Kickstart Python

Module for rapid deployment of Python

 September 2017

Introduction

- Created by Guido Van Rossum and first released in 1991
- He named it after BBC's TV Show: Monty Python's Flying Circus
- High Level Language Programming Language
- Emphasis on code readability using whitespace instead of keywords
- Designed initially for *Programming*
- Later evolved to perform *Scripting* and finally for *Data Analysis*

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

- Created by Wes McKinney in 2008 while working at AQR Capital Mgmt.
- Created a tool for high performance quantitative analysis, especially while handling timestamp data
- ***Pandas*** was derived from Panel Data an econometric term for multidimensional, structured dataset

Van Rossum

Wes McKinney

Python: Classes & Objects

- ***Procedure Oriented Programming:*** Program designed around blocks of code that operate on data. Such blocks are called Functions
- ***Object Oriented Programming:*** Focusses on functions which are wrapped around data
- ***Objects and Classes***
  - Objects are instances of Classes
  - Functions packaged with an object are called Methods
  - Attributes contained within a Method are called Arguments

```
class PartyAnimal:
    x = 0
    def party(self) :
    self.x = self.x + 1
    print "So far",self.x

an = PartyAnimal()

an.party()
an.party()
an.party()
```

Class: Defines the type of object in Python. Example all types of dogs belong to Class: DOG

Object: Defines one instance of the class. Example Class: DOG, Object: Doberman

Method: Defines abilities of an object, what all it can do. Example Class: DOG, Object: Doberman, Method: Bark()

Arguments: Within methods for python objects, we have attributes which customize the result of output. Example Class: DOG, Object: Doberman, Method: Bark() Attribute: Bark(Tone=High, Medium, Low)

NOTE: This module is developed using Python 3.x

Jupyter Notebook

- Was known as ***iPython Notebook***
- Allows block wise coding & interpretation by viewing output directly below the code
- Great for line wise debugging and therefore useful for Data Analysis
- Interactive Interface including
  - *Inline Plotting, Printing and Viewing*
  - *HTML Markdown for commenting and notes*

In [13]: Input Code Block
Out [13]: Output of respective code block

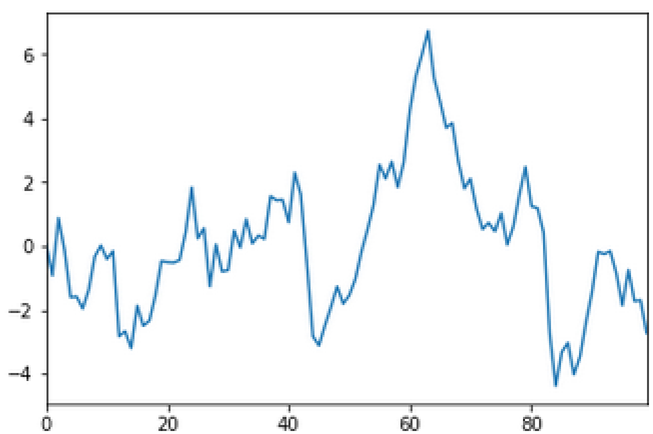
```
In [13]: x.values[:15]

Out[13]: array([ 0.04088493, -0.91920131,  0.88183568, -0.12742566, -1.60482979,
                -1.58480819, -1.9593462 , -1.36336217, -0.32561307,  0.01902615,
                -0.40668233, -0.15964083, -2.82747895, -2.67311602, -3.19894915])

In [14]: x.plot()

Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd642a45650>
```

Gray cell is the block where line(s) of code can be typed



Keyboard Shortcuts:
Adding New Cell Above : ***Esc + A***
Adding New Cell Below : ***Esc + B***
Deleting a cell: ***Esc + D + D***

Inline Plotting can be done using Matplotlib. Datasets can be viewed in the output block by simply typing the name of the dataset object and pressing **Shift + Enter**

Basic Methods For Python Objects

Python objects include Lists, Dictionary, Tuples, Strings etc. These are useful for handling small chunks of data. These are particularly useful when manipulating individual cell level data. Various methods within respective objects provide user with powerful tools to extract the data they want. The process can then be replicated for all rows using an iterator. (Discussed Later)

Basic Python Objects

Python Object	Feature	Mutability
List	Collection of numeric or character variables which are referenced using integer indices	*Mutable*
Strings	Collection of characters including whitespaces which have separate methods of their own and are referenced as (and operated upon) as list	*Immutable*
Tuples	List like objects, defined using round brackets instead of square brackets	*Immutable*
Dictionary	Dictionary is like a list, except that its items are referenced through used defined 'Keys"	*Mutable*

- **Mutable Objects**: Can change original value post reassignment
- **Immutable Objects**: Do not change original value post reassignment
- Conceptually, Lists and Tuples are similar except for the following
  - Lists are mutable, tuples are immutable
  - List is declared using Square Brackets, Tuple through round brackets
  - Many List functions do not have a parallel function for Tuple

NOTE: Refer to Appendix 1 for detailed explanation

Lists (Basic Methods)

Method	Output	Functionality
A = [1, 2, 'Hello' , '3'] ; print(A)	[1, 2, 'Hello' , '3']	Declaring and printing a list
type(A)	list	Function to display class of object
print(A[0])	1	Prints the first element of list
print(A[2])	'Hello'	Prints the third element of the list
print (A[-1])	'3'	Prints the Last element of the list
print(A[-4])	1	Prints the 4th elements from last. Same as A[0]
len(A)	4	Prints length of list. String is considered 1 element
lst = [1,3,2,7,4,6,9,5]; lst.sort() ; print(lst)	[1,2,3,4,5,6,7,9]	Sorts elements of the list if they all are numeric
lst = [1,3,2,7,4,6,9,5]; lst.reverse() ; print(lst)	[5,9,6,4,7,2,3,1]	Reverses all elements of the list
lst = [1,2,3]; lst.append(4); print(lst)	[1,2,3,4]	Used to append a particular value in a list
lst = [1,2,3]; del lst[0]; print(lst)	[2,3]	Deletes the element of the list specified
lst = [1,2,3]; lst.remove(3); print(lst)	[2,3]	Deletes the element passed in the function
print ([1,2,3] + [5,6,7])	[1,2,3,5,6,7]	' + ' is used for concatenation of two lists
print (['Hi'] * 4)	['Hi','Hi','Hi','Hi']	' * ' is used for repeating element in list
3 in [1,2,4,5,3]	True	Bool comparison and looping on list elements
max([1,2,3,4])	4	Returns maximum element in a list

Lists (Indexing and Slicing)

CODE

```
lst = [1,2,'New York','Delhi','Justin']

#From Starting, Index=0
print("First  Element :",lst[0])
print("Second Element :",lst[1])
print("Third  Element :",lst[2])
print("--------------------------------")

#From Ending, Index=-1
print("Last Element   : ",lst[-1])
print("Second Last Element  : ",lst[-2])
print("Third  Last Element  : ",lst[-3])
print("--------------------------------")

#First Three Elements
print("First Three Elements",lst[:3])
print("--------------------------------")

#Other than first three Elements
print("Other than first three elements",lst[3:])

#Last Three Accounts
print("Last Three Accounts",lst[-3:])

#Other than Last Three Accounts
print("Other than Last Three Accounts",lst[:-3])
```

OUTPUT

```
First  Element : 1
Second Element : 2
Third  Element : New York
-------------------------------
Last Element   :  Justin
Second Last Element  :  Delhi
Third  Last Element  :  New York
-------------------------------
First Three Elements [1, 2, 'New York']
-------------------------------
Other than first three elements ['Delhi', 'Justin']
Last Three Accounts ['New York', 'Delhi', 'Justin']
Other than Last Three Accounts [1, 2]
```

Function	Syntax
N^{th} Element from Last	*List [-N]*
First N elements	*List [: N]*
Everything other than first N elements	*List [N :]*
Last N Elements	*List [-N :]*
Everything other than Last N Elements	*List [: -N]*

NOTE: List objects are mutable, hence *List[2]='Hello'* will change the value in the original list as well

Strings Methods

Functionality	Code	Output
Check if entire string is lowercase/uppercase	str.islower() str.isupper()	False False
Converting entire string to upper/lower	str.upper() str.lower()	"HELLO WORLD" "hello world"
Checking if string starts with a string **[1]**	str.startswith("Hello") str.startswith("hello")	True False
Checking if string ends with a certain string **[1]**	str.endswith("World")	True
Splitting based on some delimiter	str.split(" ")	['Hello","World"]
Printing String in Reverse **[3]**	str[::-1]	'dlrow olleH'
Length of String **[2]**	len(str)	11
Individual element list from string	List(str)	['H','e','l','l'…….
Replacing strings within strings	str.replace('H','F')	'Fello World'
Removing Trailing whitespaces	str.strip()	"Hello World"
Removing internal whitespace	str.replace(" ","")	"HelloWorld"
Printing String with Text	print("I want to say %s",str)	I want to say Hello World

str = "Hello World"

[1]: Strings to be checked are all case sensitive
[2]: Length of string includes whitespaces
[3]: String slicing is same as list slicing

Dictionary: Methods

Method	Output	Functionality
A = { 'Key1' : [1,2,3,4], 'Key2' : 2, 'Key3' : 'Hello' } ; print(A)	{'Key1':[1,2,3,4], 'Key2':2,'Key3':'Hello'}	Declaring, initializing and printing a dictionary
type(A)	dict	Function to display class of object
print(A['Key1'])	[1,2,3,4]	Prints the element
print(list (A.keys()))	['Key1' , 'Key2' , 'Key3']	Provides a list of all keys of dictionary
A['Key4'] = 'Test' ; len(A)	4	Declaring a new element via its key
Del A['Key4']; len(A)	3	Deletes the element of the dictionary
A.clear()	{ }	Empty Dictionary. Removes all elements
A.values()	Dict_values([[1,2,3,4],2,' Hello',])	Returns all values assigned with a key. Values returned in declaration order

NOTE: ***Get Function***

Get function allows dynamic creation of keys within a dictionary if they are not present. In the given code, get function initializes Key as the alphabet and the count as its value. Default value is 0 (as in the code)

```
count = {} # Empty Dictionary
lst = ['A','B','C','D','A','B','B','I']
for items in lst:
    count[items]=count.get(items,0)+1
count
```

```
{'A': 2, 'B': 3, 'C': 1, 'D': 1, 'I': 1}
```

Control Flow and Loops

A program's control flow is the order in which program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls

1

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

- ***1. If*** statement is similar to the ones in any other programming language except for the indentation
- Instead of multiple cascading of ***if-else*** statement, we have ***elif*** (short for ***else-if***) for cascading
- Final case is handled by final ***else*** statement

- ***2. For*** loop operates using ***range*** function where one can specifically define the number of iterations
- Alternately, any iterable item such as string, list dictionary etc. can be used to identify values as well as number of iteration on the loop
- Example: List = [1,2,3,4]; for I in List: print(i)

2

```
for target in iterable:
    statement(s)
```

```
for letter in "ciao":
    print "give me a", letter, "..."
```

```
lst  = ['D','E','L','H','I'] #List
Test = ''    #Initializing Empty String
for i in lst:
    Test = Test + i
print(Test)
```

```
DELHI
```

3

CODE

```
count = 1
while count>0.001:
    count = count/2
    print(count)
```

OUTPUT

```
0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.00390625
0.001953125
0.0009765625
```

- 3. **While** loop operates till the condition mentioned is deemed true. This condition can be modified within the loop or till all iterations get exhausted

Exercise 1: Strings, Lists and Dictionaries

1. Read the phrase "***Quick Brown fox jumped over the lazy DOG***"
   - Count the number of alphabets and words in the phrase
   - Count the frequency of all alphabets in the phrase
   - Convert the first alphabet of all words to capital
   - Print the phrase with every alternate word in reverse

2. Read the phrase "***Everybody likes to eat pizzas and ice creams***"
   - Separate all words into a list
   - Create a dictionary with the words and length of each word
   - Combine the phrases mentioned above with comma as a separator

3. Create a program to read a phrase and highlight the word which does not contain any vowel. Run this code on the phrases mentioned above.

4. Create the sum of the infinite series $X = \frac{1}{2} + \frac{1}{4} + \ldots\ldots$ using while loop with convergence criteria for error <0.0001

5. Store maximum possible decimals for **e** and **pi** and find the frequency of digits in the decimal places

NUMPY

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- A powerful N-dimensional array objects
- Sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Introduction to Numpy

CODE

```
print("")
print("Multiplication by 3 :")
print(arr * 3)
print("")
print("Addition of Arr to itself :")
print(arr + arr)
print("")
print("Shape Function :")
print(arr.shape)
print("")
print("Creating an Array of Zeros :")
print(np.zeros(10))
print("")
print("Matrix of Zeros :")
print(np.zeros((3,6)))
print("")
print("Creating Array of Numbers :")
print(np.arange(10))
print("")
print("Creating Array of Random Numbers :")
print(np.random.randn(10))
```

OUTPUT

```
Multiplication by 3 :
[  3.     6.     10.2    17.4    27.69   14.64]

Addition of Arr to itself :
[  2.     4.      6.8    11.6    18.46    9.76]

Shape Function :
(6,)

Creating an Array of Zeros :
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

Matrix of Zeros :
[[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]

Creating Array of Numbers :
[0 1 2 3 4 5 6 7 8 9]

Creating Array of Random Numbers :
[-0.95081388 -0.97396997  1.92126233 -0.88190847 -0.20723312 -2.00430279
 -0.40070267 -0.03053227  0.23405867  0.85471097]
```

np.random.randint() returns random set of integers

ZIP Function in Numpy:
Creates a list of parallel elements for flexible and dynamic data processing

```
a = np.array([1,2,3,4,5])
b = np.array(['ABC','BCD','DEF','EFG','FGH'])
c = list(zip(a,b))
c
```

```
[(1, 'ABC'), (2, 'BCD'), (3, 'DEF'), (4, 'EFG'), (5, 'FGH')]
```

NOTE: ZIP function works well with both Lists and Arrays

Exercise 2:Numpy

Demonstrating the *CLT: Central Limit Theorem*

1. Creating a Sampling Mean Distribution

- Create an array of 10,000 random numbers from chi-square distribution.
  This we would call *Population Sample*
  (***Hint*** : Use *np.random.chisquare(df=5,size=10000) to generate numbers)*
- Create permutation of the population sample created
  (***Syntax*** : *np.random.permutation(array))*
- Create a Test sample using first 1000 values of the permuted population sample
- Calculated the mean of the Test Sample and store it in a list
- Loop last 3 steps 10,000 times

2. Visualizing Sampling Mean Distribution and Population Distribution

- Type: *import matplotlib.pyplot as plt*
- In next line, type *%pylab inline*
- Plotting Population Distribution: plt.hist(bins=100, <Array of chisq distribution>)
- Plotting Sample Mean Distribution: plt.hist(bins=100, <Array of Sampling Means>)

What you will observe is that irrespective of the population distribution, the sampling mean distribution would approximately be equal to a normal distribution. Results work best if the sample contains > 30 observations. This activity can be repeated by taking Log-Normal Distribution

PANDAS

Pandas is built on top of numpy, and is designed to eliminate the need for writing loops for any filtering or aggregation work. It is implemented in C, so is around 15x faster than base python

Key Features:

- Easy handling of ***missing data.*** (dropna, fillna, ffill, isnull, notnull)
- Simple ***mutations*** of tables (add/remove columns)
- Easy ***slicing*** of data (fancy indexing and subsetting)
- Automatic ***data alignment*** (by index)
- Powerful ***split-apply-combine*** (groupby)
- Intuitive ***merge/join*** (concat, join)
- Reshaping and ***Pivoting*** (stack, pivot)
- ***Hierarchical Labeling*** of axes indices
- Robust ***I/O tools*** to work with csv, Excel, flat files, ***databases and HDFS***
- Easy plotting (plot)

DataFrames and Series

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
[illegible]	[illegible]	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	S
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1	0	237736	30.0708	NaN	C

DataFrames

- <u>**DataFrames**</u> and <u>**Series**</u> are two main objects in Pandas.
- DataFrames correspond to Tables and Series correspond to vectors of individual (or combination of) columns
- Unique Rows can be referred through index
- Most methods applicable for DataFrames also apply for Series
- Both these objects are mutable objects

```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
5                                     Moran, Mr. James
6                              McCarthy, Mr. Timothy J
7                       Palsson, Master. Gosta Leonard
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
9                  Nasser, Mrs. Nicholas (Adele Achem)
Name: Name, dtype: object
```

Series

Data Processing and Basic Analysis in Python

Apart from these there are additional steps meant to swiftly execute the desired task. Refer to respective sections in Appendix

1. Data Importing

- Importing from- CSV
- Associated attributes for importing

2. Data Exploration

- DataFrame:, Count, describe etc.
- DataFrame Slicing: Boolean, Conditional Slicing
- Bivariate Analysis: Group-by Function
- Variable Transformation: .apply, Lambda function
- Data Merge, Append, Concatenate

3. Advanced Features

- csv export Saving graphs
- Datetime()
- Binning and Ranking

4. Visualization

- Histogram
- Pie Chart
- Bar Chart
- Scatter Chart

1. Data Importing (CSV): Creating DataFrame

For this training a popular freely available dataset called the Titanic Dataset is going to be used. Refer to **Appendix 3** for data dictionary

1

```
import pandas as pd
import numpy as np
```

```
DF = pd.read_csv('train.csv')
```

1. pd.read_csv(): Used to read .csv files (Should be present in the same directory)

2

```
DF.head()
```

2. DF.head(): Displays top 5 rows of the DataFrame. **DF.tail()** will display last 5 rows

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

3

```
DF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex
Age
SibSp
Parch
Ticket
Fare
Cabin
Embarked
dtypes: flo
memory usage: 83.6+ KB
```

3. DF.info(): Gives details of all variables and their data-types

4

```
DF.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000					
50%	446.000000	0.000000					
75%	668.500000	1.000000					
max	891.000000	1.000000					

4. DF.describe(): Gives univariate analysis of all numeric variables

NOTE: Refer to **Appendix 2** for arguments for **read_csv()**

2. Data Exploration: DataFrames and Series

Basic data exploration in Pandas can be done using DataFrames and Series. This page tells how to extract Series out of a DataFrame and how to access values of each column(s)

```
DF.tail()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.00	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.00	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.45	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.00	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.75	NaN	Q

Series

```
DF['Sex']
```

```
0      male
1    female
2    female
3    female
4      male
5      male
6      male
7      male
8    female
9    female
```

```
DF['Fare']
```

```
0      7.2500
1     71.2833
2      7.9250
3     53.1000
4      8.0500
5      8.4583
6     51.8625
7     21.0750
8     11.1333
9     30.0708
10    16.7000
```

```
type(DF['Fare'])
```

```
pandas.core.series.Series
```

Pandas Series object can be accessed by simply calling out the name of the Variable inside [ ] brackets within quotes. **Works only for a single variable**. Each series object has the values of the variable along with index to highlight its respective position in the DataFrame

DataFrame

```
type(DF[['Age','Fare']])
```

```
pandas.core.frame.DataFrame
```

Pandas DataFrame object can be accessed by calling the name of the DataFrame. To access a slice (like couple of columns) we can simple pass the variables names in double rectangular brackets.

```
DF[['Age','Fare']]
```

	Age	Fare
0	22.0	7.2500
1	38.0	71.2833
2	26.0	7.9250
3	35.0	53.1000
4	35.0	8.0500
5	NaN	8.4583
6	54.0	51.8625

Difference between [ ] and [[ ]] is that the former always produces a Series object and latter produces a DataFrame Object

Note

2. Data Exploration: Data Sanity Checks

```
DF.count()/len(DF)*100
```

```
PassengerId    100.000000
Survived       100.000000
Pclass         100.000000
Name           100.000000
Sex            100.000000
Age             80.134680
SibSp          100.000000
Parch          100.000000
Ticket         100.000000
Fare           100.000000
Cabin           22.895623
Embarked        99.775533
dtype: float64
```

1

1.Count(): gives count of all variables. Does not count missing variables

Fill Rate: Dividing individual variable count by length of DataFrame provides us the fill rate of each variable

2

```
DF.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	max
PassengerId	891.0	446.000000	257.353842	1.00	223.5000	446.0000	668.5	891.0000
Survived	891.0	0.383838	0.486592	0.00	0.0000	0.0000	1.0	1.0000
Pclass	891.0	2.308642	0.836071	1.00	2.0000	3.0000	3.0	3.0000
Age	714.0	29.699118	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
SibSp	891.0	0.523008	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Parch	891.0	0.381594	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Fare	891.0	32.204208	49.693429	0.00	7.9104	14.4542	31.0	512.3292

2.Describe().Transpose(): Converts rows to columns and vice versa

3

```
DF['Pclass'].value_counts()
```

```
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

```
DF['Sex'].value_counts()
```

```
male      577
female    314
Name: Sex, dtype: int64
```

3. .value_counts(): Gives the frequency of categorical variables in the Dataframe

Output returned by this function Is a Series Object with Indices as the individual categories

4

4. Attributes of Describe Function

- ***Include=[np.object]***: Used to describe strings, equivalent to proc freq in sas
- ***Include=[np.number]***: *Used to describe numeric variables. equivalent to proc means*

DF.columns : will provide you will names of all columns
len(DF) : will give length of DataFrame
DF.index : will give indices of all rows

Exercise 3: Introduction to Titanic Dataset

1. Read the Titanic Dataset and write a code to extract values of third last and second last column.
   (*Assume you do not know the name of the variables*)
2. Lst = ['Embarked',' Sex',' Pclass ']
   Print the value counts of these variables from the DataFrame
   (Hint: Take care of the whitespaces in variable name !)

3. Create a code to find which variables of Titanic Dataset are categorical:
   - Categorical variables are generally character variables
   - If a Numeric variable (*dtype = 'int64' or 'float64'*), then it can be categorical if number of categories of the variable are < 5

4. In a manner similar to fill rate, calculate the missing rate of all variables

Data Slicing

Data Slicing refers to cutting the existing data into chunks of more usable datasets. They can be used to divide the data and pick independent rows. Slicing can broadly be categorized into:

1. Index Based Slicing: Slicing of data on the basis of its position or label
2. Conditional Slicing: Slicing Data Frame on the basis of some condition

2. Data Exploration: Data Slicing

```
DF.loc[1]
```

```
PassengerId                                                    2
Survived                                                       1
Pclass                                                         1
Name           Cumings, Mrs. John Bradley (Florence Briggs Th...
Sex                                                       female
Age                                                           38
SibSp                                                          1
Parch                                                          0
Ticket                                                  PC 17599
Fare                                                     71.2833
Cabin                                                        C85
Embarked                                                       C
Name: 1, dtype: object
```

```
DF.loc[1,'Ticket']
```

```
'PC 17599'
```

Basics of Slicing

Basic data manipulation requires slicing. To locate a single value from
(Row X Column)
We will be using two main functions for the same

- .loc[index, 'Col Name']
- .iloc [index]['Col Name']

```
DF.iloc[0][['Age','Sex','Parch']]
```

```
Age         22
Sex       male
Parch        0
Name: 0, dtype: object
```

Conditional Slicing

```
DF[DF['Age']>50].head(3)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S
15	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55.0	0	0	248706	16.0000	NaN	S

```
DF[(DF['Sex']=='male')&(DF['Age']>50)].head(3)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
33	34	0	2	Wheadon, Mr. Edward H	male	66.0	0	0	C.A. 24579	10.5000	NaN	S
54	55	0	1	Ostby, Mr. Engelhart Cornelius	male	65.0	0	1	113509	61.9792	B30	C

Conditional Slicing is slicing based on some pre defined condition. This can be done by creating Boolean flags which pick specific rows

DF[DF['Age']>50]

Can be broken into

1. DF['Age']>50 which returns a True False Series
2. That series then picks the rows corresponding to True values

2. Data Exploration: Data Slicing

- DataFrame & Series slicing can be done using
- **.loc**
  - *Syntax*: DF.loc[<Label Name>,['Column Names']]
  - Used for Label based DataFrame slicing
  - Can operate using conditional slicing
- **.iloc**
  - *Syntax*: DF.iloc[Index Number][['Column Name(s)']]
  - Used for positional based DataFrame slicing
  - Cannot be operated using conditional slicing
- **.ix**
  - *Syntax*: DF.ix[<Label Name/Position Number>,['Column Names()']]
  - Used for Label & positional based DataFrame slicing
  - Can operate using conditional slicing
  - Use is deprecated and .loc or .iloc is preferred over .ix

LINK

Single Argument

```
#Single Argument
DF.loc[0]
```

```
# Single Argument
DF.iloc[0] #Works just as .loc
```

```
PassengerId                          1
Survived                             0
Pclass                               3
Name           Braund, Mr. Owen Harris
Sex                               male
Age                                 22
SibSp                                1
Parch                                0
Ticket                       A/5 21171
Fare                              7.25
Cabin                              NaN
Embarked                             S
Name: 0, dtype: object
```

OUTPUT

CODE

```
New_ID = ['A','B','C','D','E']
Test.index = New_ID
Test
```

```
#Single Argument .loc
Test.loc[0]
#Throws an error
#since there is no Label = 0
```

```
Test.iloc[0]
#Works
```

```
Test.loc[Test['Survived']==1] #This Works.
```

Difference between label based slicing and position based slicing

Conditional Slicing

```
#Conditional Slicing
DF.loc[DF['Age']>70,['Name','Pclass','Fare','Age']]
```

OUTPUT

	Name	Pclass	Fare	Age
96	Goldschmidt, Mr. George B	1	34.6542	71.0
116	Connors, Mr. Patrick	3	7.7500	70.5
493	Artagaveytia, Mr. Ramon	1	49.5042	71.0
630	Barkworth, Mr. Algernon Henry Wilson	1	30.0000	80.0
851	Svensson, Mr. Johan	3	7.7750	74.0

```
DF.iloc[DF['Age']>70][['Name','Age','Fare']] #Does not Work
```

2. Data Exploration: Data Slicing

Query Function: They are used to create SQL like queries on the DataFrame or Series object. SQL like arguments can be passed within quotes to generate the sliced DataFrame

```
#Rows where Age > 70
#Method 1
DF[DF['Age']>70]
DF.query("Age>70")
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
96	97	0	1	Goldschmidt, Mr. George B	male	71.0	0	0	PC 17754	34.6542	A5	C
116	117	0	3	Connors, Mr. Patrick	male	70.5	0	0	370369	7.7500	NaN	Q
493	494	0	1	Artagaveytia, Mr. Ramon	male	71.0	0	0	PC 17609	49.5042	NaN	C
630	631	1	1	Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	27042	30.0000	A23	S
851	852	0	3	Svensson, Mr. Johan	male	74.0	0	0	347060	7.7750	NaN	S

```
(DF
.query("Age>50 & Sex=='female'")
.apply(lambda x:x['Name'].upper(), axis=1)
.head())
```

```
11                              BONNELL, MISS. ELIZABETH
15                     HEWLETT, MRS. (MARY D KINGCOME)
195                                LURETTE, MISS. ELISE
268    GRAHAM, MRS. WILLIAM THOMPSON (EDITH JUNKINS)
275                   ANDREWS, MISS. KORNELIA THEODOSIA
dtype: object
```

Tip

To put functions in multiple lines, one needs to put brackets around the whole query. Multiple and long lines of code can also be written in separate lines using back slashes (Refer to Slide 33)

Basic Data Functions

Basic functions include activities like renaming, sorting, deleting rows and columns from a DataFrame. These methods will help in reshaping and molding the data as per user requirements and are some of the most frequently performed transformational changes on datasets

2. Data Exploration: Basic Functions

```
print("-------------SORTING-------------")
print("")
DF.sort_values(by=['Survived'], ascending=True, inplace=True)
print(DF[['Survived','Name','Embarked']].head(3))
print("")
print(DF[['Survived','Name','Embarked']].tail(3))
print("")

print("-------------RENAMING------------")
print("")
DF.rename(columns={'Age':'age','Ticket':'TID'},inplace=True)
print(DF.columns)
print("")

print("----------DELETING-COLUMN---------")
print("")
DF.drop(['Parch'],axis=1,inplace=True)
print(DF.columns)
print("")
```

```
print("------------DELETING-ROWS-----------")
print("")
print(DF[['Survived','Name','Embarked']].head(3))
DF.drop(0,axis=0,inplace=True)
print("")
print(DF[['Survived','Name','Embarked']].head(3))
print("")

print("---------SELECTING EMPTY VALUES-------")
print("")
Empty_Age     = DF[pd.isnull(DF['age'])==1]
Non_Empty_Age = DF[pd.isnull(DF['age'])==0]
print("EMPTY AGE : ",Empty_Age['age'].head(3))
print("NON-EMPTY AGE : ",Non_Empty_Age['age'].hea
print("")

print("------------RESETTING INDEX-----------")
print("")
DF1 = DF[DF['age']>50] #Conditional Slicing
print("Index of Sliced DataFrame",DF1.index)
DF1.reset_index(inplace=True,drop=True)
print("New Index of Sliced DataFrame",DF1.index)
print("")
```

SORTING: DF.sort_values()
By = [Column Names]
Ascending = True/False
Inplace = True (to reflect changes in DF permanently

RENAMING: DF.rename ()
Columns = {'old name': 'new name'}   A dictionary !!
inplace = True (to reflect changes in DF permanently

DELETING: DF.drop ()
[<Column Name>]
Axis = 1 (1 for Columns, 0 for Rows)

Empty Values (Empty values are NaN for Python)
pd.isnull(DF[<Column name>])==1
Creates Boolean True/False Series
This Series can be put in DF[] for picking resp. rows

Resetting Index: DF.reset_index ()
inplace = True (to reflect changes in DF permanently
Drop=True (If you want to drop the original index)
Used to reset index of a sliced DataFrame
Sliced DataFrame retains its original index

2. Data Exploration: Basic Functions

```
lst = list(np.random.randint(800,size=7))
print(" This is list of 20 Random Integers between 1-800 :",lst)
#Matches Index from DataFrame
DF1 = DF[DF.index.isin(lst)]
DF1
```

1

```
 This is list of 20 Random Integers between 1-800 : [87, 284, 393, 476, 118, 229, 782]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
87	88	0	3	Slocovski, Mr. Selman Francis	male	NaN	0	0	SOTON/OQ 392086	8.0500	NaN	S
118	119	0	1	Baxter, Mr. Quigg Edmond	male	24.0	0	1	PC 17558	247.5208	B58 B60	C
229	230	0	3	Lefebre, Miss. Mathilde	female	NaN	3	1	4133	25.4667	NaN	S
284	285	0	1	Smith, Mr. Richard William	male	NaN	0	0	113056	26.0000	A19	S
393	394	1	1	Newell, Miss. Marjorie	female	23.0	1	0	35273	113.2750	D36	C
476	477	0	2	Renouf, Mr. Peter Henry	male	34.0	1	0	31027	21.0000	NaN	S
782	783	0	1	Long, Mr. Milton Clyde	male	29.0	0	0	113501	30.0000	D6	S

1) .isin(): This returns a Boolean value if the value is within the contained list

TIP: isin() is a powerful tool when a list of values has to be compared for conditional slicing in DataFrames

```
#Dropping Duplicate Values
print(len(DF1.columns))
DF1.drop(['Sex'],axis=1, inplace=True)
print(len(DF1.columns))
```

2

```
12
11
```

2) .drop() :

- ['Var1']: Name of Column to be dropped
- Axis= 1 (1 for Columns, 0 for rows)

```
#Checks for the Values according to combination of Columns
DF1.drop_duplicates(subset=['Sex'])
```

3

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
87	88	0	3	Slocovski, Mr. Selman Francis	male	NaN	0	0	SOTON/OQ 392086	8.0500	NaN
229	230	0	3	Lefebre, Miss. Mathilde	female	NaN	3	1	4133	25.4667	NaN

3) .drop_duplicates() : Subset=['Var1'…] Takes name of variables according to which duplicates have to be checked

Exercise 4: Basic Data Functions

1. Create a new column called "FARE-INR" where the current fare is converted to INR
   (Hint: Multiply the column with exchange rate and the conversion will be done)

2. Delete the columns "Cabin and Ticket Number"

3. Create two DataFrames: Survived & Non-Survived
   - Sort both the datasets by Age and then Fare
   - Find the largest and smallest fare according to the Pclass

4. Creating DataFrame with Empty Values:
   - Find a list of all variable names
   - Check for each variable if there are any missing values
   - If there is a missing value, then create a DataFrame named as <Variable Name>_EMPTY
   - Create a list with elements as [<Variable Name>, <Length of Empty DataFrame>]

   (*Hint: Automatic DataFrame names cannot be created in Pandas, you would need to pass the DataFrame in a dictionary and then access it*)

Apply & Lambda Function

Apply and Lambda function are generally used in combination. They are primarily used to map a function to all values of a column within a DataFrame.
For instance we want to split all string values of a column on the basis of whitespaces. Then we would use a combination of apply and lambda functions.

2. Data Exploration: Apply-Lambda Functions

1) Lambda Functions: They are mini functions which allow single line manipulation instead of writing full blown functions

1 **Problem Statement**

Create two new columns with contain First and Last Names from the Full Name column of the DataFrame imported

```
DF['LastName'] = DF['Name'].apply(lambda x: x.split(",")[0])
DF['FirstName']= DF['Name'].apply(lambda x: x.split(",")[1])
DF.head(3)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	LastName	FirstName
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	Braund	Mr. Owen Harris
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	Cumings	Mrs. John Bradley (Florence Briggs Thayer)
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	Heikkinen	Miss. Laina

2) .apply Function: They are used to map operations to all items of the DataFrame or Series object instead of looping over all items. They mimic vectorized operation of Numpy arrays

2 **Problem Statement**

Create New Column which maps **0** to ***NA*** and **1** to ***Survivor*** on the column: **Survived**

```
DF['New_Col'] = DF['Pclass'].apply(lambda x: 'Survivor' if x ==1 else 'NA')
DF.head(3)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	LastName	FirstName	New_Col
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	Braund	Mr. Owen Harris	NA
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	Cumings	Mrs. John Bradley (Florence Briggs Thayer)	Survivor

Problem Statements for next Slide

1: Modify two columns and sum them using **.apply** & **.def** function, **2**: Create a column with Title Name as a separate column, **3**: Use if-else statement in .apply function to create Age-Buckets

2. Data Exploration: Apply-Lambda Functions

```
DF['Age_Weight'] = (DF['Age'].max()/DF['Age'])*100
DF['Fare_Weight'] = DF['Fare']/(DF['Fare'].max())*100
```

```
def func(x,y):
    return (x+y)
DF['Weight_Sum'] = DF.apply(lambda x: func(x['Age_Weight'],x['Fare_Weight']),
```

1.1) Broadcasting Values: They are done in a manner similar to broadcasting and vectorization in nd arrays

1.2) Using Def Function: Can be used to create user defined function to be mapped on individual values of DataFrames

```
(DF
 .head()
 .apply(lambda x: x['Name']\
        .split(",")[1]\
        .split(".")[0],axis=1))
```

```
0        Mr
1       Mrs
2      Miss
3       Mrs
4        Mr
dtype: object
```

```
(DF
 .head()['Name']
 .apply(lambda x: x\
        .split(",")[1]\
        .split(".")[0]))
```

```
0        Mr
1       Mrs
2      Miss
3       Mrs
4        Mr
Name: Name, dtype: object
```

2) Lambda function takes in the value on which they are operated. If they are operated on Series, then they take value of that series and if they are operated on DataFrame, then x can be subset to access values of individual rows. In order to do this, we need to mention, axis=1

3) IF Condition : Lambda Functions can be used to define multiple if conditions where

- ***Syntax*** : <If Value> IF <expressions> Else <Else Value>

Multiple or, and expressions can be used within IF condition. For multiple conditions, use Def function

```
#Vulnerable Age Buckets
#Vul = Age<10 or Age>70
DF['Bucket']=\
(DF
 .apply(lambda x: 'Vul' \
        if\
        ((x['Age']<10) or (x['Age']>70))\
        else 'Non-Vul',axis=1))
DF.tail()['Bucket']
```

```
886    Non-Vul
887    Non-Vul
888    Non-Vul
889    Non-Vul
890    Non-Vul
Name: Bucket, dtype: object
```

2. Data Exploration: Apply-Lambda Functions

1. Combining multiple functions in sequential manner and
2. Creating age buckets using User Defined Function (Def Function)

1

```
#Applying Lambda Function on Mutiple Columns
(DF
.query("Age > 70")
.apply(lambda x: print(x['Name'],"paid $"\
                      ,x['Fare'],"for"\
                      ,x['Pclass'],"Class"),axis=1))
```

```
Goldschmidt, Mr. George B paid $ 34.6542 for 1 Class
Connors, Mr. Patrick paid $ 7.75 for 3 Class
Artagaveytia, Mr. Ramon paid $ 49.5042 for 1 Class
Barkworth, Mr. Algernon Henry Wilson paid $ 30.0 for 1 Class
Svensson, Mr. Johan paid $ 7.775 for 3 Class
```

2

```
#How to write concatenated If Statements
def Age_Bucket(x):
    if pd.isnull(x)==1:
        return('Null_Bucket')
    elif (x>0) and (x<10):
        return('Under-10')
    elif (x>=10) and (x<20):
        return('10-20')
    elif (x>=20) and (x<30):
        return('20-30')
    elif (x>=30) and (x<40):
        return('30-40')
    else:
        return('>40')
DF['Age_Bucket']=DF['Age'].apply(lambda x: Age_Bucket(x))
DF.head(3).drop(['PassengerId','Ticket','Cabin'],axis=1)
```

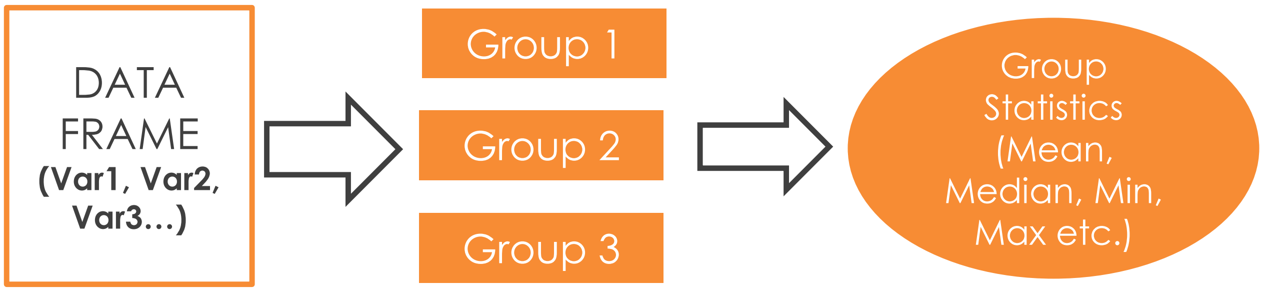
TIP: Combining multiple columns, lambda function and *Def* function together. the user can perform any conditional operation on the DataFrame

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Fare	Embarked	Bucket	Age_Bucket
0	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	7.2500	S	Non-Vul	20-30
1	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	71.2833	C	Non-Vul	30-40
2	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	7.9250	S	Non-Vul	20-30

Groupby and Agg Function

While analyzing data, we generally need to produce summary statistics which need to be grouped by certain attributes and requires summaries by some other attributes. Groups can include grouping by certain categorical variables and summary statistics can include minimum values, median value, average value, sum value or any other user defined summary statistic. Groupby and Agg function together help us in achieving this

2. Data Exploration: Group-By Objects



Pandas a **Groupby** function splits a Dataframe into multiple groups, applies the statistic on each group and summarizes value by group

```
DF.groupby(['Sex']).count()['Survived']
```

```
Sex
female     314
male       577
Name: Survived, dtype: int64
```

```
func = {'Survived':['count','mean'],
        'Fare':['mean']}
```

```
DF.groupby(['Sex','Pclass']).agg(func)
```

		Survived		Fare
		count	mean	mean
Sex	Pclass			
female	1	94	0.968085	106.125798
	2	76	0.921053	21.970121
	3	144	0.500000	16.118810
male	1	122	0.368852	67.226127
	2	108	0.157407	19.741782
	3	347	0.135447	12.661633

Groupby(): DF.groupby([‘Var1’,’Var2’,…]).function()[‘Stat_Variable]

- **DF** : Name of DataFrame
- **([‘Var1’,’Var2’])** : Combination of variables for grouping
- **function** : count, min, max, mean, sum etc.
- **[‘Stat_variable’]** : Variable on which function is to be applied

Agg() : Function helps in creating summary statistics across multiple variables.

- **Syntax:** Requires defining a dictionary with Key as the variables on which statistics are required. Functions (for required statistics) are to passed in form of list

Refer to Appendix 2 for detailed list of functions

Exercise 5: Working with Data using Pandas

1. Use .apply function to create a new column where the title names (Mr. Mrs. Ms. Dr. ....) of the all the passengers is mentioned in the dataset
   (*Hint: Refer to Slide 32*)

2. Create a value counts on the column created in Step 1

3. Create an aggregated function and apply it on a groupby object:
   - Agg function should calculate min and max Age, Average fare & Survival Rate
   - Grouping should be done on "*Pclass*" and "*Sex*"

4. Create Age Buckets for Ages where
   - if Age is missing: Bucket is "Age Missing"
   - Other Buckets should be "<10" , "10-30", "30-50" & ">50"

5. Calculate the survival rate for each of the Age Buckets calculated
   (Hint: Use Group-by Object for this step)

6. Create an indicator for all those passengers whose ticket number contains an Alphabet

Merge, Append, Concat

Transforming and working with multiple datasets requires joining merging and working with multiple datasets. Merge, Append and Concat commands help the users in joining two datasets in a variety of ways

2. Data Exploration: Merge, Append, Concat

df1

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

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

Result = df1.append (df2)
Result = Result.append(df3)

APPEND

Frames = [df1, df2, df3]
Result = pd.concat(frames)

CONCAT

Frames = [df1, df2, df3]
Result = pd.concat(frames, axis=1)

Result

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

Frames = [df1, df2, df3]
Result = pd.concat(frames, axis=1,join='inner')

Result

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

MERGE

DF1.merge(DF2, how='left/right/outer/inner', on='Variable Name)

NOTE: When axis =1, then Data-frames are concatenated along columns else along the row

3. Advanced Features: Binning

Binning is used to analyze continuous numeric variables by dividing them into different categories using a histogram like division. These can be done using **cut** and **qcut**

BINNING

Equal Width Bins

.cut( )
Divides the continuous variable into groups of equal width . User can also define bin edges for cutting the continuous variable

Equal Height Bins

.qcut( )
Divides the continuous variable into groups of with equal quartiles. User can define quartiles required or can get edges for the number of quartiles asked

Pd.cut can be used in two ways

1. It can be used to **N equal sized** bins and categorize the data accordingly
2. It can be used to define **bin edges** and then classify the data according to those edges

```
bins = 10
lst  = DF['Age'].tolist()
Test = pd.Series(pd.cut(lst, bins))
Test.value_counts(sort=False)
```

```
(0.34, 8.378]        54
(8.378, 16.336]      46
(16.336, 24.294]    177
(24.294, 32.252]    169
(32.252, 40.21]     118
(40.21, 48.168]      70
(48.168, 56.126]     45
(56.126, 64.084]     24
(64.084, 72.042]      9
(72.042, 80.0]        2
dtype: int64
```

Providing # of Bins

```
bins = [0,10,20,30,40,50,60,70,80,90]
lst  = DF['Age'].tolist()
Test = pd.Series(pd.cut(lst, bins))
Test.value_counts(sort=False)
```

```
(0, 10]      64
(10, 20]    115
(20, 30]    230
(30, 40]    155
(40, 50]     86
(50, 60]     42
(60, 70]     17
(70, 80]      5
(80, 90]      0
dtype: int64
```

Providing Bin Edges

3. Advanced Features: Binning

Pd.qcut too can be used in two ways
1. It can be used to define the **N equal sized quintiles** bins and categorize the data accordingly
2. It can be used to define specific quintiles required.

```
bins = 10
lst  = DF['Age'].tolist()
Test = pd.Series(pd.qcut(lst, bins))
Test.value_counts(sort=False)
```

```
(0.419, 14.0]     77
(14.0, 19.0]      87
(19.0, 22.0]      67
(22.0, 25.0]      70
(25.0, 28.0]      61
(28.0, 31.8]      66
(31.8, 36.0]      91
(36.0, 41.0]      53
(41.0, 50.0]      78
(50.0, 80.0]      64
dtype: int64
```

Providing Percentiles

```
bins = [0,0.25,0.50,0.75,1]
lst  = DF['Age'].tolist()
Test = pd.Series(pd.qcut(lst, bins))
Test.value_counts(sort=False)
```

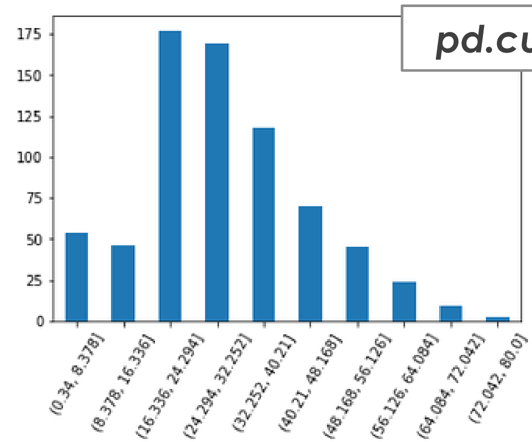
```
(0.419, 20.125]    179
(20.125, 28.0]     183
(28.0, 38.0]       175
(38.0, 80.0]       177
dtype: int64
```

Providing Percentile edges

Creating a bar plot of the data which we have generated, we see for the ***.cut***, we have equal width bins with different frequencies and for ***.qcut***, we have unequal width bins with ***almost*** the same frequencies

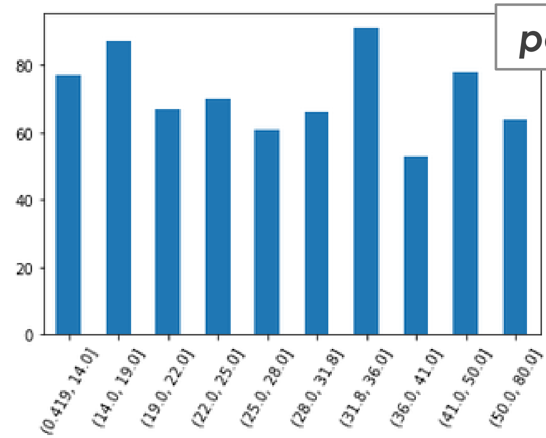
```
bins = 10
lst  = DF['Age'].tolist()
Test = pd.Series(pd.cut(lst, bins))
Test.value_counts(sort=False).plot(kind='bar',rot=60)
plt.show()
```

pd.cut()



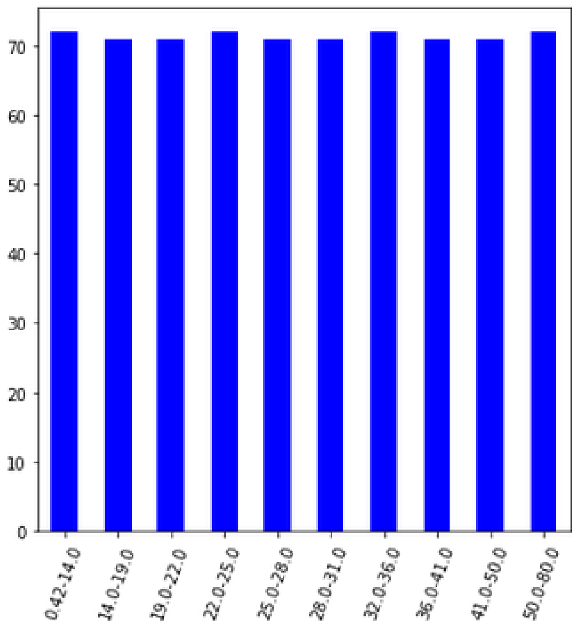

```
bins = 10
lst  = DF['Age'].tolist()
Test = pd.Series(pd.qcut(lst, bins))
Test.value_counts(sort=False).plot(kind='bar',rot=60)
plt.show()
```

pd.qcut()



3. Advanced Features: Binning

```python
bins = 10
DF['Rank']= pd.qcut(DF['Age'].rank(method='first'),
                    bins,labels=[i for i in range(bins)])
A = DF.groupby(['Rank']).agg(['mean','min',
                              'max','count'])['Age'].reset_index()
A.index = A['min'].map(str)+'-'+A['max'].map(str)
fig = plt.figure(figsize=(6,6))
ax  = fig.add_subplot(1,1,1)
ax  = A['count'].plot.bar(rot=70, color="b")
plt.show() #Sort = False implies that Bin Ranges
```



SAS-Proc Rank !!

Qcut creates groups based on equal frequencies, which can be quratiles, quantiles, percentiles etc. However, it would still produce groups of unequal frequencies when the list of continuous variable contains numerous repeating values.

In such a case, we would need to rank the values and then create the bins using qcut. Rank method creates ranking of all variables with various methods defined to break ties:

1. **average**: average rank of tied groups
2. **min**: lowest Rank of tied group
3. **max**: Highest rank of tied group
4. **first**: ranks assigned in the order they appear in array

http://datasciencefree.com/pandas.pdf

3. Advanced Features: Data Exporting

Data Exporting in Pandas is extremely simple. Two most important commands for doing the same are:

1. pd.to_csv()
2. pd.ExcelWriter()

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

To delimit by a tab you can use the `sep` argument of `to_csv`:

```
df.to_csv(file_name, sep='\t')
```

To use a specific encoding (e.g. 'utf-8') use the `encoding` argur

```
df.to_csv(file_name, sep='\t', encoding='utf-8')
```

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Refer to Pandas documentation and various Stack-Overflow links for gathering more information on advanced I/O Techniques

3. Advanced Features: Datetime and Time Series

Python includes robust features to handle Datetime and time series data. The important library is Datetime, which is included in the Pandas package

1. **Declaring Datetime Object**
Format should be integer values in format YY:MM:DD:hh:mm:ss

```
A = pd.datetime(2017,10,20,10,45,36)
```

```
print(A.year)
print(A.month)
print(A.day)
print(A.hour)
print(A.minute)
print(A.second)
```

1

```
2017
10
20
10
45
36
```

2. **Declaring Time Series**
Pd.date_range() : Creates an array of dates from the initial date. User defines ***Period*** (Number of dates) and ***Freq:*** (gap between the created dates)

```
#Format: freq= 'D' : Days, 'S': Seconds, 'H': Hours
d_range = pd.date_range('1/1/2016',periods=3,freq='D')
d_range
```

2

```
DatetimeIndex(['2016-01-01', '2016-01-02', '2016-01-03'], dtype='datetime64[ns]', freq='D')
```

```
d_range = pd.date_range('1/1/2016',periods=2,freq='H')
d_range
```

```
DatetimeIndex(['2016-01-01 00:00:00', '2016-01-01 01:00:00'], dtype='datetime64[ns]', freq='H')
```

```
d_range = pd.date_range('1/1/2016',periods=2,freq='S')
d_range
```

```
DatetimeIndex(['2016-01-01 00:00:00', '2016-01-01 00:00:01'], dtype='datetime64[ns]', freq='S')
```

3. **Timedelta Object**
Pandas has an inbuilt data type to handle difference of two dates called the Timedelta Object

```
A = pd.date_range('1/1/2017',periods=5,freq='D')
B = pd.datetime(2017,2,10,10,45,36) - A
pd.Series(B)
```

```
0    40 days 10:45:36
1    39 days 10:45:36
2    38 days 10:45:36
3    37 days 10:45:36
4    36 days 10:45:36
dtype: timedelta64[ns]
```

3

```
D = pd.Timedelta('35 days 10:45:30')
pd.Series(A+D)
```

```
0    2017-02-05 10:45:30
1    2017-02-06 10:45:30
2    2017-02-07 10:45:30
3    2017-02-08 10:45:30
4    2017-02-09 10:45:30
dtype: datetime64[ns]
```

```
A = pd.datetime(2017,1,1,10,45)
B = pd.datetime(2017,1,1,10,50)
(B-A).total_seconds()
```

```
300.0
```

4

4. total_seconds() : This command uses time delta object and converts it into number of seconds. Dividing by 60, 3600 and so on would create total hours, days and so on for the time delta object worked on

3. Advanced Features: Datetime Index

For Time-Series data handling and manipulation in Pandas, we can leverage Datetime Object and use it as index. This allows powerful and simple data sampling and slicing. Pandas was originally built to handle Time Series financial data and hence has a number of features to handle such data

```
TS = pd.Series([i for i in np.arange(500)],
               index=pd.date_range('1-1-2017',
                                   periods=500,freq='D'))
```

1

1. Creating a Series object with Datetime object as its index creates a dynamic Time Series object which can be indexed using specific year, month argument or providing a range of dates

```
#Slicing using single value input
TS['2017-1']
```

```
2017-01-01     0
2017-01-02     1
2017-01-03     2
2017-01-04     3
2017-01-05     4
2017-01-06     5
2017-01-07     6
```

```
#Slicing Using Range of Dates
TS[pd.datetime(2017,2,22):pd.datetime(2017,3,2)]
```

```
2017-02-22    52
2017-02-23    53
2017-02-24    54
2017-02-25    55
2017-02-26    56
2017-02-27    57
2017-02-28    58
2017-03-01    59
2017-03-02    60
Freq: D, dtype: int64
```

2. .resample(): Groupby for Datetime Object

Using Datetime object as index allows resampling function to Groupby the dates by 'A': Yearly basis, 'M' Monthly basis, 'W': Weekly basis and so on

```
#Powerful feature to perform Groupby
TS.resample('Q').sum()
```

2

```
2017-03-31     4005
2017-06-30    12285
2017-09-30    20838
2017-12-31    29302
2018-03-31    36855
2018-06-30    21465
Freq: Q-DEC, dtype: int64
```

```
TS.resample('M').sum()
```

```
2017-01-31      465
2017-02-28     1246
2017-03-31     2294
2017-04-30     3135
2017-05-31     4185
2017-06-30     4965
2017-07-31     6076
2017-08-31     7037
2017-09-30     7725
2017-10-31     8928
2017-11-30     9555
2017-12-31    10819
2018-01-31    11780
2018-02-28    11466
```

This enables **Group-By** without creating another column for month/quarter etc. and then doing **Group-By** using that column

Visualization

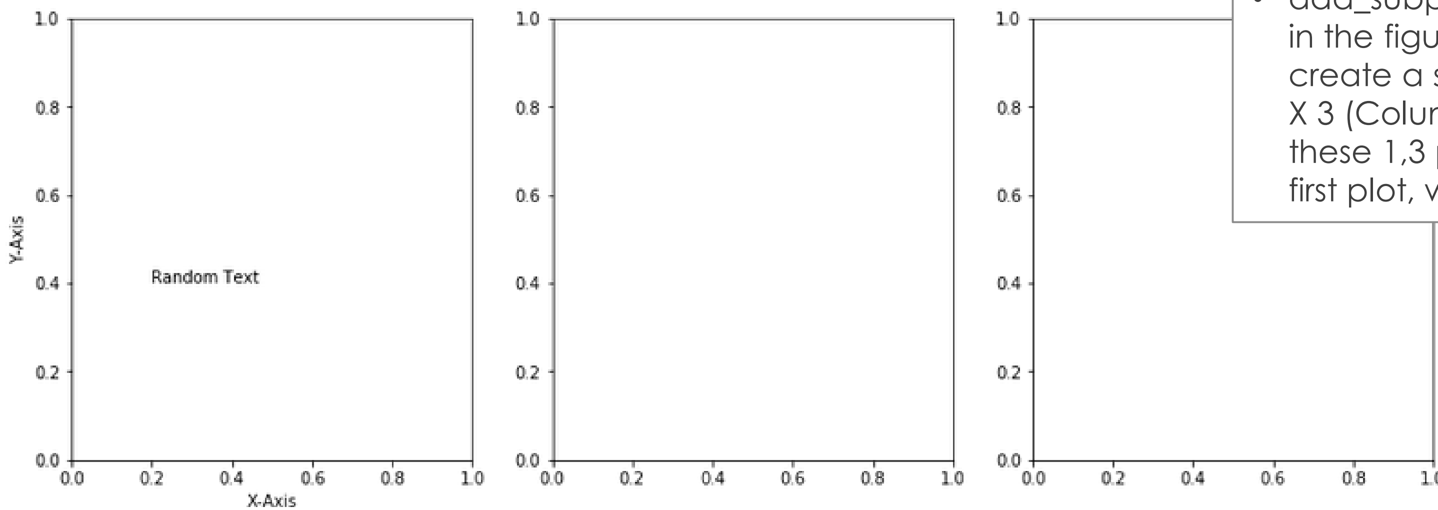
A key outcome of Data Analysis is conveying message portrayed by data through visuals. These visuals include graphs, charts, scatter plots etc. Matplotlib is the python library which allows basic data visualization. For more interactive and graphic visuals, Python libraries like Seaborn, Mayavi and Bokeh can be used

4. Data Visualization: Introduction to Matplotlib

There are many libraries in Python for plotting. Matplotlib is the most basic of those libraries used to create basic line, bar, histogram, box plots etc.

```
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(1,3,1)
ax1.set_xlabel('X-Axis')
ax1.set_ylabel('Y-Axis')
ax1.text(0.2,0.4,'Random Text')
ax2 = fig.add_subplot(1,3,2)
ax3 = fig.add_subplot(1,3,3)
plt.show()
```

- Generally for a good plot, we would like to create sub-plots with varying sizes. Add_subplot() function helps us in doing that.
- add_subplots(1,3,1) implies that in the figure object we need to create a set of 3 plots :1 (Row) X 3 (Column) wise plots and of these 1,3 plots to address the first plot, we mention 1,3,1



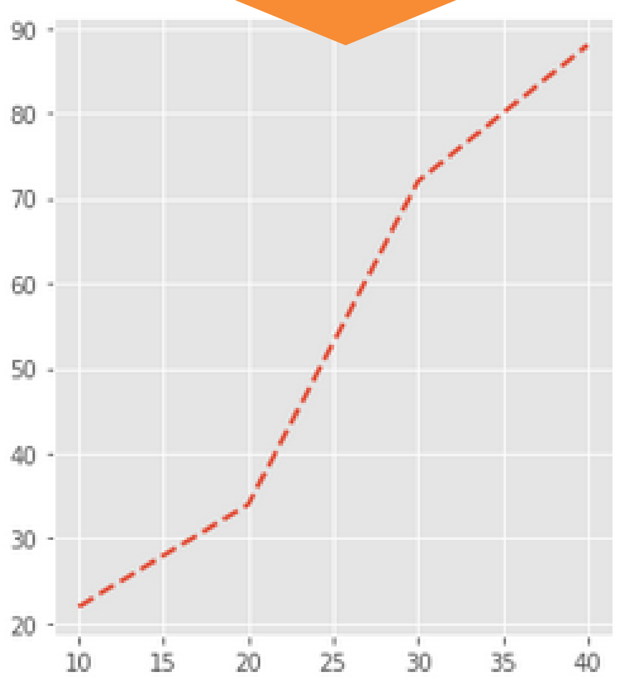
Some additional functions
- set_xlabel/set_ylabel: Use to provide labels for X and Y axis
- plt.style.use(<name of style>: In order to change the style of plot
- Plt.style.context((‘dark_background’ : Used to create a black background to the plot

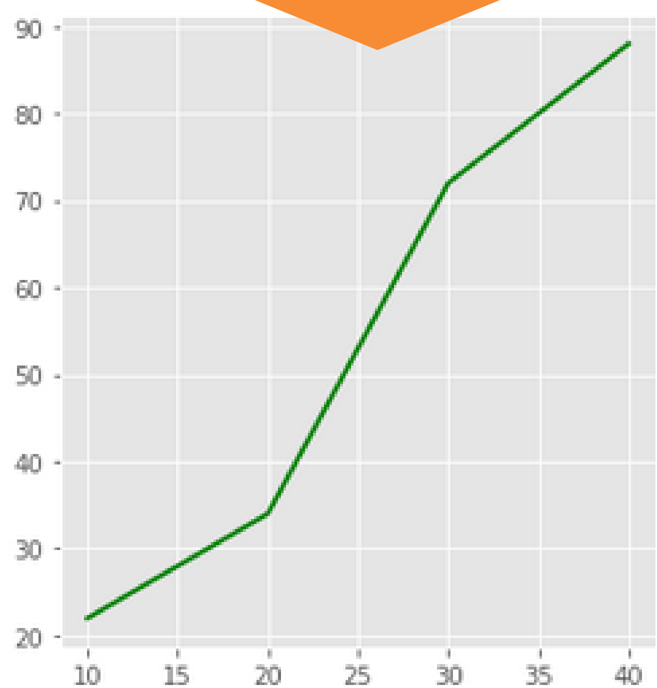
4. Data Visualization: Line Charts

Line plot is one the most basic plots and requires a simply a list of X and Y points. It has a number of attributes which can be used to fine tune the graph. Some of those attributes are demonstrated below

```
X = [10,20,30,40]
Y = [22,34,72,88]
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(1,3,2)
line, = ax1.plot(X,Y,linewidth=2)
line.set_linestyle('--')
plt.show()
```

```
X = [10,20,30,40]
Y = [22,34,72,88]
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(1,3,2)
line, = ax1.plot(X,Y)
plt.setp(line, color='g', linewidth=2.0)
plt.show()
```

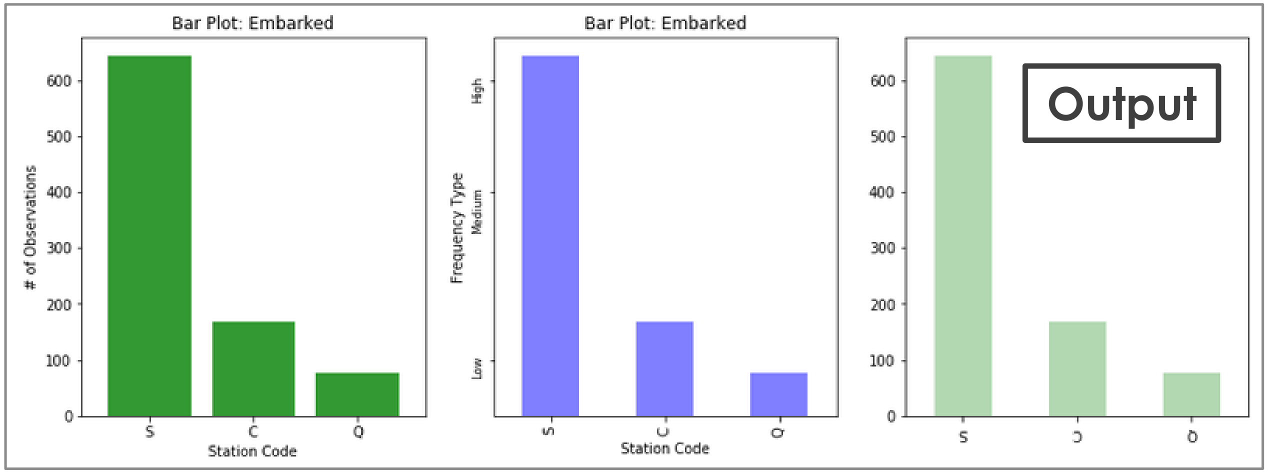




Line Object: This object creates a python line object which can then be operated upon to define various attributes

- **.set_linestyle**('—'): Creates unique line pattern
- **plt.setp**: Setup function works on Matplotlib's plt object which then can be used to create attributes for is line object. Attributes including linewidth, style color etc.

4. Data Visualization: Bar Charts



- Plots are made in the figure object of class Matplotlib
- Each figure can contain multiple plots which are defined using add_subplots
- .plot function contains the following plots
  - Scatter
  - Bar
  - Histogram
  - KDE

Code

```
A = DF1['Embarked'].value_counts()
fig = plt.figure(figsize=(15,5))

#AXIS = 1
ax1  = fig.add_subplot(1,3,1)
A.plot.bar(ax=ax1,color='g',alpha=0.8,rot=0, width=0.8)
ax1.set_xlabel('Station Code')
ax1.set_ylabel('# of Observations')
ax1.set_title('Bar Plot: Embarked')

#AXIS = 2
ax2  = fig.add_subplot(1,3,2)
A.plot.bar(ax=ax2,color='b',alpha=0.5,rot=90)
ax2.set_xlabel('Station Code')
ax2.set_ylabel('Frequency Type')
ax2.set_title('Bar Plot: Embarked')
ax2.set_yticks([100,400,600])
ax2.set_yticklabels(['Low','Medium','High'],rotation=90,fontsize='small')

#AXIS = 3
ax3  = fig.add_subplot(1,3,3)
A.plot.bar(ax=ax3,color='g',alpha=0.3,rot=180)

plt.savefig('Fig.png')# To Save Image in Local Directory
```

- Useful Arguments of .plot function
- **Ax**: Defines which axis should the figure be plotted on
- **Color**: Defines color of the plot
- **Alpha**: Defines opacity of color
- **Rot**= Rotation of Labels
- **Width**: defines width of the bar

4. Data Visualization: Histograms, KDE

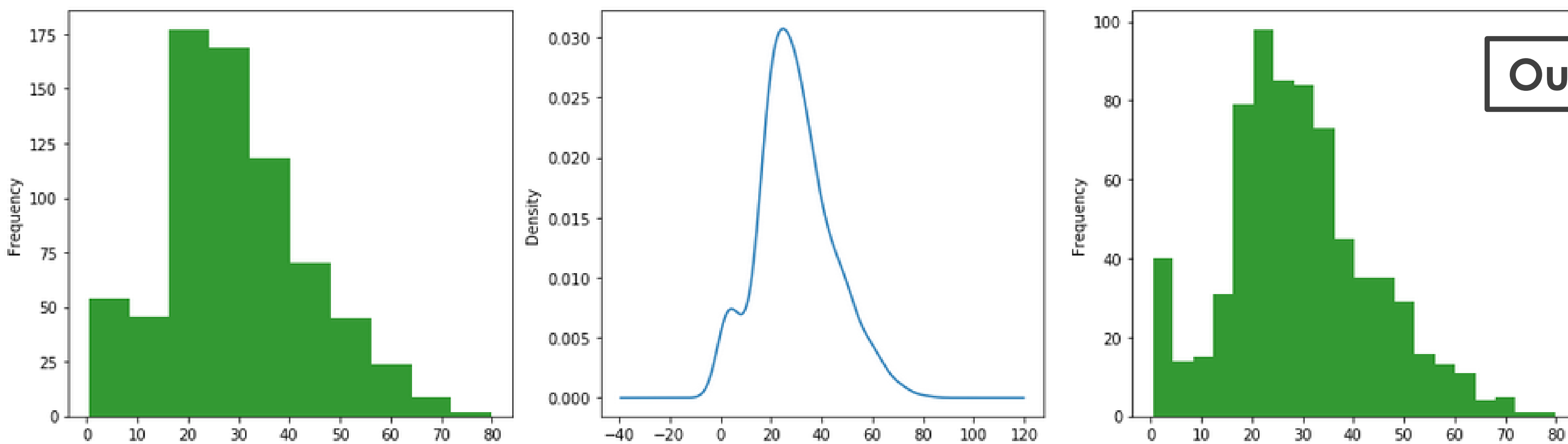
Code

```
#Histogram and KDE
fig = plt.figure(figsize=(18,5))
ax1 = fig.add_subplot(1,3,1)
DF1['Age'].plot.hist(bins=10,color='g',alpha=0.8)
ax2 = fig.add_subplot(1,3,2)
DF1['Age'].plot(kind='kde')
ax3 = fig.add_subplot(1,3,3)
DF1['Age'].plot.hist(bins=20,color='g',alpha=0.8)
print(" Bins Sizes Are:")
np.histogram(DF['Age'].tolist(),bins=10)
```

NOTE: KDE stands for **Kernel Density Estimation**: A non-parametric way to estimate pdf (probability density function) of a random variable !!

```
<matplotlib.axes._subplots.AxesSubplot at 0x11ca52e8>
```

Output



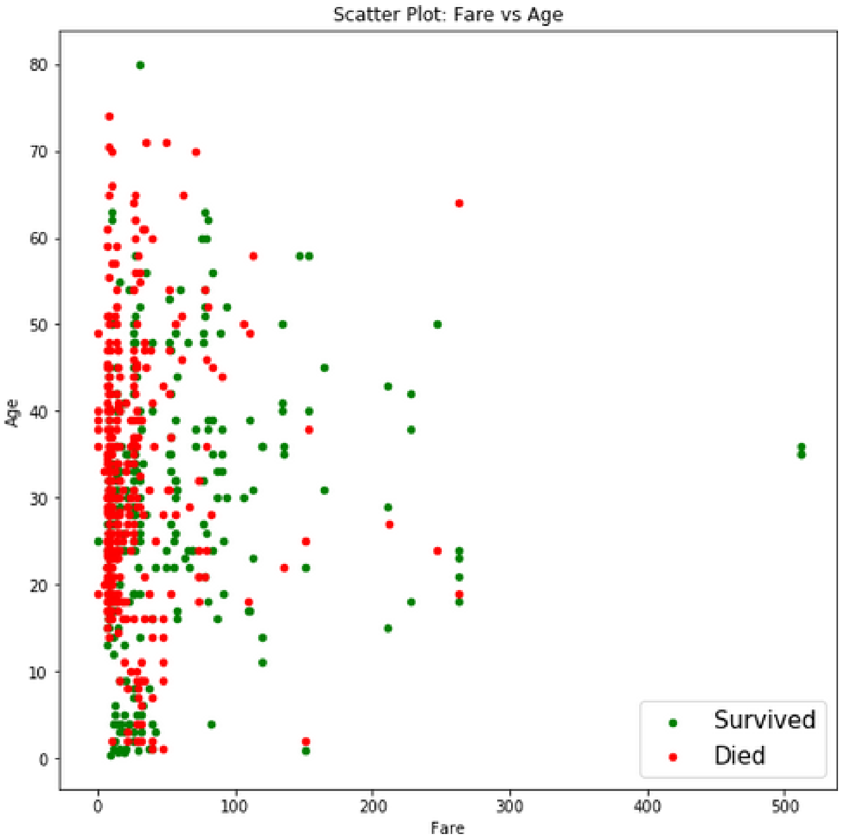
- Useful Attributes of .plot function
- **Bins**: Defines number of equal sized group to create for bucketing a continuous variable

NOTE: Categorical variables are plotted using bar plots and continuous variables are plotted using histograms !!

https://matplotlib.org/1.2.1/examples/pylab_examples/histogram_demo.html

4. Data Visualization: Scatter Plots

```
#Scatter Plot
fig = plt.figure(figsize=(9,9))
ax1 = fig.add_subplot(1,1,1)
ax1.scatter(DF1[DF1['Survived']==1]['Fare'],DF1[DF1['Survived']==1]['Age'],color='g',s=20, label='Survived')
ax1.scatter(DF1[DF1['Survived']==0]['Fare'],DF1[DF1['Survived']==0]['Age'],color='r',s=20, label='Died')
ax1.set_xlabel('Fare')
ax1.set_ylabel('Age')
ax1.set_title('Scatter Plot: Fare vs Age')
ax1.legend(loc=4, prop={'size': 15}) # Loc 1:(Top,Right) 2:(Top,Left) 3:(Bottom,Left) 4:(Bottom,Right)
```



- Useful Attributes of .plot function
- **s**: Defines size of the round shaped scatter dot
- **Label:** Defines the label for the scatter point

Conditional Scatter Plotting

- In order to create conditional scatter plots, conditional slicing of DataFrame can be done while passing the X and Y values
- In order to superimpose plotting of multiple items, use the same subplot and the new images are plotted over the previous ones

Useful Subplot Functions
- Ax.set_xlabel: Naming X Axis
- Ax.set_ylabel: Naming Y Axis
- Ax.set_title: Title of the plot
- Ax.legend: Defines location and size of legend

Exercise 6: Visualization

1. Import Titanic Dataset and run value_counts on EMBARKED, create a bar chart using that variable

2. Using a loop, create bar and horizontal charts for variables: [‘Sex’,’Pclass’,’Embarked’]
   Hint: .plot(kind = ‘barh

3. Create a histogram for Fare and Age, vary the bin size from 10, 20 to 30

4. Normalize the Age and Fare variables using the maximum values present for each variable. Then plot the histograms of these two normalized variables

APPENDIX

APPENDIX 1:MUTABLE vs. IMMUTABLE Types

HOME

```
lst  = [1,2,3]
lst1 = lst
lst.append('Hello')
print(lst)
print(lst1)
```

```
[1, 2, 3, 'Hello']
[1, 2, 3, 'Hello']
```

```
lst  = 'New'
lst1 = lst
lst  = lst + 'York'
print(lst)
print(lst1)
```

```
NewYork
New
```

Mutable objects are the ones which change the original value for all assignments. In this example, **lst** and **lst1** both point to single value ([1,2,3]) and hence a change in this value reflects everywhere. Whereas in second example we have two strings where the original value is retained on re-assignment

Mutability and Immutability is related to but separate from reassignment. Reassignment only creates an ID pointing to the value. Immutable objects point towards two different instances of the value. Mutable object point towards same instance of that value

The string objects themselves are immutable.

The variable, `a`, which points to the string, is mutable.

Consider:

```
a = "Foo"
# a now points to "Foo"
b = a
# b points to the same "Foo" that a points to
a = a + a
# a points to the new string "FooFoo", but b still points to the old "Foo"

print a
print b
# Outputs:

# FooFoo
# Foo

# Observe that b hasn't changed, even though a has.
```

APPENDIX 2: read_csv()

HOME

Additional arguments for pd.read_csv() function

Reading DataFrame From CSV

- `pd.read_csv()` , its argument are:
  - If column names are missing
    - `header = None` : Creates Integer Column Names
    - `names = ['Col1','Col2' ...]`: Provides Names to Columns
  - To manually provide datatype
    - `dtype = {'Column Name':integer/float/object]`
    - `parse_dates = [Col1,Col2...]` (Converts Columns to Datetime Stamp)
  - To select column which serves as Index
    - `index_col=['Column Name']`
  - If the file is too large, read chunks iteratively (`nrows=` and `chunksize=`)
  - Skipping over rows/footer (`skiprows=`)

APPENDIX 3: Titanic Dataset-Data Dictionary

HOME

Titanic Dataset is being used in this presentation. This dataset is one of the most frequently used dataset for getting started with Data Analytics. Google “Titanic Dataset” and you would obtain a sample data with about 900 rows and 10 columns as mentioned below. This data is good enough to work with for this module

Variable	Description
PassengerId	Unique Integer Values to identify each passenger
Survived	0/1 Indicator against each record, 1: Survived, 0: Not Survived
Pclass	Travel class of the passenger aboard the Titanic
Name	Name of the Passenger
Sex	Sex of the Passenger
Age	Age of the Passenger, few columns are missing
SibSp	Siblings of the passengers travelling
Parch	Parents and Children of the passengers
Ticket	Ticket number of the passenger
Fare	Fare paid by the passenger to obtain the ticket
Cabin	Cabin number of the passengers
Embarked	Destination from which each passenger emabarked aboard the Titanic

APPENDIX 4: Python-SAS Equivalent Codes

Tables	python	SAS
frequency	• `x.value_counts(dropna = False).sort_index()` • `pd.crosstab(df.A, df.B).apply(lambda r: r/r.sum(), axis = 1)`	`proc freq;`
drop/deep	• `df.drop(['d1', 'd2', 'd3'], axis = 1)` • `df.loc[:, ['k1', 'k2']]`	• `data a(drop=d1 d2 d2);` • `data a(keep=k1 k2 k3);`
rename	• `df.columns = ['a1', 'a2', 'a3']` • `df.rename(columns={'orig1':'new1', 'orig1':'new2'})`	`data a(rename=(orig1=new1 orig2=new2));`
summarize	• `df.x.describe()`	`proc summary;`
bin	• `pd.cut(x, [min, cut1, …, cutk, max])` • `np.digitize(x, [cut1, cut2, …, cutk])`	`proc rank;`
row select	• `df.loc[(cond1) & (cond2), :]` • `df.iloc[:, [1, 3, 5]]`	`where cond1 and cond2;`
merge	• `pd.merge(df1, df2, on = , how= )` • `pd.concat([df1, df2, df3], axis = 0, ignore_index = True)` • `df1.join(df1, how = )`	`merge df1 df2; by col1;`
sort	• `df.sort(['sort_by1', 'sort_by2'], ascending = [True, False])`	`proc sort; by sortby1 descending sort_by2;`

count missing	• `df.count()` • `df.isnull().sum()`	`proc means n nmiss min mean median max;`

format	• 1 <-> 1 mapping:
	`df.column1.replace(zip(old_value, new_value))`
	• `hgc2012.sic_code.replace(dict(zip(sic_indust.sic, sic_indust.indust)))`
	• interval bin/mapping, like from PD to risk ranking:
	`[ranking[j] for j in np.digitize(x, intervals)]`
	• `pd.cut(vectorx, [-np.inf, .25, .5, 1, 1.5, 2, 3, 3.6, 4.2, np.inf], labels = range(9))`
		`proc format;`

columns start with	• `[x for x in list(df) if x.startswith('score')]`	`var scores: ;`
index	• `df.reset_index(inplace = True)` • `df.set_index('Column_name', inplace = True)`	
apply / map / applymap	• `pd.Series.map(func, dict, Series)` • `pd.Series.apply()` • `pd.DataFrame.apply(func, axis = 0)`	`data transformation, fotmat, aggregate`

- Some useful parallels between SAS and Python for moving between both environments

Reference: http://songhuiming.github.io/pages/2016/07/12/python-vs-sas/

APPENDIX 5: Python Operators

Basic Mathematical Operators in Python

Operator	Syntax	a=10, b=20	Output
Remainder	%	b%a	0
Power	**	a**b	100000….
Equal	==	a==b	False
Not Equal	!=	a!=b	True
Greater Than	>	a>b	False
Lesser Than	<	a<b	True
Greater than Equal to	>=	a>=b	false
Less than Equal to	<=	a<=b	True

Special Referential based operation in Python

Operator	a=10, b=20	Output
=	c=a+b	c=30
+=	c+=a	c=c+a
-=	c-=a	c=c-a
*=	c*=a	c=c*a
in	‘one’ in [‘one’,’two’]	True
not in	‘one’ not in [‘one’,’two’]	false

Cheat-Sheet for Visualization

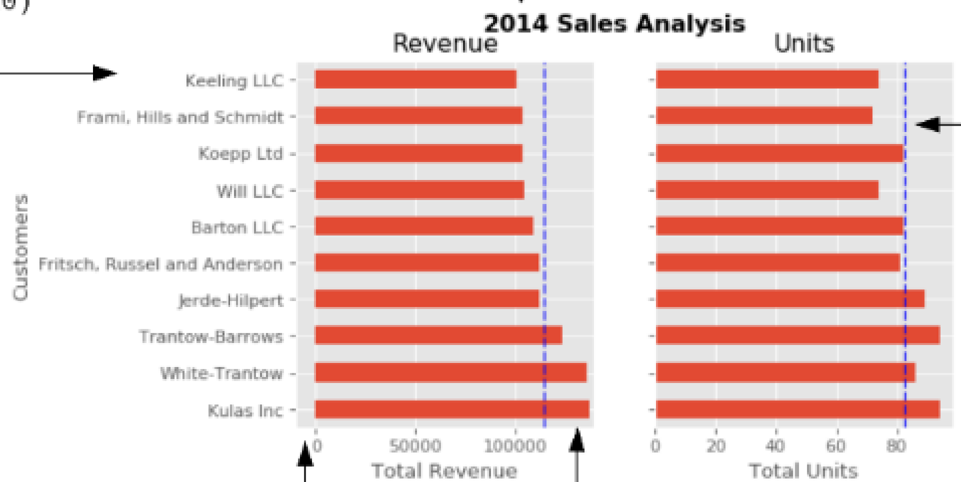
matplotlib customization example

```
fig, (ax0, ax1) = plt.subplots(nrows=1,ncols=2, sharey=True, figsize=(7, 4))
```

```
fig.suptitle('2014 Sales Analysis', fontsize=14, fontweight='bold')
```

```
top_10.plot(kind='barh', y="Sales",
          x="Name", ax=ax0)
```

```
top_10.plot(kind='barh', y="Purchases",
          x="Name", ax=ax1)
```




```
ax1.axvline(x=avg, color='b',
           label='Average',
           linestyle='--',
           linewidth=1)
```

```
ax0.set_xlim([-10000, 140000])
```

```
ax0.set(title='Revenue', xlabel='Total
      Revenue', ylabel='Customers')
```

```
ax1.set(title='Units',
       xlabel='Total Units', ylabel='')
```

```
fig.savefig('sales.png', transparent=False, dpi=80, bbox_inches="tight")
```

pbpython.com

For queries, contact

Chetan Chauhan

Consultant- Analytics
Chetan.chauhan2@exlservice.com
Mobile: 91-9958446965