In [ ]:

```
loading
cleaning
transforming
rearranging
take up 80% of an analyst's time

Data may not be in the right format
1.Handling missing data
2.Data Transformation

Missing data occurs in many applications.

NA means either data does not exist or that exists but was not observed.
when cleaning up data for analysis, do nanalysis on the missing data to identify data c
potential biases in the data caused by missing data.

NA handling methods:

1.dropna
2.fillna
3.isnull
4.notnull
```

In [4]:

```
import pandas as pd
names=pd.Series(['ramu','samu','hemu','nemu','kemu'])
print(names)
print(names.isnull())
names[0]='sita'
print(names.isnull())
```

```
0    ramu
1    samu
2    hemu
3    nemu
4    kemu
dtype: object
0    False
1    False
2    False
3    False
4    False
dtype: bool
```

In [8]:

```
# A. Filtering out missing data


#It returns the Series with only the non-null data and index values
from numpy import nan as NA
data=pd.Series([1,NA,3,5,NA,7])
print(data)
d=data.dropna()
print(d)
```

```
0    1.0
1    NaN
2    3.0
```

```
3    5.0
4    NaN
5    7.0
dtype: float64
0    1.0
2    3.0
3    5.0
5    7.0
dtype: float64
```

In [13]:

```
#Drop rows containing any NAs or those containing all NAs
df=pd.DataFrame([[1,6.5,3.],[1.,NA,NA],[NA,NA,NA],[NA,6.6,3.]])
print(df,'\n')
df1=df.dropna()
print(df1,'\n')
df2=df.dropna(how='all')
print(df2)
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.6  3.0

     0    1    2
0  1.0  6.5  3.0

     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.6  3.0
```

In [5]:

```
#Drop columns containing any NAs or those containing all NAs
import pandas as pd
from numpy import nan as NA
df=pd.DataFrame([[1.,6.5,NA],[1.,NA,NA],[NA,NA,NA],[NA,6.6,NA]])
print(df,'\n')

df2=df.dropna(axis=1)
print('\n',df2,'\n')

df3=df.dropna(axis=1,how='all')
print('\n',df3,'\n')

#keep only rows containing a certain number of observations
print(df.dropna(thresh=2))
```

```
     0    1   2
0  1.0  6.5 NaN
1  1.0  NaN NaN
2  NaN  NaN NaN
3  NaN  6.6 NaN


 Empty DataFrame
Columns: []
Index: [0, 1, 2, 3]
```

```
     0    1
0  1.0  6.5
1  1.0  NaN
2  NaN  NaN
3  NaN  6.6

     0    1   2
0  1.0  6.5 NaN
```

In [26]:

```
# B. Filling in missing data

#Rather than filtering out missing data, fill in the "holes" in number of ways
#Calling fillna with a constant replaces missing value with that value
print(df,'\n')
df1=df.fillna(0)
print(df1,'\n')

#Use a different fill value for each column
df2=df.fillna({1:0,2:5})
print(df2)
```

```
     0    1   2
0  1.0  6.5 NaN
1  1.0  NaN NaN
2  NaN  NaN NaN
3  NaN  6.6 NaN
     0    1    2
0  1.0  6.5  0.0
1  1.0  0.0  0.0
2  0.0  0.0  0.0
3  0.0  6.6  0.0
     0    1    2
0  1.0  6.5  5.0
1  1.0  0.0  5.0
2  NaN  0.0  5.0
3  NaN  6.6  5.0
```

In [8]:

```
#fillna returns a new object, but you can modify the existing object in-place
df=pd.DataFrame([[1,6.5,3.],[1.,NA,NA],[NA,NA,NA],[NA,6.6,3.]])
print(df)
df.fillna(0,inplace=True)
print(df)
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.6  3.0
     0    1    2
0  1.0  6.5  3.0
1  1.0  0.0  0.0
2  0.0  0.0  0.0
3  0.0  6.6  3.0
```

In [20]:

```
#Interpolation methods can be used with fillna:
df=pd.DataFrame([[1,6.5,3.],[1.,NA,NA],[NA,NA,NA],[NA,NA,3.]])
print(df,'\n')
df.fillna(method='ffill',inplace=True)
```

```
print(df)

df=pd.DataFrame([[1,6.5,3.],[1.,NA,NA],[NA,NA,NA],[NA,NA,3.]])
df2=df.fillna(method='ffill',limit=2)
print(df2)
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  NaN  3.0

     0    1    2
0  1.0  6.5  3.0
1  1.0  6.5  3.0
2  1.0  6.5  3.0
3  1.0  6.5  3.0
     0    1    2
0  1.0  6.5  3.0
1  1.0  6.5  3.0
2  1.0  6.5  3.0
3  1.0  NaN  3.0
```

In [23]:

```
#Pass mean or median value of a series
data=pd.Series([1.,NA,3.5,NA,7])
data.fillna(data.mean(),inplace=True)
print(data)
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

In [30]:

```
#Data Transformation

#Removing duplicates
#Duplicate rows may be found in a DataFrame
df=pd.DataFrame([[1,6.5,3.],[1.,NA,NA],[NA,NA,NA],[1,6.5,3.]])
print(df)
print(df.duplicated())
df.drop_duplicates(inplace=True)
print(df)


df=pd.DataFrame([[1,6.5,3.],[1,6.5,3.],[NA,NA,NA],[NA,6.6,3.]])
df[4]=pd.Series([1,1.0,NA,NA])
print(df)
df1=df.drop_duplicates()
print(df1)

df2=df.drop_duplicates(keep='last')
print(df2)
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
```

```
3  1.0  6.5  3.0
0    False
1    False
2    False
3     True
dtype: bool
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
     0    1    2    4
0  1.0  6.5  3.0  1.0
1  1.0  6.5  3.0  1.0
2  NaN  NaN  NaN  NaN
3  NaN  6.6  3.0  NaN
     0    1    2    4
0  1.0  6.5  3.0  1.0
2  NaN  NaN  NaN  NaN
3  NaN  6.6  3.0  NaN
     0    1    2    4
1  1.0  6.5  3.0  1.0
2  NaN  NaN  NaN  NaN
3  NaN  6.6  3.0  NaN
```

In [8]:

```
#Transforming data using a function or mapping
import pandas as pd
data=pd.DataFrame({'food':['bacon','pulled pork','bacon','pastrami','corned beef','Baco
                          'pastrami','honey ham','nova lox'],
                  'ounces':[4,3,12,6,7.5,8,3,5,6]})
print(data)

meat_to_animal={
    'bacon':'pig',
    'pulled pork': 'pig',
    'pastrami': 'cow',
    'corned beef': 'cow',
    'honey ham': 'pig',
    'nova lox': 'salmon'
}
print(meat_to_animal)

lc=data['food'].str.lower()
print(lc)
data['animal']=lc.map(meat_to_animal)
print(data)

#function can be used for this purpose
#map is a convenient way to perform element-wise transformations and other data cleanin
#print(data['food'].map(lambda x:meat_to_animal[x.lower()]))
```

```
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
```

```
8     nova lox     6.0
{'bacon': 'pig', 'pulled pork': 'pig', 'pastrami': 'cow', 'corned beef': 'cow', 'honey h
am': 'pig', 'nova lox': 'salmon'}
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

In [19]:

```
#Replacing values
import numpy as np
data=pd.Series([1.,-999.,2.,-999.,-1000.,3.])
print(data)
data.replace(-999.,np.nan,inplace=True)
print(data)

#replace multiple values
data=pd.Series([1.,-999.,2.,-999.,-1000.,3.])
data.replace([-999,-1000],np.nan,inplace=True)
print(data)

#different replacement for each value
data=pd.Series([1.,-999.,2.,-999.,-1000.,3.])
data.replace([-999,-1000],[np.nan,0],inplace=True)
#or
#argument passed can be a dict
data=pd.Series([1.,-999.,2.,-999.,-1000.,3.])
d1=data.replace({-999:np.nan,-1000:0})
print(d1)
```

```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

In [2]:

```
#5-1-2023
# Renaming Axis Indexes

#Like values in a Series, axis labels can be similarly transformed by a function or map
#ping of some form to produce new, differently labeled objects.
import pandas as pd
import numpy as np

data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   index=['Ohio', 'Colorado', 'New York'],
   columns=['one', 'two', 'three', 'four'])
print(data)
transform = lambda x: x[:4].upper()
data.index.map(transform)

#You can assign to index, modifying the DataFrame in-place:
data.index = data.index.map(transform)
data
```

```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
New York    8    9     10    11
```

Out[2]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [5]:

```
#If you want to create a transformed version of a dataset without modifying the origina
#a useful method is rename:
#rename saves you from the chore of copying the DataFrame manually and assigning #to it
#and columns attributes.

data.rename(index=str.title, columns=str.upper)

#rename can be used in conjunction with a dict-like object providing new values for a s
#the axis labels:
data.rename(index={'OHIO': 'INDIANA'},
   columns={'three': 'peekaboo'})
```

Out[5]:

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [7]:

```
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
data
```

Out[7]:

	one	two	three	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [8]:

```
#Discretization and Binning

#discretization is a process of transformation, to handle large quantities of data
#generated in sequence
#transform this data into discrete categories, for example,by dividing the range of val
#occurrence or statistics in them.
results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]

#The first contains the values between 0 and 25, the second between 26 and 50, the thir
#51 and 75, and the last between 76 and 100.
bins = [0,25,50,75,100]

cat = pd.cut(results, bins)
cat
```

Out[8]:

```
[(0, 25], (25, 50], (50, 75], (50, 75], (25, 50], ..., (75, 100], (0, 25], (25, 50], (7
5, 100], (75, 100]]
Length: 17
Categories (4, interval[int64]): [(0, 25] < (25, 50] < (50, 75] < (75, 100]]
```

In [11]:

```
#The object returned by the cut() function is a special object of Categorical type
#You can consider it as an array of strings indicating the name of the bin

cat.categories
cat.codes

#to know the occurrences for each bin, that is, how many results fall into each
#category, you have to use the value_counts() function
pd.value_counts(cat)
```

Out[11]:

```
(75, 100]    5
(0, 25]      4
(25, 50]     4
(50, 75]     4
dtype: int64
```

In [12]:
```
#You can give names to various bins by calling them first in an array of strings and th
#assigning to the labels options inside the cut() function

bin_names = ['unlikely','less likely','likely','highly likely']
pd.cut(results, bins, labels=bin_names)
```

Out[12]:
```
['unlikely', 'less likely', 'likely', 'likely', 'less likely', ..., 'highly likely', 'un
likely', 'less likely', 'highly likely', 'highly likely']
Length: 17
Categories (4, object): ['unlikely' < 'less likely' < 'likely' < 'highly likely']
```

In [13]:
```
#this will divide the range of values of the array in many intervals as specified by th
pd.cut(results, 5)
```

Out[13]:
```
[(2.904, 22.2], (22.2, 41.4], (60.6, 79.8], (41.4, 60.6], (22.2, 41.4], ..., (79.8, 99.
0], (22.2, 41.4], (41.4, 60.6], (79.8, 99.0], (79.8, 99.0]]
Length: 17
Categories (5, interval[float64]): [(2.904, 22.2] < (22.2, 41.4] < (41.4, 60.6] < (60.6,
79.8] < (79.8, 99.0]]
```

In [16]:
```
#pandas provides another method for binning: qcut().
#qcut() will ensure that the number of occurrences for each bin is equal,
#but the edges of each bin vary.  #
quintiles = pd.qcut(results, 5)
quintiles
pd.value_counts(quintiles)
#No of results not divisible by 5
```

Out[16]:
```
(2.999, 24.0]    4
(62.6, 87.0]     4
(24.0, 46.0]     3
(46.0, 62.6]     3
(87.0, 99.0]     3
dtype: int64
```

In [9]:
```
#17-1-2023

#Detecting and filtering outliers
#An Outlier is a data-item/object that deviates significantly
#from the rest of the objects.
import numpy as np
import pandas as pd
np.random.seed(12345)
data = pd.DataFrame(np.random.randn(5, 4))
data
```

Out[9]:

	0	1	2	3
0	-0.204708	0.478943	-0.519439	-0.555730
1	1.965781	1.393406	0.092908	0.281746
2	0.769023	1.246435	1.007189	-1.296221
3	0.274992	0.228913	1.352917	0.886429
4	-2.001637	-0.371843	1.669025	-0.438570

In [12]:
```
#find values in one of the columns exceeding three in magnitude
col = data[3]
col[np.abs(col) > 1]
```

Out[12]:
```
2   -1.296221
Name: 3, dtype: float64
```

In [14]:
```
#select all rows having a value exceeding 3 or -3, you can use the any method on a
#boolean DataFrame

data[(np.abs(data) > 1).any(1)]
```

Out[14]:

	0	1	2	3
1	1.965781	1.393406	0.092908	0.281746
2	0.769023	1.246435	1.007189	-1.296221
3	0.274992	0.228913	1.352917	0.886429
4	-2.001637	-0.371843	1.669025	-0.438570

In [16]:
```
#code to cap values outside the interval -3 to 3
data[np.abs(data) > 3] = np.sign(data) * 3
data
```

Out[16]:

	0	1	2	3
0	-0.204708	0.478943	-0.519439	-0.555730
1	1.965781	1.393406	0.092908	0.281746
2	0.769023	1.246435	1.007189	-1.296221
3	0.274992	0.228913	1.352917	0.886429
4	-2.001637	-0.371843	1.669025	-0.438570

In [2]:
```
#28-1-2023
#Detecting and Filtering Outliers
#Permutation
#The operations of permutation (random reordering) of a series or the rows of a
#dataframe are easy to do using the numpy.random.permutation() function

import pandas as pd
import numpy as np
df1 = pd.DataFrame(np.arange(25).reshape(5,5))
df1
```

Out[2]:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9

	0	1	2	3	4
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

In [5]:

```
new_order=np.random.permutation(5)
print(new_order)
df1.take(new_order)
```

```
[1 3 4 2 0]
```

Out[5]:

	0	1	2	3	4
1	5	6	7	8	9
3	15	16	17	18	19
4	20	21	22	23	24
2	10	11	12	13	14
0	0	1	2	3	4

In [6]:

```
#You can submit even a portion of the entire dataframe to a permutation
norder=[3,4,2]
df1.take(norder)
```

Out[6]:

	0	1	2	3	4
3	15	16	17	18	19
4	20	21	22	23	24
2	10	11	12	13	14

In [14]:

```
#Random Sampling
#Sometimes, when you have a huge dataframe, you may need to sample
#it randomly
sample = np.random.randint(0, len(df1), size=3)
print(sample)
df1.take(sample)
```

```
[0 1 3]
```

Out[14]:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
3	15	16	17	18	19

In [ ]:

```
#An Outlier is a data-item/object that deviates significantly from the rest of the obje
#When exploring data, the outliers are the extreme values within the dataset
'''For data that follows a normal distribution, the values
```

```
that fall more than three standard deviations from the mean are typically considered ou

#Load data into a dataframe
#drop the unnecessary columns
#Using pandas describe() to find outliers

Finding outliers in your data should follow a process that combines multiple techniques
Follow this plan:

.Use data visualization techniques to inspect the data’s distribution and verify the pr
.Use a statistical method to calculate the outlier data points.
.Apply a statistical method to drop or transform the outliers.

The common industry practice is to use 3 standard deviations away from the mean to diff
By using 3 standard deviations we remove the 0.3% extreme cases.
Depending on your use case, you may want to consider using 4 standard deviations which

The most common approach for removing data points from a dataset is the standard deviat

#df = df[(df[col] <= mean+(n_std*sd))]
```

In [ ]:

```
```

In [ ]:

```
```