In [1]:
```
!pip install pandas
```

```
Requirement already satisfied: pandas in c:\programdata\anaconda3\lib\site-packages
(1.3.4)
Requirement already satisfied: numpy>=1.17.3 in c:\programdata\anaconda3\lib\site-pa
ckages (from pandas) (1.20.3)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\programdata\anaconda3\li
b\site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in c:\programdata\anaconda3\lib\site-pac
kages (from pandas) (2021.3)
Requirement already satisfied: six>=1.5 in c:\programdata\anaconda3\lib\site-package
s (from python-dateutil>=2.7.3->pandas) (1.16.0)
```

In [2]:
```
import pandas as pd
import numpy as np
```

In [3]:
```
pd.__version__
```

Out[3]:
```
'1.3.4'
```

Pandas Series Example

In [4]:
```
#Creating a series from array

data = np.arange(10,20) #10-19
ser = pd.Series(data)

print(ser)
```

```
0    10
1    11
2    12
3    13
4    14
5    15
6    16
7    17
8    18
9    19
dtype: int32
```

In [5]:
```
print(type(ser))
```

```
<class 'pandas.core.series.Series'>
```

In [6]:
```
#Creating a series from Lists:

# a simple list
list = ['a', 's', 'h', 'i', 's', 'h']

# create series form a list
ser = pd.Series(list)
print(ser)
```



```
1    s
```

```
2    h
3    i
4    s
5    h
dtype: object
```

In [7]:
```
ser[0:3]
```

Out[7]:
```
0    a
1    s
2    h
dtype: object
```

Accessing Element from Series with Position

In [8]:
```
# creating simple array
data = np.array(['a','s','h','i','s','h', 'v','i','s','h','w','a','k','a','r','r','a
ser = pd.Series(data)

#retrieve the first element
print(ser)
```

```
0     a
1     s
2     h
3     i
4     s
5     h
6     v
7     i
8     s
9     h
10    w
11    a
12    k
13    a
14    r
15    r
16    a
dtype: object
```

In [9]:
```
#retrieve the first 5 element
print(ser[0:6])
```

```
0    a
1    s
2    h
3    i
4    s
5    h
dtype: object
```

Accessing Element from Series with Label

In [10]:
```
# creating simple array
```

In [11]:
```
ser
```

Out[11]:
```
a       Punit
b      Ramesh
c      Suresh
d    Abhishek
e       Rahul
dtype: object
```

In [12]:
```
#accessing a element using index element
print(ser['a':'c'])
print(ser[0:3])
```

```
a     Punit
b    Ramesh
c    Suresh
dtype: object
a     Punit
b    Ramesh
c    Suresh
dtype: object
```

Pandas Dataframe Example

In [13]:
```
#%config Completer.use_jedi = False
```

In [76]:
```
#creating dataframe from csv file
data = pd.read_csv("Bank_churn.csv")
```

In [15]:
```
type(data)
```

Out[15]:
```
pandas.core.frame.DataFrame
```

In [79]:
```
data.head(6)
```

Out[79]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	N
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	
2	3	15619304	Onio	502	France	Female	42	8	159660.80	
3	4	15701354	Boni	699	France	Female	39	1	0.00	
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	
5	6	15574012	Chu	645	Spain	Male	44	8	113755.78	

In [17]:
```
# Print records from bottom
data.tail()
#data.tail(10)
```

Out[17]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balanc
9995	9996	15606229	Obijiaku	771	France	Male	39	5	0.0(
9996	9997	15569892	Johnstone	516	France	Male	35	10	57369.6
9997	9998	15584532	Liu	709	France	Female	36	7	0.0(
9998	9999	15682355	Sabbatini	772	Germany	Male	42	3	75075.3
9999	10000	15628319	Walker	792	France	Female	28	4	130142.7

In [18]:
```
# See the shape
data.shape
```

Out[18]:
```
(10000, 14)
```

In [19]:
```
# Data information about columns and non-null
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   RowNumber        10000 non-null  int64  
 1   CustomerId       10000 non-null  int64  
 2   Surname          10000 non-null  object 
 3   CreditScore      10000 non-null  int64  
 4   Geography        10000 non-null  object 
 5   Gender           10000 non-null  object 
 6   Age              10000 non-null  int64  
 7   Tenure           10000 non-null  int64  
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64  
 10  HasCrCard        10000 non-null  int64  
 11  IsActiveMember   10000 non-null  int64  
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64  
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

In [20]:
```
# Get datatype for all columns
data.dtypes
```

Out[20]:
```
RowNumber            int64
CustomerId           int64
Surname             object
CreditScore          int64
Geography           object
Gender              object
Age                  int64
Tenure               int64
Balance            float64
NumOfProducts        int64
HasCrCard            int64
IsActiveMember       int64
EstimatedSalary    float64
```

```
Exited               int64
dtype: object
```

In [21]:
```
type(data.Age)
```

Out[21]:
```
pandas.core.series.Series
```

In [22]:
```
type(data.Tenure)
```

Out[22]:
```
pandas.core.series.Series
```

In [23]:
```
#Data Description
#We get 5 point summary for only NUMERIC data
data.describe()
```

Out[23]:

	RowNumber	CustomerId	CreditScore	Age	Tenure	Balance	NumOf
count	10000.00000	1.000000e+04	10000.000000	10000.000000	10000.000000	10000.000000	1000
mean	5000.50000	1.569094e+07	650.528800	38.921800	5.012800	76485.889288	
std	2886.89568	7.193619e+04	96.653299	10.487806	2.892174	62397.405202	
min	1.00000	1.556570e+07	350.000000	18.000000	0.000000	0.000000	
25%	2500.75000	1.562853e+07	584.000000	32.000000	3.000000	0.000000	
50%	5000.50000	1.569074e+07	652.000000	37.000000	5.000000	97198.540000	
75%	7500.25000	1.575323e+07	718.000000	44.000000	7.000000	127644.240000	
max	10000.00000	1.581569e+07	850.000000	92.000000	10.000000	250898.090000	

In [71]:
```
# Print the name of columns
data.columns
```

Out[71]:
```
Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography',
       'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
       'IsActiveMember', 'EstimatedSalary', 'Exited'],
      dtype='object')
```

In [72]:
```
# Find unique values with counts
data.Geography.value_counts()
```

Out[72]:
```
France     5014
Germany    2509
Spain      2477
Name: Geography, dtype: int64
```

In [73]:
```
data.Gender.value_counts()
```

Out[73]:
```
Male      5457
Female    4543
Name: Gender, dtype: int64
```

In [74]:
```
data.Geography.value_counts(dropna=True)
```

Out[74]:
```
France     5014
Germany    2509
Spain      2477
Name: Geography, dtype: int64
```

In [27]:
```
# Find mean of specific columns
data.Age.mean()
```

Out[27]:
```
38.9218
```

In [28]:
```
# Find out Max Value
data.Age.max()
```

Out[28]:
```
92
```

In [29]:
```
# Find out Min Value
data.Age.min()
```

Out[29]:
```
18
```

In [30]:
```
#find out summazation
data.Age.sum()
```

Out[30]:
```
389218
```

In [31]:
```
#find out median number
data.Age.median()
```

Out[31]:
```
37.0
```

In [33]:
```
# Returns non-null values
data.count()
```

Out[33]:
```
RowNumber          10000
CustomerId         10000
Surname            10000
CreditScore        10000
Geography          10000
Gender             10000
Age                10000
Tenure             10000
Balance            10000
NumOfProducts      10000
HasCrCard          10000
IsActiveMember     10000
EstimatedSalary    10000
Exited             10000
dtype: int64
```

In [34]:
```
#Find out correlation between all columns
data.corr()
```

Out[34]:

	RowNumber	CustomerId	CreditScore	Age	Tenure	Balance	NumOfProd
RowNumber	1.000000	0.004202	0.005840	0.000783	-0.006495	-0.009067	0.007
CustomerId	0.004202	1.000000	0.005308	0.009497	-0.014883	-0.012419	0.016
CreditScore	0.005840	0.005308	1.000000	-0.003965	0.000842	0.006268	0.012
Age	0.000783	0.009497	-0.003965	1.000000	-0.009997	0.028308	-0.030
Tenure	-0.006495	-0.014883	0.000842	-0.009997	1.000000	-0.012254	0.013
Balance	-0.009067	-0.012419	0.006268	0.028308	-0.012254	1.000000	-0.304
NumOfProducts	0.007246	0.016972	0.012238	-0.030680	0.013444	-0.304180	1.000
HasCrCard	0.000599	-0.014025	-0.005458	-0.011721	0.022583	-0.014858	0.003
IsActiveMember	0.012044	0.001665	0.025651	0.085472	-0.028362	-0.010084	0.009
EstimatedSalary	-0.005988	0.015271	-0.001384	-0.007201	0.007784	0.012797	0.014
Exited	-0.016571	-0.006248	-0.027094	0.285323	-0.014001	0.118533	-0.047

Create Dataframe from List

In [75]:
```
# list of strings
lst = ['Pandas', 'is', 'a', 'Python', 'Library', 'used', 'for', 'working','with','da

# Calling DataFrame constructor on list
df = pd.DataFrame(lst)
df.head()
```

Out[75]:

	0
0	Pandas
1	is
2	a
3	Python
4	Library

In [36]:
```
df.columns = ['Sentence']
```

In [37]:
```
df.head()
```

Out[37]:

	Sentence
0	Geeks
1	For
2	Geeks
3	is

Create Dataframe from Dict

In [38]:
```
dict = {
            "country": ["Brazil", "Russia", "India", "China", "South Africa"],
            "capital": ["Brasilia", "Moscow", "Delhi" ,"Beijing", "Pretoria"],
            "area": [8.516, 17.10, 3.286, 9.597, 1.221],
            "population": [200.4, 143.5, 1252, 1357, 52.98]
        }
```

In [39]:
```
dict
```

Out[39]:
```
{'country': ['Brazil', 'Russia', 'India', 'China', 'South Africa'],
 'capital': ['Brasilia', 'Moscow', 'Delhi', 'Beijing', 'Pretoria'],
 'area': [8.516, 17.1, 3.286, 9.597, 1.221],
 'population': [200.4, 143.5, 1252, 1357, 52.98]}
```

In [40]:
```
brics = pd.DataFrame(dict)
brics.head()
```

Out[40]:

	country	capital	area	population
0	Brazil	Brasilia	8.516	200.40
1	Russia	Moscow	17.100	143.50
2	India	Delhi	3.286	1252.00
3	China	Beijing	9.597	1357.00
4	South Africa	Pretoria	1.221	52.98

In [41]:
```
brics.capital.dtype
```

Out[41]:
```
dtype('O')
```

Data Accessing Example

In [42]:
```
# how to print sepecifc columns
#brics.country

#brics.country
brics['country']
```

Out[42]:
```
0          Brazil
1          Russia
2           India
3           China
4    South Africa
Name: country, dtype: object
```

In [43]:
```
#Print specific rows from datafram
brics[0:4]
```

Out[43]:

	country	capital	area	population
0	Brazil	Brasilia	8.516	200.4
1	Russia	Moscow	17.100	143.5
2	India	Delhi	3.286	1252.0
3	China	Beijing	9.597	1357.0

In [44]:

```
#select two columns
data.head()
data[['Age','Tenure','Surname']]
```

Out[44]:

	Age	Tenure	Surname
0	42	2	Hargrave
1	41	1	Hill
2	42	8	Onio
3	39	1	Boni
4	43	2	Mitchell
...	...	...	...
9995	39	5	Obijiaku
9996	35	10	Johnstone
9997	36	7	Liu
9998	42	3	Sabbatini
9999	28	4	Walker

10000 rows × 3 columns

In [45]:

```
# Add new column in Dataframe
brics.head()
```

Out[45]:

	country	capital	area	population
0	Brazil	Brasilia	8.516	200.40
1	Russia	Moscow	17.100	143.50
2	India	Delhi	3.286	1252.00
3	China	Beijing	9.597	1357.00
4	South Africa	Pretoria	1.221	52.98

In [46]:

```
#Create new column as has_beachs or not
has_beaches = ['yes','yes','yes','yes','no']
brics['has_beaches'] = has_beaches
```

In [47]:

```
brics.head()
```

Out[47]:

	country	capital	area	population	has_beaches
0	Brazil	Brasilia	8.516	200.40	yes
1	Russia	Moscow	17.100	143.50	yes
2	India	Delhi	3.286	1252.00	yes
3	China	Beijing	9.597	1357.00	yes
4	South Africa	Pretoria	1.221	52.98	no

In [48]:

```
# Delete column from dataframe
# This will NOT delete the dataframe.. It will just print
brics.drop(columns=['has_beaches','area'])
```

Out[48]:

	country	capital	population
0	Brazil	Brasilia	200.40
1	Russia	Moscow	143.50
2	India	Delhi	1252.00
3	China	Beijing	1357.00
4	South Africa	Pretoria	52.98

In [49]:

```
brics.head()
```

Out[49]:

	country	capital	area	population	has_beaches
0	Brazil	Brasilia	8.516	200.40	yes
1	Russia	Moscow	17.100	143.50	yes
2	India	Delhi	3.286	1252.00	yes
3	China	Beijing	9.597	1357.00	yes
4	South Africa	Pretoria	1.221	52.98	no

In [50]:

```
# This will delete in reality from source

brics.drop(columns=['has_beaches'],inplace=True)
```

In [51]:

```
brics.head()
```

Out[51]:

	country	capital	area	population
0	Brazil	Brasilia	8.516	200.40
1	Russia	Moscow	17.100	143.50
2	India	Delhi	3.286	1252.00
3	China	Beijing	9.597	1357.00
4	South Africa	Pretoria	1.221	52.98

In [52]:
```
brics.columns
```

Out[52]:
```
Index(['country', 'capital', 'area', 'population'], dtype='object')
```

loc and iloc Example

loc Example

We will create a sample student dataset consisting of 5 columns – age, section, city, gender, and favorite color. This dataset will contain both numerical as well as categorical variables

In [53]:
```
# crete a sample dataframe
student_data = pd.DataFrame({
    'age' :     [ 10, 22, 13, 21, 12, 11, 17],
    'section' : [ 'A', 'B', 'C', 'B', 'B', 'A', 'A'],
    'city' :    [ 'Gurgaon', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai'
    'gender' :  [ 'M', 'F', 'F', 'M', 'M', 'M', 'F'],
    'favourite_color' : [ 'red', np.NaN , 'yellow', np.NAN, 'black', 'green', 'red']
})
```

In [54]:
```
student_data
```

Out[54]:

	age	section	city	gender	favourite_color
0	10	A	Gurgaon	M	red
1	22	B	Delhi	F	NaN
2	13	C	Mumbai	F	yellow
3	21	B	Delhi	M	NaN
4	12	B	Mumbai	M	black
5	11	A	Delhi	M	green
6	17	A	Mumbai	F	red

In [55]:
```
# Selecing single record
student_data.loc[3]
```

Out[55]:
```
age                  21
section               B
city              Delhi
gender                M
favourite_color     NaN
Name: 3, dtype: object
```

In [56]:
```
# Selecing single record
student_data.loc[1:3,'city']
```

Out[56]:
```
1     Delhi
2    Mumbai
3     Delhi
Name: city, dtype: object
```


In [57]:
```
student_data.age >= 15
```

Out[57]:
```
0    False
1     True
2    False
3     True
4    False
5    False
6     True
Name: age, dtype: bool
```

In [58]:
```
#Find all the rows based on any condition in a column
# Let's try to find the rows where the value of age is greater than or equal to 15:
student_data.loc[student_data.age >= 15]
```

Out[58]:

	age	section	city	gender	favourite_color
1	22	B	Delhi	F	NaN
3	21	B	Delhi	M	NaN
6	17	A	Mumbai	F	red

In [59]:
```
#Find all the rows with more than one condition
#select with multiple conditions
student_data.loc[(student_data.age >= 12) & (student_data.gender == 'M')]
```

Out[59]:

	age	section	city	gender	favourite_color
3	21	B	Delhi	M	NaN
4	12	B	Mumbai	M	black

In [60]:
```
#Select a range of rows using loc
#Using loc, we can also slice the Pandas dataframe over a range of indices.
#And if the indices are not numbers, then we cannot slice our dataframe.
# Both numbers are inclusive
student_data.loc[1:4]
```

Out[60]:

	age	section	city	gender	favourite_color
1	22	B	Delhi	F	NaN
2	13	C	Mumbai	F	yellow
3	21	B	Delhi	M	NaN
4	12	B	Mumbai	M	black

In [61]:
```
#Select only required columns with a condition
# select few columns with a condition
student_data.loc[1:4 , ['age','section']]
```

Out[61]:

	age	section
1	22	B
2	13	C

	age	section
4	12	B

Update the values of a particular column on selected rows

We often have to update values in our dataset based on a certain condition. For example, if the values in age are greater than equal to 12, then we want to update the values of the column section to be “M”. We can do this by running a for loop as well but if our dataset is big in size, then it would take forever to complete the task. Using loc in Pandas, we can do this within seconds, even on bigger datasets! We just need to specify the condition followed by the target column and then assign the value with which we want to update

In [62]:
```
# update a column with condition
student_data.loc[(student_data.age >= 12), ['section']] = 'M'
student_data
```

Out[62]:

	age	section	city	gender	favourite_color
0	10	A	Gurgaon	M	red
1	22	M	Delhi	F	NaN
2	13	M	Mumbai	F	yellow
3	21	M	Delhi	M	NaN
4	12	M	Mumbai	M	black
5	11	A	Delhi	M	green
6	17	M	Mumbai	F	red

Update the values of multiple columns on selected rows

In [63]:
```
#If we want to update multiple columns with different values, then we can use the be

student_data.loc[(student_data.age >= 20), ['section', 'city']] = ['S','Pune']
student_data
```

Out[63]:

	age	section	city	gender	favourite_color
0	10	A	Gurgaon	M	red
1	22	S	Pune	F	NaN
2	13	M	Mumbai	F	yellow
3	21	S	Pune	M	NaN
4	12	M	Mumbai	M	black
5	11	A	Delhi	M	green
6	17	M	Mumbai	F	red

In [64]:
```
student_data[1:4]
```

Out[64]:

	age	section	city	gender	favourite_color
1	22	S	Pune	F	NaN
2	13	M	Mumbai	F	yellow
3	21	S	Pune	M	NaN

Select rows with indices using iloc

When we are using iloc, we need to specify the rows and columns by their integer index. If we want to select only the first and third row, we simply need to put this into a list in the iloc statement with our dataframe

In [65]:
```
# Selecting two rows number 1 and number 3
student_data.iloc[[1,3]]
```

Out[65]:

	age	section	city	gender	favourite_color
1	22	S	Pune	F	NaN
3	21	S	Pune	M	NaN

In [66]:
```
student_data
```

Out[66]:

	age	section	city	gender	favourite_color
0	10	A	Gurgaon	M	red
1	22	S	Pune	F	NaN
2	13	M	Mumbai	F	yellow
3	21	S	Pune	M	NaN
4	12	M	Mumbai	M	black
5	11	A	Delhi	M	green
6	17	M	Mumbai	F	red

In [67]:
```
# Select rows with particular indices and particular columns
# Selecting rows 0 and 2 and selecting column number 1 and 3

student_data.iloc[[0,2],[1,3]]
```

Out[67]:

	section	gender
0	A	M
2	M	F

In [68]:
```
# Selecting range of rows from rows 0 to rows 4.
# So total 5 rows
# high number is exclusive
student_data.iloc[0:5]
```

Out[68]:

	age	section	city	gender	favourite_color
0	10	A	Gurgaon	M	red
1	22	S	Pune	F	NaN
2	13	M	Mumbai	F	yellow
3	21	S	Pune	M	NaN
4	12	M	Mumbai	M	black

In [69]:
```
# Select a range of rows and columns using iloc
# select a range of rows and columns
# high numbers are exclusive

student_data.iloc[1:3 , 2:4]
```

Out[69]:

	city	gender
1	Pune	F
2	Mumbai	F

In [70]:
```
col_lst = list(student_data.columns)
col_lst
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\SAGARC~1\AppData\Local\Temp/ipykernel_7484/3575098561.py in <module>
----> 1 col_lst = list(student_data.columns)
      2 col_lst

TypeError: 'list' object is not callable
```

In [ ]:
```
col_lst[0] = 'age_1'
```

In [ ]:
```
col_lst
```

In [ ]:
```
student_data.columns = col_lst
```

In [ ]:
```
student_data
```

GroupBy Example

In [ ]:
```
data.head()
```

In [ ]:
```
data.groupby('Geography')
```

GroupBy has conveniently returned a DataFrameGroupBy object. It has split the data into separate groups. However, it won't do anything unless it is being told explicitly to do so.

In [ ]:
```
# Finding count for each job category
data.groupby('Geography').count()
```

In [ ]:
```
# Group by on multiple columns
data.groupby(['Geography','Gender']).count()
```

In [ ]:
```
# Find count for specific column
# As per job category what is the total balance

data.groupby('Geography')['Balance'].mean()
```

In [ ]:
```
# Find count for specific column
# As per job category what is the average balance

data.groupby('job')['balance'].mean()
```

In [ ]:
```
# Loop over groupby groups
grp = data.groupby('Geography')
```

In [ ]:
```
grp
```

In [ ]:
```
# Print groups
grp.groups
```

In [ ]:
```
# Datatype for groups
type(grp.groups)
```

In [ ]:
```
#We can even iterate over all of the groups

for name,group in grp:
    print(name,'contains',group.shape[0],'rows')
```

In [ ]:
```
# We can get even specifc group

grp.get_group('France')
```

In [ ]:
```
grp.get_group('management')
```

In [ ]:
```
#Aggregation function over group by
#agg() function in Pandas gives us the flexibility to perform several statistical co
```

In [ ]:
```
import numpy as np
data.groupby('Geography')['Balance'].agg([np.mean, np.sum, np.min, np.max])
```

In [ ]:
```
# Agg function on multiple columns
data.groupby(['Geography','Gender'])['Balance'].agg([np.mean,np.sum])
```

Working with Missing Data in Pandas

In Pandas missing data is represented by two value: None: None is a Python singleton object that is often used for missing data in Python code. NaN : NaN (an acronym for Not a Number), is a special floating-point value recognized by all systems that use the standard IEEE floating-point representationisnull() notnull() dropna() fillna() replace() interpolate()

In [ ]:
```
# dictionary of lists
dict = {'First Score':['Male', 'Female', np.nan, 'Female'],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from list
df = pd.DataFrame(dict)
```

In [ ]:
```
df
```

In [ ]:
```
# Check if any value in DF is null
df.isnull()
```

In [ ]:
```
# Get count of NaN values from all columns
df.isnull().sum()
```

In [ ]:
```
# Check for NaN in one specific column
pd.isnull(df['First Score'])
```

In [ ]:
```
# opposite of isnull
df.notnull()
```

Filling missing values

In [ ]:
```
# Fill missing value with 0
df.fillna(df.mean())
```

In [ ]:
```
#Filling null values with the previous ones
df.fillna(method='ffill')
```

In [ ]:
```
#Filling null values with the next ones
df.fillna(method='bfill')
```

In [ ]:
```
# Use method Replace
df.replace(to_replace=np.nan,value=100)
```

Dropping the Missing Values

In [ ]:
```
# Drop all rows with NA values

df.dropna()
#df.dropna(inplace=True)
```

In [ ]:
```
#drop all duplicates

data.drop_duplicates(keep='first')
```

Apply function Example

In [ ]:
```
data.head()
```

In [ ]:
```
# defining function to check balance
def fun(num):

    if num<2000:
        return "Low"
    elif num>= 2000 and num<4000:
        return "Normal"
    else:
        return "High"
```

In [ ]:
```
# Create new column and apply function on each value of column Balance
data['TEMP'] = data.Balance.apply(lambda x : fun(x))
```

In [ ]:
```
data.head(10)
```

In [ ]:
```
# Example of Lambda
#Stateless function
x = lambda a : a + 10

def x(a):
    return a + 10
```

In [ ]:
```
print(x(20))
```

Merging, Joining, and Concatenating DataFrame

Concatenating DataFrame using .concat() :

The concat() function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say "if any" because there is only a single possible axis of concatenation for Series.

In [ ]:
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])


df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4, 5, 6, 7])


df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])

#Putting all DF's in list
frames = [df1, df2, df3]
#Concatinating them
result = pd.concat(frames)

print(result)
```

Set logic on the other axes

In [ ]:
```
#Common indexes
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                   index=[2, 3, 6, 7])


result = pd.concat([df1, df4], axis=1)

print(result)
```

In [ ]:
```
#Providing option for join
result = pd.concat([df1, df4], axis=1, join='inner')

print(result)
```

Concatenating with Series

In [ ]:
```
s2 = pd.Series(['_0', '_1', '_2', '_3'])
result = pd.concat([df1, s2, s2, s2], axis=1)

print(result)
```

Concatenating DataFrame using .append() :

A useful shortcut to concat() are the append() instance methods on Series and DataFrame. These methods actually predated concat. They concatenate along axis=0, namely the index.

In [4]:
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])


df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4, 5, 6, 7])

result = df1.append(df2)

print(result)
```

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```

Example of multiple DF's with common indexes

In [5]:
```
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                   index=[2, 3, 6, 7])

result = df1.append(df4, sort=False)

print(result)
```

```
     A   B    C   D    F
0   A0  B0   C0  D0  NaN
1   A1  B1   C1  D1  NaN
2   A2  B2   C2  D2  NaN
3   A3  B3   C3  D3  NaN
2  NaN  B2  NaN  D2   F2
3  NaN  B3  NaN  D3   F3
6  NaN  B6  NaN  D6   F6
7  NaN  B7  NaN  D7   F7
```

append may take multiple objects to concatenate

In [ ]:
```
result = df1.append([df2, df3])
print(result)
```


Appending rows to a DataFrame

In [6]:
```
s2 = pd.Series(['X0', 'X1', 'X2', 'X3'], index=['A', 'B', 'C', 'D'])
result = df1.append(s2, ignore_index=True)

print(result)
```

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3
4  X0  X1  X2  X3
```

Concatenating DataFrame using .merge() :

When you want to combine data objects based on one or more keys in a similar way to a relational database, merge() is the tool you need. More specifically, merge() is most useful when you want to combine rows that share data.

**pandas provides a single function, merge(), as the entry point for all standard database join operations between DataFrame or named Series objects. **

There are THREE types of operation in merge

one-to-one joins: for example when joining two DataFrame objects on their indexes (which must contain unique values).
many-to-one joins: for example when joining an index (unique) to one or more columns in a different DataFrame.
many-to-many joins: joining columns on columns.

In [7]:
```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})


right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})


result = pd.merge(left, right, on='key')

print(result)
```

```
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```

Here is a more complicated example with multiple join keys. Only the keys appearing in left and right are present (the intersection), since how='inner' by default.

In [8]:
```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})
```

```
right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                      'key2': ['K0', 'K0', 'K0', 'K0'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})


result = pd.merge(left, right, on=['key1', 'key2'])

print(result)
```

```
  key1 key2   A   B   C   D
0   K0   K0  A0  B0  C0  D0
1   K1   K0  A2  B2  C1  D1
2   K1   K0  A2  B2  C2  D2
```

The how argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names: Merge method SQL Join Name Description left LEFT OUTER JOIN Use keys from left frame only right RIGHT OUTER JOIN Use keys from right frame only outer FULL OUTER JOIN Use union of keys from both frames inner INNER JOIN Use intersection of keys from both frames

In [9]:
```
#merge() accepts the argument indicator.
#If True, a Categorical-type column called _merge will be added to the output object

result = pd.merge(left, right, how='left', on=['key1', 'key2'], indicator = True)
print(result)
```

```
  key1 key2   A   B    C    D     _merge
0   K0   K0  A0  B0   C0   D0       both
1   K0   K1  A1  B1  NaN  NaN  left_only
2   K1   K0  A2  B2   C1   D1       both
3   K1   K0  A2  B2   C2   D2       both
4   K2   K1  A3  B3  NaN  NaN  left_only
```

Concatinating Dataframe using .join():

In [10]:
```
# 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'])
```

In [11]:
```
df
```

Out[11]:

	Name	Age
K0	Jai	27
K1	Princi	24

	Name	Age
K2	Gaurav	22
K3	Anuj	32

In [ ]:
```
df1
```

In [ ]:
```
# Joining Dataframe
# Based on initial DF, you will see indexes
df.join(df1)
```

In [ ]:
```
df1.join(df)
```

In [ ]:
```
# Outer Join
df.join(df1, how='outer')
```

In [ ]:
```
df1.join(df,how="outer")
```

In [ ]:
```
# Outer Join
df.join(df1, how='inner')
```

In [ ]:
```
df.join(df1, how='inner',sort=False)
```

In [ ]:
```
# Example on rsuffix and lsuffix
```

In [ ]:
```
# Define a dictionary containing employee data
data1 = {'key': ['K0', 'K1', 'K2', 'K3'],
         'key1': ['K0', 'K1', 'K0', 'K1'],
         'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
        'Age':[27, 24, 22, 32],
        'Group' : ['A','B','C','D']}

# Define a dictionary containing employee data
data2 = {'key': ['K0', 'K1', 'K2', 'K3'],
         'key1': ['K0', 'K0', 'K0', 'K0'],
         'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],
        'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons'],
        'Group' : ['A','B','C','D']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])

# Convert the dictionardfy into DataFrame
df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])
```

In [ ]:
```
df
```

In [ ]:
```
df1
```

In [ ]:
```
# Error on same column name
df.join(df1)
```

In [ ]:
```
df.join(df1,on=['key'],lsuffix='_left',rsuffix='_right',how='inner')
```

Working with Date and Time

Pandas has a built-in function called to_datetime() that can be used to convert strings to datetime. Let's take a look at some examples

In [ ]:
```
df = pd.DataFrame({'date': ['3/10/2000', '3/11/2000', '3/12/2000'],
                   'value': [2, 3, 4]})
df['date'] = pd.to_datetime(df['date'])
df
```

Day first format

By default, to_datetime() will parse string with month first (MM/DD, MM DD, or MM-DD) format, and this arrangement is relatively unique in the United State. In most of the rest of the world, the day is written first (DD/MM, DD MM, or DD-MM). If you would like Pandas to consider day first instead of month, you can set the argument dayfirst to True.

In [ ]:
```
df = pd.DataFrame({'date': ['3/10/2000', '3/11/2000', '3/12/2000'],
                   'value': [2, 3, 4]})
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df
```

Custome format

In [ ]:
```
df = pd.DataFrame({'date': ['2016-6-10 20:30:0',
                            '2016-7-1 19:45:30',
                            '2013-10-12 4:5:1'],
                   'value': [2, 3, 4]})
df['date'] = pd.to_datetime(df['date'], format="%Y-%d-%m %H:%M:%S")
df
```

Assemble a datetime from multiple columns

to_datetime() can be used to assemble a datetime from multiple columns as well. The keys (columns label) can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same.

In [ ]:
```
df = pd.DataFrame({'year': [2015, 2016],
```

```
df['date'] = pd.to_datetime(df)
df
```

Get year, month, and day

dt.year, dt.month and dt.day are the inbuilt attributes to get year, month , and day from Pandas datetime object.

In [ ]:
```
df = pd.DataFrame({'name': ['Tom', 'Andy', 'Lucas'],
                   'DoB': ['08-05-1997', '04-28-1996', '12-16-1995']})
df['DoB'] = pd.to_datetime(df['DoB'])

df
```

In [ ]:
```
df['year']= df['DoB'].dt.year
df['month']= df['DoB'].dt.month
df['day']= df['DoB'].dt.day
df
```

Get the week of year, the day of week and leap year

Similarly, dt.week, dt.dayofweek, and dt.is_leap_year are the inbuilt attributes to get the week of year, the day of week, and leap year

In [ ]:
```
df['week_of_year'] = df['DoB'].dt.week
df['day_of_week'] = df['DoB'].dt.dayofweek
df['is_leap_year'] = df['DoB'].dt.is_leap_year
df
```

Get the age from the date of birth

In [ ]:
```
today = pd.to_datetime('today')
df['age'] = today.year - df['DoB'].dt.year
df
```

Select data with a specific year and perform aggregation

In [ ]:
```
df = pd.read_csv('/Users/punitshah/Downloads/city_sales.csv', parse_dates=['date'])
df.head()
df = df.set_index(['date'])
```

In [ ]:
```
df.loc['2018']
```

In [ ]:
```
df.loc['2018','num'].sum()
```

Select data with a specific month and a specific day of

In [ ]:
```
df.loc['2018-5']
```

In [ ]:
```
df.loc['2018-5-1']
```

Select data between two dates

In [ ]:
```
#Select data between 2016 and 2018
df.loc['2016' : '2018']
```

In [ ]:
```
#Select data between 10 and 11 o'clock on the 2nd May 2018
df.loc['2018-5-2 10' : '2018-5-2 11' ]
```

In [ ]:
```
#select data between time, we should use between_time(), for example, 10:30 and 10:4
df.between_time('10:30','10:45')
```