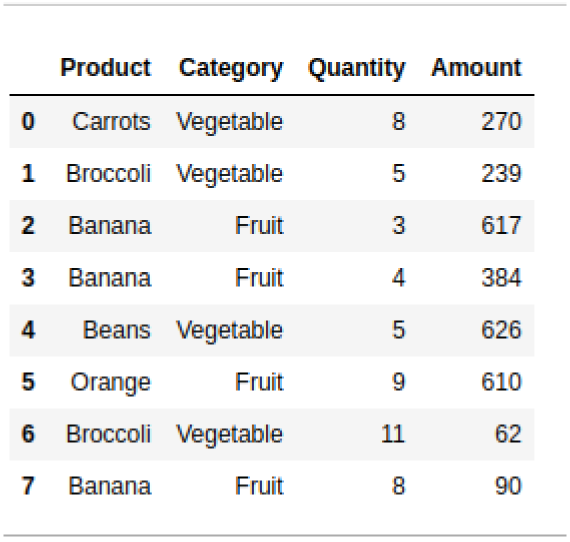
values: Column(s) for populating new frame's values.

aggfunc: function, list of functions, dict, default numpy.mean

Let's first create a dataframe that includes Sales of Fruits.

```
# importing pandas

import pandas as pd
# creating dataframe
df = pd.DataFrame({'Product' : ['Carrots', 'Broccoli', 'Banana', 'Banana',
                                'Beans', 'Orange', 'Broccoli', 'Banana'],
                   'Category' : ['Vegetable', 'Vegetable', 'Fruit', 'Fruit',
                                 'Vegetable', 'Fruit', 'Vegetable', 'Fruit'],
                   'Quantity' : [8, 5, 3, 4, 5, 9, 11, 8],
                   'Amount' : [270, 239, 617, 384, 626, 610, 62, 90]})
df
```

OUTPUT:

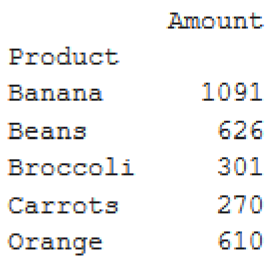


	Product	Category	Quantity	Amount
0	Carrots	Vegetable	8	270
1	Broccoli	Vegetable	5	239
2	Banana	Fruit	3	617
3	Banana	Fruit	4	384
4	Beans	Vegetable	5	626
5	Orange	Fruit	9	610
6	Broccoli	Vegetable	11	62
7	Banana	Fruit	8	90

Get the total sales of each product

```
# creating pivot table of total sales

# product-wise aggfunc = 'sum' will

# allow you to obtain the sum of sales

# each product

pivot = df.pivot_table(index =['Product'],

                       values =['Amount'],

                       aggfunc ='sum')

print(pivot)
```

Output:

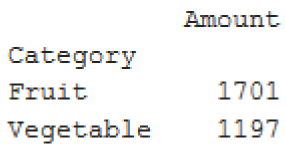


```
          Amount
Product
Banana      1091
Beans        626
Broccoli     301
Carrots      270
Orange       610
```

Get the total sales of each category

```
# creating pivot table of total

# sales category-wise aggfunc = 'sum'

# will allow you to obtain the sum of

# sales each product

pivot = df.pivot_table(index =['Category'],

                       values =['Amount'],

                       aggfunc ='sum')

print(pivot)
```

Output:



```
           Amount
Category
Fruit        1701
Vegetable    1197
```

Get the total sales of by category and product both

```
# creating pivot table of sales

# by product and category both
# aggfunc = 'sum' will allow you

# to obtain the sum of sales each

# product

pivot = df.pivot_table(index =['Product', 'Category'],

                                    values =['Amount'], aggfunc ='sum')

print (pivot)
```

Output –

```
                       Amount
Product  Category
Banana   Fruit           1091
Beans    Vegetable        626
Broccoli Vegetable        301
Carrots  Vegetable        270
Orange   Fruit            610
```

Get the Mean, Median, Minimum sale by category

```
# creating pivot table of Mean, Median,

# Minimum sale by category aggfunc = {'median',

# 'mean', 'min'} will get median, mean and

# minimum of sales respectively

pivot = df.pivot_table(index =['Category'], values =['Amount'],

                                    aggfunc ={'median', 'mean', 'min'})

print (pivot)
```

Output –

```
           Amount
             mean median   min
Category
Fruit      425.25  497.0  90.0
Vegetable  299.25  254.5  62.0
```

Get the Mean, Median, Minimum sale by product

```
# creating pivot table of Mean, Median,

# Minimum sale by product aggfunc = {'median',

# 'mean', 'min'} will get median, mean and

# minimum of sales respectively

pivot = df.pivot_table(index =['Product'], values =['Amount'],
```

aggfunc ={'median', 'mean', 'min'})

print (pivot)

Output:

```
              Amount
                mean median    min
Product
Banana    363.666667  384.0   90.0
Beans     626.000000  626.0  626.0
Broccoli  150.500000  150.5   62.0
Carrots   270.000000  270.0  270.0
Orange    610.000000  610.0  610.0
```

Output:

Aim: Program using Pandas to Vectorized String Operations.

Program: Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes

```
import pandas as pd
data=[ ‵ peter’,‘Paul’,‘None’,‘MARY’,‘gUIDO’]
names = pd.Series(data)
names//printing names
0   peter
1    Paul
2    None
3    MARY
4   gUIDO
dtype: object
```

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()
0   Peter
1    Paul
2    None
3    Mary
4   Guido
dtype: object
```

Tables of Pandas String Methods

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()
zfill()	index()	isalpha()	split()
strip()	rindex()	isdigit()	rsplit()
rstrip()	capitalize()	isspace()	partition()
lstrip()	swapcase()	istitle()	rpartition()

```
monte.str.lower()
0   graham chapman
1      john cleese
```

```
2    terry gilliam
3        eric idle
4      terry jones
5    michael palin
dtype: object
monte.str.len()
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
monte.str.startswith('T')
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
monte.str.split()
0    [Graham, Chapman]
1       [John, Cleese]
2     [Terry, Gilliam]
3         [Eric, Idle]
4       [Terry, Jones]
5     [Michael, Palin]
dtype: object
```

Methods using regular expressions

Method	Description
match()	Call re.match() on each element, returning a boolean.
extract()	Call re.match() on each element, returning matched groups as strings.
findall()	Call re.findall() on each element
replace()	Replace occurrences of pattern with some other string
contains()	Call re.search() on each element, returning a boolean
count()	Count occurrences of pattern
split()	Equivalent to str.split(), but accepts regexps
rsplit()	Equivalent to str.rsplit(), but accepts regexps

```
monte.str.extract('([A-Za-z]+)', expand=False)
0     Graham
1       John
2      Terry
3       Eric
4      Terry
5    Michael
dtype: object
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
0    [Graham Chapman]
1                  []
2     [Terry Gilliam]
3                  []
4       [Terry Jones]
5     [Michael Palin]
```

```
dtype: object
```

Vectorized item access and slicing

The get() and slice() operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using str.slice(0, 3). Note that this behavior is also available through Python's normal indexing syntax–for example, df.str.slice(0, 3) is equivalent to df.str[0:3]:

```
monte.str[0:3]
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

These get() and slice() methods also let you access elements of arrays returned by split(). For example, to extract the last name of each entry, we can combine split() and get():

```
monte.str.split().str.get(-1)
0    Chapman
1     Cleese
2    Gilliam
3       Idle
4      Jones
5      Palin
dtype: object
```

WEEK 8

Aim: Program using Pandas to Working with Time Series
Example: Visualizing Seattle Bicycle Counts.

Program:

```
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',

    '2015-07-04', '2015-08-04'])

data = pd.Series([0, 1, 2, 3], index=index)
data
```

Output:

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
data['2014-07-04':'2015-07-04']
```

Output:

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
dtype: int64
data['2015']
```

Output:

```
2015-07-04    2
2015-08-04    3
dtype: int64
dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                        '2015-Jul-6', '07-07-2015', '20150708'])
dates
```

output:

```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                    '2015-07-08'],
                  dtype='datetime64[ns]', freq=None)
```
dates.to_period('D')

Output:
```
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
         '2015-07-08'],
        dtype='int64', freq='D')
```
dates - dates[0]

Output:
```
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'], dtype='timedelta64[ns]', freq=No
ne)
```
pd.date_range('2015-07-03', '2015-07-10')

Output:
```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
          '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
         dtype='datetime64[ns]', freq='D')
```
Alternatively, the date range can be specified not with a start and endpoint, but with a startpoint and a number of periods:

pd.date_range('2015-07-03', periods=8)

Output:
```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
          '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
         dtype='datetime64[ns]', freq='D')
```
The spacing can be modified by altering the freq argument, which defaults to D. For example, here we will construct a range of hourly timestamps:

pd.date_range('2015-07-03', periods=8, freq='H')

Output:
```
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
          '2015-07-03 02:00:00', '2015-07-03 03:00:00',
          '2015-07-03 04:00:00', '2015-07-03 05:00:00',
          '2015-07-03 06:00:00', '2015-07-03 07:00:00'],
         dtype='datetime64[ns]', freq='H')
```
To create regular sequences of Period or Timedelta values, the very similar pd.period_range() and pd.timedelta_range() functions are useful. Here are some monthly periods:

pd.period_range('2015-07', periods=8, freq='M')

Output:
```
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
         '2016-01', '2016-02'],
        dtype='int64', freq='M')
```
And a sequence of durations increasing by an hour:

pd.timedelta_range(0, periods=10, freq='H')

Output:
```
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
           '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
          dtype='timedelta64[ns]', freq='H')
```
Week-9: Implement the python program for the following matplotlib features

Color bars. Annotation Matplotlib to Text. Histograms Scatter Plot Box plot Color bars.

```
# Python Program illustrating
# pyplot.colorbar() method
import numpy as np
import matplotlib.pyplot as plt
# Dataset
# List of total number of items purchased
# from each products
purchaseCount = [100, 200, 150, 23, 30, 50,
        156, 32, 67, 89]
# List of total likes of 10 products
likes = [50, 70, 100, 10, 10, 34, 56, 18, 35, 45]
# List of Like/Dislike ratio of 10 products
ratio = [1, 0.53, 2, 0.76, 0.5, 2.125, 0.56,
    1.28, 1.09, 1.02]
# scatterplot
plt.scatter(x=purchaseCount, y=likes, c=ratio, cmap="summer")
plt.colorbar(label="Like/Dislike Ratio", orientation="horizontal")
plt.show()
```

Annotation

```
      # Implementation of matplotlib.pyplot.annotate()
# function
import matplotlib.pyplot as plt
import numpy as np
fig, geeeks = plt.subplots()
t = np.arange(0.0, 5.0, 0.001)
s = np.cos(3 * np.pi * t)
line = geeeks.plot(t, s, lw = 2)
# Annotation
geeeks.annotate('Local Max', xy =(3.3, 1),
        xytext =(3, 1.8),
        arrowprops = dict(facecolor ='green',
                shrink = 0.05),)
geeeks.set_ylim(-2, 2)
# Plot the Annotation in the graph
plt.show()
```

Matplotlib to Text.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-10, 10, 0.01)
y = x**2
#adding text inside the plot
plt.text(-5, 60, 'Parabola $Y = x^2$', fontsize = 22)
plt.plot(x, y, c='g')
plt.xlabel("X-axis", fontsize = 15)
plt.ylabel("Y-axis",fontsize = 15)

plt.show()
```

Histograms

```
from matplotlib import pyplot as plt
import numpy as np
# Creating dataset
a = np.array([22, 87, 5, 43, 56,
      73, 55, 54, 11,
      20, 51, 5, 79, 31,
      27])
# Creating histogram
fig, ax = plt.subplots(figsize =(10, 7))
ax.hist(a, bins = [0, 25, 50, 75, 100])
# Show plot
plt.show()
```

Scatter Plots

```
import matplotlib.pyplot as plt
# dataset-1
x1 = [89, 43, 36, 36, 95, 10,
  66, 34, 38, 20]
y1 = [21, 46, 3, 35, 67, 95,
  53, 72, 58, 10]
# dataset2
x2 = [26, 29, 48, 64, 6, 5,
  36, 66, 72, 40]
```

```
y2 = [26, 34, 90, 33, 38,
  20, 56, 2, 47, 15]
plt.scatter(x1, y1, c ="pink",
      linewidths = 2,
      marker ="s",
      edgecolor ="green",
      s = 50)
plt.scatter(x2, y2, c ="yellow",
      linewidths = 2,
      marker ="^",
      edgecolor ="red",
      s = 200)
```

<u>Box plot</u>

```
# Import libraries
import matplotlib.pyplot as plt
import numpy as np
# Creating dataset
np.random.seed(10)
data = np.random.normal(100, 20, 200)
fig = plt.figure(figsize =(10, 7))
# Creating plot
plt.boxplot(data)
# show plot
plt.show()
```

10. Write the python program to implement various sub packages of Scipy.

SciPy in Python is an open-source library used for solving mathematical, scientific, engineering, and technical problems. It allows users to manipulate the data and visualize the data using a wide range of high-level Python commands. SciPy is built on the Python NumPy extention.

<u>scipy constants</u>

```
from scipy import constants
print(constants.minute)      #60.0
print(constants.hour)        #3600.0
print(constants.day)         #86400.0
print(constants.week)        #604800.0
print(constants.year)        #31536000.0
print(constants.Julian_year) #31557600.0
from scipy import constants
print(constants.inch)              #0.0254
print(constants.foot)              #0.30479999999999996
print(constants.yard)              #0.9143999999999999
print(constants.mile)              #1609.3439999999998
print(constants.mil)               #2.5399999999999997e-05
print(constants.pt)                #0.00035277777777777776
print(constants.point)             #0.00035277777777777776
print(constants.survey_foot)       #0.3048006096012192
print(constants.survey_mile)       #1609.3472186944373
print(constants.nautical_mile)     #1852.0
print(constants.fermi)             #1e-15
print(constants.angstrom)          #1e-10
print(constants.micron)            #1e-06
print(constants.au)                #149597870691.0
print(constants.astronomical_unit) #149597870691.0
print(constants.light_year)        #9460730472580800.0
print(constants.parsec)            #3.0856775813057292e+16
```

<u>Sci py linear Algebra</u>

```
#import numpy library
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,8]])
# importing linalg function from scipy
from scipy import linalg

# Compute the determinant of a matrix
linalg.det(A)
eigen_values, eigen_vectors = linalg.eig(A)
```

```
print(eigen_values)
print(eigen_vectors)
```

Scipy-Integration

```
import scipy.integrate
f= lambda x:np.exp(-x**2)
# print results
i = scipy.integrate.quad(f, 0, 1)
print(i)
```

Week11: Write a Python program to create a parent class and child class along with their own methods. Access parent class members in child class to implement the following sceneries.

a) Constructors & destructors

b) Polymorphism

Example:

Create a class ATM and define ATM operations to create account, deposit, check_balance, withdraw and delete account. Use constructor to initialize members.

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the __init__() method is called the constructor and is always called when an object is created.

```
    class Addition:
        first = 0
        second = 0
        answer = 0
        # parameterized constructor
        def __init__(self, f, s):
                self.first = f
                self.second = s
        def display(self):
                print("First number = " + str(self.first))
                print("Second number = " + str(self.second))
                print("Addition of two numbers = " + str(self.answer))
        def calculate(self):
                self.answer = self.first + self.second
    # creating object of the class
    # this will invoke parameterized constructor
    obj = Addition(1000, 2000)
    # perform Addition
    obj.calculate()
    # display result
    obj.display()
```

# Python program to illustrate destructor

Destructors are called when an object gets destroyed. The **__del__()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted. The destructor was called **after the program ended** or when all the references to object are deleted i.e when the reference count becomes zero, not when object went out of scope

```
class Employee:

        # Initializing
        def __init__(self):
                print('Employee created')

        # Calling destructor
        def __del__(self):
                print("Destructor called")
```

```
def Create_obj():
        print('Making Object...')
        obj = Employee()
        print('function end...')
        return obj
print('Calling Create_obj() function...')
obj = Create_obj()
print('Program End...')
```

What is Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

```
class Bird:
def intro(self):
        print("There are many types of birds.")
def flight(self):
        print("Most of the birds can fly but some cannot.")
class sparrow(Bird):
def flight(self):
        print("Sparrows can fly.")
class ostrich(Bird):
def flight(self):
        print("Ostriches cannot fly.")
obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()
obj_bird.intro()
obj_bird.flight()
obj_spr.intro()
obj_spr.flight()
obj_ost.intro()
obj_ost.flight()
# Python program to create Bankaccount class
# with both a deposit() and a withdraw() function
class Bank_Account:
        def __init__(self):
                self.balance=0
                print("Hello!!! Welcome to the Deposit & Withdrawal Machine")
        def deposit(self):
                amount=float(input("Enter amount to be Deposited: "))
                self.balance += amount
                print("\n Amount Deposited:",amount)
        def withdraw(self):
                amount = float(input("Enter amount to be Withdrawn: "))
                if self.balance>=amount:
                        self.balance-=amount
                        print("\n You Withdrew:", amount)
                else:
                        print("\n Insufficient balance ")
        def display(self):
                print("\n Net Available Balance=",self.balance)
# Driver code
# creating an object of class
s = Bank_Account()
```

# Calling functions with that class object
s.deposit()
s.withdraw()
s.display()

Week-12: Implement the various data cleaning steps of example data sets using python nympy and pandas

Data cleaning is the process of correcting or removing corrupt, incorrect, or unnecessary data from a data set before data analysis. Here are the basic data cleaning tasks :

1. Importing Libraries
2. Input Customer Feedback Dataset
3. Locate Missing Data
4. Check for Duplicates
5. Detect Outliers
6. Normalize Casing

```
from pandas import read_csv
from numpy import set_printoptions
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
import pandas as pd
# load data
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pd.read_csv('https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-
diabetes.csv', names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
test = SelectKBest(score_func=f_classif, k=4)
fit = test.fit(X, Y)
# summarize scores
set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X)
# summarize selected features
print(features[0:5,:])
# Feature Extraction with RFE
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
# feature extraction
model = LogisticRegression(solver='lbfgs')
rfe = RFE(model,n_features_to_select=3)
fit = rfe.fit(X, Y)
print("Num Features: %d" % fit.n_features_)
print("Selected Features: %s" % fit.support_)
print("Feature Ranking: %s" % fit.ranking_)
# Feature Extraction with PCA
import numpy
from sklearn.decomposition import PCA
# feature extraction
pca = PCA(n_components=3)
fit = pca.fit(X)
# summarize components
print("Explained Variance: %s" % fit.explained_variance_ratio_)
print(fit.components_)
```

Week13: Implement the feature selection of data set using appropriate sklearn libraries.

```
# mount the data sets
from google.colab import drive
drive.mount("/content/gdrive")
#Importing Libraries and data set
import pandas as pd
data=pd.read_csv('/content/gdrive/My Drive/Datasets/feedback.csv')
data
# Locate Missing Data
data.isnull()
```

```
# count the missing values
data.isnull().sum()
# Drop the data with higher missing values
remove = ['Review ID','Date']
data.drop(remove, inplace =True, axis =1)
data
# Input missing data
data['Review'] = data['Review'].fillna('No review')
data
#Check for Duplicates
data.duplicated()
# drop duplicates
data.drop_duplicates()
#Detect Outliers
data['Rating'].describe()
data.loc[10,'Rating'] = 1
data
#Normalize Casing
data['Review Title'] = data['Review Title'].str.lower()
data
#Normalize Casing
data['Customer Name'] = data['Customer Name'].str.title()
data
```