Python for Data Scientists
L14 : Data science libraries

Part 2

Numerical Python Pandas

Pandas?

Pandas contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.

- It provides built-in data structures which simplify the manipulation and analysis of data sets.

- http://pandas.pydata.org/pandas-docs/stable/

Pandas?

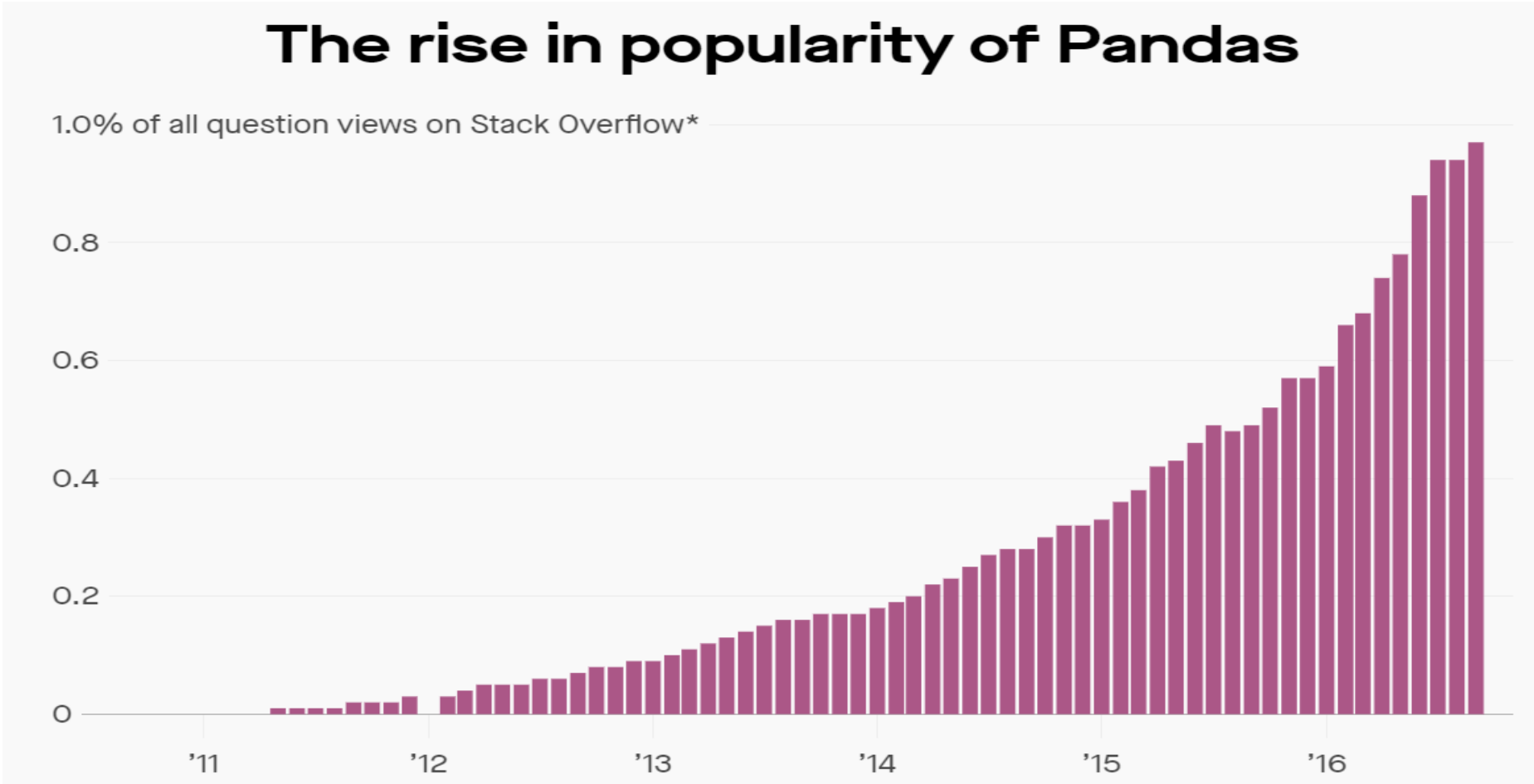
Example: explore a dataset stored in a CSV

→Pandas will extract the data from that CSV into a DataFrame (a table)

→ You can :

- Calculate statistics and answer questions about the data by doing things like removing missing values and filtering rows or columns by some criteria (data cleaning)
- Visualize the data with help from Matplotlib.
- Store the cleaned, transformed data back into a CSV, other file or database

Pandas



https://theatlas.com/charts/rJ9sZ5syf

Pandas vs NumPy

While Pandas adopts many coding idioms from NumPy, the difference is :

- Pandas is designed for working with tabular or heterogenous data.

- NumPy is best suited for working with homogeneous numerical array data

Pandas Data Structures

- A Series is a named Python list (dict with list as value).
  { ‘grades’ : [50,90,100,45] }

- A DataFrame is a collection of Series (dict of series):
  { { ‘names’ : [‘bob’,’ken’,’art’,’joe’]}
  { ‘grades’ : [50,90,100,45] }
  }

Series

Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.).

The axis labels are collectively called *index*.

Series

	NAME	AGE	DESIGNATION
1	a	20	VP
2	b	27	CEO
3	c	35	CFO
4	d	55	VP
5	e	18	VP
6	f	21	CEO
7	g	35	MD

Series

Pandas Series

https://towardsdatascience.com/pandas-series-a-lightweight-intro-b7963a0d62a2

Series

Example

```
import pandas as pd

obj1 = pd.Series([1,5,9,0])
print(obj1)

obj2 = pd.Series([1,5,9,0],
index=['a','b','c','d'])
print(obj2)


print(obj2.values)
print(obj2.index)
```

```
0   1
1   5
2   9
3   0
dtype: int64

a   1
b   5
c   9
d   0
dtype: int64

[1 5 9 0]
Index(['a', 'b', 'c', 'd'], dtype='object')
```

Series

A Pandas Series can be created out of a :

- Python list
- NumPy array

```
import pandas as pd
import numpy as np

series_list = pd.Series([1,2,3,4,5,6])
series_np = pd.Series(np.array([1,2,3,4,5,6]))
```

```
0  1
1  2
2  3
3  4
4  5
5  6
```

Series will always contain data of the same type. This makes NumPy array a better candidate for creating a pandas series

Series: indexing

Compared to NumPy arrays, labels in the index can be used to select single or a set of values:

```
import pandas as pd

obj1 = pd.Series([3,7,1,8], index=['a','b','c','d'])

print(obj1['c'])

print(obj1[['d', 'b', 'a']])
```

```
a  3
b  7
c  1
d  8
dtype: int64

1

d  8
b  7
a  3
dtype: int64
```

Series: functions

NumPy functions are applicable:

```
import pandas as pd
import numpy as np

obj1 = pd.Series([3,7,1,8], index=['a','b','c','d'])

print(obj1>3)

print(obj1/2)

print(np.sqrt(obj1))
```

```
a    3
b    7
c    1
d    8
dtype: int64

a    False
b     True
c    False
d     True
dtype: bool

a    1.5
b    3.5
c    0.5
d    4.0
dtype: float64

a    1.732051
b    2.645751
c    1.000000
d    2.828427
dtype: float64
```

Series: mapping

Series: mapping of index values to data values with a fixed length

→ Dict

```
import pandas as pd


d = {'Stockholm' : 1515017, 'Gothenburg' :599011,
'Malmö': 316588, 'Uppsala': 160462}

obj1 = pd.Series(d)
print(obj1)

cities={'Stockholm', 'Gothenburg' , 'Malmö',
'Uppsala'}
obj2= pd.Series(d, index=cities)
print(obj2)
```

```
Stockholm    1515017
Gothenburg    599011
Malmö         316588
Uppsala       160462
dtype: int64
```

```
Stockholm    1515017
Gothenburg    599011
Malmö         316588
Uppsala       160462
dtype: int64
```

Series: mapping

```
import pandas as pd


d = {'Stockholm' : 1515017, 'Gothenburg' :599011,
'Malmö': 316588, 'Uppsala': 160462}

cities={'Linköping', 'Malmö','Stockholm' ,
'Uppsala'}
obj2= pd.Series(d, index=cities)
print(obj2)
```

```
Uppsala      160462.0
Linköping         NaN
Malmö        316588.0
Stockholm   1515017.0
dtype: float64
```

To detect missing data, you can use the isnull and notnull functions: pd.isnull(obj)

Series: Attributes

Series object can have name attribute:

```
import pandas as pd


d = {'Stockholm' : 1515017, 'Gothenburg' :599011,
'Malmö': 316588, 'Uppsala': 160462}


obj1= pd.Series(d)
obj1.name = 'population'
obj1.index.name= 'cities'
print(obj1)
```

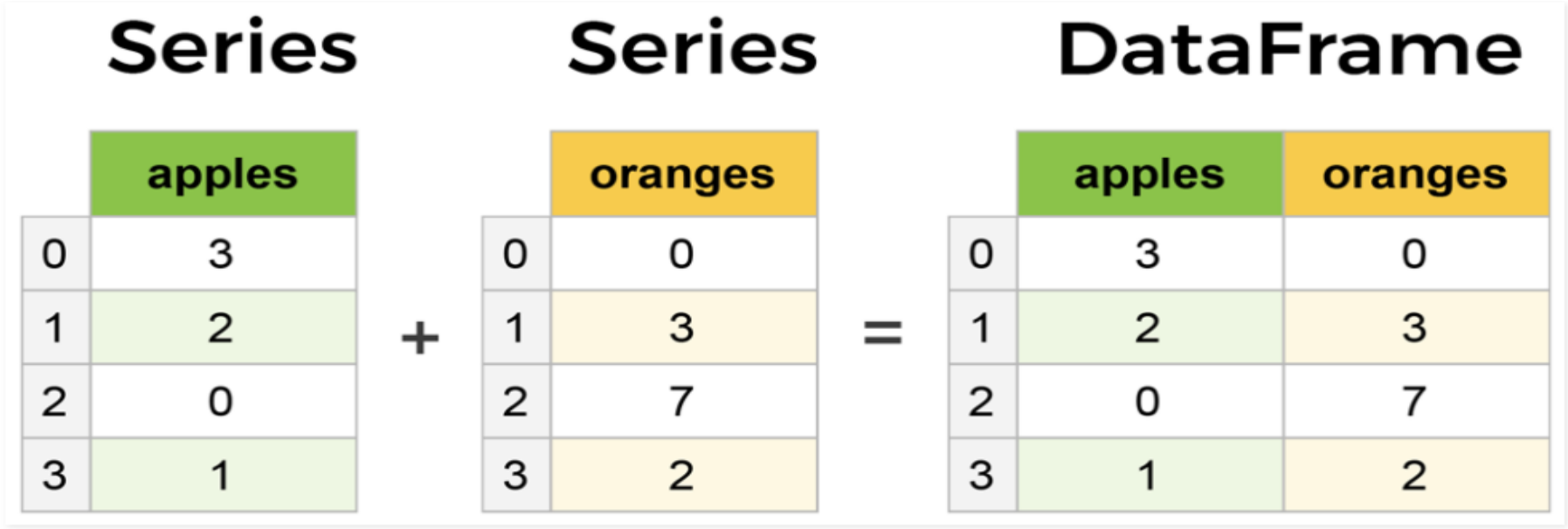
```
cities
Stockholm    1515017
Gothenburg    599011
Malmö        316588
Uppsala      160462
Name: population, dtype: int64
```

DataFrame

- It is a 2-dimensional labeled data structure with columns of potentially different types.

- It has both a row and column index.

DataFrame



https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/

DataFrame: from dictionary

```
import pandas as pd

d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
   'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}


df1 = pd.DataFrame(d)
print(df1)

df2 = pd.DataFrame(d, index=['d', 'b', 'a'])
print(df2)

df3 = pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
print(df3)
```

```
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

DataFrame: from nested dictionary of dictionaries

For the nested dict, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices

You can also swap rows and columns with T (transpose)

```
import pandas as pd

nd = {'Stockholm' : {2019: 1500000,
2020:1515017}, 'Gothenburg' :{2019: 580945,
2020:599011}, 'Malmö': {2019: 314879, 2020:
316588}}

df = pd.DataFrame(nd)
print(df)
```

	Stockholm	Gothenburg	Malmö
2019	1500000	580945	314879
2020	1515017	599011	316588

```
df1= df.T
print(df1)
```

	2019	2020
Stockholm	1500000	1515017
Gothenburg	580945	599011
Malmö	314879	316588

DataFrame: from dictionary of ndarrays/lists

```
import pandas as pd
d = {'one': [1., 2., 3., 4.],'two': [4., 3., 2., 1.]}
df1= pd.DataFrame(d)
print(df1)
```

```
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
```

```
d2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df2= pd.DataFrame(d2)
print(df2)
```

```
   a   b   c
0  1   2   NaN
1  5  10  20.0
```

DataFrame: viewing data

```
import pandas as pd

d = {'one': pd.Series([1., 2., 3., 4., 5., 6.],
index=['a', 'b', 'c', 'd', 'e', 'f']),
     'two': pd.Series([1., 2., 3., 4., 5., 6., 7.],
index=['a', 'b', 'c', 'd', 'e', 'f', 'g'])}

df1 = pd.DataFrame(d)
print(df1.head(2))
print(df1.tail(2))
print(df1.info)
print(df1.shape)
```

```
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  4.0  4.0
e  5.0  5.0
f  6.0  6.0
g  NaN  7.0
```

head: returns only the first five rows
tail: returns the five last raws
Info: returns info about your data
Shape: returns a tuple of (rows, columns)

```
   one  two
a  1.0  1.0
b  2.0  2.0

   one  two
f  6.0  6.0
g  NaN  7.0

<bound method
DataFrame.info of
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  4.0  4.0
e  5.0  5.0
f  6.0  6.0
g  NaN  7.0>

(7, 2)
```

DataFrame: viewing data

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

DataFrame: Methods

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

DataFrame: Essential Functionality

Column selection

```
import pandas as pd

d = {'one': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e']),
   'two': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e'])}


df1 = pd.DataFrame(d)

s1= df1.one
print(type(s1),s1)

s2= df1['one']
print(type(s1),s2)
```

```
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  4.0  4.0
e  5.0  5.0
```

```
<class 'pandas.core.series.Series'>
a   1.0
b   2.0
c   3.0
d   4.0
e   5.0
Name: one, dtype: float64

<class 'pandas.core.series.Series'>
a   1.0
b   2.0
c   3.0
d   4.0
e   5.0
Name: one, dtype: float64
```

DataFrame: Essential Functionality

Column selection

```
import pandas as pd

d = {'one': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e']),
        'two': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e'])}


df1 = pd.DataFrame(d)
print(df1[2:4])
print(df1[3:])
print(df1['c':'e'])
print(df1[df1>3])
```

```
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  4.0  4.0
e  5.0  5.0
```

```
   one  two
c  3.0  3.0
d  4.0  4.0

   one  two
d  4.0  4.0
e  5.0  5.0

   one  two
c  3.0  3.0
d  4.0  4.0
e  5.0  5.0

   one  two
a  NaN  NaN
b  NaN  NaN
c  NaN  NaN
d  4.0  4.0
e  5.0  5.0
```

DataFrame: Essential Functionality

Column addition

```
import pandas as pd


d = {'one': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e']),
   'two': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e'])}


df = pd.DataFrame(d, columns=['one', 'two',
'three', 'four'])
print(df)

df['three'] = df['one'] * df['two']
df['four'] = df['one'] > 2
print(df)
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	4.0	4.0
e	5.0	5.0

	one	two	three	four
a	1.0	1.0	NaN	NaN
b	2.0	2.0	NaN	NaN
c	3.0	3.0	NaN	NaN
d	4.0	4.0	NaN	NaN
e	5.0	5.0	NaN	NaN

	one	two	three	four
a	1.0	1.0	1.0	False
b	2.0	2.0	4.0	False
c	3.0	3.0	9.0	True
d	4.0	4.0	16.0	True
e	5.0	5.0	25.0	True

DataFrame: Essential Functionality

Column deletion:

Columns can be deleted or popped like with a dict

```
import pandas as pd

d = {'one': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e']),
   'two': pd.Series([1., 2., 3., 4., 5.],
index=['a', 'b', 'c', 'd', 'e'])}

df = pd.DataFrame(d, columns=['one', 'two',
'three', 'four'])

del df['three']
df.pop('four')

print(df)
```

```
one  two three four
a  1.0  1.0   NaN  NaN
b  2.0  2.0   NaN  NaN
c  3.0  3.0   NaN  NaN
d  4.0  4.0   NaN  NaN
e  5.0  5.0   NaN  NaN
```

```
 one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  4.0  4.0
e  5.0  5.0
```

DataFrame: Essential Functionality

Reindexing

→ Create a new object with the data conformed to a new index

```
import pandas as pd

d = {'Stockholm' : 1515017, 'Gothenburg' :599011,
'Malmö': 316588, 'Uppsala': 160462}
cities=['Stockholm', 'Gothenburg', 'Malmö',
'Uppsala']

df1= pd.Series(d, index=cities)
print(df1)

df2 = df1.reindex(['Stockholm', 'Gothenburg',
'Malmö', 'Uppsala', 'Linköping'])
print(df2)
```

```
Stockholm    1515017
Gothenburg    599011
Malmö         316588
Uppsala       160462
dtype: int64

Stockholm    1515017.0
Gothenburg    599011.0
Malmö         316588.0
Uppsala       160462.0
Linköping          NaN
dtype: float64
```

Data Frames: *groupby* method

→ Split the data into groups based on some criteria
→ Calculate statistics (or apply a function) to each group

```
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',
   'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],
   'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
   'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
   'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

print (df)

grouped = df.groupby('Year')
for name,group in grouped:
   print (name)
   print (group)
```

	Team	Rank	Year	Points
0	Riders	1	2014	876
1	Riders	2	2015	789
2	Devils	2	2014	863
…				
9	Royals	4	2014	701
10	Royals	1	2015	804
11	Riders	2	2017	690

2014

	Team	Rank	Year	Points
0	Riders	1	2014	876
2	Devils	2	2014	863
4	Kings	3	2014	741
9	Royals	4	2014	701

2015

	Team	Rank	Year	Points
1	Riders	2	2015	789
3	Devils	3	2015	673
5	kings	4	2015	812
10	Royals	1	2015	804
…				

Data Frames: *groupby* method Poll

What will be the output of this code?

```
import pandas as pd
data1 = {'Name': ['Adam', 'Anna', 'Alexander', 'Ahmed',
                  'Gustaf', 'Lenna', 'Eya', 'Dag'],
         'Age': [27, 24, 27, 32,
                 33, 36, 27, 32],
         'Qualification': ['Msc', 'MA', 'MCA', 'Phd',
                           'B.Tech', 'B.com', 'Msc', 'Phd']}
df = pd.DataFrame(data1)
print(df)

grouped=df.groupby(['Age', 'Qualification'])
for name,group in grouped:
        print (name)
        print (group)
```

A- Grouped by age and then qualification

B- Grouped by both of them

C- None of the above

```
    Name  Age Qualification
0   Adam   27         Msc
1   Anna   24          MA
2  Alexander   27        MCA
3   Ahmed   32         Phd
4  Gustaf   33      B.Tech
5   Lenna   36       B.com
6    Eya   27         Msc
7    Dag   32         Phd
```

```
(24, 'MA')
  Name  Age Qualification
1  Anna   24          MA

(27, 'MCA')
      Name  Age Qualification
2  Alexander   27        MCA

(27, 'Msc')
  Name  Age Qualification
0 Adam   27         Msc
6  Eya   27         Msc

(32, 'Phd')
   Name  Age Qualification
3 Ahmed   32         Phd
7   Dag   32         Phd

(33, 'B.Tech')
    Name  Age Qualification
4  Gustaf   33      B.Tech

(36, 'B.com')
   Name  Age Qualification
5 Lenna   36       B.com
```

Data Frames: *groupby* method

groupby performance notes:

- no grouping/splitting occurs until it's needed. Creating the *groupby* object only verifies that you have passed a valid mapping

- by default the group keys are sorted during the *groupby* operation. You may want to pass sort=False for potential speedup

Data Frames: *groupby* method

```
import pandas as pd

data1 = {'Name': ['Adam', 'Anna', 'Alexander', 'Ahmed',
                  'Gustaf', 'Lenna', 'Eya', 'Dag'],
         'Age': [27, 24, 27, 32,
                 33, 36, 27, 32],
         'Qualification': ['Msc', 'MA', 'MCA', 'Phd',
                           'B.Tech', 'B.com', 'Msc', 'Phd']}
df = pd.DataFrame(data1)
print(df)

grouped=df.groupby(['Age', 'Qualification'],sort = False)
for name,group in grouped:
        print (name)
        print (group)
```

	Name	Age	Qualification
0	Adam	27	Msc
1	Anna	24	MA
2	Alexander	27	MCA
3	Ahmed	32	Phd
4	Gustaf	33	B.Tech
5	Lenna	36	B.com
6	Eya	27	Msc
7	Dag	32	Phd

```
(27, 'Msc')
   Name  Age Qualification
0  Adam   27         Msc
6   Eya   27         Msc

(24, 'MA')
   Name  Age Qualification
1  Anna   24         MA

(27, 'MCA')
      Name  Age Qualification
2  Alexander   27         MCA

(32, 'Phd')
    Name  Age Qualification
3  Ahmed   32         Phd
7    Dag   32         Phd

(33, 'B.Tech')
    Name  Age Qualification
4  Gustaf   33      B.Tech

(36, 'B.com')
   Name  Age Qualification
5  Lenna   36       B.com
```

Arithmetic and Data Alignment

Example: Adding objects

```
import pandas as pd

s1 = pd.Series([3.4,2.5,8.9,6.7],
index=['a','b','c','d'])
s2 = pd.Series([-0.4, 3.8,-8.4,5.7,4.5],
index=['a','b','c','d','e'])

sumS1S2= s1+s2
print(sumS1S2)
```

```
a    3.0
b    6.3
c    0.5
d   12.4
e    NaN
dtype: float64
```

The internal data alignment introduces missing values in the label locations that don’t overlap.

Arithmetic and Data Alignment

Example: Adding objects

```
import pandas as pd
import numpy as np

df1=
pd.DataFrame(np.arange(9.).reshape(
(3,3)),columns=list('bcd'))
df2=
pd.DataFrame(np.arange(12.).reshape
((4,3)),columns=list('bde'))

print(df1+df2)
```

```
   b   c   d
0  0.0 1.0 2.0
1  3.0 4.0 5.0
2  6.0 7.0 8.0
   b   d   e
0  0.0 1.0 2.0
1  3.0 4.0 5.0
2  6.0 7.0 8.0
3  9.0 10.0 11.0
```

```
    b   c    d   e
0   0.0 NaN  3.0 NaN
1   6.0 NaN  9.0 NaN
2  12.0 NaN 15.0 NaN
3   NaN NaN  NaN NaN
```

With DataFrame, alignment is performed on both rows and columns

Arithmetic and Data Alignment

Example: Adding objects

```
import pandas as pd
import numpy as np

df1=
pd.DataFrame(np.arange(9.).reshape((3,3
)),columns=list('bcd'))
df2=
pd.DataFrame(np.arange(16.).reshape((4,
4)),columns=list('bcde'))

df3= df1.add(df2, fill_value=0)
print (df3)
```

	b	c	d	e
0	0.0	2.0	4.0	3.0
1	7.0	9.0	11.0	7.0
2	14.0	16.0	18.0	11.0
3	12.0	13.0	14.0	15.0

Arithmetic methods with fill values: fill with a special value, like 0, when an axis label is found in one object but not the other.

Arithmetic and Data Alignment

Other methods:

- Subtraction: sub
- Division: div
- Multiplication: mul
- Exponentiation: pow
- …

Function application and mapping

NumPy unfuncs (element-wise array methods) work with pandas objects

```
import  pandas as pd
import  numpy as np

frame = pd.DataFrame(np.random.randn(4,3),
columns=list('bde'), index=('one', 'two', 'three',
'four'))
print(frame)

f1= np.abs(frame)
print(f1)
```

```
            b         d         e
one  -0.811599  0.007935  1.661863
two  -0.752944 -0.266001  0.250081
three 1.233754 -0.285695 -0.143235
four  0.627463 -0.318166 -0.072572
```

```
            b         d         e
one   0.811599  0.007935  1.661863
two   0.752944  0.266001  0.250081
three 1.233754  0.285695  0.143235
four  0.627463  0.318166  0.072572
```

Function application and mapping

Applying a function on one-dimensional arrays to each column or row:

```
import  pandas as pd
import  numpy as np

frame = pd.DataFrame(np.random.randn(4,3),
columns=list('bde'), index=('one', 'two', 'three',
'four'))
print(frame)

fun = lambda x: x.max() - x.min()
f1= frame.apply(fun)
print(f1)
```

```
             b         d         e
one   0.449232  1.581346  2.104580
two  -0.425239 -0.750867  0.738536
three 0.232608 -0.813960  0.059527
four -0.208886 -0.312999  0.518480
```

```
b    0.874471
d    2.395306
e    2.045053
dtype: float64
```

Function application and mapping

Applying element-wise Python functions

```
import  pandas as pd
import  numpy as np

frame = pd.DataFrame(np.random.randn(4,3),
columns=list('bde'), index=('one', 'two',
'three', 'four'))
print(frame)

fun = lambda x: '%.2f' % x
f1= frame.applymap(fun)
print(f1)
```

	b	d	e
one	-0.283752	-0.987356	0.008827
two	0.068133	0.352152	0.640658
three	1.196239	-1.115327	-0.674795
four	-0.243760	0.897512	0.455745

	b	d	e
one	-0.28	-0.99	0.01
two	0.07	0.35	0.64
three	1.20	-1.12	-0.67
four	-0.24	0.90	0.46

Applymap : used to apply an element-wise functions

Sorting

- We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is returned.

- Sort by index on either axis: frame.sort_index(), frame.sort_index(axis=1)

- Sort by value: frame.sort_values()

Sorting Poll

What will be the output of this code?

```
import pandas as pd
df = pd.DataFrame({
    'col1': [2, 1, 9, 8, 7, 4],
    'col2': [0, 1, 9, 4, 2, 3],
})

df.sort_values(by='col1', ascending=False)
print(df)
```

```
   col1  col2
0     2     0
1     1     1
2     9     9
3     8     4
4     7     2
5     4     3
```

	col1	col2
2	9	9
3	8	4
4	7	2
5	4	3
0	2	0
1	1	1

Sorting Poll

What will be the output of this code?

```
import pandas as pd
df = pd.DataFrame([1, 2, 4, 3, 5], index=[100, 29, 234, 1, 150],
                  columns=['A'])
df1=df.sort_index()
print(df1)
```

	A
100	1
29	2
234	4
1	3
150	5

	A
100	1
29	2
1	3
234	4
150	5

	A
1	3
29	2
100	1
150	5
234	4

Missing values

Missing values are marked as NaN

```
import pandas as pd

# Read a dataset with missing values
flights =
pd.read_csv("http://rcs.bu.edu/examples/python/data_
analysis/flights.csv")
# Select the rows that have at least one missing
value
df1= flights[flights.isnull().any(axis=1)].head()

print(df1)
```

```
 year  month  day  dep_time  ...  air_time  distance  hour
minute
330  2013      1    1    1807.0  ...      NaN      2425  18.0
7.0
403  2013      1    1       NaN  ...      NaN      1389   NaN
NaN
404  2013      1    1       NaN  ...      NaN      1096   NaN
NaN
855  2013      1    2    2145.0  ...      NaN      1068  21.0
45.0
858  2013      1    2       NaN  ...      NaN      2475   NaN
NaN

[5 rows x 16 columns]
```

Missing values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- Missing values in GroupBy method are excluded
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default

Missing values

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Aggregation functions in Pandas

- Aggregation - computing a summary statistic about each group, i.e.
  - compute group sums or means
  - compute group sizes/counts

- Common aggregation functions:
  - min, max
  - count, sum, prod
  - mean, median, mode, mad
  - std, var

Aggregation functions in Pandas

```
df2= flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
print(df2)
```

	dep_delay	arr_delay
min	-33.000000	-75.000000
mean	9.463773	2.094537
max	1014.000000	1007.000000

Basic descriptive statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean

MatPlotlib

Matplotlib?

- Visualization library in Python for 2D plots of arrays
- Built on NumPy arrays and it can be used in python scripts, shell, web application servers and other graphical user interface toolkits.
- One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals.

Matplotlib?

- a set of functionalities similar to those of MATLAB
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

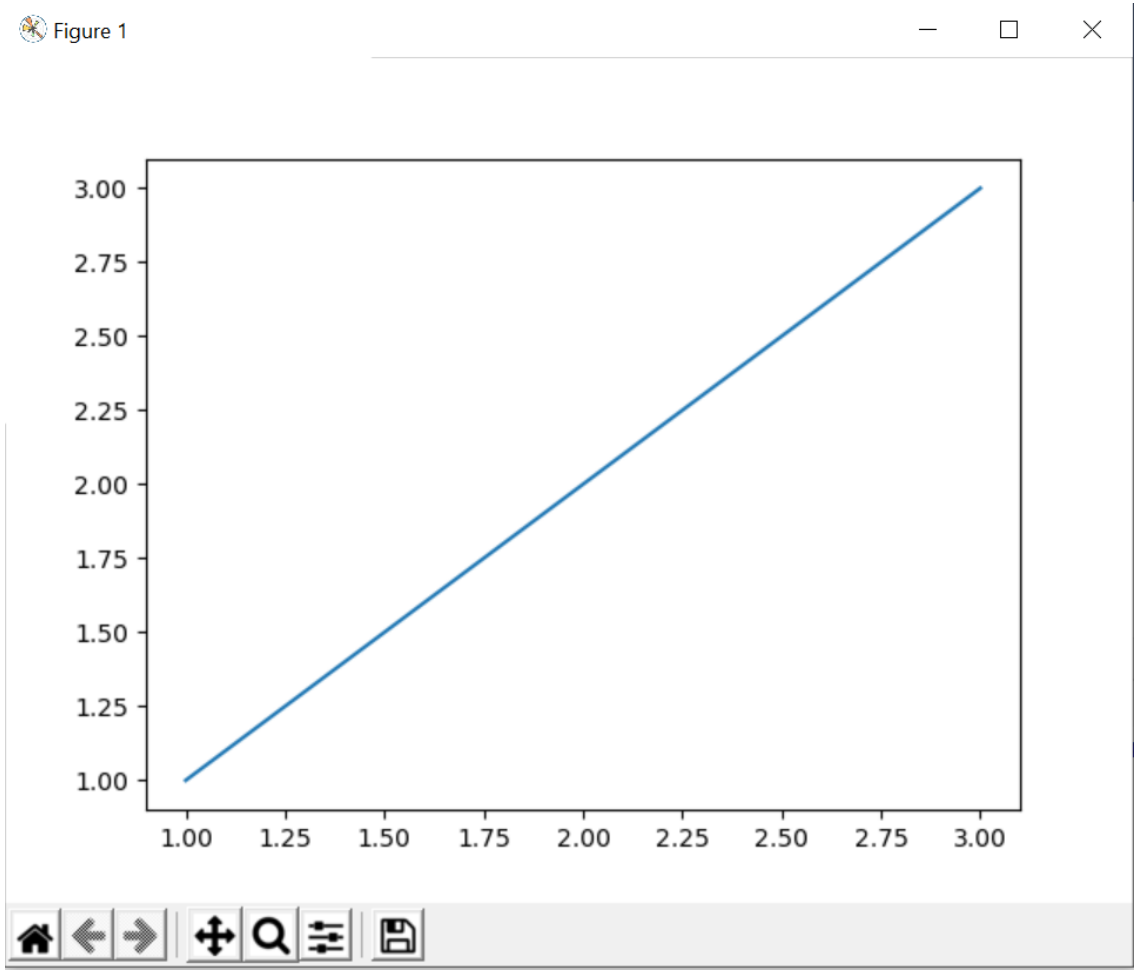
Basic plots in Matplotlib

- Matplotlib comes with a wide variety of plots.

- Plots helps to understand trends, patterns, and to make correlations.

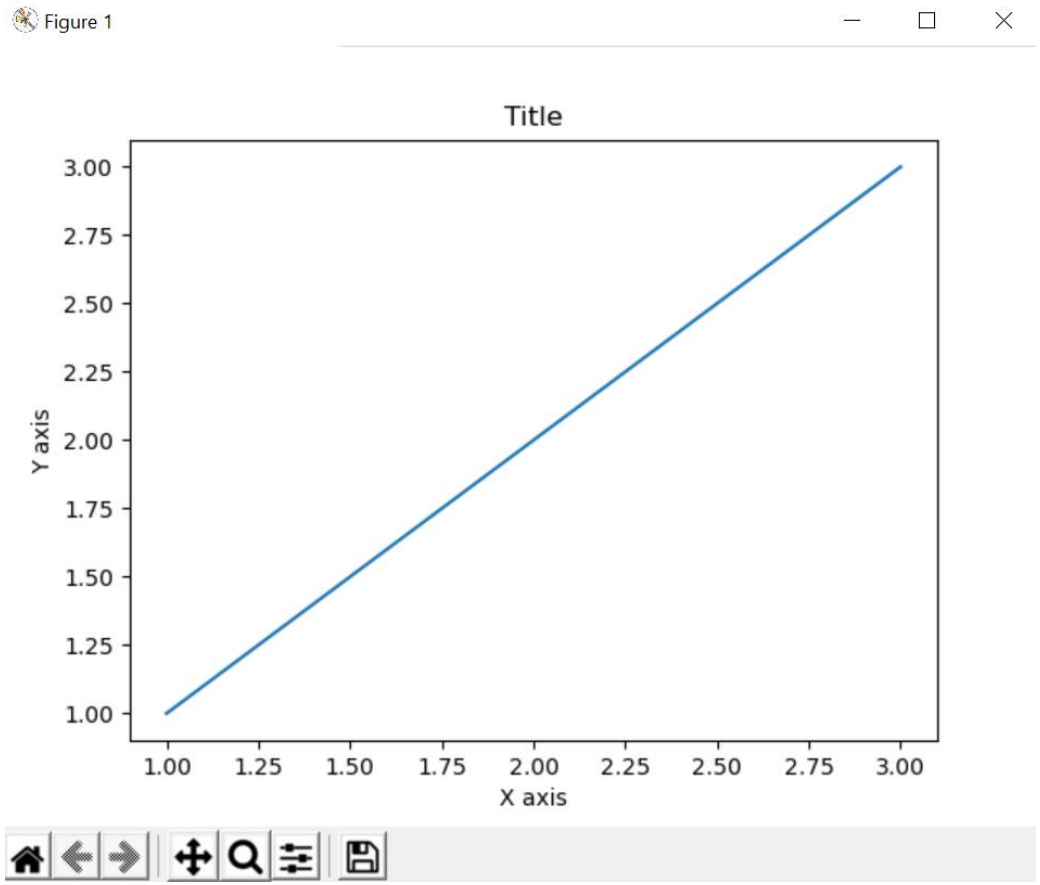
Basics of Matplotlib

```
from matplotlib import pyplot as plt

# Plotting
plt.plot([1, 2, 3], [1, 2, 3])

# Showing the plot
plt.show()
```



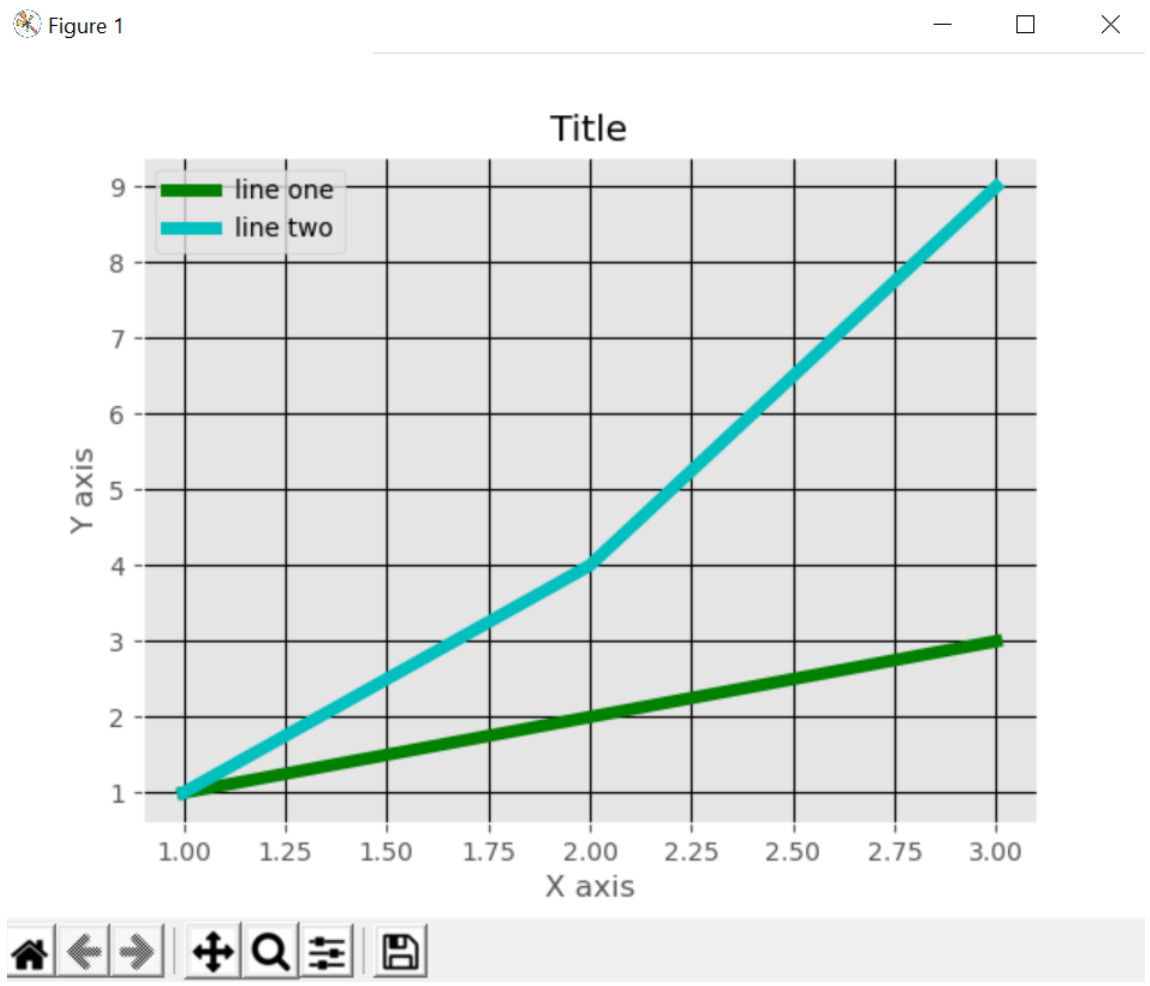
Basics of Matplotlib

```
from matplotlib import pyplot as plt

x = [1, 2, 3]
y = [1, 2, 3]
plt.plot(x, y)
plt.title('Title')
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.show()
```



Basics of Matplotlib

```
from matplotlib import pyplot as plt
from matplotlib import style

style.use('ggplot')
x = [1, 2, 3]
y = [1, 2, 3]
x2 = [1, 2, 3]
y2 = [1, 4, 9]
plt.plot(x, y, 'g', label='line one',
linewidth=5)
plt.plot(x2, y2, 'c', label='line two',
linewidth=5)
plt.title('Title')
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.legend()
plt.grid(True, color='k')
plt.show()
```



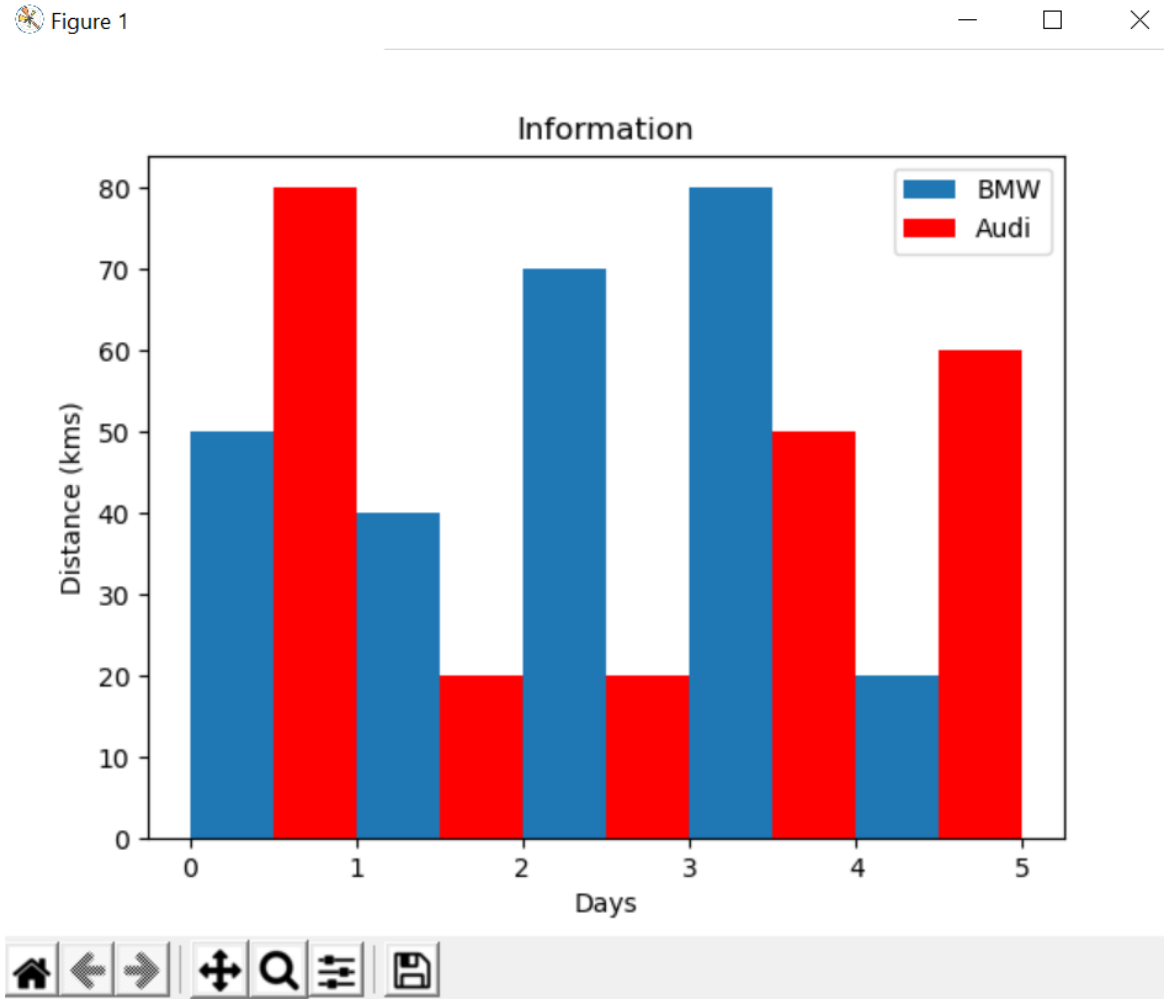
Bar plot

- Uses bars to compare data among different categories.

- It is useful when we want to measure the changes over a period of time (longer the bar, greater is the value).

Bar plot

```
from matplotlib import pyplot as plt

plt.bar([0.25, 1.25, 2.25, 3.25, 4.25],
[50, 40, 70, 80, 20],
        label="BMW", width=.5)
plt.bar([.75, 1.75, 2.75, 3.75, 4.75],
[80, 20, 20, 50, 60],
        label="Audi", color='r',
width=.5)
plt.legend()
plt.xlabel('Days')
plt.ylabel('Distance (kms)')
plt.title('Information')
plt.show()
```



https://www.edureka.co/blog/python-matplotlib-tutorial/

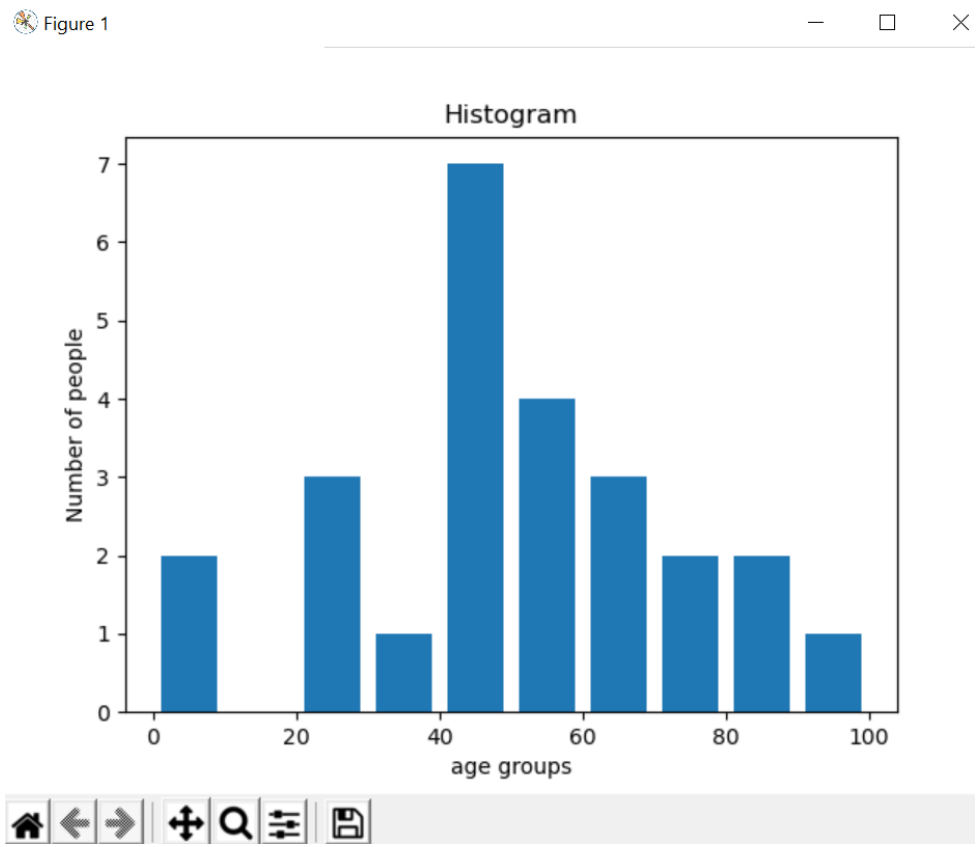
Histogram

- Histograms are used to show a distribution whereas a bar chart is used to compare different entities.

- Histograms are useful when you have arrays or a very long list.

Histogram

```
import matplotlib.pyplot as plt
population_age =
[22,55,62,45,21,22,34,42,42,4,2,102,95,85,55,110,
120,70,65,55,111,115,80,75,65,54,44,43,42,48]
bins = [0,10,20,30,40,50,60,70,80,90,100]
plt.hist(population_age, bins, histtype='bar',
rwidth=0.8)
plt.xlabel('age groups')
plt.ylabel('Number of people')
plt.title('Histogram')
plt.show()
```



https://www.edureka.co/blog/python-matplotlib-tutorial/

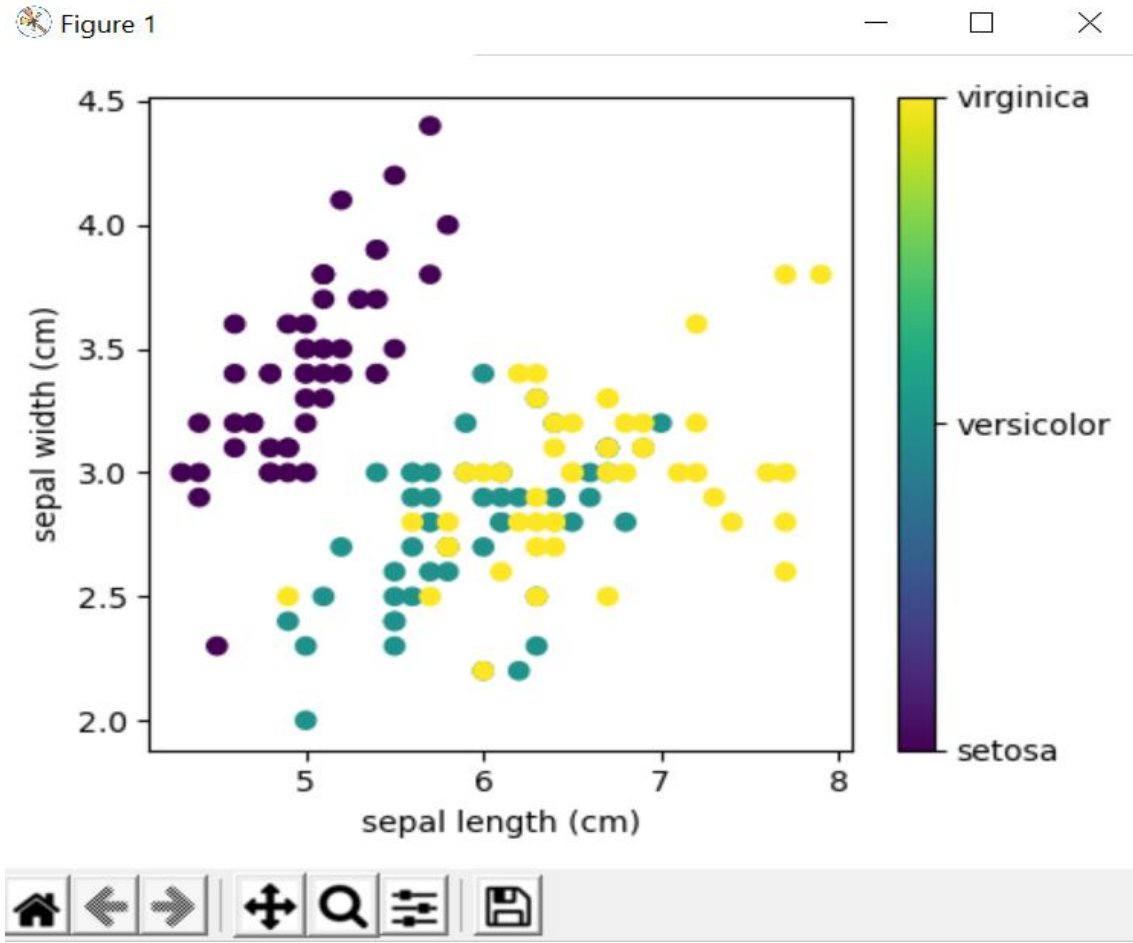
Scatter Plot

- Used to compare variables

- The data is displayed as a collection of points, each having the value of one variable which determines the position on the horizontal axis and the value of other variable determines the position on the vertical axis.

Scatter Plot

```
# Load the data
from sklearn.datasets import load_iris
iris = load_iris()
from matplotlib import pyplot as plt

# The indices of the features that we are plotting
x_index = 0
y_index = 1

# this formatter will label the colorbar with the correct
target names
formatter = plt.FuncFormatter(lambda i, *args:
iris.target_names[int(i)])

plt.figure(figsize=(5, 4))
plt.scatter(iris.data[:, x_index], iris.data[:, y_index],
c=iris.target)
plt.colorbar(ticks=[0, 1, 2], format=formatter)
plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index])
plt.tight_layout()
plt.show()
```



https://scipy-lectures.org/packages/scikit-learn/auto_examples/plot_iris_scatter.html

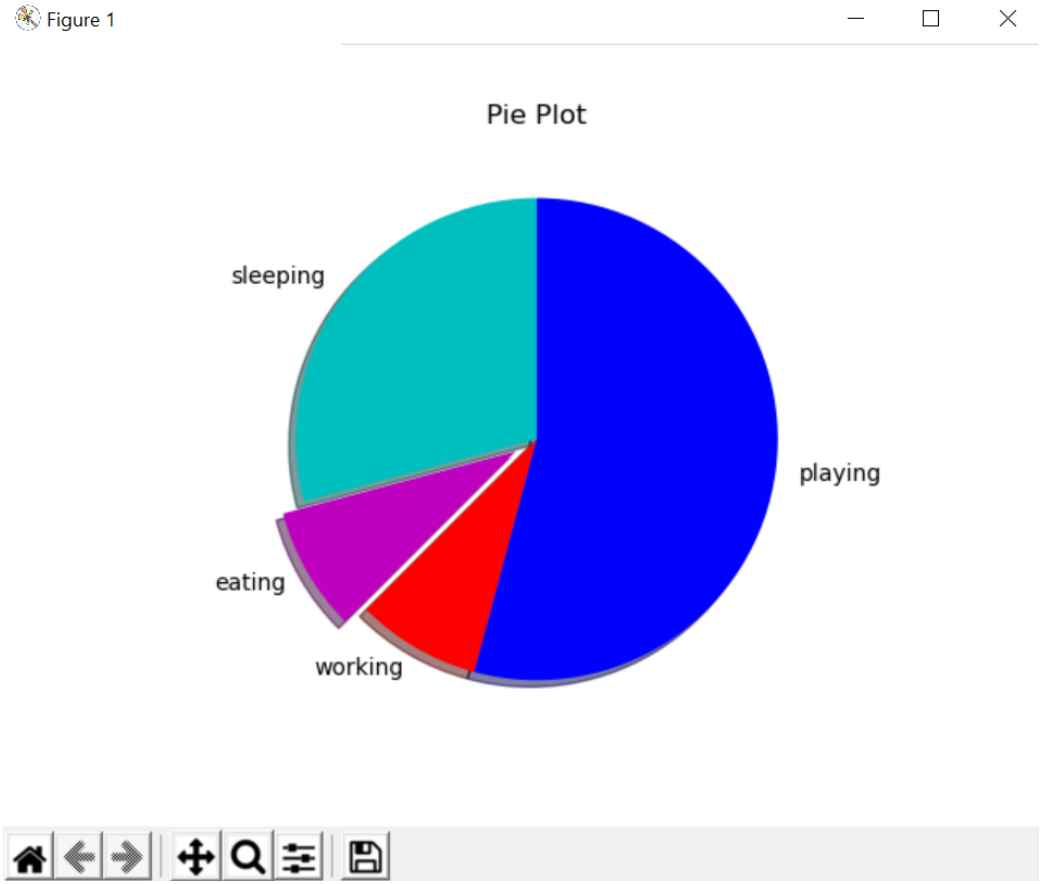
Pie Chart

- Refers to a circular graph which is broken down into segments.

- Used to show the percentage or proportional data where each slice of pie represents a category.

Pie Chart

```
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5]
sleeping = [7, 8, 6, 11, 7]
eating = [2, 3, 4, 3, 2]
working = [7, 8, 7, 2, 2]
playing = [8, 5, 7, 8, 13]
slices = [7, 2, 2, 13]
activities = ['sleeping', 'eating', 'working',
'playing']
cols = ['c', 'm', 'r', 'b']

plt.pie(slices,
        labels=activities,
        colors=cols,
        startangle=90,
        shadow=True,
        explode=(0, 0.1, 0, 0))


plt.title('Pie Plot')
plt.show()
```



https://www.edureka.co/blog/python-matplotlib-tutorial/

Multiple Plots

- Matplotlib allows us easily create multi-plots on the same figure using the .subplot() method.

- This .subplot() method takes in three parameters, namely:
  - nrows: the number of rows the Figure should have.
  - ncols: the number of columns the Figure should have.
  - plot_number : which refers to a specific plot in the Figure.

Multiple Plots

```
import matplotlib.pyplot as plt

x = [1, 2, 3]
y = [1, 2, 3]
plt.subplot(1,2,1)
plt.plot(x,y, 'red')

x2 = [1, 2, 3]
y2 = [1, 4, 9]
plt.subplot(1,2,2)
plt.plot(x,y, 'green')
plt.show()
```