Data Pre-Processing

Data Cleaning and Preperation is known as data pre-processing.

The major data preprocessing tasks are:

- *Handling missing values*
- *Data transformation*
- *String manipulation*

Handling Missing Values

In [1]:
```
# Why to handle missing values?
# By default all the descriptive methods in Pandas exclude missing values. Hence it is neccessary to handle missing va
lues before
# analysing further.
```

In [ ]:
```
# Let's see the doc_string of four important methods used to handle 'missing' values
```

In [7]:
```
pd.isnull?
```

In [8]:
```
pd.notnull?
```

In [14]:
```
pd.Series.fillna?
```

In [16]:
```
pd.Series.dropna?
```

In [3]:
```
# For numeric data, pandas uses the floating-point value NaN (Not a Number) to represent missing data
# This is referred as the sentinel value in Pandas
import pandas as pd
import numpy as np
ser_miss = pd.Series(['Python', 'Java', 'C', 'Ruby', np.nan])
ser_miss
```

Out[3]:
```
0    Python
1      Java
2         C
3      Ruby
4       NaN
dtype: object
```

In [4]:
```
# In Pandas 'NA' stands for not available, the a convention used in the R programming language by referring to missing data as NA
# To get the boolean object with missing values we use 'isnull' method
ser_miss.isnull()
```

Out[4]:
```
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

In [5]:
```
# The Python's built in value 'None' is also treated as missing value in Pandas
# Let's insert missing value for one of the element in ser_miss
ser_miss[0] = None
ser_miss
```

Out[5]:
```
0    None
1    Java
2       C
3    Ruby
4     NaN
dtype: object
```

In [6]:
```
# The 'isnull' method will identify the 'None' value as the 'missing' data
ser_miss.isnull()
```

Out[6]:
```
0     True
1    False
2    False
3    False
4     True
dtype: bool
```

Filtering the Missing Values

In [17]:
```
# The pandas 'dropna' method drops all the missing values and returns the non-null object
data = pd.Series([1, np.nan, 3.5, np.nan, 7])
data
```

Out[17]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [18]:
```
data.dropna()
```

Out[18]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [19]:
```
# data.notnull can be passed to object index to obtain the same
data[data.notnull()]
```

Out[19]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [31]:
```
# For DataFrame object the 'dropna' method will drop any row containing missing values
data_df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                       [3, 4, 5, 1],
                       [np.nan, np.nan, np.nan, 5],
                       [np.nan, 3, np.nan, 4]],
                       columns=list('ABCD'))
data_df
```

Out[31]:

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	5.0	1
2	NaN	NaN	NaN	5
3	NaN	3.0	NaN	4

In [32]:
```
data_df.dropna()
```

Out[32]:

	A	B	C	D
1	3.0	4.0	5.0	1

In [33]:
```
# Passing how='all' will only drop rows that are all NA
data_df.loc[2, 'D']= np.nan
data_df
```

Out[33]:

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	5.0	1.0
2	NaN	NaN	NaN	NaN
3	NaN	3.0	NaN	4.0

In [34]:
```
data_df.dropna(how='all')
```

Out[34]:

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	5.0	1.0
3	NaN	3.0	NaN	4.0

In [36]:
```
# we can drop columns in the sam way by passing 'axis=1'
data_df['E'] = np.nan
data_df
```

Out[36]:

	A	B	C	D	E
0	NaN	2.0	NaN	0.0	NaN
1	3.0	4.0	5.0	1.0	NaN
2	NaN	NaN	NaN	NaN	NaN
3	NaN	3.0	NaN	4.0	NaN

In [37]:
```
data_df.dropna(how='all', axis=1)
```

Out[37]:

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	5.0	1.0
2	NaN	NaN	NaN	NaN
3	NaN	3.0	NaN	4.0

In [42]:
```
# To keep only the rows containing certain number of observations we can use 'thresh=n':
print(data_df)
data_df.dropna(thresh=1)
```

```
     A    B    C    D   E
0  NaN  2.0  NaN  0.0 NaN
1  3.0  4.0  5.0  1.0 NaN
2  NaN  NaN  NaN  NaN NaN
3  NaN  3.0  NaN  4.0 NaN
```

Out[42]:

	A	B	C	D	E
0	*NaN*	*2.0*	*NaN*	*0.0*	*NaN*
1	*3.0*	*4.0*	*5.0*	*1.0*	*NaN*
3	*NaN*	*3.0*	*NaN*	*4.0*	*NaN*

In [41]:
```
print(data_df)
data_df.dropna(thresh=2)
```

```
     A    B    C    D   E
0  NaN  2.0  NaN  0.0 NaN
1  3.0  4.0  5.0  1.0 NaN
2  NaN  NaN  NaN  NaN NaN
3  NaN  3.0  NaN  4.0 NaN
```

Out[41]:

	A	B	C	D	E
0	*NaN*	*2.0*	*NaN*	*0.0*	*NaN*
1	*3.0*	*4.0*	*5.0*	*1.0*	*NaN*
3	*NaN*	*3.0*	*NaN*	*4.0*	*NaN*

In [43]:
```
print(data_df)
data_df.dropna(thresh=4)
```

```
     A    B    C    D   E
0  NaN  2.0  NaN  0.0 NaN
1  3.0  4.0  5.0  1.0 NaN
2  NaN  NaN  NaN  NaN NaN
3  NaN  3.0  NaN  4.0 NaN
```

Out[43]:

	A	B	C	D	E
1	3.0	4.0	5.0	1.0	NaN

Filling The Missing Values

In [44]:
```
# The pandas fillna method will fill the missing values with the specified fill value
data_df.fillna(50)
```

Out[44]:

	A	B	C	D	E
0	50.0	2.0	50.0	0.0	50.0
1	3.0	4.0	5.0	1.0	50.0
2	50.0	50.0	50.0	50.0	50.0
3	50.0	3.0	50.0	4.0	50.0

In [46]:
```
# We can use pass dictionary like object to fill different values for different columns
data_df.fillna({'A':20, 'B':50, 'C':40, 'D':60, 'E':70})
```

Out[46]:

	A	B	C	D	E
0	20.0	2.0	40.0	0.0	70.0
1	3.0	4.0	5.0	1.0	70.0
2	20.0	50.0	40.0	60.0	70.0
3	20.0	3.0	40.0	4.0	70.0

In [47]:
```
# 'fillna' method returns new object with filled values; we can use 'inplace=True' to avoid returning new object
data_df.fillna({'A':20, 'B':50, 'C':40, 'D':60, 'E':70}, inplace=True)
```

In [48]:
```
# The interpolation methods available for 'reindex' can be used with fillna method
data_rand = pd.DataFrame(np.random.randn(5, 4))
data_rand
```

Out[48]:

	0	1	2	3
0	2.195054	0.757316	1.737008	-0.366094
1	-0.197048	2.167639	-1.038619	0.584105
2	-0.735918	-1.545185	0.485349	1.377621
3	-1.818897	-0.437082	-0.156616	1.199250
4	0.875698	1.469395	-0.981955	0.790918

In [52]:
```
data_rand.iloc[2:, 2]=np.nan
data_rand.iloc[:2, 3]=np.nan
data_rand
```

Out[52]:

	0	1	2	3
0	2.195054	0.757316	1.737008	NaN
1	-0.197048	2.167639	-1.038619	NaN
2	-0.735918	-1.545185	NaN	1.377621
3	-1.818897	-0.437082	NaN	1.199250
4	0.875698	1.469395	NaN	0.790918

In [53]:
```
# Let's use 'ffill' and 'bfill' methods
data_rand.fillna(method='ffill')
```

Out[53]:

	0	1	2	3
0	2.195054	0.757316	1.737008	NaN
1	-0.197048	2.167639	-1.038619	NaN
2	-0.735918	-1.545185	-1.038619	1.377621
3	-1.818897	-0.437082	-1.038619	1.199250
4	0.875698	1.469395	-1.038619	0.790918

In [54]:
```
data_rand.fillna(method='bfill')
```

Out[54]:

	0	1	2	3
0	2.195054	0.757316	1.737008	1.377621
1	-0.197048	2.167639	-1.038619	1.377621
2	-0.735918	-1.545185	NaN	1.377621
3	-1.818897	-0.437082	NaN	1.199250
4	0.875698	1.469395	NaN	0.790918

In [55]:
```
# we can limit the number values to be 'ffill' or 'bfill' using 'limit=n', by default it is None; we can set to any ot
her n
data_rand.fillna(method='ffill', limit=2)
```

Out[55]:

	0	1	2	3
0	2.195054	0.757316	1.737008	NaN
1	-0.197048	2.167639	-1.038619	NaN
2	-0.735918	-1.545185	-1.038619	1.377621
3	-1.818897	-0.437082	-1.038619	1.199250
4	0.875698	1.469395	NaN	0.790918

In [57]:
```
ser_miss = pd.Series([1, 2, np.nan, 4, np.nan])
ser_miss
```

Out[57]:
```
0    1.0
1    2.0
2    NaN
3    4.0
4    NaN
dtype: float64
```

In [58]:
```
# for mean calculation the non null values are considered and missing values are treated as NA
ser_miss.fillna(ser_miss.mean())
```

Out[58]:
```
0    1.000000
1    2.000000
2    2.333333
3    4.000000
4    2.333333
dtype: float64
```

Data Transformation

Data tranformation includes several useful operations:

How To Remove Duplicate Rows?

In [ ]:
```
# Let's see the doc_string of 'duplicated' and 'drop_duplicates' used to work with the duplicate row values
```

In [110]:
```
pd.DataFrame.duplicated?
```

In [109]:
```
pd.DataFrame.drop_duplicates?
```

In [78]:
```
dup = pd.DataFrame({'key1': ['one', 'three'] * 3 + ['three', 'two'],
                    'key2': [1, 1, 2, 3, 3, 4, 4, 5]})
dup
```

Out[78]:

	key1	key2
0	one	1
1	three	1
2	one	2
3	three	3
4	one	3
5	three	4
6	three	4
7	two	5

In [79]:
```
# Pandas 'duplicated()': it returns a boolean object indicating each row is a duplicate or not
dup.duplicated()
```

Out[79]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
7    False
dtype: bool
```

In [80]:
```
# Pandas 'drop_duplicates()': it returns a object with duplicate row values removed
dup.drop_duplicates()
```

Out[80]:

	key1	key2
0	one	1
1	three	1
2	one	2
3	three	3
4	one	3
5	three	4
7	two	5

In [81]:
```
# By default both 'duplicated' and 'drop_duplicates' consider all the columns
dup['key3'] = np.arange(8)
dup
```

Out[81]:

	key1	key2	key3
0	one	1	0
1	three	1	1
2	one	2	2
3	three	3	3
4	one	3	4
5	three	4	5
6	three	4	6
7	two	5	7

In [82]:
```
# Let's remove duplicates by specific column
dup.drop_duplicates(['key2'])
```

Out[82]:

	key1	key2	key3
0	one	1	0
2	one	2	2
3	three	3	3
5	three	4	5
7	two	5	7

In [83]:
```
# Let's do this for column 'key3'
dup.drop_duplicates(['key3'])
```

Out[83]:

	key1	key2	key3
0	one	1	0
1	three	1	1
2	one	2	2
3	three	3	3
4	one	3	4
5	three	4	5
6	three	4	6
7	two	5	7

In [85]:
```
# Once again both the above methods consider the first observed value combination; we can consider the lastobserved va
lue combinations using
# 'keep=last'
# We can pass a single 'column' also
dup.drop_duplicates(['key1', 'key2'], keep='last')
```

Out[85]:

	key1	key2	key3
0	one	1	0
1	three	1	1
2	one	2	2
3	three	3	3
4	one	3	4
6	three	4	6
7	two	5	7

How To Transform Data Usig Functions and/or Mapping?

In [ ]:
```
# Data transformation based on the values in an array, Series or DataFrame columns can be done using 'Functions' and
 'Mapping'
# Let's see the example
```

In [95]:
```
# The map method on a Series accepts a function or dict-like object containing a mapping values or data
pd.Series.map?
```

In [99]:
```
data_tr = pd.DataFrame({'Names': ['Raja', 'vali', 'Salu',
                                  'Balu', 'Vali', 'mali'],
                        'Score': [4, 3, 2, 6, 5, 1,]})
data_tr
```

Out[99]:

	Names	Score
0	Raja	4
1	vali	3
2	Salu	2
3	Balu	6
4	Vali	5
5	mali	1

In [101]:
```
# Let's match the 'Names' of the above DataFrame with their favorite colour:
match_data_tr = {'raja':'Yellow', 'vali':'Red', 'salu':'Green', 'balu':'Green', 'mali':'Dark'}
match_data_tr
```

Out[101]:
```
{'raja': 'Yellow',
 'vali': 'Red',
 'salu': 'Green',
 'balu': 'Green',
 'mali': 'Dark'}
```

In [96]:
```
# Notice that the 'data_tr' consists of lowercase and uppercase string characters in their 'Names' column
# For that reason we use Python's string method to convert uppercase to lowercase first and then apply 'map' method
```

In [103]:
```
lower_str = data_tr['Names'].str.lower()
lower_str
```

Out[103]:
```
0    raja
1    vali
2    salu
3    balu
4    vali
5    mali
Name: Names, dtype: object
```

In [104]:
```
data_tr['Color'] = lower_str.map(match_data_tr)
data_tr
```

Out[104]:

	Names	Score	Color
0	*Raja*	*4*	*Yellow*
1	*vali*	*3*	*Red*
2	*Salu*	*2*	*Green*
3	*Balu*	*6*	*Green*
4	*Vali*	*5*	*Red*
5	*mali*	*1*	*Dark*

In [106]:
```
# we can also use 'lambda' function to do all the work at once inside the 'map' method
data_tr['Color'] = data_tr['Names'].map(lambda x: match_data_tr[x.lower()])
data_tr
```

Out[106]:

	Names	Score	Color
0	*Raja*	*4*	*Yellow*
1	*vali*	*3*	*Red*
2	*Salu*	*2*	*Green*
3	*Balu*	*6*	*Green*
4	*Vali*	*5*	*Red*
5	*mali*	*1*	*Dark*

How To Replace Values?

In [ ]:
```
# Pandas 'replace' method is a convinient method to replace a specific values with specific data
# It also provides 'limit' argument to replace the limited number of data elements
```

In [113]:
```
pd.DataFrame.replace?
```

In [114]:
```
ser_data = pd.Series([1., -9., 2., -9., -1., 3.])
ser_data
```

Out[114]:
```
0    1.0
1   -9.0
2    2.0
3   -9.0
4   -1.0
5    3.0
dtype: float64
```

In [115]:
```
# Let's replace -9 by the missing sentinel value NA or NaN
ser_data.replace(-9, np.nan)
```

Out[115]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4   -1.0
5    3.0
dtype: float64
```

In [116]:
```
# we can pass inplace=True to avoid returning new object
```

In [117]:
```
# To remove multiple values at once; pass a list of values and then the substitute value
ser_data.replace([-9, -1], np.nan)
```

Out[117]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

In [118]:
```
# To use a different replacement for each value, pass a list of substitutes
ser_data.replace([-9, -1], [100, 500])
```

Out[118]:
```
0      1.0
1    100.0
2      2.0
3    100.0
4    500.0
5      3.0
dtype: float64
```

In [119]:
```
# we can also pass dictionary object to replace the values
ser_data.replace({-9:50, -1:60})
```

Out[119]:
```
0     1.0
1    50.0
2     2.0
3    50.0
4    60.0
5     3.0
dtype: float64
```

In [ ]:
```
# Similarly you can do more on replace method: Take this as an assignment
```

How To Rename Axis Indexes?

In [ ]:
```
# Before introducing 'rename' method let's work with the 'map' method first for better understanding
```

In [122]:
```
data_tor = pd.DataFrame(np.arange(12).reshape((3, 4)),
              index=['Apple', 'Banana', 'Grapes'],
              columns=['one', 'two', 'three', 'four'])
data_tor
```

Out[122]:

	one	two	three	four
Apple	0	1	2	3
Banana	4	5	6	7
Grapes	8	9	10	11

In [125]:
```
# Let's rename the DataFrame's index labels by uppercae letters using lambda and map method
upper = lambda x: x[:5].upper()
data_tor.index.map(upper)
```

Out[125]:
```
Index(['APPLE', 'BANAN', 'GRAPE'], dtype='object')
```

In [129]:
```
# By default it returns new  index object: we can make it inplace using 'DataFrame.index' method
data_tor.index = data_tor.index.map(upper)
```

In [130]:
```
data_tor
```

Out[130]:

	one	two	three	four
APPLE	0	1	2	3
BANAN	4	5	6	7
GRAPE	8	9	10	11

In [131]:
```
# Let's see how 'rename' works
pd.DataFrame.rename?
```

In [132]:
```
# Now let's use 'rename' method to rename the index labels as lowercase type and column names to title type
data_tor.rename(index=str.lower, columns=str.title)
```

Out[132]:

	One	Two	Three	Four
apple	0	1	2	3
banan	4	5	6	7
grape	8	9	10	11

In [136]:
```
# we can use dictionary like object to rename the index and column names
# data_tor.rename(index={}, columns={})
data_tor.rename(index={'APPLE': 'Appale'}, columns={'two':2})
# we can use 'inplace=True' to avoid returning the new object
```

Out[136]:

	one	2	three	four
Appale	0	1	2	3
BANAN	4	5	6	7
GRAPE	8	9	10	11

How To Descretize and/or Bin The Data?

In [137]:
```
#### Continuous data is often descretzed or separated into small bins for analysis purpose
```

In [170]:
```
pd.cut?
```

In [171]:
```
pd.qcut?
# Quantile-based discretization function. Discretize variable into equal-sized buckets based on rank or based on sampl
e quantiles. For example-
# - 1000 values for 10 quantiles would produce a Categorical object indicating quantile membership for each data poin
t.
```

In [154]:
```
# Assume that a certain number of students scored marks between some range
s_m = [21, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
s_m
```

Out[154]:
```
[21, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

In [155]:
```
# Let's divide these into bins: 20 To 35, 36 To 45, 45 To 60 and finally 61 To end of the value
# To do this Pandas provides 'cut' method
```

In [ ]:

In [156]:
```
bin_size = [20, 35, 45, 60, 100]
```

In [157]:
```
binned = pd.cut(s_m, bin_size)
```

In [158]:
```
print(s_m)
binned
```

```
[21, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Out[158]:
```
[(20, 35], (20, 35], (20, 35], (20, 35], (20, 35], ..., (20, 35], (60, 100], (35, 45], (35, 45], (20, 35]]
Length: 12
Categories (4, interval[int64]): [(20, 35] < (35, 45] < (45, 60] < (60, 100]]
```

In [159]:
```
# internally it contains a categories array specifying the distinct category names along with a labeling for the s_m d
ata in the codes attribute:
binned.codes
```

Out[159]:
```
array([0, 0, 0, 0, 0, 0, 1, 0, 3, 1, 1, 0], dtype=int8)
```

In [160]:
```
# it has a categorical data internally
binned.categories
```

Out[160]:
```
IntervalIndex([(20, 35], (35, 45], (45, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')
```

In [161]:
```
# Let's count the values fall under different bins i.e., computing the frequency of occurences of values within bin si
ze
binned.value_counts()
# it is also reffered as bin counts of Pandas cut object
```

Out[161]:
```
(20, 35]     8
(35, 45]     3
(45, 60]     0
(60, 100]    1
dtype: int64
```

In [162]:
```
# Here the '(' represents 'the side is open' and ']' represents 'the side is closed'
# we can change this by passing 'right=False' in cut method
```

In [163]:
```
# We can also pass our own bin names by passing a list or array to the labels option
```

In [164]:
```
rebinned = pd.cut(s_m, bin_size, labels=['Fail', 'Pass', 'SC', 'FCorFCD'])
```

In [165]:
```
rebinned
```

Out[165]:
```
[Fail, Fail, Fail, Fail, Fail, ..., Fail, FCorFCD, Pass, Pass, Fail]
Length: 12
Categories (4, object): [Fail < Pass < SC < FCorFCD]
```

In [166]:
```
# we can also pass an integer number of bins to 'cut' method to bin the data into equal length based on minimum and ma
ximum values
rd = np.random.rand(10)
rd
```

Out[166]:
```
array([0.33844381, 0.84754648, 0.58312366, 0.23053346, 0.94907906,
       0.72503993, 0.80126111, 0.35455327, 0.99030319, 0.1171436 ])
```

In [169]:
```
# Let's pass integer number for bin length: here I'll pass 4 (to get 4 equal bins based on minimum and maximum values)
# 'precision=1' is passed to limit the decimal precision to 1 for better analysis
pd.cut(rd, 4, precision=1)
```

Out[169]:
```
[(0.3, 0.6], (0.8, 1.0], (0.6, 0.8], (0.1, 0.3], (0.8, 1.0], (0.6, 0.8], (0.8, 1.0], (0.3, 0.6], (0.8, 1.0], (0.1, 0.
3]]
Categories (4, interval[float64]): [(0.1, 0.3] < (0.3, 0.6] < (0.6, 0.8] < (0.8, 1.0]]
```

In [ ]:
```
# Let's see the 'Quantile' based descritization of normally distributed data
```

In [173]:
```
ran_data = np.random.randn(1000)
ran_data[:20]
```

Out[173]:
```
array([ 0.71074878,  0.30920852, -0.17024288, -1.94199464,  2.21777608,
        1.09738176,  1.76850996, -1.31775008, -1.12988772,  0.96738624,
        0.48648407,  1.00817878,  1.31828806, -1.09274611, -0.53590507,
        0.63986658,  0.19294962,  2.92242288,  2.66923232,  2.04384581])
```

In [174]:
```
# Let's cut this data into 6 equally sized data
quantiles_bins = pd.qcut(ran_data, 6)
```

In [175]:
```
quantiles_bins
```

Out[175]:
```
[(0.459, 1.04], (-0.0162, 0.459], (-0.439, -0.0162], (-3.024, -0.994], (1.04, 3.529], ..., (-0.994, -0.439], (-3.024,
-0.994], (-3.024, -0.994], (0.459, 1.04], (0.459, 1.04]]
Length: 1000
Categories (6, interval[float64]): [(-3.024, -0.994] < (-0.994, -0.439] < (-0.439, -0.0162] < (-0.0162, 0.459] < (0.4
59, 1.04] < (1.04, 3.529]]
```

In [176]:
```
# Let's see how this data is descretized roughly into 6 interval bins
quantiles_bins.value_counts()
```

Out[176]:
```
(-3.024, -0.994]     167
(-0.994, -0.439]     167
(-0.439, -0.0162]    166
(-0.0162, 0.459]     167
(0.459, 1.04]        166
(1.04, 3.529]        167
dtype: int64
```

In [177]:
```
# we can also pass our own quantiles to 'qcut' similar ot 'cut' method
```

How To Detect and Filter Outliers?

In [196]:
```
ug_data = pd.DataFrame({'A':pd.Series(np.arange(10)), 'B':pd.Series(np.arange(5, 15)),'C':pd.Series(np.arange(10, 20
)),
              'D':pd.Series(np.arange(15, 25))})
ug_data
```

Out[196]:

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

In [197]:
```
ug_data.describe()
```

Out[197]:

	A	B	C	D
count	10.00000	10.00000	10.00000	10.00000
mean	4.50000	9.50000	14.50000	19.50000
std	3.02765	3.02765	3.02765	3.02765
min	0.00000	5.00000	10.00000	15.00000
25%	2.25000	7.25000	12.25000	17.25000
50%	4.50000	9.50000	14.50000	19.50000
75%	6.75000	11.75000	16.75000	21.75000
max	9.00000	14.00000	19.00000	24.00000

In [199]:
```
col = ug_data['A']
col[np.abs(col) > 4]
```

Out[199]:
```
5    5
6    6
7    7
8    8
9    9
Name: A, dtype: int32
```

In [201]:
```
ug_data[(np.abs(ug_data) > 20).any(1)]
```

Out[201]:

	A	B	C	D
6	6	11	16	21
7	7	12	17	22
8	8	13	18	23
9	9	14	19	24

How To Reorder and Select Rondomly?

In [ ]:
```
# In order to reorder a Series or Rows of a DataFrame randomly, we can use 'numpy.random.permutation' function
```

In [205]:
```
np.random.permutation?
```

In [207]:
```
# Let's create a sample DataFrame
ran_df = pd.DataFrame(np.arange(20).reshape((5, 4)))
ran_df
```

Out[207]:

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

In [209]:
```
# Now compute the random array for index reordering
sampler = np.random.permutation(5)
sampler
```

Out[209]:
```
array([0, 4, 3, 2, 1])
```

In [217]:
```
# The 'take' function: Return the elements in the given *positional* indices along an axis
# This means that we are not indexing according to actual values in the index attribute of the object. We are indexing
according to the
# actual position of the element in the object.
pd.DataFrame.take?
```

In [210]:
```
# the array 'sampler' then can be used to reorder the DataFrame index using Pandas 'take' function
ran_df.take(sampler)
```

Out[210]:

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

In [212]:
```
# The 'sample' funtion: Return a random sample of items from an axis of object.
pd.DataFrame.sample?
```

In [214]:
```
ran_df.sample(n=2)
```

Out[214]:

	0	1	2	3
2	8	9	10	11
4	16	17	18	19

In [216]:
```
# To repeat the DataFrame or Series with the existing sample of data we can pass 'replace=True' to the 'sample' functi
on
rep = ran_df.sample(n=10, replace=True)
rep
```

Out[216]:

	0	1	2	3
0	0	1	2	3
4	16	17	18	19
3	12	13	14	15
0	0	1	2	3
0	0	1	2	3
3	12	13	14	15
4	16	17	18	19
0	0	1	2	3
0	0	1	2	3
0	0	1	2	3

How To Compute Indicator/Dummy Variables?

For statistical or machine learing applications, it is often necessary to convort a categorical varibles into a 'indicator' or 'dummy' variables matrix.

In [2]:
```
# If a column in a DataFrame has k distinct values,we can derive a matrix or DataFrame with k columns containing all 1
s and 0s.
```

In [33]:
```
import pandas as pd
import numpy as np
```

In [12]:
```
# Convert categorical variable into dummy/indicator variables
pd.get_dummies?
```

In [16]:
```
# Now first create a DataFrame for some statistical or machine learning application
data_dummies = pd.DataFrame({'key': ['b', 'a', 'a', 'c', 'a'], 'data1': range(5)})
data_dummies
```

Out[16]:

	key	data1
0	b	0
1	a	1
2	a	2
3	c	3
4	a	4

In [17]:
```
# Let's create a dummy variables
pd.get_dummies(data_dummies['key'])
```

Out[17]:

	a	b	c
0	0	1	0
1	1	0	0
2	1	0	0
3	0	0	1
4	1	0	0

In [23]:
```
# we can add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data
pd.get_dummies(data_dummies['key'], prefix='Key')
```

Out[23]:

	Key_a	Key_b	Key_c
0	0	1	0
1	1	0	0
2	1	0	0
3	0	0	1
4	1	0	0

In [24]:
```
# Let's join the 'data1' column and the 'dummies' columns to get the original DataFrame in the form of dummies DataFra
me
data_dummies_df = data_dummies[['data1']].join(pd.get_dummies(data_dummies['key'], prefix='Key'))
data_dummies_df
```

Out[24]:

	data1	Key_a	Key_b	Key_c
0	0	0	1	0
1	1	1	0	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0

In [ ]:
```
# For stastistical application we can combine 'get_dummies' with the 'pd.cut' method to apply it for decritixzed value
s
```

In [32]:
```
np.random.seed(42)
v = np.random.rand(10)
v
```

Out[32]:
```
array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864,
       0.15599452, 0.05808361, 0.86617615, 0.60111501, 0.70807258])
```

In [34]:
```
bins = [0.1, 0.3, 0.5, 0.7, 1.0]
```

In [35]:
```
pd.get_dummies(pd.cut(v, bins))
# we can observe the 6th value dummies all are zeros because 0.058.. won't belongs even 0.1-0.3 range
```

Out[35]:

	(0.1, 0.3]	(0.3, 0.5]	(0.5, 0.7]	(0.7, 1.0]
0	0	1	0	0
1	0	0	0	1
2	0	0	0	1
3	0	0	1	0
4	1	0	0	0
5	1	0	0	0
6	0	0	0	0
7	0	0	0	1
8	0	0	1	0
9	0	0	0	1

In [ ]:
```
# we can work on this in detail in the project session of this course
```

How To Manipulate With Strings?

In [36]:
```
# Python is very popular in handling 'text' and 'string' data.
# Text processing and string manipulation is very interesting topic for many applications
# Hence Pandas cooperate with the 'RegularExpressions' that makes us to work with the 'text' and 'string' with pandas
 objects easily
```

Let's see some of the 'String Object Methods' first

In [38]:
```
# In many situations built-in string methods are sufficient to work with the string manipulation and other scripting a
pplications
# Let's see the ',' separated string object
Python_sentence = 'python,Is, a programming, Language'
Python_sentence
```

Out[38]: 'python,Is, a programming, Language'

In [ ]:
```
# We can use split method to split or broken the string into a number of pieces
```

In [97]:
```
# Return a list of the words in the string, using sep as the delimiter string.
str.split?
```

In [39]:
```
#
Python_sentence.split(sep=',')
```

Out[39]: ['python', 'Is', ' a programming', ' Language']

In [ ]:
```
# split is often combined with strip to trim whitespace (including line breaks)
```

In [149]:
```
cs = [x.strip() for x in Python_sentence.split(',')]
cs
```

Out[149]: ['python', 'Is', 'a programming', 'Language']

In [152]:
```
# join: Concatenate any number of strings.
str.join?
```

In [153]:
```
':'.join(cs)
```

Out[153]: 'python:Is:a programming:Language'

In [55]:
```
# The unpacked substrings can be combined each other using any string object using 'addition' method
one, two, three, four = cs
one
```

Out[55]: 'python'

In [57]:
```
one + ':::' + two + '...>' + three + '#1' + four
```

Out[57]: 'python:::Is...>a programming#1Language'

In [59]:
```
# To detect a substring we can use Python's 'in' kerword
'python' in cs
```

Out[59]: True

In [ ]:
```
# we can also use 'index' and 'find' methods to detect a substring based on its position it appears
```

In [66]:
```
str.index?
```

In [ ]:
```
str.find?
```

In [76]:
```
# The 'index' method gives the position of the substring in first appearence
print(Python_sentence)
Python_sentence.index(',')
```

```
python,Is, a programming, Language
```

Out[76]: 6

In [69]:
```
# If the substring is not found in the string 'index' method returns ValueError
Python_sentence.index(';')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-69-7885e523a804> in <module>()
----> 1 Python_sentence.index(';')

ValueError: substring not found
```

In [70]:
```
# Find method also works similar to like 'index' method but in the absence of substring it returns '-1'
Python_sentence.find(',')
```

Out[70]: 6

In [71]:
```
Python_sentence.find(';')
```

Out[71]: -1

In [73]:
```
# we can also count the substrings in a given string using 'count' method
Python_sentence.count(',')
```

Out[73]: 3

In [74]:
```
# 'replace' method will replace another substring in place of excisting substring
Python_sentence.replace(',', ':')
```

Out[74]: 'python:Is: a programming: Language'

In [75]:
```
Python_sentence.replace(',', '')
```

Out[75]: 'pythonIs a programming Language'

Now Let's see how to work with the Regular Expressions

In [65]:
```
# Regular expressions provide a flexible way to search or match (often more complex) string patterns in text
# A single expression, commonly called a regex, is a string formed according to the regular expression language
# Python'a re module is used to apply regular expressions on strings
```

In [82]:
```
import re
```

In [91]:
```
text = "python Is \ta programming\t Language"
text
```

Out[91]: 'python Is \ta programming\t Language'

In [154]:
```
# Split the source string by the occurrences of the pattern, returning a list containing the resulting substrings.  If
# capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of t
he
# resulting list.
re.split?
```

In [92]:
```
# '\s+' : it describes the one or more whitspace characters
re.split('\s+', text)
```

Out[92]: ['python', 'Is', 'a', 'programming', 'Language']

In [93]:
```
# In order to use the regex object again and again or as a reusable object first compile the 'regex' object and then u
se
regex = re.compile('\s+')
```

In [94]:
```
regex.split(text)
```

Out[94]: ['python', 'Is', 'a', 'programming', 'Language']

In [98]:
```
# Return a list of all non-overlapping matches in the string.
re.findall?
```

In [95]:
```
regex.findall(text)
```

Out[95]: [' ', ' \t', ' ', '\t ']

In [99]:
```
# Note: To avoid unwanted escaping with \ in a regular expression, use raw string literals like r'E:\x' instead of the
equivalent 'E:\\x'
```

In [100]:
```
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
```

In [101]:
```
# # re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [104]:
```
# Let's have text form of email details
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com"""
text
```

Out[104]: 'Dave dave@google.com \nSteve steve@gmail.com \nRob rob@gmail.com \nRyan ryan@yahoo.com'

In [105]:
```
# Now we can use the compiled 'regex' to find the string that matches the string pattern
# So it results all the email addresses matching the pattern
regex.findall(text)
```

Out[105]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']

In [108]:
```
# Scan through string looking for a match, and return a corresponding match object instance.
regex.search?
```

In [106]:
```
# search returns a special match object for the first email address in the text
regex.search(text)
```

Out[106]: <re.Match object; span=(5, 20), match='dave@google.com'>

In [109]:
```
# Matches zero or more characters at the beginning of the string.
regex.match?
```

In [107]:
```
# If we observe the text object the first strings are not matchig the pattern to match; hence we will get None for thi
s example
# regex.match returns None, as it only will match if the pattern occurs at the start of the string
print(regex.match(text))
```

None

In [110]:
```
# sub will return a new string with occurrences of the pattern replaced by the a new string object.
regex.sub?
```

In [111]:
```
# Let's apply it for the 'text' object
print(regex.sub('Python', text))
```

```
Dave Python
Steve Python
Rob Python
Ryan Python
```

In [112]:
```
# in order to introduce the new string objects for the matching objects like username, domain name and domain suffix f
or the
# email address's we can use string pattern with in '()'
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
```

In [113]:
```
# Now compile the 'regex' with the pattern with case sensitive flag activated, by default it is None
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [114]:
```
# A match object produced by this modified regex returns a tuple of the pattern components with its groups method
m = regex.match('Paru@hotmail.com')
m
```

Out[114]:
```
<re.Match object; span=(0, 16), match='Paru@hotmail.com'>
```

In [115]:
```
# Let's group the individual elements of the pattern as tuple of elements for further analysis
m.groups()
```

Out[115]:
```
('Paru', 'hotmail', 'com')
```

In [116]:
```
# Now it is the time to find all the matching pattern objects with this modified 'regex', so let's use text object to
 find all
# the matching pattern
regex.findall(text)
```

Out[116]:
```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

In [117]:
```
# Now finally we can go ahead and print the matching pattern with the new string objects
# sub also has access to groups in each match using special symbols like \1 and \2.
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

```
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

How To Work With The Vectorized String Functions in Pandas?

In [ ]:
```
# some times it is necessary to apply 'regex' method to retrieve each element based on the pattern or function like
# to lower all the charecters of the string we use lambda funcion and so on
```

In [119]:
```
# Let's create a vectorized data with null value to get experience on how to process the string object vectorially.
maild = {'Pruthvi': 'pruthvi@google.com', 'Stella': 'stella@gmail.com', 'Roby': 'roby@gmail.com', 'Navar': np.nan}
maild
```

Out[119]:
```
{'Pruthvi': 'pruthvi@google.com',
 'Stella': 'stella@gmail.com',
 'Roby': 'roby@gmail.com',
 'Navar': nan}
```

In [120]:
```
# The column of the series now has the missing values
maild_s = pd.Series(maild)
maild_s
```

Out[120]:
```
Pruthvi    pruthvi@google.com
Stella       stella@gmail.com
Roby           roby@gmail.com
Navar                     NaN
dtype: object
```

In [121]:
```
# we knew that 'isnull' method of Pandas will detect the missing values in Series or DataFrame and return the boolean
 object
maild_s.isnull()
```

Out[121]:
```
Pruthvi    False
Stella     False
Roby       False
Navar       True
dtype: bool
```

In [124]:
```
# To handle missing values pandas Series has array-oriented methods that skip the NA values like str.contains()
# So let's check each string object has the 'gmail' in its matched pattern
maild_s.str.contains('gmail')
```

Out[124]:
```
Pruthvi    False
Stella      True
Roby        True
Navar        NaN
dtype: object
```

In [125]:
```
# 'regex' can also be used
pattern
```

Out[125]:
```
'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'
```

In [128]:
```
# Find all occurrences of pattern or regular expression in the Series/Index.
pd.Series.str.findall?
```

In [127]:
```
maild_s.str.findall(pattern, flags=re.IGNORECASE)
```

Out[127]:
```
Pruthvi    [(pruthvi, google, com)]
Stella       [(stella, gmail, com)]
Roby           [(roby, gmail, com)]
Navar                           NaN
dtype: object
```

In [130]:
```
# Still we can use many other techniques for the vectorized string operations like 'match' method
matching = maild_s.str.match(pattern, flags=re.IGNORECASE)
matching
```

Out[130]:
```
Pruthvi    True
Stella     True
Roby       True
Navar       NaN
dtype: object
```

In [ ]:
```
# To access elements in the embedded lists, we can pass an index to either of these functions
```

In [131]:
```
# Extract element from each component at specified position.
# Extract element from lists, tuples, or strings in each element in the Series/Index.
pd.Series.str.get?
```

In [146]:
```
print(maild_s)
maild_s.str.get(2)
```

```
Pruthvi    pruthvi@google.com
Stella       stella@gmail.com
Roby           roby@gmail.com
Navar                     NaN
dtype: object
```

Out[146]:
```
Pruthvi      u
Stella       e
Roby         b
Navar      NaN
dtype: object
```

In [147]:
```
# we can also use slicing method to extract a number of charecters from the vectorized strings
maild_s.str[:4] # upper index is excluded in Python's implicit slicing ':4' means from 0 To 3
```

Out[147]:
```
Pruthvi    prut
Stella     stel
Roby       roby
Navar       NaN
dtype: object
```

In [ ]:
```
# A lot more can be done with the strings with both Python's and Pandas objects. We will see those methods further in
 this course.
```