DATA MANIPULATION USE CASE

TYPICAL DATA MANIPULATION STEPS(Operations on Data Frame)

- Sub Setting Data or Filtering Data or Slicing Data
  - Using [] brackets
  - Using indexing or referring with column names/rows
  - Using functions
- Mutation of table (Adding/deleting columns)
- Binning data (Binning numerical variables in to categorical variables using cut() and qcut() functions)
- Dropping rows & columns
- Renaming columns or rows
- Sorting
  - by data/values, index
  - By one column or multiple columns
  - Ascending or Descending
- Type conversions
- Setting index
- Applying functions to all the variables in a data frame (broadcasting)
- Handling missing values – detect, filter, replace
- Handling duplicates
- Handling outliers
- Creating dummies from categorical data (using get_dummies())
- Table manipulation (One table and multiple tables)-
  - Aggregation – Group by processing
  - Merge/ Join (left, right, outer, inner)
  - Concatenate (appending) – Binding or stacking or union all
  - Reshaping & Pivoting data – stack/unstack, pivot table, summarizations
  - Standardize the variables
- Random Sampling (with replacement/with out replacement)
- Handling Time Series Data
- Handling text (string) data
  - with functions
  - with Regular expressions

Data Importing

In [1]:
```
#import all necessary libraries
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
#install if seabon is not available using cmd prompt - conda install seaborn
```

In [2]:
```
os.getcwd()
```

Out[2]:
```
'C:\\Users\\admin\\pandas'
```

In [3]:
```
os.chdir('C:\\Users\\admin\\pandas')
```

In [4]:
```
# Data importing (we will use data from Pandas Case Study)
cust_data = pd.read_csv("DataSets/Cust_data.csv")
cust_demo = pd.read_csv("DataSets/Cust_demo.csv")
cust_new  = pd.read_csv("DataSets/cust_new.csv")
stores = pd.read_csv("DataSets/stores.csv")
```

Data Understanding

In [ ]:
```
# Data Understanding
print cust_data.head(5)
print cust_demo.head(5)
print cust_new.head(5)
```

In [5]:
```
cust_demo.columns
```

Out[5]:
```
Index([u'ID', u'Location', u'Gender', u'age', u'Martial_Status',
       u'NumberOfDependents', u'Own_House', u'No_Years_address'],
      dtype='object')
```

In [6]:
```
cust_demo.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 149956 entries, 0 to 149955
Data columns (total 8 columns):
ID                    149956 non-null int64
Location              149956 non-null object
Gender                149956 non-null int64
age                   149956 non-null int64
Martial_Status        149956 non-null object
NumberOfDependents    146033 non-null float64
Own_House             149956 non-null int64
No_Years_address      149956 non-null int64
dtypes: float64(1), int64(5), object(2)
memory usage: 9.2+ MB
```

In [7]:
```
cust_demo.shape
```
Out[7]:
```
(149956, 8)
```

In [8]:
```
cust_demo.size
```
Out[8]:
```
1199648
```

In [9]:
```
cust_demo.dtypes
```
Out[9]:
```
ID                      int64
Location               object
Gender                  int64
age                     int64
Martial_Status         object
NumberOfDependents    float64
Own_House               int64
No_Years_address        int64
dtype: object
```

In [10]:
```
cust_demo.get_dtype_counts()
```
Out[10]:
```
float64    1
int64      5
object     2
dtype: int64
```

In [11]:
```
cust_demo.count() # no of non null values per column
```
Out[11]:
```
ID                    149956
Location              149956
Gender                149956
age                   149956
Martial_Status        149956
NumberOfDependents    146033
Own_House             149956
No_Years_address      149956
dtype: int64
```

In [12]:
```
cust_demo.memory_usage()
```
Out[12]:
```
Index                      72
ID                    1199648
Location              1199648
Gender                1199648
age                   1199648
Martial_Status        1199648
NumberOfDependents    1199648
Own_House             1199648
No_Years_address      1199648
dtype: int64
```

In [13]:
```
cust_demo.ndim
```
Out[13]:
```
2
```

In [14]:
```
import numpy as np
```

In [15]:
```
#Getting specific list of data types
cust_demo.select_dtypes(include = ['category'] ).head(5)
cust_demo.select_dtypes(include = ['number'] ).head(5)
cust_demo.select_dtypes(include = ['floating'] ).head(5)
cust_demo.select_dtypes(include = ['integer'] ).head(5)
cust_demo.select_dtypes(include = ['object'] ).head(5)

cust_demo.select_dtypes(exclude = ['object'] ).head(5)

#cust_demo.select_dtypes(include=None, exclude=None)
#Return a subset of a DataFrame including/excluding columns based on their ``dtyp
```

...

In [16]:
```
cust_demo.describe()
```

Out[16]:

	ID	Gender	age	NumberOfDependents	Own_House	No_Yea
count	149956.000000	149956.000000	149956.000000	146033.000000	149956.000000	149
mean	75001.373563	0.374977	52.296814	0.757219	0.399344	
std	43300.796939	0.484119	14.769803	1.115036	0.489765	
min	1.000000	0.000000	0.000000	0.000000	0.000000	
25%	37503.750000	0.000000	41.000000	0.000000	0.000000	
50%	75002.500000	0.000000	52.000000	0.000000	0.000000	
75%	112499.250000	1.000000	63.000000	1.000000	1.000000	
max	150000.000000	1.000000	109.000000	20.000000	1.000000	

Data Manipulation

Sub Setting Data

In [17]:
```
cust_data.columns
```

Out[17]:
```
Index([u'ID', u'SeriousDlqin2yrs', u'RevolvingUtilization', u'No_of_30_59_DPD',
       u'DebtRatio', u'MonthlyIncome', u'No_Of_OpenCreditLines',
       u'No_of_90_DPD', u'No_of_RealEstateLoans', u'No_of_60_89_DPD'],
      dtype='object')
```

In [18]:
```
cust_data.MonthlyIncome
```

Out[18]:
```
0         11500.0
1         14166.0
2          6733.0
3         13316.0
4          2557.0
5             NaN
6          5000.0
7          7000.0
8         11833.0
9             NaN
10        20000.0
11         6150.0
12         6200.0
13         5700.0
14            NaN
15         7500.0
16         3900.0
17         7726.0
18            NaN
19        10400.0
20        26574.0
21         1666.0
22            NaN
23         7126.0
24         5000.0
25        10000.0
26            NaN
27         5083.0
28            NaN
29         7002.0
           ...
149872     1720.0
149873        NaN
149874     4816.0
149875     1000.0
149876        NaN
149877        NaN
149878     4583.0
149879     4800.0
149880        NaN
149881    21083.0
149882        NaN
149883        NaN
149884     5606.0
149885     3333.0
149886     9583.0
149887     2950.0
149888     8165.0
149889     5328.0
149890     2648.0
149891     3600.0
149892    10500.0
149893        0.0
149894        NaN
149895    16666.0
```

```
149896     3693.0
149897     2650.0
149898     1962.0
149899    10016.0
149900     1750.0
149901    12400.0
Name: MonthlyIncome, Length: 149902, dtype: float64
```

```
# Series
Ser1.iloc[]
```

```
dataframe.loc[rows,columns] # .loc can take row ID
# can pass row numbers to .loc if the row ix are numbers
dataframe.iloc[row_ix,columns_ix]
```

In [ ]:
```
#subsetting data
cust_data[["MonthlyIncome","ID"]].head(5)
```

In [ ]:
```
cust_data.head(3)
```

In [ ]:
```
cust_data.iloc[:,3:7:2] # 4th, 6th column.
```

In [ ]:
```
cust_data.iloc[0:100:10,:]
```

In [ ]:
```
cust_data.loc[:,'ID'].head(2)
```

In [ ]:
```
cust_data.loc[:10,'ID']
```

In [ ]:
```
cust_data.loc[1:10,['ID', 'DebtRatio']]
```

Creating New Variables

In [ ]:
```
cust_data["NewColumn"] = ""
# numpy - randn(), randint()
# range()
# numpy.arange()
```

In [ ]:
```
cust_data.NewColumn.head(10)
```

In [ ]:
```
stores["GrandTotalSales"] = stores["TotalSales"] * stores["Total_Customers"]

# NetProfit = TotalSales - OperatingCost
```

In [ ]:
```
# stores -> TotalSales, OperatingCost, Total_Customers, AcqCostperCust

# GTSales = TotalSales * Total_Customers
# TotalExpenses = OperatingCost + AcqCostPerCust
# NetProfit = GTSales - TotalExpenses
stores = pd.read_csv("DataSets/stores.csv")

stores = stores.assign(GTSales = stores.TotalSales * stores.Total_Customers,Total
stores
```

In [ ]:
```
#Method-1
#Creating new Columns variable No_of_30_Plus_DPD =  No_of_30_59_DPD + No_of_60_89_
cust_data['No_of_30_Plus_DPD'] = cust_data['No_of_30_59_DPD']+cust_data['No_of_60_
print cust_data1.head(2)
```

In [ ]:
```
#Method-2
# df.assign will give you new data frame with old and new variables.
# You can create as many as variables
cust_data1=cust_data.assign(No_of_30_Plus_DPD=cust_data.No_of_30_59_DPD+cust_data
                            MonthlySavings=cust_data.MonthlyIncome*0.15)
```

In [ ]:
```
cust_data1.columns
```

Dropping Variables

In [ ]:
```
#Creating new column monthly_savings = MonthlyIncome * 0.15
cust_data['MonthlySavings1'] = cust_data['MonthlyIncome'] * 0.15
cust_data.head(2)
```

In [ ]:
```
cust_data = cust_data.drop("MonthlySavings1",axis = 1).head(3)
# or use inplace = True
# cust_data.drop("MonthlySavings1",axis = 1,inplace = True).head(3)
```

In [ ]:
```
#dropping columns
cust_data1.drop('MonthlySavings1', axis=1).head(2) # it creates new data
#cust_data.drop('monthly_savings', inplace=True, axis=1) # it modify the existing
cust_data.head(2)
```

In [ ]:
```
stores = pd.read_csv("DataSets/stores.csv")
stores
```

In [ ]:
```
stores.columns
```

In [ ]:
```
stores.drop("GrandTotalSales",axis = 1,inplace=True)
# inplace =  True - make the changes and save it back to object
```

Binning data: Converting numeric variables into categorical variables

The `pd.cut()` and `pd.qcut()` functions are used; they take as arguments the following;

- `var`, the continuous variable to discretize
- `bins`, specified as a number (equal sized bins will be computed based on min/max) or a list of bin edges
- `right=True`, a boolean to include the edge or not
- `labels=`, for naming the bins
- `precision=`

> `pd.cut` -> same distributions as the underlying data
> `pd.qcut` -> uniform distribtion

In [ ]:
```
stores["TotalSales"].head(5)
```

In [ ]:
```
stores["SalesCat"] = pd.cut(stores["TotalSales"],4, labels = ["C1","C2","C3","C4"
stores.head(2)
stores[["SalesCat","TotalSales"]]
```

In [ ]:
```
cust_data1['MonthlyIncome'].head(10)
```

In [ ]:
```
pd.cut(cust_data1['MonthlyIncome'],3, labels = ['0-33', '33-66','66-100'])
```

In [ ]:
```
# Automatic Binning
#pd.cut(cust_data['MonthlyIncome'], 5)[:10]

# Specifying bins manually
#pd.cut(cust_data['MonthlyIncome'], bins=range(-100000, 3000000, 100000))

cust_data1['IncomeBuckets']=pd.cut(cust_data1['MonthlyIncome'], 3, labels=['0to33
```

In [ ]:
```
cust_data1.head(10)
```

In [ ]:
```
pd.cut(cust_data1['MonthlyIncome'], 3, labels=['one', 'two', 'three'], retbins=Tr
```

In [ ]:
```
cust_data1['MonthlyIncome'].describe()
```

In [ ]:
```
cust_data1['IncomeBuckets']=pd.cut(cust_data1['MonthlyIncome'], range(0, 1000000,

#0 - 10000
#10000 - 20000
#...
#- 10000000
```

In [ ]:
```
cust_data1['IncomeBuckets'].value_counts()
```

In [ ]:
```
# Binning into quantiles
cust_data1['Deciles']=pd.qcut(cust_data1['MonthlyIncome'], 10, labels=range(1,11,
#pd.qcut(cust_data['MonthlyIncome'], 10).value_counts().plot.bar()
```

In [ ]:
```
# Data Types
```

In [ ]:
```
cust_data.dtypes
```

In [ ]:
```
MI = cust_data.MonthlyIncome
MI.dtypes
cust_data.MonthlyIncome.dtype
```

In [ ]:
```
# Numbers :
# int64,float64,long,int and float

# Text :
# object, str

# Logical :
# bool

# Dates :
# datetime64
```

In [ ]:
```
cust_data.MonthlyIncome.astype(str)
```

In [ ]:
```


#stores1.TotalSales.astype(str)

cust_data.MonthlyIncome = cust_data.MonthlyIncome.astype(str)
cust_data.dtypes


# Strings - "123", "abs123"
table.column.astype(float)

# floats can never be converted to int

stores.TotalSales.astype(str)
```

Renaming Columns (sngle or multiple)

In [ ]:
```
#renaming column "RevolvingUtilization with Rev_Utilization" and "SeriousDlqin2yr
print cust_data1.columns
cust_data1=cust_data1.rename(columns={'RevolvingUtilization':'Rev_Utilization', '
```

In [ ]:
```
print cust_data1.columns
```

Sorting Data (single, multiple columns) in ascending and descending

In [ ]:
```
## Sorting the data
cust_data.sort_values(by='MonthlyIncome', ascending=False).head(10)

# ascending=False - get the sort in desc order
```

In [ ]:
```
Sort1 = stores.sort_values(by = "TotalSales")
```

In [ ]:
```
Sort2 = stores.sort_values(by = ["Location","TotalSales"])
```

In [ ]:
```
# Location in asc and TotalSales in desc
Sort3 = stores.sort_values(by = ["Location","TotalSales"], ascending = [True,Fals
```

In [ ]:
```
cust_demo.columns
```

In [ ]:
```
#Sorting Data with multiple columns
cust_demo.sort_values(by=['Location', 'Gender'], ascending=[False, True]).head(50
#cust_demo.sort_values(by=['Location', 'Gender'], ascending= False).head(5)
```

Type Conversions(Convert Data types of columns)

In [ ]:
```
#while importing
#df = pd.DataFrame(a, dtype='float')
#df[['col2','col3']] = df[['col2','col3']].apply(pd.to_numeric)
#There is also pd.to_datetime and pd.to_timedelta for conversion to dates and tim
#df.convert_objects(convert_dates='coerce', convert_numeric=True)

cust_data1.dtypes
#cust_data1.convert_objects(convert_dates='coerce', convert_numeric=True)
#cust_data1[['No_of_30_59_DPD', 'No_Of_OpenCreditLines']].apply(pd.to_numeric)

cust_data1['No_of_30_59_DPD'] = cust_data1['No_of_30_59_DPD'].astype('str')
```

In [ ]:
```
cust_data1.dtypes
```

In [ ]:
```
cust_data1['Deciles']=cust_data1['Deciles'].astype('str')
```

Resetting Index

It is used to create a DF with the data *conformed* to a new index.

If we subset a Series or DataFrame with an index object,

the data is *rearranged* to obey this new index and missing values are introduced wherever the data was not present

In [ ]:
```
cust_data1.info()
```

In [ ]:
```
cust_data1['ID']
```

In [ ]:
```
cust_data2=cust_data1.set_index("ID")
#cust_data1.set_index("ID", inplace=True)
```

In [ ]:
```
cust_data2.info()
```

In [ ]:
```
cust_data2.reset_index(inplace=True) #create variable
```

Handling Duplicates

- `df.duplicated()` Returns boolean Series denoting duplicate rows, optionally only considering certain columns
- `df.drop_duplicates()` Returns DataFrame with duplicate rows removed, optionally only considering certain columns

In [ ]:
```
print cust_data.assign(Dups = cust_data.duplicated()).head(5)
#cust_data['Dups']=cust_data.duplicated()
# Creates a boolean series to indicate which rows have dups
```

In [ ]:
```
print cust_data[cust_data.duplicated()]
# Retain the rows that are duplicates
```

In [ ]:
```
print cust_data[-cust_data.duplicated()].head(5)
# ignore duplicates
```

In [ ]:
```
print cust_data.drop_duplicates().head(3)
# retain the first occurrence of each row (drop dups)
```

In [ ]:
```
print cust_data.drop_duplicates(keep='last').head(3)
# retain the last occurrence of each row (drop dups)

print cust_data.drop_duplicates(keep=False).head(3)
```

In [ ]:
```
#To find the number of duplicated rows
cust_demo.duplicated().value_counts()
#cust_data.duplicated('ID').value_counts()
```

In [ ]:
```
cust_demo.columns
```

In [ ]:
```
cust_demo[cust_demo['ID'].duplicated()]
```

Handling Missing Data

Pandas treats the numpy `NaN` and the Python `None` as missing values.

--- These can be **detected** in a Series or DataFrame using **`obj.notnull()`**, **`obj.isnull()`** which returns a boolean.

--- **To filter out missing data** from a Series, or to remove rows (default action) or columns with missing data in a DataFrame, we use **`obj.dropna()`**

--- Missing Value **imputation** is done using the **`obj.fillna()`** method.

Missing Data in Numeric Variables

- Fill missing values with the
  - Drop rows where data is missing (if you have LOTS of data >1 mn rows)
  - Mean (if the distribution is symmetric)
  - Median (if the distribution is skewed)
  - Zeros (for data that indicates absence of a metric.)
  - ffill & bfill (eg. for time series data)
  - use a ML algorithm (KNN, Regression method) to predict the missing values using all other columns with non-missing data

Missing Data in Object Variables

- Fill missing values with
  - the Mode
  - a label that denotes missing values

In [ ]:
```
s12 = cust_data['MonthlyIncome']
# Detect missing values
#zip(s12, s12.isnull(), s12.notnull())
```

In [ ]:
```
sum(s12.isnull())
len(s12)
```

In [ ]:
```
# Replace missing values with 0
s12.fillna(0)

# Fill with median
s12.fillna(s12.median())

# dropping the observations
s12.dropna()
```

Handling Outliers

- Treating Outliers:

> clip_upper, clip_lower can be used to clip outliers at a threshold value. All values lower than the one supplied to clip_lower , or higher than the one supplied to clip_upper will be replaced by that value.
> This function is especially useful in treating outliers when used in conjunction with .quantile()
> (Note: In data wrangling, we generally clip values at the 1st-99th Percentile (or the 5th-95th percentile))

- Replacing Values:

> replace is an effective way to replace source values with target values by suppling a dictionary with the required substitutions

In [ ]:
```
cust_data.columns
```

In [ ]:
```
#Handling Outliers - Method1
print cust_data['MonthlyIncome'].head(10)
print cust_data['MonthlyIncome'].clip_upper(10000)
print cust_data['MonthlyIncome'].clip_lower(0)

#Handling Outliers-Method2
print cust_data['MonthlyIncome'].quantile(0.95)
print cust_data['MonthlyIncome'].quantile(0.05)
print cust_data['MonthlyIncome'].head(10).clip_upper(cust_data['MonthlyIncome'].q
print cust_data['MonthlyIncome'].head(10).clip_lower(cust_data['MonthlyIncome'].q
```

Handling Categorical variables for analysis - Create Dummies for a Categorical Variable

Create a (n x k) matrix of binary variables from a categorical variable of length n with k levels.

`pd.get_dummies(var)` does this.

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

In [ ]:
```
pd.get_dummies(cust_demo['Martial_Status'], prefix="D").head(10)
```

Steps for Creating Dummies

1. Identify the categorical variables you want to create dummies from
2. Create the dummies for n-1 categories for each
3. Join the dummies in to the original table
4. Drop the categorical variables in step 1.

In [ ]:
```
df_G = pd.DataFrame({'key': list('bbacccb'),
                 'val': np.random.randn(7) }).round(2)
df_G

#DataFrame({'key': df_G['key']}).assign(dummy_a = lambda x: x['key'] == 'a',
#                                       dummy_b = lambda x: x['key'] == 'b',
#                                       dummy_c = lambda x: x['key'] == 'c')

# Create and merge dummies in the same DF
print df_G,'\n'
print df_G.join(pd.get_dummies(df_G['key'], prefix='dummy')).drop('key', axis=1).
```

Special Case: Get Dummies from a Numeric

- Numeric ---> Categorical [by Cutting]
- Categorical ---> Binary [by Dummifying]

In [ ]:
```
# Create a categorical variable from a numeric and then compute dummies
df_G.val = np.random.rand(7).round(1)
df_G

pd.get_dummies(pd.cut(df_G['val'], 3, labels=list('XYZ')), prefix='dummy')
```

Apply functions to each element/rows or columns of a DataFrame

Using

- **`s.map()`**, apply a func to each element of a Series
- **`df.applymap()`** apply a func to each element of a DF
- **`df.apply()`** apply a func to rows/columns of a DF

> Lambda functions are also known as ANONYMOUS functions because they typically do not have a name.

- The are used extensively in Python, and even more with the `map(), applymap() and apply()` methods

In [ ]:
```
data = {'name': ['Dinkar', 'Vikalp', 'Sumeet', 'Shubham', 'Ramesh'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [4, 24, 31, 2, 3],
        'coverage': [25, 94, 57, 62, 70]}
df = pd.DataFrame(data)
df = df.iloc[:,[1,0,2,3]]
```

In [ ]:
```
capitalizer = lambda x: x.upper()

#def capitalizer(x):
#    return x.upper
```

In [ ]:
```
df['name'].apply(capitalizer)
```

In [ ]:
```
def Cube(x):
    return x ** 3

df_n = df.iloc[:,1:]
df_n.apply(Cube)
```

In [ ]:
```
df_n.applymap(Cube)
```

In [ ]:
```

```

In [ ]:
```
# Create a dataframe to work with
df = pd.DataFrame(np.random.randn(25).reshape(5,5),
                index=list('abcde'),
                columns=list('vwxyz')).round(2);
df
```

In [ ]:
```
# Write a function that formats a number to 2 decimal places
format8 = lambda x: '%.2f' %x

# SAME AS
# def format8(x):
#     return '%.2f' %x


# Apply this function to each element of the Series
print df.applymap(format8)

# Apply the function over columns
# Get the RANGE of each column
print df.apply(lambda x: x.max() - x.min())
```

In [ ]:
```
# Use a general function that returns multiple values
def func8(x):
    return pd.Series([x.min(), x.mean(), x.max()],
                  index=['MIN.', 'MEAN.', 'MAX.'])

df.apply(lambda x: func8(x))
```

In [ ]:
```
# Over Rows
df.apply(func8, axis=1)
```

In [ ]:
```
# Use a general function that returns multiple values
def var_summary(x):
    return pd.Series([x.count(), x.isnull().sum(), x.sum(), x.mean(), x.median(),
                  index=['N', 'NMISS', 'SUM', 'MEAN','MEDIAN', 'STD', 'VAR', 'MIN

df.apply(lambda x: var_summary(x)).T
```

In [ ]:
```
cust_data._get_numeric_data().apply(lambda x: var_summary(x)).T
```

Table Manipulations

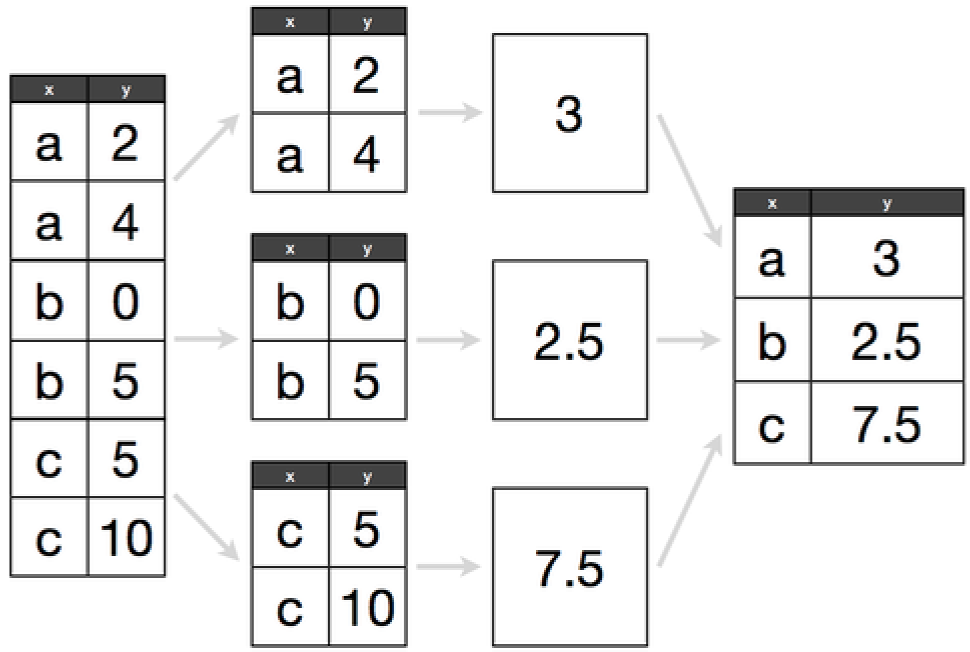
Implementing Split-Apply-Combine: The *groupby* method

- You may group along the rows or columns.
- Returns the groupby object that stores info on how to split the data
- To this object, we implement Aggregations (reduce size of data) or Transformations (no change in size) or Apply

Split - Apply - Combine!

In Data Analysis workflows, operations like `loading, cleaning and merging` are usually following by `summarizations` using some grouping variable(s). This includes *summary statistics* over variables or groups within variables, within-group *transformations* (like variable standardization), computing *pivot-tables* and group analyses.

- *Split:*
  - A DataFrame can be split up by rows(`axis=0`)/columns(`axis=1`) into **groups**.
  - We use `pd.groupby()` to create a groupby object
- *Apply:*
  - A function is applied to each group.
- *Combine:*
- The results of applying functions to groups are put together into an object
  - data types of returned objects are handled gracefully by pandas



In [ ]:
```
cust_data.columns
```

In [ ]:
```
## ANALOGY
#SELECT x, avg(y) as avg_y
#FROM Table_1
#GROUP BY x

#df_1.groupby('x').mean()
```

In [ ]:
```
# Or you can created a generic groupby object and re-use it

grouped = cust_data[['RevolvingUtilization','SeriousDlqin2yrs']].groupby('Serious
print type(grouped)

print grouped.max()
print grouped.min()
print grouped.mean()
```

In [ ]:
```
pd.DataFrame(pd.concat([grouped.max(), grouped.min(), grouped.mean(), grouped.std
          columns=['Max', 'Min', 'Mean', 'Stddev'])
```

GroupBy objects

- `DataFrame.groupby(<key>)` will produce a groupby object
- have a `.size()` method, which returns the count of elements in each group.
- can be subsetted using column names (or arrays of column names) to select variables for aggregation
- have optimized methods for general aggregation operations like -
    - `count, sum`

- `mean, median, std, var`
- `first, last`
- `min, max`
- methods like `.describe` apply to these objects

By far, the most important `GroupBy Object` methods are `.agg()` `.transform()`, and `.apply()`

In [ ]:
```
df = pd.DataFrame({'k1': list('abcd' * 25),
                   'k2': list('xy' * 25 + 'yx' * 25),
                   'v1': np.random.rand(100),
                   'v2': np.random.rand(100)}).round(2)
df[:15]
```

In [ ]:
```
print '\n', df.groupby('k1').mean()
print '\n', df.groupby('k2').sum()
```

In [ ]:
```
# Group by two keys
df.groupby(['k1', 'k2']).mean()
```

In [ ]:
```
grpd = df.groupby(['k1', 'k2'])
```

In [ ]:
```
type(grpd)
```

In [ ]:
```
print grpd['v1'].sum()
print
print grpd['v2'].median()
```

In [ ]:
```
obj = df.groupby(['k1'])
```

In [ ]:
```
obj.size()
```

In [ ]:
```
len(obj)
# there are 4 groups
```

In [ ]:
```
obj.groups.keys()
# names of the 4 groups
```

In [ ]:
```
obj.get_group('b')[:5]
```

In [ ]:
```
obj.get_group('c')[:5]
```

In [ ]:
```
print obj.mean()
```

In [ ]:
```
print df.groupby(df.k1).agg('mean').add_prefix('mu_')
```

In [ ]:
```
pd.concat([df.groupby(df.k1).agg('mean').add_prefix('mu_'),
           df.groupby(df.k1).agg('std').add_prefix('sigma_')], axis=1)
```

In [ ]:
```
cust_demo.columns
```

In [ ]:
```
cust_demo[['Location', 'Gender','age']].groupby(['Location', 'Gender']).agg('mean
```

In [ ]:
```
cust_demo[['Location', 'Gender','age']].groupby(['Location', 'Gender']).agg(['mea
```

In [ ]:
```
pd.concat([cust_demo[['Location', 'Gender','age']].groupby(['Location', 'Gender']
cust_demo[['Location', 'Gender','age']].groupby(['Location', 'Gender']).agg('std'
```

In [ ]:
```
cust_data.columns
```

The `.apply()` method

takes as argument the following:

- a general or user defined function
- any other parameters that the function would take

Syntax: `dataFrame.groupby('column').apply(udf, udf-par_1, udf_par_2 ...)`

In [ ]:
```
#Finding top 5 customers with income by each category in SeriousDlqin2yrs

def topN(data, col, N):
    """
    Takes a dataframe, and returns N rows for given column (sorted by this column
    """
    #return data.sort_values(by=col, ascending=False).loc[:, col].head(N)
    return data.sort_values(by=col, ascending=False).head(N)
```

In [ ]:
```
cust_data.groupby('SeriousDlqin2yrs').apply(topN, col='MonthlyIncome', N=5)
```

In [ ]:
```
#Example 2: want to find the number of rows and columns

def analyze(df):
    """
    RETURNS THE NROWS AND NCOLS OF A DATAFRAME
    """
    return pd.Series({"nrow": len(df), "ncol": len(df.columns)})
```

In [ ]:
```
# Rows and Cols per group
print cust_data.groupby('SeriousDlqin2yrs').apply(analyze)
print
print cust_data.groupby('SeriousDlqin2yrs').apply(sum)
```

In [ ]:
```
print cust_data.groupby('SeriousDlqin2yrs').size()
```

In [ ]:
```

```

Reshaping your data with `stack`, `unstack` and `pivot_table`

Usually, for convenience, data in relational DB is stored in the **long format**

```
* fewer columns, label duplication in keys
```

For certain kinds of analysis, we might prefer to have the data in the **wide format**

```
* more columns, unique labels in keys
```

Reshaping using `stack()` and `unstack()`

Hierarchical Indexing provides a convenient way to reshape data;

```
* `stack` pivots the columns into rows
* `unstack` pivots rows into columns
```

Long to Wide

In [ ]:
```
df = pd.DataFrame({'A': list('x' * 5) + list('y' * 5),
                   'B': list('abcde' * 2),
                   'C': np.random.randint(0, 100, 10)})
df.set_index(['A', 'B'], inplace=True)
df
```

In [ ]:
```
df.unstack()
```

> To use stack/unstack, we need the values we want to shift from rows to columns or the other way around as the index

Wide to Long

In [ ]:
```
df_wide = pd.DataFrame(np.random.randint(0, 100, 25).reshape(5, 5),
                      index=list('abcde'),
                      columns=list('vwxyz'))

df_wide
```

In [ ]:
```
df_long = df_wide.stack()
```

In [ ]:
```
df_long
```

Converting data from 'long' to 'wide' format using

.pivot()

The `df.pivot()` method takes the names of columns to be used as row (`index=`) and column indexes (`columns=`) and a column to fill in the data as (`values=`)

In [ ]:
```
df = pd.DataFrame({'date': (list(pd.date_range('2000-01-03', '2000-01-05')) * 4),
          'item': (list('ABCD'*3)),
          'status': (np.random.randn(12))})
print df
```

In [ ]:
```
print df.set_index(['date', 'item'])
# The data is in Long-format
```

In [ ]:
```
print df.set_index(['date', 'item']).unstack()
# Unstacking long data into wide
```

In [ ]:
```
# Same effect using Pivot

print df.pivot(index='date', columns='item', values='status')
```

In [ ]:
```
%timeit df.set_index(['date', 'item']).unstack()
```

In [ ]:
```
%timeit df.pivot(index='date', columns='item', values='status')
```

> Note: Pivot is just a convenient wrapper function that replaces the need to create a hierarchical index using `set_index` and reshaping with `stack`

In [ ]:
```
print pd.pivot_table(data=df,
                     index='date',
                     columns='item',
                     values='status',
                     aggfunc=np.sum)
```

In [ ]:
```
print pd.pivot_table(data=df,
                     index='date',
                     columns='item',
                     values='status',
                     aggfunc='sum')
```

In [ ]:
```
print pd.pivot_table(data=df,
                     index='date',
                     columns='item',
                     values='status',
                     aggfunc='mean')
```

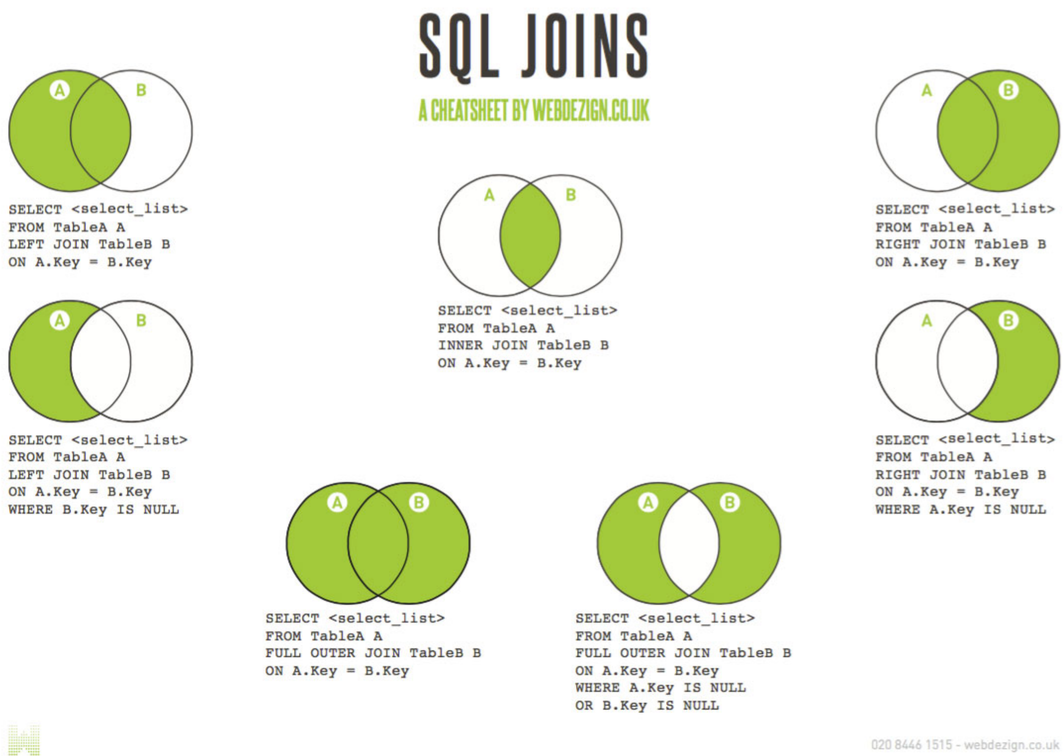
In [ ]:
```
print pd.pivot_table(data=cust_demo, index='Location', columns='Gender', values='
```

In [ ]:
```
pt=pd.pivot_table(data=cust_demo, index=['Location','Martial_Status'], columns=['
```

In [ ]:
```
pt.to_csv('C:\\Users\\ChandraMouli\\Desktop\\summary.csv')
```

In [ ]:
```
print pd.pivot_table(data=cust_data, index='SeriousDlqin2yrs', values='No_of_Real
```

MERGING - JOINING



`pandas.merge()` is similar to the *SQL join* operations; it links rows of tables using one or more *keys*

Syntax:

```
merge(df1, df2,
       how='left', on='key', left_on=None, right_on=None,
       left_index=False, right_index=False,
       sort=True, copy=True,
       suffixes=('_l', '_r'))
```

The syntax includes specifications of the following arguments

- **Which column to merge on;**

  - the `on='key'` if the same key is in the two DFs,
  - or `left_on='lkey', right_on='rkey'` if the keys have different names in the DFs
  - Note: To merge on multiple keys, pass a list of column names

- **The nature of the join;**

  - the `how=` option, with `left`, `right`, `outer`
  - By default, the merge is an `inner` join

- Tuple of string values to append to **overlapping column names** to identify them in the merged dataset

  - the `suffixes=` option
  - defaults to `('_x', '_y')`

- If you wish **to merge on the DF index**, pass `left_index=True` or `right_index=True` or both.

- Sort the result DataFrame by the join keys in lexicographical order or not;
  - `sort=` option; Defaults to True, setting to False will improve performance substantially in many cases

> *Note:* For the **official Documentation** refer http://pandas.pydata.org/pandas-docs/dev/merging.html (http://pandas.pydata.org/pandas-docs/dev/merging.html)

In [ ]:
```
# Let's define a few toy datasets to use as examples

df0 = pd.DataFrame({'key': ['a', 'b', 'c', 'd', 'e'], 'data0': np.random.randint(
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': np.rando
df2 = pd.DataFrame({'key': ['a', 'b', 'd', 'f', 'g'], 'data2': np.random.randint(
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data3': np.rand
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'], 'data4': np.random.randint(0, 100, 3

print 'df0\n', df0, '\ndf1\n', df1, '\ndf2\n', df2, '\ndf3\n', df3, '\ndf4\n', df
```

In [ ]:
```
# Inner Join (Default)

print pd.merge(df0, df2)
# or
print pd.merge(df0, df2, how='inner', on='key')

# We see that its an inner join by default (output key is the intersection of inp
# Merge happens on the column 'key' which is common to both datasets;
    # We could've written pd.merge(df1, df2, on='key', how='inner') to the same e
```

In [ ]:
```
# Outer Join
print pd.merge(df0, df2, how='outer')

# The output table has a UNION of the keys
```

In [ ]:
```
# Left Join
print pd.merge(df0, df2, how='left')

# The output table has all the key values from the left table, and matching ones j
```

In [ ]:
```
# Right Join
print pd.merge(df0, df2, how='right')

# The output table has all the key values from the right table, and matching ones
```

In [ ]:
```
# Check the common columns are exist or not
print df1.columns.tolist()
print df4.columns.tolist()

print np.intersect1d(df1.columns.tolist(), df4.columns.tolist())

#pd.merge(df1, df4)
# would yield an error because there are no matching column names to merge on
# If there are no common keys, `pd.merge` will throw a `MergeError
```

In [ ]:
```
# 2. Specifying which columns to merge on (if keys have different names in datase

pd.merge(df1, df4, left_on='key', right_on='rkey')
# still an inner join!
```

In [ ]:
```
#pd.merge(cust_data, cust_demo, how='left', on='ID').head(3)
pd.merge(cust_data, cust_demo, how='left', left_on='ID', right_on='ID').head(3)
```

The `.join()` method

.join is a convenient **DataFrame method** for combining many DataFrames objects with the same or similar indexes but non-overlapping columns into a single result DataFrame.

By default, the `join` method performs a *left join* on the join keys.

For simple **index-on-index merges** we can pass a list of DataFrames to `join`.

In [ ]:
```
df = pd.DataFrame(np.random.randint(0, 50, 32).reshape(8, 4), columns=list('WXYZ'

df1 = df.ix[2:, ['W', 'X']]
df2 = df.ix[:5, ['Y', 'Z']]

print df, '\n\n', df1, '\n\n', df2
```

In [ ]:
```
# Default actions is a left join on the indexes
df1.join(df2)
```

In [ ]:
```
pd.merge(df1, df2, how='left',
         right_index=True, left_index=True)
```

In [ ]:
```
# Create a couple more DFs with the same index
df3 = df.ix[0:3, ['X', 'Z']]
df3.columns = ['P', 'Q']

df4 = df.ix[4:6, ['W']]
df4.columns = ['R']

print df3, "\n\n", df4
```

In [ ]:
```
# Merging multiple DFs with the same index by passing a list of names to .join
df1.join([df2, df3, df4]).fillna('')
```

In [ ]:
```
df2.join([df1, df3, df4], how='outer').fillna('')
```

Concatenating DataFrames

- (aka binding, stacking, union all)

a. Series objects with small index overlap

```
* concat with axis=0 (default) will append the Series (~rbind)
* concat with axis=1 will merge the Series to produce a DF (~outer join)
```

In [ ]:
```
# Create toy Series with non-overlapping indices
s1 = pd.Series(np.random.randn(3).round(2), index=list('abc'), name='S1')
s2 = pd.Series(np.random.randn(5).round(2), index=list('cdefg'), name='S2')
s3 = pd.Series(np.random.randn(4).round(2), index=list('fghi'), name='S3')

print s1, '\n\n S2:\n', s2, '\n\n S3:\n', s3
```

In [ ]:
```
# Default action is to append the data
pd.concat([s1, s2, s3], axis=0)
```

In [ ]:
```
# concat with axis=1 (non-overlapping index)
print pd.concat([s1, s2, s3], axis=1)
```

In [ ]:
```
# Passing keys= creates a hierarchical index when appending (axis=0)

pd.concat([s1, s2, s3], axis=0, keys=['one', 'two', 'thr'])
```

In [ ]:
```
# Passing keys= gives names to columns when using axis=1

print pd.concat([s1, s2, s3], axis=1, keys=['S1', 'S2', 'S3'])
```

b. Series objects with overlapping index

- If there is an overlap on indexes, we can specify **join=** to intersect the data

- Note that the `join=` option takes only `'inner'` and `'outer'`

In [ ]:
```
s4 =pd.Series(np.random.randn(5), index=list('abcde'), name='S4')
print s4
```

In [ ]:
```
# concat with overlapping index (default join type is outer)
print pd.concat([s1, s4], axis=1)
```

In [ ]:
```
# if we specify a join type, this will be equivalent to a merge
print pd.concat([s1, s4], axis=1, join='inner')
```

In [ ]:
```
cust_demo.head(3)
```

In [ ]:
```
cust_new.head(3)
```

In [ ]:
```
# if we specify a join type, this will be equivalent to a merge
#pd.concat([cust_demo.head(3),cust_new.head(3)])
pd.concat([cust_demo.head(3),cust_new.head(3)]).fillna('')
```

Dealing with String Data

These include methods applied to string objects that

- split a string by given delimiter - `.split()`
- trim whitespace - `.strip()`
- concatenate strings - `.join()`
- detect substrings - `.find()` and `.index()`
- count occurrences - `.count()`
- find and replace - `.replace()`

In [ ]:
```
s = 'ready, set ,   go '
```

In [ ]:
```
s.split(',')
```

In [ ]:
```
s.split(' ')
```

In [ ]:
```
'_'.join(s.split(','))
```

In [ ]:
```
# String Splitting
'_'.join([x.strip() for x in s.split(',')])
```

In [ ]:
```
# Trimming whitespace
pieces = [x.strip() for x in s.split(',')]
pieces
# Also see rstrip, lstrip
```

In [ ]:
```
'_#_'.join(list('abcde'))
```

In [ ]:
```
# Concatenating Strings
print '::'.join(pieces)
print '--'.join(pieces)
print ' '.join(pieces)
```

In [ ]:
```
# Does a Substring belong to a string
print 'steady' in s
print 'set' in s
```

In [ ]:
```
# Locate a substring
s.index('go')
```

In [ ]:
```
s[15:17]
```

In [ ]:
```
#find vs index
sentence = 'the sun rises in the east'
sentence.find('east')
```

In [ ]:
```
sentence.index('east')
```

In [ ]:
```
print sentence.find('west')
#print sentence.index('west') #it will throw an error
```

In [ ]:
```
# Locate a substring
s.find(',')
```

In [ ]:
```
# Count occurrences
s.count(',')
```

In [ ]:
```
sentence.endswith('east')
```

In [ ]:
```
s2 = 'the quick brown fox jumps over the lazy dog'
s2.find('fox')

print 'lazy' in s2

print s2.endswith('dog')
```

In [ ]:
```
s.startswith('ready')
# similarly .endswith()
```

In [ ]:
```
cust_demo.columns
```

In [ ]:
```
cust_demo.Location.head(5)
```

In [ ]:
```
cust_demo[['city', 'state']]=cust_demo['Location'].str.split(',', expand=True)
```

In [ ]:
```
cust_demo.head(3)
```

Regular Expressions

A Regex is a sequence of characters that define a search pattern used in find-and-replace actions.

Example: The regex

- `\s+` describes one or more whitespaces
- `(?<=\.) {2,}(?=[A-Z])` matches at least two spaces occurring after period (.) and before an upper case letter

Note:

- Before a regex is applied to a string, it must be *compiled* to create a reusable regex object.
- The object's methods can then be called on a string.
- These include:
  - **`split`**,
  - **`findall`** (returns all matches),
  - **`match`** (checks only the beginning of the string),
  - **`search`** (returns the first occurrence)
  - **`sub`** (returns a new string with occurrences of the pattern replaced with the supplied string)

Syntax:

1. `import re`
2. `r_obj = re.compile('my-regex')`
3. `r_obj.method(my-text)`

In [ ]:
```
import re
import pandas as pd
```

In [ ]:
```
# Create a dataframe with a single column of strings
data = {'raw': ['Arizona 1 2014-12-23       3242.0',
                'Iowa 1 2010-02-23       3453.7',
                'Oregon 0 2014-06-20       2123.0',
                'Maryland 0 2014-03-14       1123.6',
                'Florida 1 2013-01-15       2134.0',
                'Georgia 0 2012-07-14       2345.6']}
df = pd.DataFrame(data, columns = ['raw'])
df
```

In [ ]:
```
# Which rows of df['raw'] contain 'xxxx-xx-xx'?
df['raw'].str.contains('....-..-..', regex=True)
```

In [ ]:
```
# In the column 'raw', extract single digit in the strings
df['female'] = df['raw'].str.extract('(\d)', expand=True)
df['female']
```

In [ ]:
```
# In the column 'raw', extract xxxx-xx-xx in the strings
df['date'] = df['raw'].str.extract('(....-..-..)', expand=True)
df['date']
```

In [ ]:
```
# In the column 'raw', extract ####.## in the strings
df['score'] = df['raw'].str.extract('(\d\d\d\d\.\d)', expand=True)
df['score']
```

In [ ]:
```
# In the column 'raw', extract the word in the strings
df['state'] = df['raw'].str.extract('([A-Z]\w{0,})', expand=True)
df['state']
```

In [ ]:
```
df
```

HANDLING TIME SERIES DATA

In [ ]:
```
pd.datetime.now()
```

In [ ]:
```
# Create a date value
# Syntax: pd.datetime(year, month, day, hour, mins)
dt_1 = pd.datetime(2016, 1, 1)
```

In [ ]:
```
# Create a date range
# Syntax: pd.date_range(start, stop, freq=)
pd.date_range(pd.datetime(2016, 1, 1), pd.datetime(2016, 7, 1), freq="W")
```

In [ ]:
```
dates = pd.date_range('1950-01', '2013-03', freq='M'); dates
```

In [ ]:
```
ts = pd.DataFrame(np.random.randn(758, 4), columns=list('ABCD'), index=dates)
```

In [ ]:
```
#sub setting time series
# Between June 1951 to Jan 1952
ts[pd.datetime(1951, 6, 1):pd.datetime(1952, 1, 1)]
```

In [ ]:
```
ts['year'] = ts.index.year
```

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

In [ ]:
```
# Aggregating data by year
print ts.groupby('year').sum().tail(5)
```

```
Time/Date Components

There are several time/date properties that one can access from Timestamp or a
collection of timestamps like a DateTimeIndex.

Property                Description
--------                ------------
year                    The year of the datetime
```

```
month                    The month of the datetime
day                      The days of the datetime
hour                     The hour of the datetime
minute                   The minutes of the datetime
second                   The seconds of the datetime
microsecond              The microseconds of the datetime
nanosecond               The nanoseconds of the datetime
date                     Returns datetime.date (does not contain timezone
information)
time                     Returns datetime.time (does not contain timezone
information)
dayofyear                The ordinal day of year
weekofyear               The week ordinal of the year
week                     The week ordinal of the year
dayofweek                The numer of the day of the week with Monday=0, Sunday=6
weekday                  The number of the day of the week with Monday=0,
Sunday=6
weekday_name             The name of the day in a week (ex: Friday)
quarter                  Quarter of the date: Jan=Mar = 1, Apr-Jun = 2, etc.
days_in_month            The number of days in the month of the datetime
is_month_start           Logical indicating if first day of month (defined by
frequency)
is_month_end             Logical indicating if last day of month (defined by
frequency)
is_quarter_start         Logical indicating if first day of quarter (defined by
frequency)
is_quarter_end           Logical indicating if last day of quarter (defined by
frequency)
is_year_start            Logical indicating if first day of year (defined by
frequency)
is_year_end              Logical indicating if last day of year (defined by
frequency)
is_leap_year             Logical indicating if the date belongs to a leap year

Furthermore, if you have a Series with datetimelike values, then you can access
these properties via the .dt accessor, see the docs
```

Random Sampling

We can use the `np.random.permutation` function (passing nrows as an argument) for randomly reordering a Series.

To select a random sample, create an index and subset the DF using it.

- **Without replacement**: slice off the first *k* rows; where *k* is the size of the subset you desire
- **With replacement**: use `np.random.randint(start, stop, size=)` to draw integers at random

Sampling using .sample() method

In [ ]:
```
# WIthout replacement
cust_demo.sample(n=700, replace=False).duplicated().value_counts()
```

In [ ]:
```
# WIth replacement
df.sample(frac=0.7, replace=True).duplicated().value_counts()
```

Data visualization in Python (Plotting & Visualization)

Python DataViz Libraries

- Matplotlib (http://matplotlib.org/gallery.html)
- Seaborn (https://stanford.edu/~mwaskom/software/seaborn/index.html)
- GGPLOT (http://ggplot.yhathq.com/)
- Altair (https://github.com/ellisonbg/altair)
- Plotly (https://plot.ly/python/)

1. matplotlib basics

- Run **`import matplotlib.pyplot as plt`**
- Create a figure object using **`plt.figure`**
- Add subplots to it using **`add_subplot`**
  - This creates **AxesSubplot** objects on which you can place plots
- Use a plotting command like **`plt.plot`** and matplotlib will place your plot on this canvas

1.1 Figure, Subplots, AxisSubplot objects and your plot

Create a 2x2 figure and add three plots to it

In [ ]:
```
## add necessary libraries
import matplotlib.pyplot as plt
#%pylab inline

%matplotlib inline
#Populating the interactive namespace from numpy and matplotlib
```

In [ ]:
```
# Create an empty figure
fig = plt.figure(figsize=(12, 8))
print type(fig)
```

In [ ]:
```
plt.figure?
```

In [ ]:
```
# Run plt.figure? to check out figure options like size, dpi, color

axsp1 = fig.add_subplot(2, 2, 1)
# There will be 2 x 2 subplots on the figure and ax1 will put your figure on subp

axsp2 = fig.add_subplot(2, 2, 2)
axsp3 = fig.add_subplot(2, 2, 3)
axsp4 = fig.add_subplot(2, 2, 4)
# Now, we have three AxesSubplot objects on our figure.
```

In [ ]:
```
# First plot: timeseries
axsp1.plot(np.random.randn(40).cumsum(), 'r--')

# Second plot: histogram
axsp2.hist(np.random.randn(400), bins=10, color='b', alpha=0.3)

# Third plot: scatterplot
axsp3.scatter(np.arange(30), 4 * np.arange(30) + 6 * np.random.randn(30))
# Note: if you make changes to the AxisSubplot object, you'll have to re-run the
```

In [ ]:
```
fig
```

Shorthand to achieve the same effect

- Create a grid figure using **`plt.subplots`**
    - Syntax: `fig, axes = plt.subplots(rows, cols, figsize = (width, height), sharex=False, sharey=False)`
- It returns an array of **AxisSubplot** objects
- Reference them using basic indexing (Saves typing!)

`plt.subplots` has some interesting options such as `sharex/sharey` which are useful when comparing data on the same scale

Run `plt.subplots?` for more.

In [ ]:
```
fig, axes = plt.subplots(2, 2, figsize = (12, 6), sharex=True)
# returns an array
```

In [ ]:
```
axes[0, 0].plot(np.random.randn(50).cumsum(), 'r--')
axes[1, 1].scatter(np.arange(30), np.log10(np.arange(30)))
fig
```

> NOTE: subplots.adjust is a Figure method that can be used to adjust figure parameters like spacing between subplots

In [ ]:
```
fig1, axes1 = plt.subplots(2, 2, figsize=(12, 4), sharex=True, sharey=True)

for i in range(2):
    for j in range(2):
        axes1[i, j].hist(np.random.randn(500), bins=15, alpha=0.4, color='c')

plt.subplots_adjust(wspace=0.1, hspace=0.1)
# comment out the plt.subplots line and re-run. See what happens
```

Plot Formatting

a. Color, Linestyle and Markers

The `plot` function takes `x, y` and optionally an abbreviation to specify `marker, color, and style`

Example: Abbreviations work as `color-marker-style`, so `'g--'` means color = 'green' and linestyle = '--'

In [ ]:
```
plt.figure(figsize=(15, 5));
plt.plot(np.sin(np.arange(50)), 'c-*');
```

b. Ticks, Labels, Legends

In [ ]:
```
f = plt.figure(figsize=(12, 5))
ax1 = f.add_subplot(1, 1, 1)
ax1.plot(4 + 2 * np.sin(np.arange(50)), 'g--', label='4 + 6*sin(x)')
```

In [ ]:
```
# Ticks
ax1.set_xticks([5, 15, 25, 35, 45])

# Chart title
ax1.set_title('This is a placeholder Plot Title')

# Axis Label
ax1.set_xlabel('Values of X')
ax1.set_ylabel('Values of Y')

f
```

In [ ]:
```
# Saving plots to file

# Add more plots
ax1.plot(np.log(np.arange(50)), 'r', label='log(x)')
ax1.plot(np.sqrt(np.arange(50)), 'b*--', label='sqrt(x)')
# Add a Legend
ax1.legend(loc='best')

plt.savefig('threePlots.png')
```

Plotting in pandas

- There are high level plotting methods that take advantage of the fact that data are organized in DataFrames (have index, colnames)
- Both `Series` and `DataFrame` objects have a `pandas.plot` method for making different plot types
- Other parameters that can be passed to `pandas.plot` are:
  - `xticks, xlim, yticks, ylim`
  - `label`
  - `style` (as an abbreviation,) and `alpha`
  - `grid=True`
  - `rot` (rotate tick labels by and angle 0-360)
  - `use_index` (use index for tick labels)

One variable (plotting a Series)

```
s = pd.Series(np.random.randn(100)).cumsum()
s.plot(figsize=(13, 4));
```

In [ ]:
```
# Chart with options
s.plot(kind='line',
       grid=True,
       legend=True,
       label='timeseries',
       title='Random Normal Numbers - Cumulative Series',
       xlim=(0, 100),
       ylim=(-10, 12),
       xticks=np.arange(0, 100, 10),
       yticks=np.arange(-10, 12, 2),
       style='go-',
       alpha=0.2,
       figsize=(15, 4)
      )
```

In [ ]:
```
# One Variable as a Histogram
pd.Series(np.random.randn(10000)).plot(kind='hist',
                                       bins=50,
                                       color='k',
                                       alpha=0.3,
                                       title='A histogram');
```

In [ ]:
```
# One Variable as a Histogram
pd.Series(np.random.randn(10000)).plot.hist(bins=50,
                                            color='r',
                                            alpha=0.3,
                                            title='Another histogram');
```

In [ ]:
```
(pd.Series(pd.cut(np.random.randint(1, 100, 1000), 5, labels=list('abcde')))
 .value_counts()
 .plot.bar(title='A Bar Chart',
          ylim=(0, 400),
          alpha=0.6))
```

In [ ]:
```
s2 = pd.Series(np.random.beta(2, 5, 1000))
s3 = pd.Series(np.random.beta(0.2, 0.5, 1000))
s2.plot.box()
```

In [ ]:
```
s3.hist(bins=30)
```

Multiple Variables (plotting a DataFrame)

We can choose between plotting

- All Variables on one plot
- Each variable on a separate plot

In addition to the parameters above, `DataFrame.plot` also takes

- `subplots=False` (default is to plot all on the same figure)
- `sharex=False, sharey=False`
- `figsize`
- `title, legend`
- `sort_columns`

a. Variables on the same plot

In [ ]:
```
df = pd.DataFrame(np.random.randn(5000, 5), columns=list('PQRST'));
print df.shape
df[:5]
```

In [ ]:
```
# Default plot
df.cumsum().plot(figsize=(14, 6))
```

In [ ]:
```
#b. Each variable on its own plot
df.cumsum().plot(figsize=(12, 16), subplots=True)
```

In [ ]:
```
# Let's make some radical changes
df.cumsum().plot(kind='line',
                 figsize=(12, 16),
                 title='Each variable is now on its own plot, but the axes are sh
                 subplots=True,
                 sharex=True,
                 sharey=True
                );
```

Barplots

This is as simple as passing `kind=bar` or `kind=barh` (for horiz bars) to `pd.plot`

One Variable (simple barplot)

In [ ]:
```
fig, axes = plt.subplots(3, 1, figsize=(8, 8))
s = pd.Series(np.random.rand(10), index=list('abcdefghij'))

s.plot(kind='bar',
       ax=axes[1],
       color='k',
       alpha=0.6)

s.plot(kind='barh',
       ax=axes[0],
       color='k')

s.plot(ax=axes[2], color='g');
```

In [ ]:
```
df = pd.DataFrame(np.random.rand(5,5), index=list('ABCDE'), columns=list('PQRST'))
print df
```

In [ ]:
```
df.plot(kind='bar', stacked=True, figsize=(12, 5))
plt.savefig('stackedBarcharts.jpeg')
```

> Note: Functions value_counts() and pd.crosstab() prove helpful to prepare data for stacked bar charts

d. Histograms & Density Plots

- *Histograms*: Pass `kind='hist'` to `pd.plot()` or use the method `pd.hist()`
- *Density Plots*: Use `kind='kde'`

In [ ]:
```
#Using the .hist() method
pd.Series(np.random.randn(1000)).hist(bins=20, alpha=0.4);
```

In [ ]:
```
#Using the .plot() method
pd.Series(np.random.randn(1000)).plot(kind='hist', bins=20, color='Y');
```

In [ ]:
```
s = pd.Series(np.random.randn(10000))
s.plot(kind='kde', color='b')
```

In [ ]:
```
# A bimodal distribution
s1 = np.random.normal(0, 1, 2000)
s2 = np.random.normal(9, 2, 2000)

v = pd.Series(np.concatenate([s1, s2]))

v.hist(bins=100, alpha=0.4, color='B', normed=True)
v.plot(kind='kde', style='k--')
```

Scatter Plots

- `.plot(kind='scatter')`
- `.scatter()`

In [ ]:
```
df = pd.DataFrame({'A': np.arange(50),
                   'B': np.arange(50) + np.random.randn(50),
                   'C': np.sqrt(np.arange(50)) + np.sin(np.arange(50)) })
print df[:10]
```

In [ ]:
```
# Two variable Scatterplot
plt.scatter(df['B'], df['C'])
plt.title('Scatterplot of X and Y')
```

In [ ]:
```
df.plot(kind='scatter', x='B', y='C', title = 'Scatterplot')
```

In [ ]:
```
df.plot.scatter(x='B', y='C', title = 'Scatterplot', color='r')
```

Scatterplot Matrix

A MOST important visual that allows you to see, for numeric variables:

- The distribution of each (histograms or kde along the diagonal)
- The relationships between variables (as pairwise scatterplots)

In [ ]:
```
pd.scatter_matrix(df, color='k', alpha=0.5, figsize=(12, 6))
tight_layout()
```

In [ ]:
```
pd.scatter_matrix(df, diagonal='kde', color='k', alpha=0.5, figsize=(12, 6))
tight_layout()
```

In [ ]:
```
pd.scatter_matrix(df);
```

In [ ]:
```
#Time Series Plots
dates = pd.date_range('1950-01', '2013-03', freq='M')
ts =pd.DataFrame(np.random.randn(758, 4), columns=list('ABCD'), index=dates)
ts['year'] = ts.index.year
# Visualize Trends over time
ts.drop('year', axis=1).cumsum().plot(figsize=(10, 6))
```

In [ ]:
```
cust_demo.age.hist(bins=50, color='R')
```

In [ ]:
```
cust_demo.age.plot(kind='hist', bins=50, color='R');
```

In [ ]:
```
cust_demo.age.plot.box()
```

In [ ]:
```
cust_demo.Martial_Status.value_counts().plot(kind='bar', color='R', alpha=0.5)
```

In [ ]:
```
cust_demo.Location.value_counts().plot(kind='barh', color='R', alpha=0.5)
```

In [ ]:
```
pd.crosstab(cust_demo.Martial_Status,cust_demo.Own_House).plot(kind='bar', color=
```

In [ ]:
```
cust_data.plot(kind='scatter', x='RevolvingUtilization', y='MonthlyIncome', title
```

In [ ]:
```
#pd.scatter_matrix(cust_data._get_numeric_data())
#pd.scatter_matrix(cust_demo.select_dtypes(include = ['number']))
pd.scatter_matrix(cust_demo.select_dtypes(include = ['number']), color='k', alpha
tight_layout()
```

In [ ]:
```

```