DIGITAL NOTES
ON
DATA VISUALIZATION

B.TECH IV YEAR
(2022-23)

Edited by Jogi John

DEPARTMENT OF COMPUTER TECHNOLOGY

LOKMANYA TILAK JANKALYAN SHIKSHAN SANSTHA'S
PRIYADARSHINI COLLEGE OF ENGINEERING
(Approved by A.I.C.T.E., New Delhi & Govt. Of Maharashtra,
Affiliated to R.T.M. Nagpur University, Nagpur)
Priyadarshini Campus, Near CRPF, Hingna Road,
Nagpur – 440 019, Maharashtra (India)

PRIYADARSHINI COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER TECHNOLOGY

Elective IV : Data Visualization (TH)

Course Objectives:

To learn different statistical methods for Datavisualization.

To understand the basics of R and Python.

To learn usage of Watson studio.

To understand the usage of the packages like Numpy, pandas and matplotlib.

To know the functionalities and usages ofSeaborn.

UNIT I

Introduction to Statistics : Introduction to Statistics, Difference between inferential statistics and Descriptive statistics, Inferential Statistics- Drawing Inferences from Data, Random Variables, Normal Probability Distribution, Sampling, Sample Statistics and Sampling Distributions.

R overview and Installation- Overview and About R, R and R studio Installation, Descriptive Data analysis using R, Description of basic functions used to describe data in R.

UNIT II

Data manipulation with R: Data manipulation packages-dplyr,data.table, reshape2, tidyr, Lubridate, Data visualization with R.

Data visualization in Watson Studio: Adding data to datarefinery, Visualization of Data on Watson Studio.

UNIT III

Python: Introduction to Python, How to Install, Introduction to Jupyter Notebook, Python Scripting basics, Numpy and Pandas-Creating and Accessing Numpy Arrays, Introduction to pandas, read and write csv, Descriptive statistics using pandas, Working with text data and datetime columns, Indexing and selecting data, groupby, Merge / Joindatasets

UNIT IV

Data Visualization Tools in Python- Introduction to Matplotlib, Basic plots using matplotlib, Specialized Visualization Tools usingMatplotlib, Advanced Visualization Tools using Matplotlib Waffle Charts, Word Clouds.

UNIT V

Introduction to Seaborn: Seaborn functionalities and usage, Spatial Visualizations and Analysis in Python with Folium, Case Study.

TEXT BOOKS:

1. Core Python Programming - Second Edition,R. Nageswara Rao, DreamtechPress.
2. Hands on programming with R by Garrett Grolemund,Shroff/O'Reilly; Firstedition
3. Fundamentals of Mathematical Statistics by S.C. Gupta, Sultan Chand &Son9

REFERENCE BOOKS:

1. Learn R for Applied Statistics: With Data Visualizations, Regressions, and Statistics by Eric Goh Ming Hui,Apress
2. Python for Data Analysis by William McKinney, Second Edition, O'Reilly Media Inc.\
3. The Comprehensive R Archive Network-https://cran.r-project.org

PRIYADARSHINI COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER TECHNOLOGY

INDEX

UNIT-1

Introduction to Statistics

Statistics is a mathematical science that includes methods for collecting, organizing, analyzing and visualizing data in such a way that meaningful conclusions can be drawn.

Statistics is also a field of study that summarizes the data, interpret the data making decisions based on the data.

Statistics is composed of two broad categories:

1. Descriptive Statistics
2. Inferential Statistics

1. Descriptive Statistics

Descriptive statistics describes the characteristics or properties of the data. It helps to summarize the data in a meaningful data in a meaningful way. It allows important patterns to emerge from the data. Data summarization techniques are used to identify the properties of data. It is helpful in understanding the distribution of data. They do not involve in generalizing beyond the data.

1.1 Two types of descriptive statistics

1. Measures of Central Tendency: (Mean , Median , Mode)
2. Measures of data spread or dispersion (range, quartiles, variance and standard deviation)

1.1.1 Measures of Central Tendency: (Mean , Median , Mode)

A measure of central tendency is a single value that attempts to describe a set of data by identifying the central position within that set of data. The mean, median and mode are all valid measures of central tendency.

Mean (Arithmetic)

The mean (or average) is the most popular and well known measure of central tendency. It can be used with both discrete and continuous data, although its use is most often with continuous data.

The mean is equal to the sum of all the values in the data set divided by the number of values in the data set. So, if we have values in a data set and they have values $x_1, x_2, \ldots x_n$, the sample mean, usually denoted by $\bar{x}$.

$\bar{x} = (x_1 x_2, \ldots x_n)/ n$.

An important property of the mean is that it includes every value in the data set as part of the calculation. In addition, the mean is the only measure of central tendency where the sum of the deviations of each value from the mean is always zero.

Median:

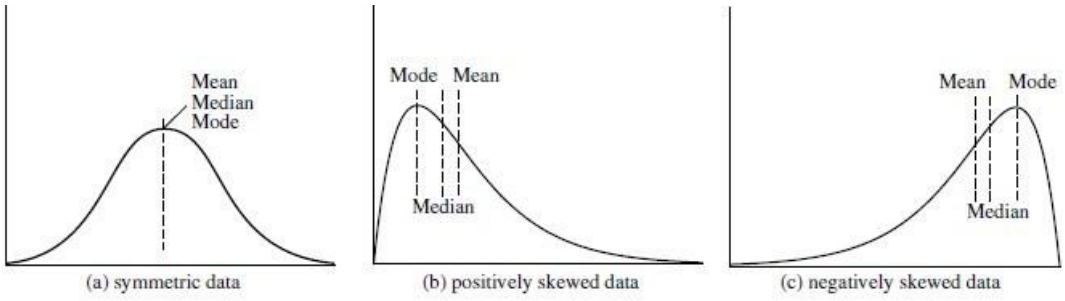
The median is the middle score for a set of data that has been arranged in order of magnitude. The median is less affected by outliers and skewed data. It is a holistic measure. It is easy method of approximation of median value of a large data set.

Mode

The mode is the most frequent score in our data set. The mode is used for categorical data where we want to know which is the most common category occurring in the population. There are possibilities for the greatest frequency to correspond to different values. This results in more than one,two or more modes in a dataset. They are called as unimodal, bimodal and multimodal datasets. If each data occurs only once then the mode is equal to zero.

Unimodal frequency curve with symmetric data distribution , the mean median and mode are all the same.

In real applications the data is not symmetrical and they are asymmetric.It might be positively skewed or negatively skewed. If positively skewed then mode is smaller than median and in negatively skewed the mode occurs at a value greater than the median.



Mean, median, and mode of symmetric versus positively and negatively skewed data.

1.1.2 Measures of spread:

Measures of spread are the ways of summarizing a group of data by describing how scores are spread out. To describe this spread, a number of statistics are available to us, including the range, quartiles, absolute deviation, variance and standard deviation.

- The degree to which numerical data tend to spread is called the dispersion, or variance of the data. The common measures of data dispersion: Range, Quartiles, Outliers, and Boxplots.

Range : Range of the set is the difference between the largest (max()) and smallest (min()) values.
Ex: Step 1: Sort the numbers in order, from smallest to largest: 7, 10, 21, 33, 43, 45, 45, 65, 67, 87, 98, 99

Step 2: Subtract the smallest number in the set from the largest number in the set:
99 – 7 = 92

The range is 92

Quartiles : Percentile : kth percentile of a set of data in numerical order is the value xi having the property that k percent of the data entries lie at or below xi

- The first quartile (Q1) is the 25th percentile;
- The third quartile (Q3) is the 75th percentile
- The distance between the first and third quartiles is the range covered by the middle half of the data.
- Interquartile range (IQR) and is defined as IQR = Q3 - Q1.
- Outliers is to single out values falling at least 1.5 *IQR above the third quartile or below the first quartile.
- Five-number summary: median, the quartiles Q1 and Q3, and the smallest and largest individual observations comprise the five number summary: Minimum; Q1; Median; Q3; Maximum

  Example : Quartiles
- Start with the following data set:
- 1, 2, 2, 3, 4, 6, 6, 7, 7, 7, 8, 11, 12, 15, 15, 15, 17, 17, 18, 20
- There are a total of twenty data points in the set. There is an even number of data values, hence the median is the mean of the tenth and eleventh values.
- the median is: (7 + 8)/2 = 7.5.
- The median of the first half of the set is found between the fifth and sixth values of:
- 1, 2, 2, 3, 4, 6, 6, 7, 7, 7
- Thus the first quartile is found to equal Q1 = (4 + 6)/2 = 5
- To find the third quartile, examine the top half of the original data set. The median of
- 8, 11, 12, 15, 15, 15, 17, 17, 18, 20
- is (15 + 15)/2 = 15. Thus the third quartile Q3 = 15.

A small interquartile range indicates data that is clumped about the median. A larger interquartile range shows that the data is more spread out

Variance and Standard Deviation

The variance of N observations, $x_1, x_2, \ldots, x_N$, is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 = \frac{1}{N}\left[\sum x_i^2 - \frac{1}{N}(\sum x_i)^2\right],$$

Inferential Statistics – Definition and Types

Inferential statistics is generally used when the user needs to make a conclusion about the whole population at hand, and this is done using the various types of tests available. It is a technique which is used to understand trends and draw the required conclusions about a large population by taking and analyzing a sample from it. Descriptive statistics, on the other hand, is only about the smaller sized data set at hand – it usually does not involve large populations. Using variables and the relationships between them from the sample, we will be able to make generalizations and predict other relationships within the whole population, regardless of how large it is.

With inferential statistics, data is taken from samples and generalizations are made about a population. Inferential statistics use statistical models to compare sample data to other samples or to previous research.

There are two main areas of inferential statistics:

1. Estimating parameters:

This means taking a statistic from the sample data (for example the sample mean) and using it to infer about a population parameter (i.e. the population mean).There may be sampling variations because of chance fluctuations, variations in sampling techniques, and other sampling errors. Estimation about population characteristics may be influenced by such factors. Therefore, in estimation the important point is that to what extent our estimate is close to the true value.

Characteristics of Good Estimator: A good statistical estimator should have the following characteristics, (i) Unbiased (ii) Consistent (iii) Accuracy

i) Unbiased

An unbiased estimator is one in which, if we were to obtain an infinite number of random samples of a certain size, the mean of the statistic would be equal to the parameter. The sample mean, (x) is an unbiased estimate of population mean (μ)because if we look at possible random samples of size N from a population, then mean of the sample would be equal to μ.

ii) Consistent

A consistent estimator is one that as the sample size increased, the probability that estimate has a value close to the parameter also increased. Because it is a consistent estimator, a sample mean based on 20 scores has a greater probability of being closer to (μ) than does a sample mean based upon only 5 scores

iii) Accuracy

The sample mean is an unbiased and consistent estimator of population mean (μ).But we should not over look the fact that an estimate is just a rough or approximate calculation. It is unlikely in any estimate that (x) will be exactly equal to population mean (μ). Whether or not x is a good estimate of (μ) depends upon the representativeness of sample, the sample size, and the variability of scores in the population.

2. **Hypothesis tests.** This is where sample data can be used to answer research questions. For example, we might be interested in knowing if a new cancer drug is effective. Or if breakfast helps children perform better in schools.

Inferential statistics is closely tied to the logic of hypothesis testing. We hypothesize that this value characterise the population of observations. The question is whether that hypothesis is reasonable evidence from the sample. Sometimes hypothesis testing is referred to as statistical decision-making process. In day-to-day situations we are required to take decisions about the population on the basis of sample information.

2.6.1 Statement of Hypothesis

A statistical hypothesis is defined as a statement, which may or may not be true about the population parameter or about the probability distribution of the parameter that we wish to validate on the basis of sample information. Most times, experiments are performed with random samples instead of the entire population and inferences drawn from the observed results are then generalised over to the entire population. But before drawing inferences about the population it should be always kept in mind that the observed results might have come due to chance factor. In order to have an accurate or more precise inference, the chance factor should be ruled out.

Null Hypothesis

The probability of chance occurrence of the observed results is examined by the null hypothesis (H0). Null hypothesis is a statement of no differences. The other way to state null hypothesis is that the two samples came from the same population. Here, we assume that population is normally distributed and both the groups have equal means and standard deviations.

Since the null hypothesis is a testable proposition, there is counter proposition to it known as alternative hypothesis and denoted by H1 . In contrast to null hypothesis, the alternative hypothesis (H1) proposes that

i) the two samples belong to two different populations,
ii) their means are estimates of two different parametric means of the respective population, and
iii) there is a significant difference between their sample means.

The alternative hypothesis (H1) is not directly tested statistically; rather its acceptance or rejection is determined by the rejection or retention of the null hypothesis. The probability ‘p’ of the null hypothesis being correct is assessed by a statistical test. If probability ‘p’ is too low, H0 is rejected and H1 is accepted.

It is inferred that the observed difference is significant. If probability ‘p’ is high, H0 is accepted and it is inferred that the difference is due to the chance factor and not due to the variable factor.

2.6.2 Level of Significance

The level of significance is defined as the probability of rejecting a null hypothesis by the test when it is really true, which is denoted as α. That is, P (Type I error) = α.

Confidence level:

Confidence level refers to the possibility of a parameter that lies within a specified range of values, which is denoted as c. Moreover, the confidence level is connected with the level of significance. The relationship between level of significance and the confidence level is $c=1-\alpha$. The common level of significance and the corresponding confidence level are given below:

- The level of significance 0.10 is related to the 90% confidence level.
- The level of significance 0.05 is related to the 95% confidence level.
- The level of significance 0.01 is related to the 99% confidence level.

The rejection rule is as follows:

- If p-value $\leq$ level of significance (α), then reject the null hypothesis H_0.
- If p-value $>$ level of significance (α), then do not reject the null hypothesis H_0.

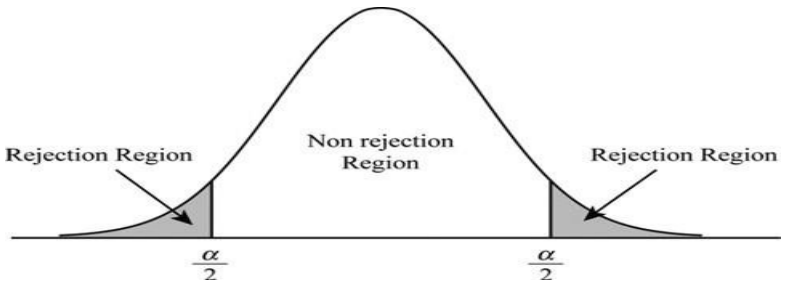
Rejection region:

The rejection region is the values of test statistic for which the null hypothesis is rejected.

Non rejection region:

The set of all possible values for which the null hypothesis is not rejected is called the rejection region.

The rejection region for two-tailed test is shown below:



The rejection region for one-tailed test is given below:

In the left-tailed test, the rejection region is shaded in left side.

In the right-tailed test, the rejection region is shaded in right side.

2.6.3 One-tail and Two-tail Test

Depending upon the statement in alternative hypothesis (H1), either a one-tail or two tail test is chosen for knowing the statistical significance. A one-tail test is a directional test. It is formulated to find the significance of both the magnitude and the direction (algebraic sign) of the observed difference between two statistics. Thus, in two-tailed tests researcher is interested in testing whether one sample mean is significantly higher (alternatively lower) than the other sample mean.

Types of Inferential Statistics Tests

There are many tests in this field, of which some of the most important are mentioned below.

1. Linear Regression Analysis

In this test, a linear algorithm is used to understand the relationship between two variables from the data set. One of those variables is the dependent variable, while there can be one or more independent variables used. In simpler terms, we try to predict the value of the dependent variable based on the available values of the independent variables. This is usually represented by using a scatter plot, although we can also use other types of graphs too.

2. Analysis of Variance

This is another statistical method which is extremely popular in data science. It is used to test and analyse the differences between two or more means from the data set. The significant differences between the means are obtained, using this test.

3. Analysis of Co-variance

This is only a development on the Analysis of Variance method and involves the inclusion of a continuous co-variance in the calculations. A co-variate is an independent variable which is continuous, and is used as regression variables. This method is used extensively in statistical modelling, in order to study the differences present between the average values of dependent variables.

4. Statistical Significance (T-Test)

A relatively simple test in inferential statistics, this is used to compare the means of two groups and understand if they are different from each other. The order of difference, or how significant the differences are can be obtained from this.

5. Correlation Analysis

Another extremely useful test, this is used to understand the extent to which two variables are dependent on each other. The strength of any relationship, if they exist, between the two variables can be obtained from this. You will be able to understand whether the variables have a strong correlation or a weak one. The correlation can also be negative or positive, depending upon the variables. A negative correlation means that the value of one variable decreases while the value of the other increases and positive correlation means that the value both variables decrease or increase simultaneously.

Differences between Descriptive and Inferential Statistics

Descriptive Statistics	Inferential Statistics
Concerned with describing the target population	Make inferences from the sample and generalize them to the population

Organise, analyse, present the data in a meaningful way	Compare, tests and predicts future outcomes
The analysed results are in the form of graphs, charts etc	The analysed results are the probability scores
Describes the data which is already known	Tries to make conclusions about the population beyond the data available
Tools: Measures of central tendency and measures of spread	Tools: Hypothesis tests, analysis of variance etc

Random Variables

A random variable, X, is a variable whose possible values are numerical outcomes of a random phenomenon. There are two types of random variables, discrete and continuous.

Example of Random variable

- A person's blood type
- Number of leaves on a tree
- Number of times a user visits LinkedIn in a day
- Length of a tweet.

Discrete Random Variables :

A discrete random variable is one which may take on only a countable number of distinct values such as 0,1,2,3,4,..... Discrete random variables are usually counts. If a random variable can take only a finite number of distinct values, then it must be discrete. Examples of discrete random variables include the number of children in a family, the Friday night attendance at a cinema, the number of patients in a doctor's surgery, the number of defective light bulbs in a box of ten.

The **probability distribution** of a discrete random variable is a list of probabilities associated with each of its possible values. It is also sometimes called the probability function or the probability mass function

Suppose a random variable X may take k different values, with the probability that $X = x_i$ defined to be $P(X = x_i) = p_i$. The probabilities p_i must satisfy the following:

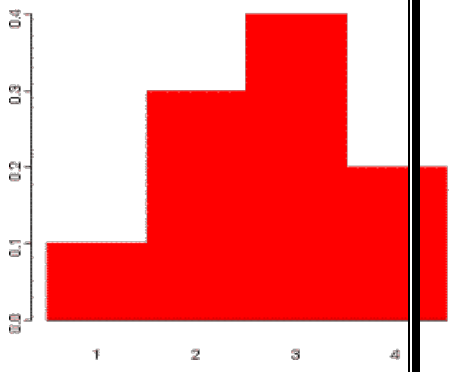
1: $0 \leq p_i \leq 1$ for each i

2: $p_1 + p_2 + ... + p_k = 1$.

Example

Suppose a variable X can take the values 1, 2, 3, or 4. The probabilities associated with each outcome are described by the following table:

Outcome	1	2	3	4
Probability	0.1	0.3	0.4	0.2



The probability that X is equal to 2 or 3 is the sum of the two probabilities: P(X = 2 or X = 3) = P(X = 2) + P(X = 3) = 0.3 + 0.4 = 0.7. Similarly, the probability that X is greater than 1 is equal to 1 - P(X = 1) = 1 - 0.1 = 0.9, by the complement rule.

Continuous Random Variables

A **continuous random variable** is one which takes an infinite number of possible values. Continuous random variables are usually measurements. Examples include height, weight, the amount of sugar in an orange, the time required to run a mile.

A continuous random variable is not defined at specific values. Instead, it is defined over an interval of values, and is represented by the **area under a curve** (known as an integral). The probability of observing any single value is equal to 0, since the number of values which may be assumed by the random variable is infinite.

Suppose a random variable X may take all values over an interval of real numbers. Then the probability that X is in the set of outcomes A, P(A), is defined to be the area above A and under a curve. The curve, which represents a function p(x), must satisfy the following:

1: The curve has no negative values ($p(x) \geq 0$ for all x)

2: The total area under the curve is equal to 1.

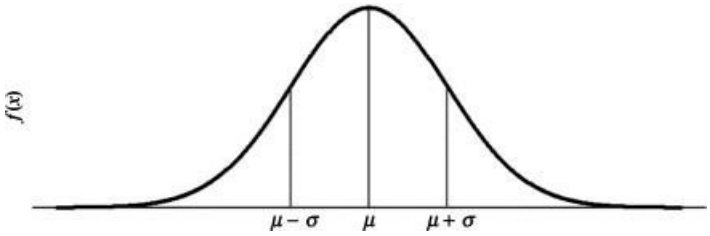
A curve meeting these requirements is known as a **density curve**.

All random variables (discrete and continuous) have a **cumulative distribution function**. It is a function giving the probability that the random variable X is less than or equal to x, for every value x. For a discrete random variable, the cumulative distribution function is found by
summing up the probabilities.

Normal Probability Distribution

The Bell-Shaped Curve

The **Bell-shaped Curve** is commonly called the **normal curve** and is mathematically referred to as the Gaussian probability distribution. Unlike Bernoulli trials which are based on discrete counts, the **normal distribution** is used to determine the probability of a continuous random variable.



The **normal** or Gaussian Probability Distribution is most popular and important because of its unique mathematical properties which facilitate its application to practically any physical problem in the real world. The constants **μ** and $\boldsymbol{\sigma^2}$ are the parameters;

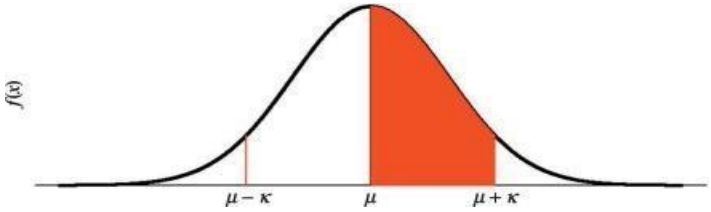
"**μ**" is the population true mean (or expected value) of the subject phenomenon characterized by the continuous random variable, **X**,

"$\boldsymbol{\sigma^2}$" is the population true variance characterized by the continuous random variable, **X**.

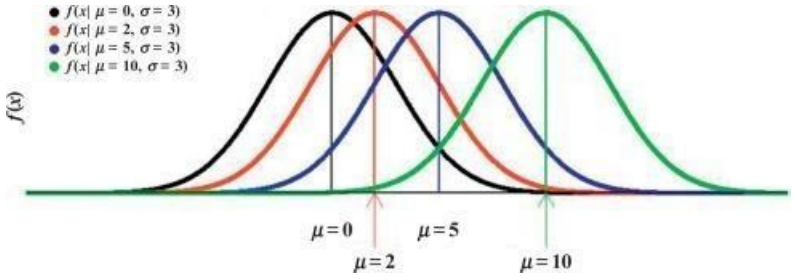
Hence, "**σ**" the population standard deviation characterized by the continuous random variable **X**;

the points located at μ−σ and μ+σ are the points of inflection; that is, where the graph changes from cupping up to cupping down

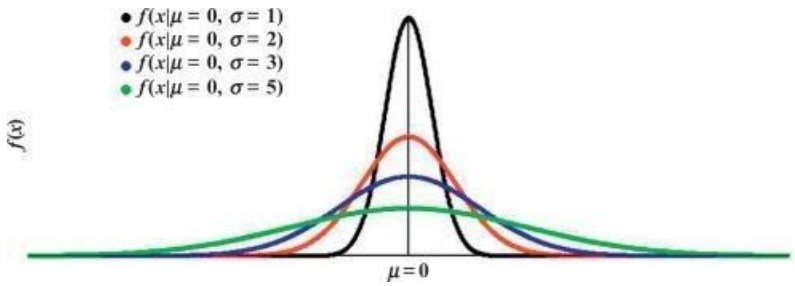
The **normal curve graph of the normal probability distribution**) is **symmetric** with respect to the mean **μ** as the **central position**. That is, the area between **μ** and **κ** units to the left of **μ** is equal to the area between **μ** and **κ** units to the right of **μ**.



There is not a unique **normal probability distribution.** The figure below is a graphical representation of the normal distribution for a fixed value of σ2 with μ varying.



The figure below is a graphical representation of the **normal distribution** for a fixed value of **μ** with varying σ².



SAMPLING and SAMPLING DISTRIBUTION

Sampling is a process used in statistical analysis in which a predetermined number of observations are taken from a larger population. It helps us to make statistical inferences about the population. A population can be defined as a whole that includes all items and characteristics of the research taken into study. However, gathering all this information is time consuming and costly. We therefore make inferences about the population with the help of samples.

Random sampling:

In data collection, every individual observation has equal probability to be selected into a sample. In random sampling, there should be no pattern when drawing a sample.

Probability sampling:

It is the sampling technique in which every individual unit of the population has greater than zero probability of getting selected into a sample.

Non-probability sampling:

It is the sampling technique in which some elements of the population have no probability of getting selected into a sample.

Cluster samples:

It divides the population into groups (clusters). Then a random sample is chosen from the clusters.

Systematic sampling : select sample elements from an ordered frame. A sampling frame is just a list of participants that we want to get a sample from.

Stratified sampling : sample each subpopulation independently. First, divide the population into homogeneous (very similar) subgroups before getting the sample. Each population member only belongs to one group. Then apply simple random or a systematic method within each group to choose the sample.

Sampling Distribution

A sampling distribution is a probability distribution of a statistic. It is obtained through a large number of samples drawn from a specific population. It is the distribution of all possible values taken by the statistic when all possible samples of a fixed size n are taken from the population.

Sampling Distributions and Inferential Statistics

Sampling distributions are important for inferential statistics. A population is specified and the sampling distribution of the mean and the range were determined. In practice, the process proceeds the other way: the sample data is collected and from these data we estimate parameters of the sampling distribution. This knowledge of the sampling distribution can be very useful.

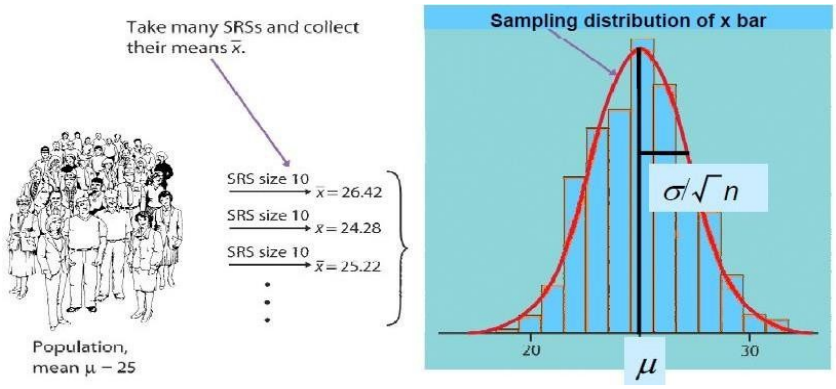
Knowing the degree to which means from different samples would differ from each other and from the population mean (this would give an idea of how close the particular sample mean is likely to be to the population mean)

The most common measure of how much sample means differ from each other is the standard deviation of the sampling distribution of the mean. This standard deviation is called the standard error of the mean.

If all the sample means were very close to the population mean, then the standard error of the mean would be small. On the other hand, if the sample means varied considerably, then the standard error of the mean would be large.

Sampling distribution of the sample mean

1. We take many random samples of a given size n from a population with mean μ and standard deviation σ.
2. Some sample means will be above the population mean μ and some will be below, making up the sampling distribution.



For any population with mean μ and standard deviation σ:

- The **mean**, or center of the sampling distribution of $\bar{x}$, is equal to the population mean μ: $\mu_{\bar{x}} = \mu$.
- The **standard deviation** of the sampling distribution is $\sigma/\sqrt{n}$, where n is the sample size: $\sigma_{\bar{x}} = \sigma/\sqrt{n}$.

Application

Hypokalemia is diagnosed when blood potassium levels are below 3.5mEq/dl. Let's assume that we know a patient whose measured potassium levels vary daily according to a normal distribution $N(\mu = 3.8, \sigma = 0.2)$.

If only one measurement is made, what is the probability that this patient will be misdiagnosed with Hypokalemia?

$$z = \frac{(x - \mu)}{\sigma} = \frac{3.5 - 3.8}{0.2} = -1.5\,, \quad P(z < -1.5) = 0.0668 \approx 7\%$$

Instead, if measurements are taken on 4 separate days, what is the probability of a misdiagnosis?

$$z = \frac{(\bar{x} - \mu)}{\sigma/\sqrt{n}} = \frac{3.5 - 3.8}{0.2/\sqrt{4}} = -3\,, \quad P(z < -3) = 0.0013 \approx 0.1\%$$

R overview and Installation

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

R is free software distributed under a GNU-style copy left, and an official part of the GNU project called **GNUs**.

Features of R

R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.

R has an effective data handling and storage facility,

R provides a suite of operators for calculations on arrays, lists, vectors and matrices.

R provides a large, coherent and integrated collection of tools for data analysis.

R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

To Install R:

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for Windows" link at the top of the page.
5. Click on the "install R for the first time" link at the top of the page.
6. Click "Download R for Windows" and save the executable file somewhere on computer. Run the .exe file and follow the installation instructions.
7. Now that R is installed, next step is to download and install RStudio.

To Install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Windows version, and save the executable file. Run the .exe file and follow the installation instructions.

R Command Prompt

Once R environment setup is done, then it's easy to start R command prompt by just typing the following command at command prompt – "$ R"

This will launch R interpreter and will get a prompt > where we can start typing your program as follows −

```
> myString <- "Hello, World!"
> print ( myString)
```

```
[1] "Hello, World!"
```

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

R Script File

execute scripts at command prompt with the help of R interpreter called **Rscript**.

```
# My first program in R Programming
myString <- "Hello, World!"
print ( myString)
```

Save the above code in a file test.R and execute it at command prompt as given below.

$ Rscript test.R

When we run the above program, it produces the following result.
"Hello, World!"

Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing actual program. Single comment is written using # in the beginning of the statement as follows −

My first program in R Programming
R does not support multi-line comments but they can be written as follows:
"This is a demo for multi-line comments and it should be put inside either a
single OR double quote"

myString <- "Hello, World!"
print (myString)
Result for above code is:
"Hello, World!"

R data types:

The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are −

Vectors
Lists
Matrices
Arrays
Factors
Data Frames

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type	Example	Verify
Logical	TRUE, FALSE	v <- TRUE print(class(v)) [1] "logical"
Numeric	12.3, 5, 999	v <- 23.5 print(class(v)) [1] "numeric"
Integer	2L, 34L, 0L	v <- 2L

		print(class(v)) [1] "integer"
Complex	3 + 2i	v <- 2+5i print(class(v)) [1] "complex"
Character	'a' , '"good", "TRUE", '23.4'	v <- "TRUE" print(class(v)) [1] "character"
Raw	"Hello" is stored as 48 65 6c 6c 6f	v<-charToRaw("Hello") print(class(v)) [1] "raw"

In R programming, the very basic data types are the R-objects called **vectors** which hold elements of different classes as shown above.

Vectors

When you want to create vector with more than one element, you should use **c()** function which means to combine the elements into a vector.

```
# Create a vector.
apple <- c('red','green',"yellow")
print(apple)
# Get the class of the vector.
print(class(apple))
```

When we execute the above code, it produces the following result −

```
"red"    "green" "yellow"
"character"
```

Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.
list1 <- list(c(2,5,3),21.3,sin)
# Print the list.
print(list1)
```

When we execute the above code, it produces the following result −

```
[[1]]
[1] 2 5 3
[[2]]
[1] 21.3
 [[3]]
```

function (x) .Primitive("sin")

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)
print(M)
```

When we execute the above code, it produces the following result −

```
     [,1] [,2] [,3]
[1,] "a"  "a"  "b"
[2,] "c"  "b"  "a"
```

Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.
a <- array(c('green','yellow'),dim = c(3,3,2))
print(a)
```

When we execute the above code, it produces the following result −

```
, , 1
     [,1]     [,2]     [,3]
[1,] "green" "yellow" "green"
[2,] "yellow" "green"  "yellow"
[3,] "green" "yellow" "green"
, , 2
     [,1]     [,2]     [,3]
[1,] "yellow" "green"  "yellow"
[2,] "green" "yellow" "green"
[3,] "yellow" "green"  "yellow"
```

Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length. Data Frames are created using the **data.frame()** function.

```
# Create the data frame.
BMI <- data.frame(    gender = c("Male", "Male","Female"),    height = c(152, 171.5, 165),
   weight = c(81,93, 78),    Age = c(42,38,26) )
print(BMI)
```

Result −

```
  gender height weight Age
1   Male  152.0     81  42
2   Male  171.5     93  38
3 Female  165.0     78  26
```

R - Variables

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name, var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

R - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operators

types of operators in R programming −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

Descriptive Data analysis using R:

R provides a wide range of functions for obtaining summary statistics. One method of obtaining descriptive statistics is to use the **sapply( )** function with a specified summary statistic.

```
sapply(mydata, mean, na.rm=TRUE)
```

Possible functions used in sapply include **mean, sd, var, min, max, median, range, and quantile**.

Check your data

You can inspect your data using the functions **head**() and **tails**(), which will display the first and the last part of the data, respectively.

```
# Print the first 6 rows
head(my_data, 6)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

R functions for computing descriptive statistics

Some R functions for computing descriptive statistics:

Description	R function
Mean	**mean**()
Standard deviation	**sd**()
Variance	**var**()
Minimum	**min**()
Maximum	**maximum**()
Median	**median**()
Range of values (minimum and maximum)	**range**()
Sample quantiles	**quantile**()
Generic function	**summary**()
Interquartile range	**IQR**()

Descriptive statistics for a single group

Measure of central tendency: mean, median, mode

Roughly speaking, the central tendency measures the “average” or the “middle” of your data. The most commonly used measures include:

the mean: the average value. It's sensitive to outliers.
the median: the middle value. It's a robust alternative to mean.
and the mode: the most frequent value

In R,

The function **mean**() and **median**() can be used to compute the mean and the median, respectively;
The function **mfv**() [in the **modeest** R package] can be used to compute the mode of a variable.

The R code below computes the mean, median and the mode of the variable Sepal.Length [in my_data data set]:

```
# Compute the mean value
mean(my_data$Sepal.Length)
[1] 5.843333

# Compute the median value
median(my_data$Sepal.Length)
[1] 5.8

# Compute the mode
# install.packages("modeest")
require(modeest)
mfv(my_data$Sepal.Length)
[1] 5
```

Measure of variability

Measures of variability gives how "spread out" the data are.

Range: minimum & maximum

Range corresponds to biggest value minus the smallest value. It gives you the full spread of the data.

```
# Compute the minimum value
min(my_data$Sepal.Length)
[1] 4.3
# Compute the maximum value
max(my_data$Sepal.Length)
[1] 7.9
# Range
range(my_data$Sepal.Length)
[1] 4.3 7.9
```

Interquartile range

The **interquartile range** (IQR) - corresponding to the difference between the first and third quartiles - is sometimes used as a robust alternative to the standard deviation.

R function:

```
quantile(x, probs = seq(0, 1, 0.25))
```

x: numeric vector whose sample quantiles are wanted.
probs: numeric vector of probabilities with values in [0,1].

Example:

```
quantile(my_data$Sepal.Length)
  0% 25% 50% 75% 100%
 4.3 5.1 5.8 6.4  7.9
```

To compute deciles (0.1, 0.2, 0.3, …., 0.9), use this:

```
quantile(my_data$Sepal.Length, seq(0, 1, 0.1))
```

To compute the interquartile range, type this:

```
IQR(my_data$Sepal.Length)
[1] 1.3
```

Variance and standard deviation

The variance represents the average squared deviation from the mean. The standard deviation is the square root of the variance. It measures the average deviation of the values, in the data, from the mean value.

```
# Compute the variance
var(my_data$Sepal.Length)

# Compute the standard deviation =
# square root of th variance
sd(my_data$Sepal.Length)
```

Computing an overall summary of a variable and an entire data frame summary() function

Summary of a single variable. Five values are returned: the mean, median, 25th and 75th quartiles, min and max in one single line call:

```
summary(my_data$Sepal.Length)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300 5.100 5.800 5.843 6.400  7.900
```

Summary of a data frame. In this case, the function **summary**() is automatically applied to each column. The format of the result depends on the type of the data contained in the column. For example:

- If the column is a numeric variable, mean, median, min, max and quartiles are returned.
- If the column is a factor variable, the number of observations in each group is returned.

```
summary(my_data, digits = 1)
  Sepal.Length  Sepal.Width  Petal.Length    Petal.Width    Species
```

```
 Min. :4 Min. :2 Min. :1 Min.  :0.1 setosa :50
 1st Qu.:5  1st Qu.:3  1st Qu.:2  1st Qu.:0.3  versicolor:50
 Median :6 Median :3 Median :4 Median :1.3 virginica :50
 Mean  :6  Mean  :3  Mean  :4  Mean  :1.2
 3rd Qu.:6 3rd Qu.:3  3rd Qu.:5  3rd Qu.:1.8
 Max. :8 Max. :4 Max. :7 Max. :2.5
```

sapply() function

```
# Compute the mean of each column
sapply(my_data[, -5], mean)
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
   5.843333     3.057333     3.758000     1.199333
#        Compute        quartiles
sapply(my_data[, -5], quantile)
    Sepal.Length Sepal.Width Petal.Length Petal.Width
0%           4.3         2.0         1.00         0.1
25%          5.1         2.8         1.60         0.3
50%          5.8         3.0         4.35         1.3
75%          6.4         3.3         5.10         1.8
100%         7.9         4.4         6.90         2.5
```

Descriptive Data Analysis using R > Description of Basic Functions used to Describe Data in R

builtins()	# List all built-in functions
help() or ? or ??	#i.e. help(boxplot)
getwd() and setwd()	# working with a file directory
q()	#To close R
ls()	#Lists all user defined objects.
rm()	#Removes objects from an environment.
demo()	#Lists the demonstrations in the packages that are loaded.
demo(package = .packages(all.available = TRUE))	#Lists the demonstrations in all installed packages.
?NA	# Help page on handling of missing data values
abs(x)	# The absolute value of "x"
append()	# Add elements to a vector
cat(x)	# Prints the arguments
cbind()	# Combine vectors by row/column (cf. "paste" in Unix)
grep()	# Pattern matching
identical()	# Test if 2 objects are *exactly* equal
length(x)	# Return no. of elements in vector x
ls()	# List objects in current environment
mat.or.vec()	# Create a matrix or vector
paste(x)	# Concatenate vectors after converting to character
range(x)	# Returns the minimum and maximum of x
rep(1,5)	# Repeat the number 1 five times

rev(x)	# List the elements of "x" in reverse order
seq(1,10,0.4)	# Generate a sequence (1 -> 10, spaced by 0.4)
sequence()	# Create a vector of sequences
sign(x)	# Returns the signs of the elements of x
sort(x)	# Sort the vector x
order(x)	# list sorted element numbers of x
tolower(),toupper()	# Convert string to lower/upper case letters
unique(x)	# Remove duplicate entries from vector
vector()	# Produces a vector of given length and mode
formatC(x)	# Format x using 'C' style formatting specifications
floor(x), ceiling(x), round(x), signif(x), trunc(x)	# rounding functions
Sys.time()	# Return system time
Sys.Date()	# Return system date
getwd()	# Return working directory
setwd()	# Set working directory

Inferential statistics using R

Simple linear regression analysis

- Regression analysis is a very widely used statistical tool to establish a relationship model between two variables
- One of these variable is called predictor variable
- The other variable is called response variable
- The general mathematical equation for a linear regression is y = mx + b

	Register_no	**Name**	**Dept**	**CGPA**	**Height**	**Weight**
1	18N312001	JOHN	IT	8.5	151	63
2	18N312005	SIM	CSE	9.2	174	81
3	18N312011	TIM	IT	9.5	138	56
4	18N312061	LILLY	IT	9.34	186	91
5	18N312099	CARL	MECH	8.12	128	47

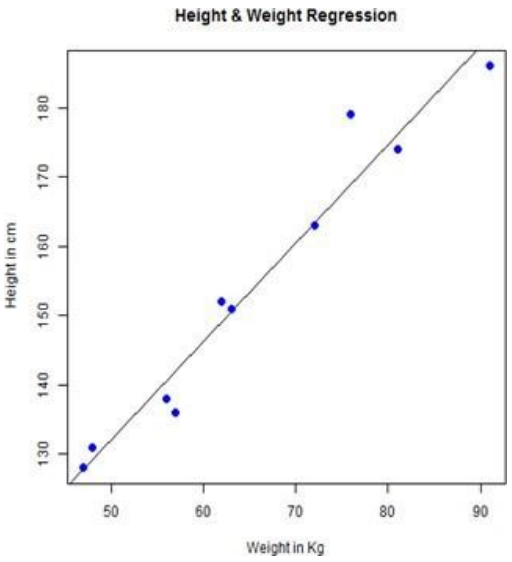
- lm() Function
- This function creates the relationship model between the predictor and the response variable.
- The basic syntax for lm() function in linear regression is −
- lm(formula,data)
- # Apply the lm() function.
- relation <- lm(stud.data$weight ~ stud.data$height)
- print(relation)

Output

Coefficients:

(Intercept (m)) x

-38.4551 0.6746



UNIT-II

Introduction

Data Manipulation is an important phase of predictive modeling. A robust predictive model cannot be built using machine learning algorithms. But, with an approach to understand the business problem, the underlying data and extracting business insights are done performing required data manipulations. Among several phases of model building, most of the time is usually spent in understanding underlying data and performing required manipulations.

Data Manipulation

It involves 'manipulating' data using available set of variables. This is done to enhance accuracy and precision associated with data. Actually, the data collection process can have many loopholes. There are various uncontrollable factors which lead to inaccuracy in data such as mental situation of respondents, personal biases, difference / error in readings of machines etc. To lessen these inaccuracies, data manipulation is done to increase the possible (highest) accuracy in data. This stage is also known as data wrangling or data cleaning.

Different Ways to Manipulate / Treat Data:

Manipulating data using inbuilt base R functions. This is the first step, but is often repetitive and time consuming. Hence, it is a less efficient way to solve the problem.
Use of packages for data manipulation. CRAN has more than 8000 packages available today. These packages are a collection of pre-written commonly used pieces of codes. They helps to perform the repetitive tasks fasts, reduce errors in coding and take help of code written by experts (across the open source eco-system for R) to make code more efficient. This is usually the most common way of performing data manipulation.
Use of Machine Learning(ML) algorithms for data manipulation. ML algorithms like tree based boosting algorithms to take care of missing data & outliers. These algorithms are less time consuming,

Note: Install packages using:

```
install.packages('package name')
```

List of Packages

1. dplyr
2. data.table
3. ggplot2
4. reshape2
5. readr
6. tidyr

7. lubridate

dplyr Package

This package is created and maintained by Hadley Wickham. This package has everything (almost) to accelerate data manipulation efforts. It is known best for data exploration and transformation. Its chaining syntax makes it highly adaptive to use. It includes 5 major data manipulation commands:

1. filter – It filters the data based on a condition
2. select – It is used to select columns of interest from a data set
3. arrange – It is used to arrange data set values on ascending or descending order
4. mutate – It is used to create new variables from existing variables
5. summarise (with group_by) – It is used to perform analysis by commonly used operations such as min, max, mean count etc

Note : 2 pre-installed R data sets namely mtcars and iris.

```
> library(dplyr)

> data("mtcars")
> data('iris')

> mydata <- mtcars
#read data
> head(mydata)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
#creating a local dataframe.

Local data frame are easier to read

> mynewdata <- tbl_df(mydata)

> myirisdata <- tbl_df(iris)

#now data will be in tabular structure

> mynewdata
```

```
Source: local data frame [32 x 11]

     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1   21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
2   21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
3   22.8     4 108.0    93  3.85 2.320 18.61     1     1     4     1
4   21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
5   18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
6   18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
7   14.3     8 360.0   245  3.21 3.570 15.84     0     0     3     4
8   24.4     4 146.7    62  3.69 3.190 20.00     1     0     4     2
9   22.8     4 140.8    95  3.92 3.150 22.90     1     0     4     2
10  19.2     6 167.6   123  3.92 3.440 18.30     1     0     4     4
..   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
```

> myirisdata	

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          (dbl)       (dbl)        (dbl)       (dbl)  (fctr)
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
..          ...         ...          ...         ...     ...
```

#use filter to filter data with required condition

> filter(mynewdata, cyl > 4 & gear > 4)

```
Source: local data frame [3 x 11]

    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1  15.8     8   351   264  4.22  3.17  14.5     0     1     5     4
2  19.7     6   145   175  3.62  2.77  15.5     0     1     5     6
3  15.0     8   301   335  3.54  3.57  14.6     0     1     5     8
```

> filter(mynewdata, cyl > 4)

```
Source: local data frame [21 x 11]

     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1   21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
2   21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
3   21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
4   18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
5   18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
6   14.3     8 360.0   245  3.21 3.570 15.84     0     0     3     4
7   19.2     6 167.6   123  3.92 3.440 18.30     1     0     4     4
8   17.8     6 167.6   123  3.92 3.440 18.90     1     0     4     4
9   16.4     8 275.8   180  3.07 4.070 17.40     0     0     3     3
10  17.3     8 275.8   180  3.07 3.730 17.60     0     0     3     3
..   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
```

> filter(myirisdata, Species %in% c('setosa', 'virginica'))

```
Source: local data frame [100 x 5]

   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          (dbl)       (dbl)        (dbl)       (dbl)  (fctr)
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
..          ...         ...          ...         ...     ...
```

#use select to pick columns by name

> select(mynewdata, cyl,mpg,hp)

```
      cyl   mpg    hp
    (dbl) (dbl) (dbl)
1       6  21.0   110
2       6  21.0   110
3       4  22.8    93
4       6  21.4   110
5       8  18.7   175
6       6  18.1   105
7       8  14.3   245
8       4  24.4    62
9       4  22.8    95
10      6  19.2   123
..    ...   ...   ...
```

#here you can use (-) to hide columns

> select(mynewdata, -cyl, -mpg)

	disp (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (dbl)	carb (dbl)
1	160.0	110	3.90	2.620	16.46	0	1	4	4
2	160.0	110	3.90	2.875	17.02	0	1	4	4
3	108.0	93	3.85	2.320	18.61	1	1	4	1
4	258.0	110	3.08	3.215	19.44	1	0	3	1
5	360.0	175	3.15	3.440	17.02	0	0	3	2
6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	360.0	245	3.21	3.570	15.84	0	0	3	4
8	146.7	62	3.69	3.190	20.00	1	0	4	2
9	140.8	95	3.92	3.150	22.90	1	0	4	2
10	167.6	123	3.92	3.440	18.30	1	0	4	4
..	...	...	...	...	...	...	...	...	...

#hide a range of columns

> select(mynewdata, -c(cyl,mpg))

	disp (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (dbl)	carb (dbl)
1	160.0	110	3.90	2.620	16.46	0	1	4	4
2	160.0	110	3.90	2.875	17.02	0	1	4	4
3	108.0	93	3.85	2.320	18.61	1	1	4	1
4	258.0	110	3.08	3.215	19.44	1	0	3	1
5	360.0	175	3.15	3.440	17.02	0	0	3	2
6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	360.0	245	3.21	3.570	15.84	0	0	3	4
8	146.7	62	3.69	3.190	20.00	1	0	4	2
9	140.8	95	3.92	3.150	22.90	1	0	4	2
10	167.6	123	3.92	3.440	18.30	1	0	4	4
..	...	...	...	...	...	...	...	...	...

#select series of columns

> select(mynewdata, cyl:gear)

	cyl (dbl)	disp (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (dbl)
1	6	160.0	110	3.90	2.620	16.46	0	1	4
2	6	160.0	110	3.90	2.875	17.02	0	1	4
3	4	108.0	93	3.85	2.320	18.61	1	1	4
4	6	258.0	110	3.08	3.215	19.44	1	0	3
5	8	360.0	175	3.15	3.440	17.02	0	0	3
6	6	225.0	105	2.76	3.460	20.22	1	0	3
7	8	360.0	245	3.21	3.570	15.84	0	0	3
8	4	146.7	62	3.69	3.190	20.00	1	0	4
9	4	140.8	95	3.92	3.150	22.90	1	0	4
10	6	167.6	123	3.92	3.440	18.30	1	0	4
..	...	...	...	...	...	...	...	...	...

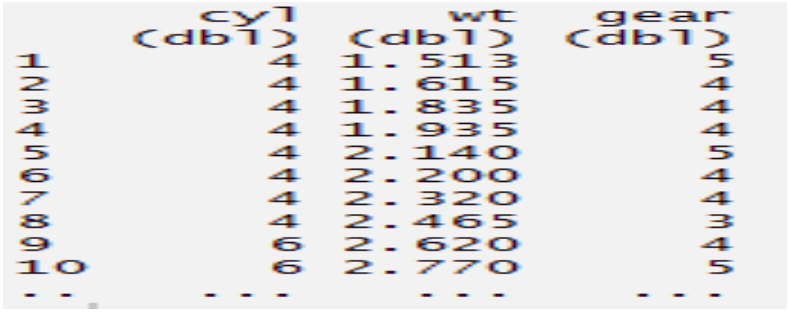
#chaining or pipelining - a way to perform multiple operations #in one line

> mynewdata %>% select(cyl, wt, gear)%>% filter(wt > 2)

	cyl (dbl)	wt (dbl)	gear (dbl)
1	6	2.620	4
2	6	2.875	4
3	4	2.320	4
4	6	3.215	3
5	8	3.440	3
6	6	3.460	3
7	8	3.570	3
8	4	3.190	4
9	4	3.150	4
10	6	3.440	4
..	...	...	...

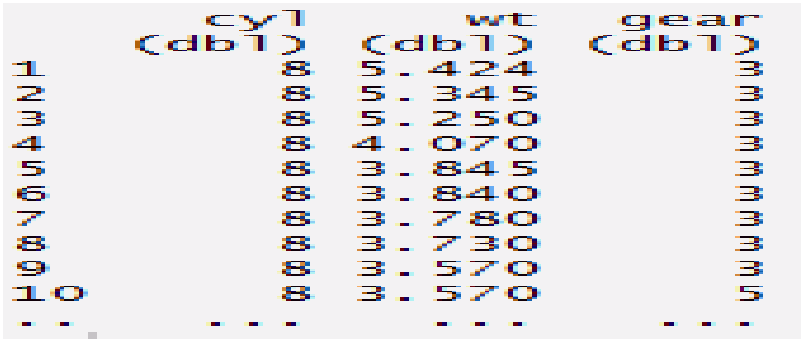
#arrange can be used to reorder rows

> mynewdata%>% select(cyl, wt, gear)%>% arrange(wt)



	cyl	wt	gear
	(dbl)	(dbl)	(dbl)
1	4	1.513	5
2	4	1.615	4
3	4	1.835	4
4	4	1.935	4
5	4	2.140	5
6	4	2.200	4
7	4	2.320	4
8	4	2.465	3
9	6	2.620	4
10	6	2.770	5
..	...	...	...

> mynewdata%>% select(cyl, wt, gear)%>% arrange(desc(wt))

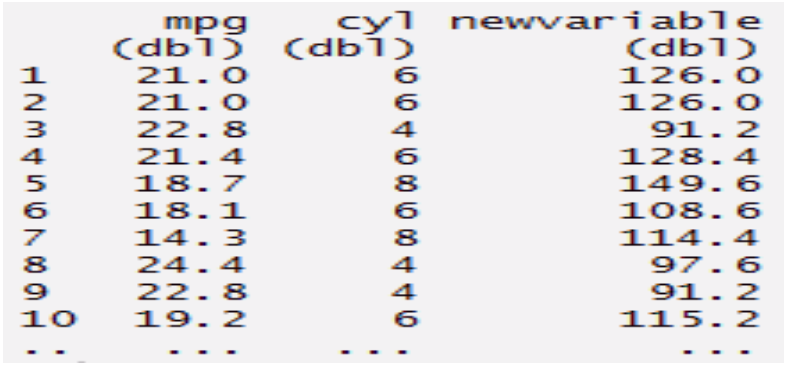


	cyl	wt	gear
	(dbl)	(dbl)	(dbl)
1	8	5.424	3
2	8	5.345	3
3	8	5.250	3
4	8	4.070	3
5	8	3.845	3
6	8	3.840	3
7	8	3.780	3
8	8	3.730	3
9	8	3.570	3
10	8	3.570	5
..	...	...	...

#mutate - create new variables

> mynewdata %>% select(mpg, cyl)%>% mutate(newvariable = mpg*cyl)

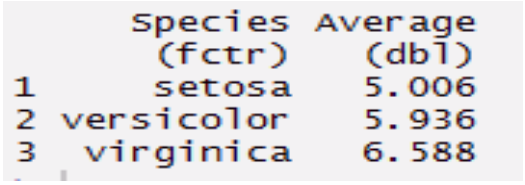
> newvariable <- mynewdata %>% mutate(newvariable = mpg*cyl)



	mpg	cyl	newvariable
	(dbl)	(dbl)	(dbl)
1	21.0	6	126.0
2	21.0	6	126.0
3	22.8	4	91.2
4	21.4	6	128.4
5	18.7	8	149.6
6	18.1	6	108.6
7	14.3	8	114.4
8	24.4	4	97.6
9	22.8	4	91.2
10	19.2	6	115.2
..	...	...	...

#summarise - this is used to find insights from data

> myirisdata%>% group_by(Species)%>% summarise(Average = mean(Sepal.Length, na.rm = TRUE))



	Species	Average
	(fctr)	(dbl)
1	setosa	5.006
2	versicolor	5.936
3	virginica	6.588

#summarise each

> myirisdata%>% group_by(Species)%>% summarise_each(funs(mean, n()), Sepal.Length, Sepal.Width)

Species	Sepal.Length_mean	Sepal.Width_mean	Sepal.Length_n	Sepal.Width_n
(fctr)	(dbl)	(dbl)	(int)	(int)
1 setosa	5.006	3.428	50	50
2 versicolor	5.936	2.770	50	50
3 virginica	6.588	2.974	50	50

rename the variables using rename command

> mynewdata %>% rename(miles = mpg)

	miles	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)
1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
..	...	...	...	...	...	...	...	...	...	...	...

data.table Package

This package allows to perform faster manipulation in a data set. A data table has 3 parts namely DT[i,j,by]. We can tell R to subset the rows using ‘i’, to calculate ‘j’ which is grouped by ‘by’. Most of the times, ‘by’ relates to categorical variable.

#load data

> data("airquality")
> mydata <- airquality
> head(airquality,6)

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

#load package

> library(data.table)

> mydata <- data.table(mydata)
> mydata

```
      Ozone Solar.R Wind Temp Month Day
  1:     41     190  7.4   67     5   1
  2:     36     118  8.0   72     5   2
  3:     12     149 12.6   74     5   3
  4:     18     313 11.5   62     5   4
  5:     NA      NA 14.3   56     5   5
 ---
149:     30     193  6.9   70     9  26
150:     NA     145 13.2   77     9  27
151:     14     191 14.3   75     9  28
152:     18     131  8.0   76     9  29
153:     20     223 11.5   68     9  30
```

> myiris <- data.table(myiris)

> myiris

```
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
  1:          5.1         3.5          1.4         0.2    setosa
  2:          4.9         3.0          1.4         0.2    setosa
  3:          4.7         3.2          1.3         0.2    setosa
  4:          4.6         3.1          1.5         0.2    setosa
  5:          5.0         3.6          1.4         0.2    setosa
 ---
146:          6.7         3.0          5.2         2.3 virginica
147:          6.3         2.5          5.0         1.9 virginica
148:          6.5         3.0          5.2         2.0 virginica
149:          6.2         3.4          5.4         2.3 virginica
150:          5.9         3.0          5.1         1.8 virginica
```

#subset rows - select 2nd to 4th row

> mydata[2:4,]

```
   Ozone Solar.R Wind Temp Month Day
1:    36     118  8.0   72     5   2
2:    12     149 12.6   74     5   3
3:    18     313 11.5   62     5   4
```

#select columns with particular values

> myiris[Species == 'setosa']

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1:	5.1	3.5	1.4	0.2	setosa
2:	4.9	3.0	1.4	0.2	setosa
3:	4.7	3.2	1.3	0.2	setosa
4:	4.6	3.1	1.5	0.2	setosa
5:	5.0	3.6	1.4	0.2	setosa
6:	5.4	3.9	1.7	0.4	setosa
7:	4.6	3.4	1.4	0.3	setosa
8:	5.0	3.4	1.5	0.2	setosa
9:	4.4	2.9	1.4	0.2	setosa
10:	4.9	3.1	1.5	0.1	setosa
11:	5.4	3.7	1.5	0.2	setosa

#select columns with multiple values. This will give you columns with Setosa #and virginica species

> myiris[Species %in% c('setosa', 'virginica')]

#select columns. Returns a vector

> mydata[,Temp]

```
  [1] 67 72 74 62 56 66 65 59 61 69 74 69 66 68 58 64 66 57 68 62 59 73 61 61 57 58 57
 [28] 67 81 79 76 78 74 67 84 85 79 82 87 90 87 93 92 82 80 79 77 72 65 73 76 77 76 76
 [55] 76 75 78 73 80 77 83 84 85 81 84 83 83 88 92 92 89 82 73 81 91 80 81 82 84 87 85
 [82] 74 81 82 86 85 82 86 88 86 83 81 81 81 82 86 85 87 89 90 90 92 86 86 82 80 79 77
[109] 79 76 78 78 77 72 75 79 81 86 88 97 94 96 94 91 92 93 93 87 84 80 78 75 73 81 76
[136] 77 71 71 78 67 76 68 82 64 71 81 69 63 70 77 75 76 68
```

> mydata[,.(Temp,Month)]

	Temp	Month
1:	67	5
2:	72	5
3:	74	5
4:	62	5
5:	56	5

149:	70	9
150:	77	9
151:	75	9
152:	76	9
153:	68	9

#returns sum of selected column

> mydata[,sum(Ozone, na.rm = TRUE)]

[1]4887

#returns sum and standard deviation

> mydata[,.(sum(Ozone, na.rm = TRUE), sd(Ozone, na.rm = TRUE))]

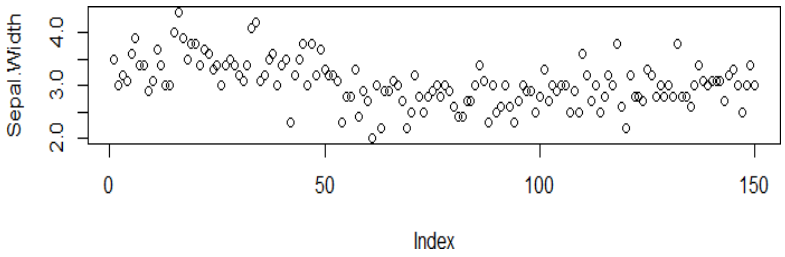
```
     V1       V2
1: 4887 32.98788
```

#print and plot

> myiris[,{print(Sepal.Length)

> plot(Sepal.Width) NULL}]

```
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5.1
 [21] 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9 4.4 5.1
 [41] 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2
 [61] 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7
 [81] 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
[101] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0
[121] 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9
[141] 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```



#grouping by a variable

> myiris[,.(sepalsum = sum(Sepal.Length)), by=Species

```
      Species sepalsum
1:     setosa    250.3
2: versicolor    296.8
3:  virginica    329.4
```

#select a column for computation, hence need to set the key on column

> setkey(myiris, Species)

#selects all the rows associated with this data point

> myiris['setosa']
> myiris[c('setosa', 'virginica')]

ggplot2 Package

ggplot offers a whole new world of colors and patterns. Plotting 3 graphs: Scatter Plot, Bar Plot, Histogram. ggplot is enriched with customized features to make visualization better. It becomes even more powerful when grouped with other packages like cowplot, gridExtra.

Scatter Plot :

A Scatter Plot is a graph in which the values of two variables are plotted along two axes, the pattern of the resulting points revealing any correlation present.

With scatter plots we can explain how the variables relate to each other. Which is defined as correlation. Positive, Negative, and None (no correlation) are the three types of correlation.

Limitations of a Scatter Diagram

Below are the few limitations of a scatter diagram:

• With Scatter diagrams we cannot get the exact extent of correlation.

• Quantitative measure of the relationship between the variable cannot be viewed. Only shows the quantitative expression.

• The relationship can only show for two variables.

Advantages of a Scatter Diagram

Below are the few advantages of a scatter diagram:

• Relationship between two variables can be viewed.

• For non-linear pattern, this is the best method.

• Maximum and minimum value, can be easily determined.

• Observation and reading is easy to understand

• Plotting the diagram is very simple.

Bar Plot

A barplot (or barchart) is one of the most common type of graphic. It shows the relationship between a numeric variable and a categoric variable.

Bar Plot are classified into four types of graphs - bar graph or bar chart, line graph, pie chart, and diagram.

Limitations of Bar Plot:

When we try to display changes in speeds such as acceleration, Bar graphs wont help us.

Advantages of Bar plot:

• Bar charts are easy to understand and interpret.

• Relationship between size and value helps for in easy comparison.

• They're simple to create.

• They can help in presenting very large or very small values easily.

Histogram

A histogram represents the frequency distribution of continuous variables. while, a bar

graph is a diagrammatic comparison of discrete variables.

Histogram presents numerical data whereas bar graph shows categorical data. The histogram is drawn in such a way that there is no gap between the bars.

Limitations of Histogram:

A histogram can present data that is misleading as it has many bars.

Only two sets of data are used, but to analyze certain types of statistical data, more than two sets of data are necessary

Advantages of Histogram:

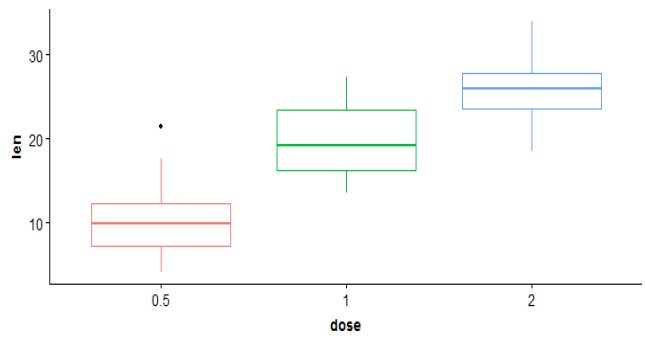
Histogram helps to identify different data, the frequency of the data occurring in the dataset and categories which are difficult to interpret in a tabular form. It helps to visualize the distribution of the data.

```
> library(ggplot2)
> library(gridExtra)
> df <- ToothGrowth
> df$dose <- as.factor(df$dose)
> head(df)
```

	len	supp	dose
1	4.2	VC	0.5
2	11.5	VC	0.5
3	7.3	VC	0.5
4	5.8	VC	0.5
5	6.4	VC	0.5
6	10.0	VC	0.5

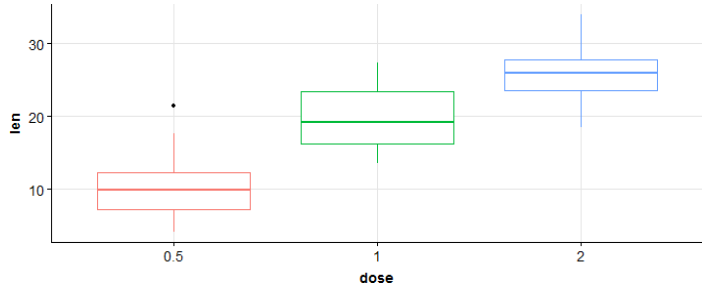
BOX PLOT

```
> bp <- ggplot(df, aes(x = dose, y = len, color = dose)) + geom_boxplot() +
theme(legend.position = 'none')

> bp
```



#add gridlines

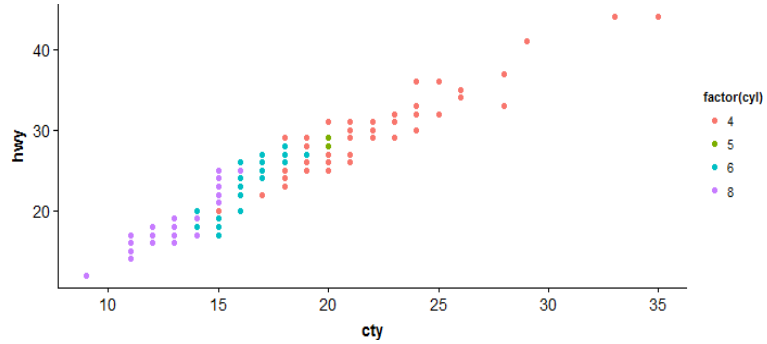
> bp + background_grid(major = "xy", minor = 'none')



SCATTER PLOT

> sp <- ggplot(mpg, aes(x = cty, y = hwy, color = factor(cyl)))+geom_point(size = 2.5)

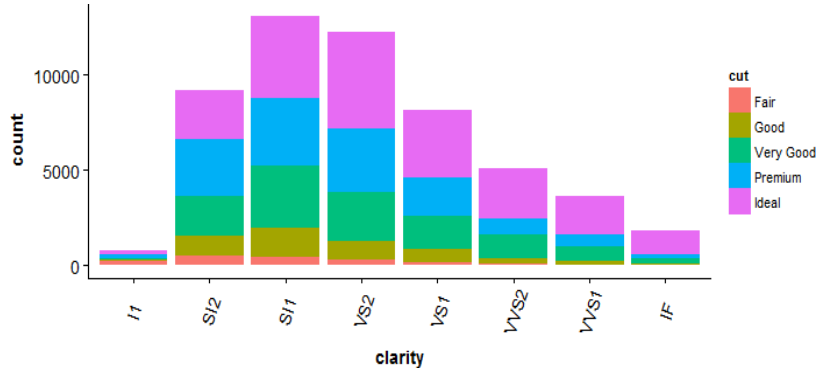
> sp



BAR PLOT

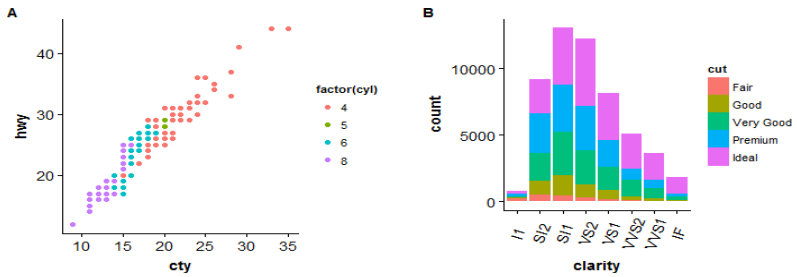
> bp <- ggplot(diamonds, aes(clarity, fill = cut)) + geom_bar() +theme(axis.text.x = element_text(angle = 70, vjust = 0.5))

> bp



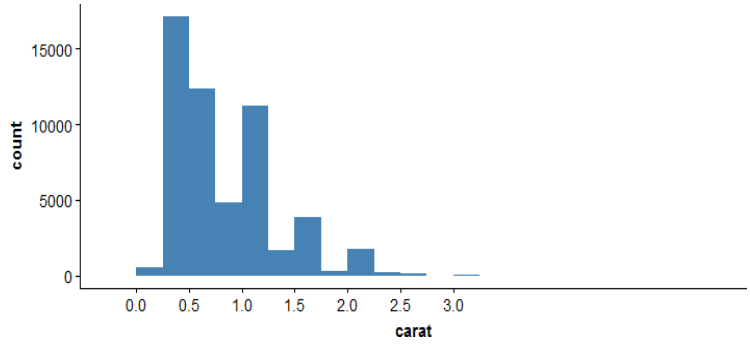
#compare two plots

> plot_grid(sp, bp, labels = c("A","B"), ncol = 2, nrow = 1)



#histogram

> ggplot(diamonds, aes(x = carat)) + geom_histogram(binwidth = 0.25, fill = 'steelblue')+scale_x_continuous(breaks=seq(0,3, by=0.5))



reshape2 Package

As the name suggests, this package is useful in reshaping data. The data come in many forms. Hence, we are required to shape it according to our need. Usually, the process of reshaping data in R is tedious. R base functions consist of 'Aggregation' option using which data can be reduced and rearranged into smaller forms, but with reduction in amount of information. Aggregation includes tapply, by and aggregate base functions. The reshape package overcomes these problems. It has 2 functions namely melt and cast.

melt : This function converts data from wide format to long format. It's a form of restructuring where multiple categorical columns are 'melted' into unique rows.

#create a data

> ID <- c(1,2,3,4,5)

> Names <- c('Joseph','Matrin','Joseph','James','Matrin')

> DateofBirth <- c(1993,1992,1993,1994,1992)

> Subject<- c('Maths','Biology','Science','Psycology','Physics')

> thisdata <- data.frame(ID, Names, DateofBirth, Subject)

> data.table(thisdata)

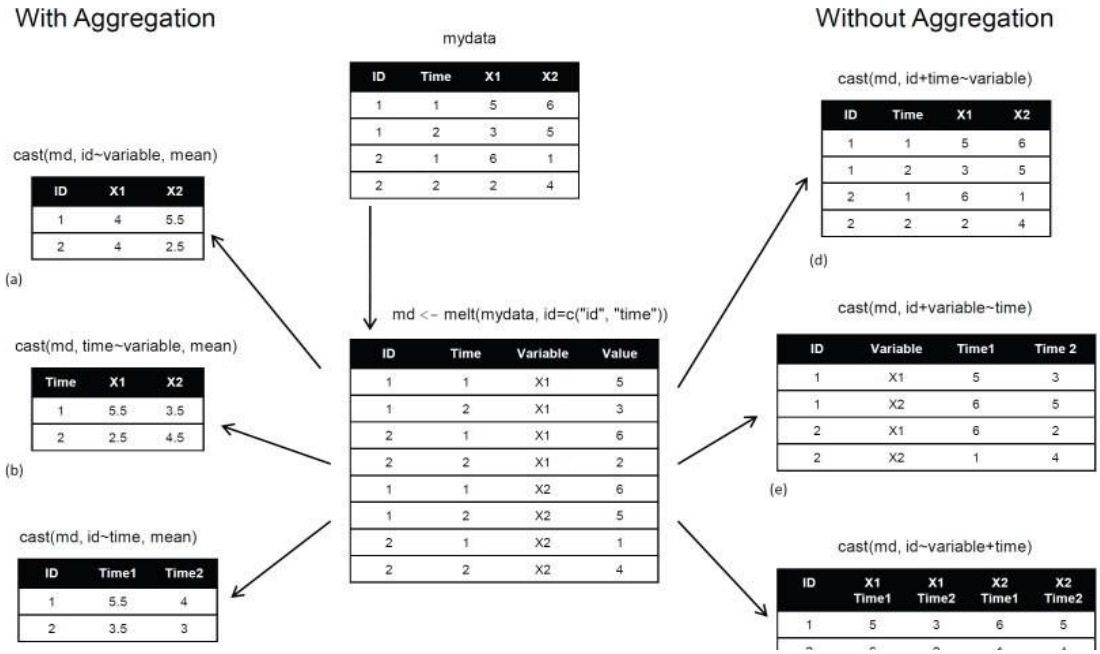
```
   ID  Names DateofBirth   Subject
1:  1 Joseph        1993     Maths
2:  2 Matrin        1992   Biology
3:  3 Joseph        1993   Science
4:  4  James        1994 Psycology
5:  5 Matrin        1992   Physics
```

#load package
> install.packages('reshape2')
> library(reshape2)
#melt
> mt <- melt(thisdata, id=(c('ID','Names')))
> mt

	ID	Names	variable	value
1	1	Joseph	DateofBirth	1993
2	2	Matrin	DateofBirth	1992
3	3	Joseph	DateofBirth	1993
4	4	James	DateofBirth	1994
5	5	Matrin	DateofBirth	1992
6	1	Joseph	Subject	Maths
7	2	Matrin	Subject	Biology
8	3	Joseph	Subject	Science
9	4	James	Subject	Psycology
10	5	Matrin	Subject	Physics

cast : This function converts data from long format to wide format. It starts with melted data and reshapes into long format. It's just the reverse of melt function. It has two functions namely, dcast and acast. dcast returns a data frame as output. acast returns a vector/matrix/array as the output.
> mcast <- dcast(mt, DateofBirth + Subject ~ variable)
> mcast

	DateofBirth	Subject	DateofBirth	Subject
1	1992	Biology	1992	Biology
2	1992	Physics	1992	Physics
3	1993	Maths	1993	Maths
4	1993	Science	1993	Science
5	1994	Psycology	1994	Psycology



tidyr Package

This package can make the data look 'tidy'. It has 4 major functions to accomplish this task. The 4 functions are:

gather() – it ‘gathers’ multiple columns. Then, it converts them into key:value pairs.
This function will transform wide from of data to long form. You can use it as in alternative to ‘melt’ in reshape package.
spread() – It does reverse of gather. It takes a key:value pair and converts it into separate columns.
separate() – It splits a column into multiple columns.

unite() – It does reverse of separate. It unites multiple columns into single column

```
#load package
> library(tidyr)
#create a dummy data set
> names <- c('A','B','C','D','E','A','B')
> weight <- c(55,49,76,71,65,44,34)
> age <- c(21,20,25,29,33,32,38)

> Class <- c('Maths','Science','Social','Physics','Biology','Economics','Accounts')

#create data frame
> tdata <- data.frame(names, age, weight, Class)

> tdata
```

	names	age	weight	Class
1	A	21	55	Maths
2	B	20	49	Science
3	C	25	76	Social
4	D	29	71	Physics
5	E	33	65	Biology
6	A	32	44	Economics
7	B	38	34	Accounts

#using gather function

```
> long_t <- tdata %>% gather(Key, Value, weight:Class)
> long_t
```

	names	age	Key	Value
1	A	21	weight	55
2	B	20	weight	49
3	C	25	weight	76
4	D	29	weight	71
5	E	33	weight	65
6	A	32	weight	44
7	B	38	weight	34
8	A	21	Class	Maths
9	B	20	Class	Science
10	C	25	Class	Social
11	D	29	Class	Physics
12	E	33	Class	Biology
13	A	32	Class	Economics
14	B	38	Class	Accounts

Separate Command

#create a data set

Time <- c("27/01/2015 15:44","23/02/2015 23:24", "31/03/2015 19:15", "20/01/2015 20:52", "23/02/2015 07:46", "31/01/2015 01:55")

#build a data frame

> d_set <- data.frame(Humidity, Rain, Time)

#using separate function we can separate date, month, year

> separate_d <- d_set %>% separate(Time, c('Date', 'Month','Year'))

> separate_d

	Humidity	Rain	Date	Month	Year
1	37.79	0.9713604	27	01	2015
2	42.34	1.1096972	23	02	2015
3	52.16	1.0644759	31	03	2015
4	44.57	0.9531834	20	01	2015
5	43.83	0.9887885	23	02	2015
6	44.59	0.9396761	31	01	2015

Unite Command

#using unite function - reverse of separate

> unite_d <- separate_d%>% unite(Time, c(Date, Month, Year), sep = "/")

> unite_d

	Humidity	Rain	Time
1	37.79	0.9713604	27/01/2015
2	42.34	1.1096972	23/02/2015
3	52.16	1.0644759	31/03/2015
4	44.57	0.9531834	20/01/2015
5	43.83	0.9887885	23/02/2015
6	44.59	0.9396761	31/01/2015

Spread Function (reverse of gather command)

#using spread function - reverse of gather

> wide_t <- long_t %>% spread(Key, Value)

> wide_t

	names	age	weight	Class
1	A	21	55	Maths
2	A	32	44	Economics
3	B	20	49	Science
4	B	38	34	Accounts
5	C	25	76	Social
6	D	29	71	Physics
7	E	33	65	Biology

readr Package

'readr' helps in reading various forms of data into R. With 10x faster speed. Here, characters are never converted to factors. This package can replace the traditional read.csv() and read.table() base R functions. It helps in reading the following data:

Delimited files with read_delim(), read_csv(), read_tsv(), and read_csv2().
Fixed width files with read_fwf(), and read_table().
Web log files with read_log()

If the data loading time is more than 5 seconds, this function will show you a progress bar too.

> install.packages('readr')	> library(readr)
	> read_csv('test.csv',col_names = TRUE)
specify the data type of every column loaded in data	> read_csv("iris.csv", col_types = list(Sepal.Length = col_double(), Sepal.Width = col_double(), Petal.Length = col_double(), Petal.Width = col_double(), Species = col_factor(c("setosa", "versicolor", "virginica"))))
choose to omit unimportant columns	> read_csv("iris.csv", col_types = list(Species = col_factor (c("setosa", "versicolor", "virginica")))

Lubridate Package

Lubridate package reduces the pain of working of data time variable in R. The inbuilt function of this package offers a nice way to make easy parsing in dates and times. This package is frequently used with data comprising of timely data.

> install.packages('lubridate')
> library(lubridate)

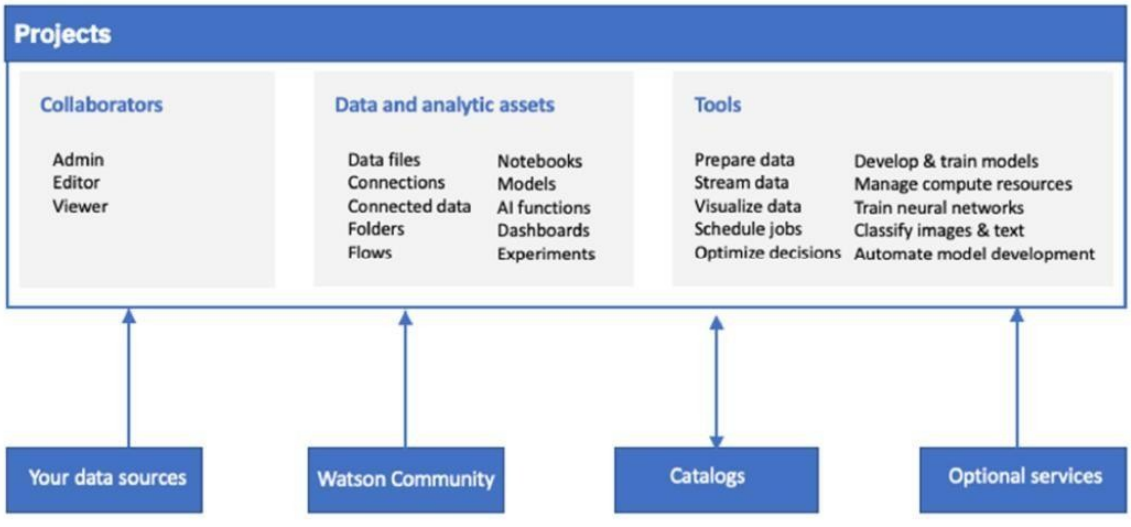
#current date and time	> now()	[1] "2015-12-11 13:23:48 IST"
#assigning current date and time to variable n_time	> n_time <- now()	
#using update function	>n_update<-update(n_time, year = 2013, month = 10) > n_update	[1] "2013-10-11 13:24:28 IST"

#add days, months, year, seconds	> d_time <- now() > d_time + ddays(1)	[1] "2015-12-12 13:24:54 IST"
	> d_time + dweeks(2)	[1] "2015-12-12 13:24:54 IST"
	> d_time + dyears(3)	[1] "2018-12-10 13:24:54 IST"
	> d_time + dhours(2)	[1] "2015-12-11 15:24:54 IST"
	> d_time + dminutes(50)	[1] "2015-12-11 14:14:54 IST"
	> d_time + dseconds(60)	[1] "2015-12-11 13:25:54 IST"
#extract date,time	> n_time$hour <- hour(now()) > n_time$minute <- minute(now()) > n_time$second <- second(now()) > n_time$month <- month(now()) > n_time$year <- year(now())	
#check the extracted dates in separate columns	> new_data <- data.frame(n_time$hour, n_time$minute, n_time$second, n_time$month, n_time$year) > new_data	n_time.hour n_time.minute n_time.second n_time.month 13 27 41.65723 12

WATSON STUDIO

Watson Studio provides you with the environment and tools to solve your business problems by collaboratively working with data. You can choose the tools you need to analyze and visualize data, to cleanse and shape data, to ingest streaming data, or to create and train machine learning models.

This illustration shows how the architecture of Watson Studio is centered around the project. A project is where you organize your resources and work with data.



Visualizing information in graphical ways can give you insights into your data. By enabling you to look at and explore data from different perspectives, visualizations can help you identify patterns, connections, and relationships within that data as well as understand large amounts of information very quickly.

Create a project -
To create a project :

Click New project on the Watson Studio home page or your My Projects page.

Choose whether to create an empty project or to create a project based on an exported project file or a sample project.

If you chose to create a project from a file or a sample, upload a project file or select a sample project. See Importing a project.

On the New project screen, add a name and optional description for the

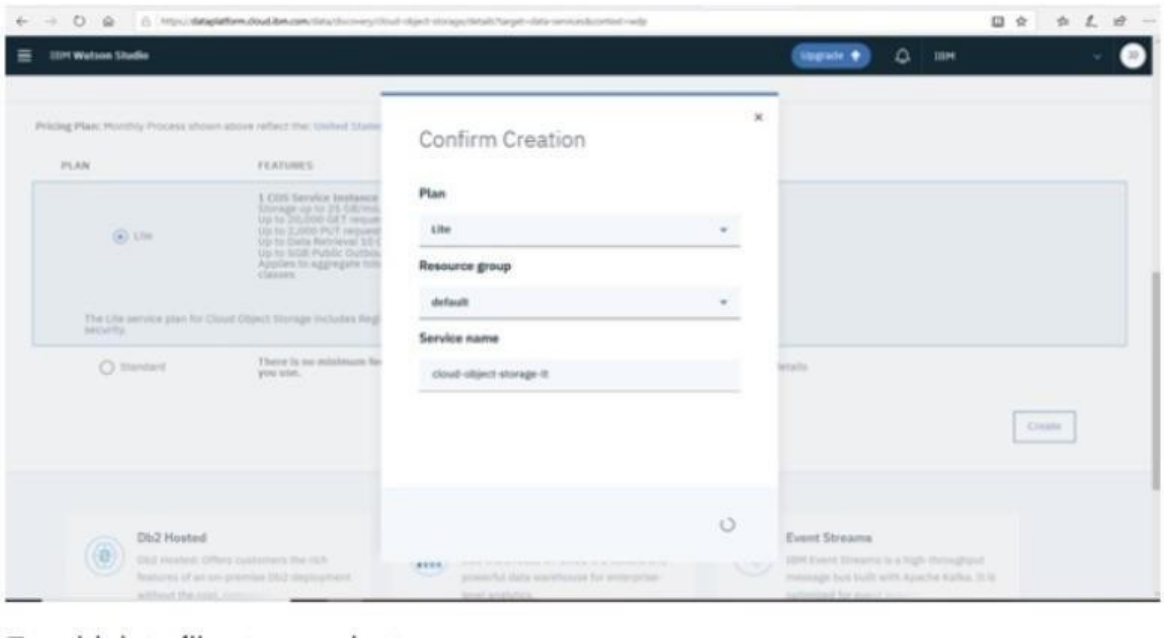
project.

Select the Restrict who can be a collaborator check box to restrict collaborators to members of your organization or integrate with a catalog. The check box is selected by default if you are a member of a catalog. You can’t change this setting after you create the project.

If prompted, choose or add any required services.

Choose an existing object storage service instance or create a new one.

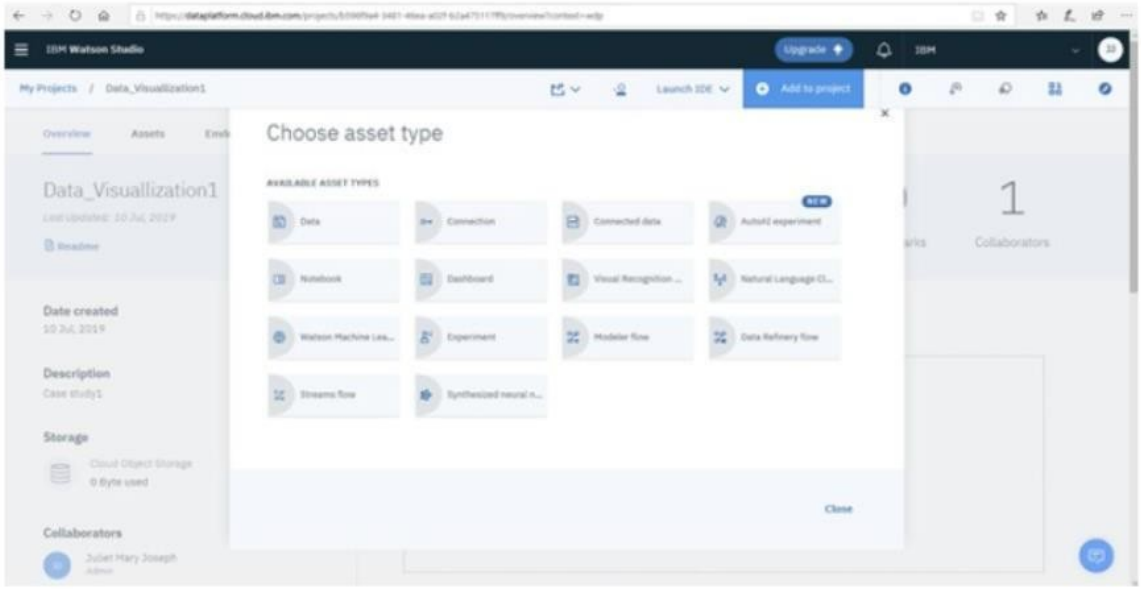
Click Create. You can start adding resources if your project is empty or begin working with the resources you imported.



To add data files to a project:

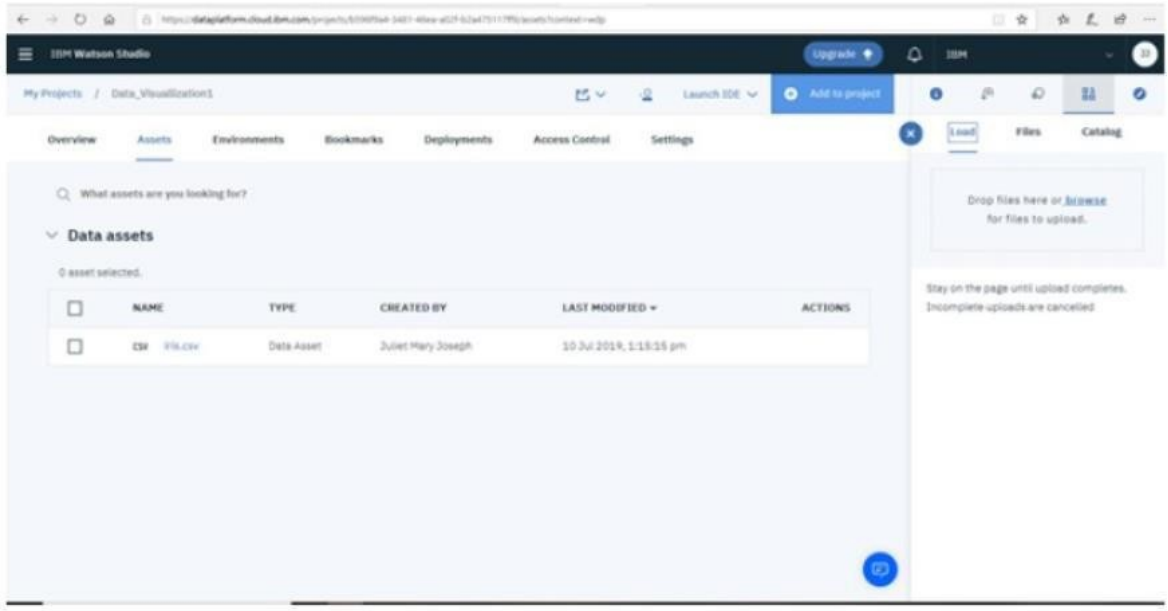
From your project’s Assets page, click Add to project > Data or click the Find and add data icon ().You can also click the Find and add data icon from within a notebook or canvas.

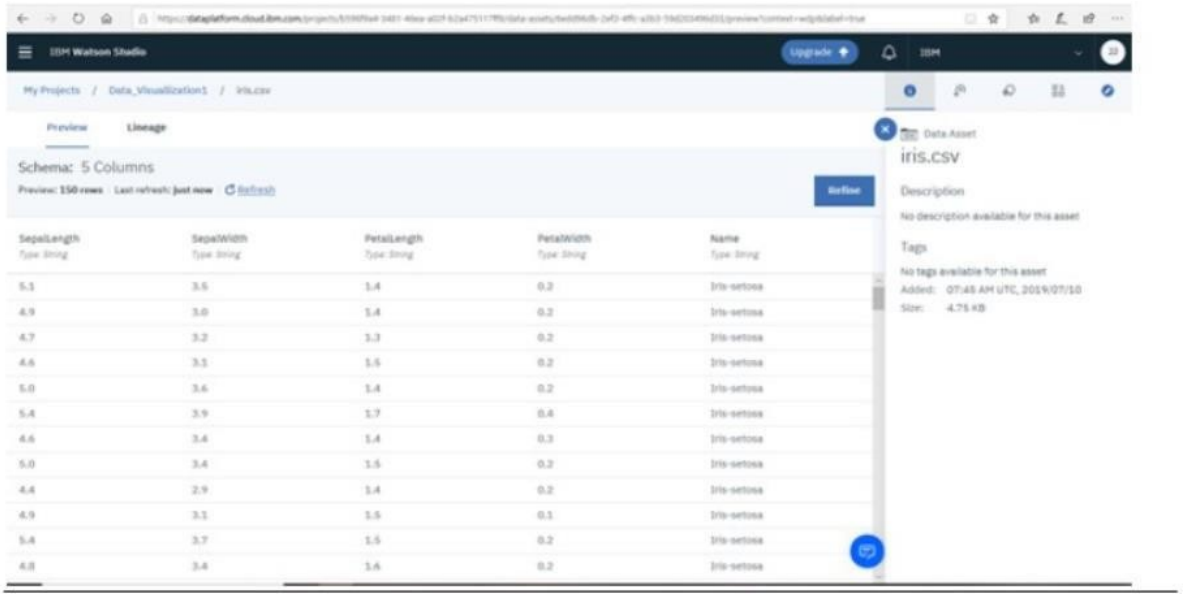
In the Load pane that opens, browse for the files or drag them onto the pane. You must stay on the page until the load is complete. You can cancel an ongoing load process if you want to stop loading a file.



Case Study:

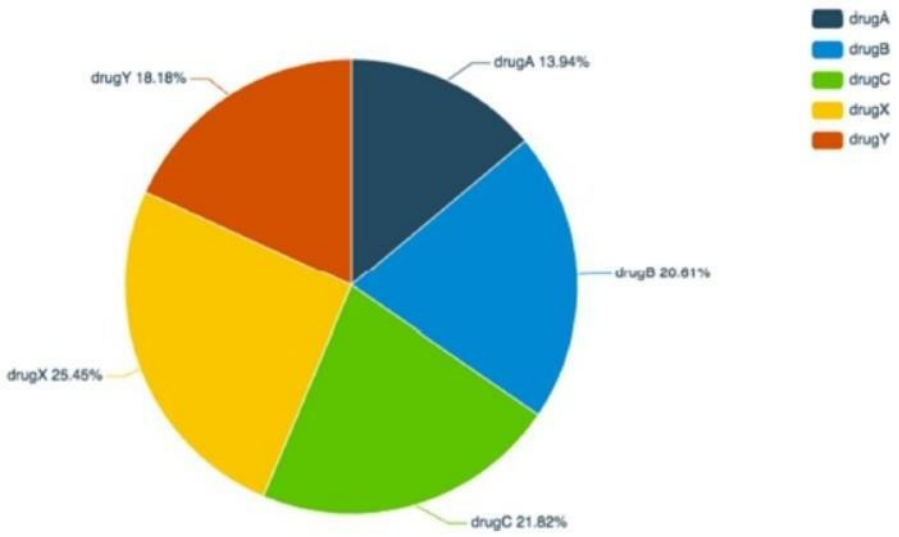
Let us take the Iris Data set to see how we can visualize the data in Watson studio.





Adding Data to Data Refinery

Visualizing information in graphical ways can give you insights into your data. By enabling you to look at and explore data from different perspectives, visualizations can help you identify patterns, connections, and relationships within that data as well as understand large amounts of information very quickly. You can also visualize your data with these same charts in an SPSS Modeler flow. Right-click a node and select **Profile.**



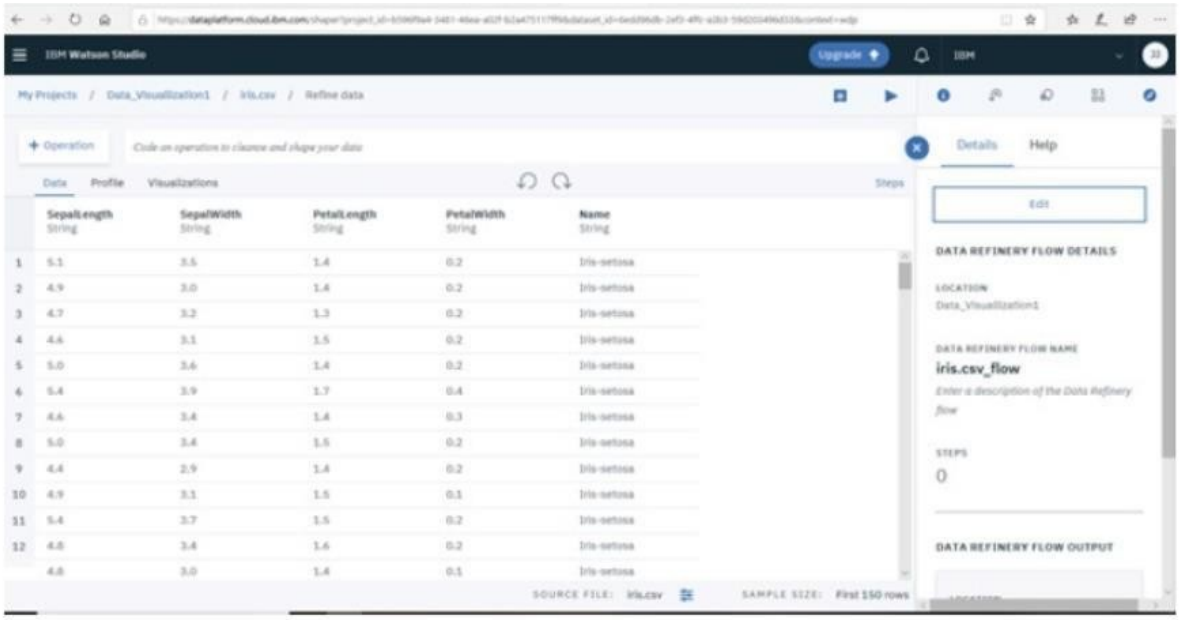
To visualize your data:

From Data Refinery, click the **Visualizations** tab.

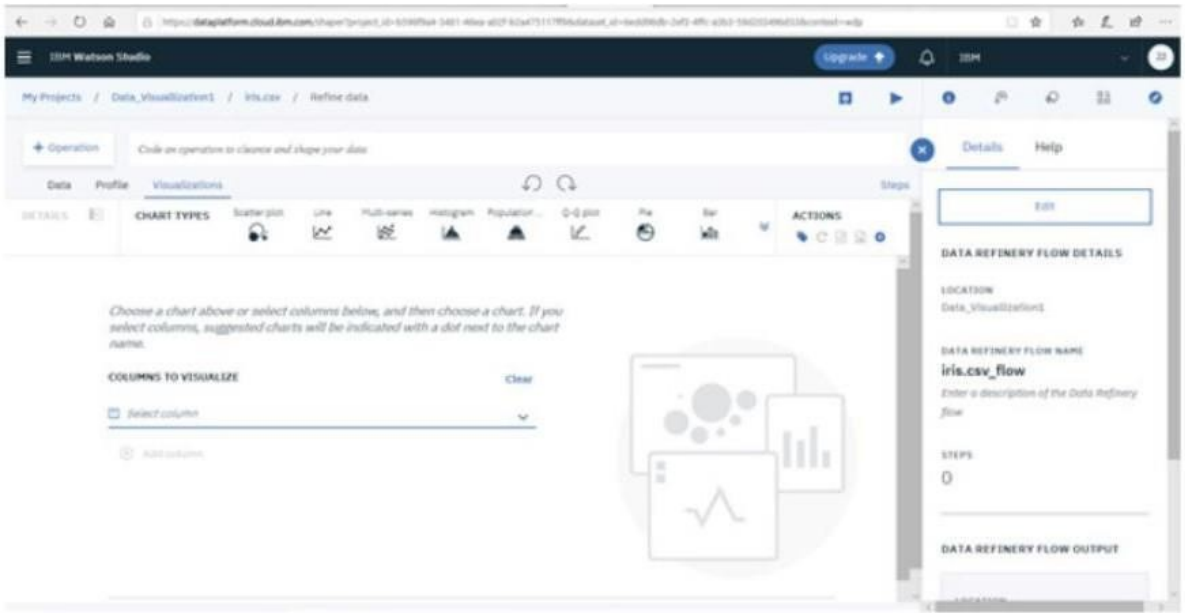
Start with a chart or select columns.

1. Click any of the available charts. Then add columns in the **DETAILS** panel that opens on the left side of the page.
2. Select the columns that you want to work with. Suggested charts will be indicated with a dot next to the chart name. Click a chart to visualize your data.

Click on refine



Click on Visualization tab:



Add the columns by selecting.

UNIT – III

Introduction to Anaconda -

Anaconda is a package manager, an environment manager, and Python distribution that contains a collection of many open source packages.

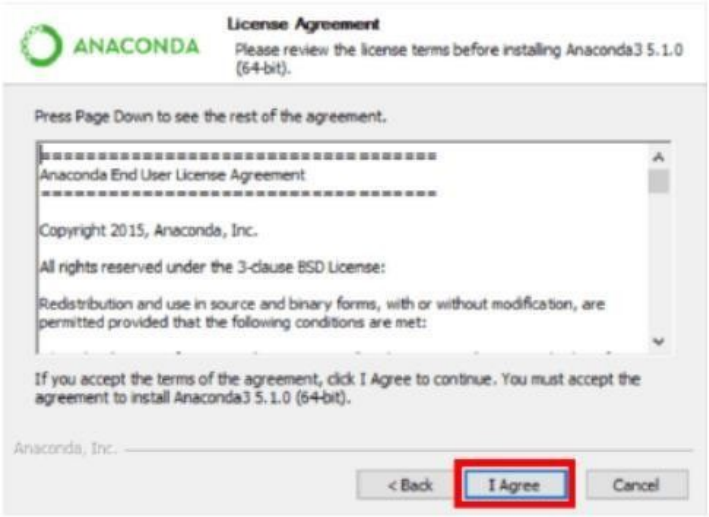
Anaconda Installation -

Go to the Anaconda Website and choose a Python 3.x graphical installer (A) or a Python 2.x graphical installer.

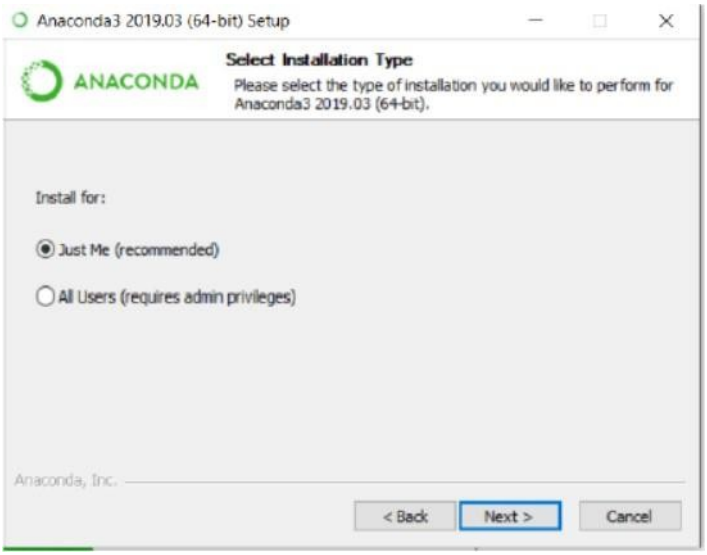
- Locate your download and double click it.

When the screen below appears, click on Next

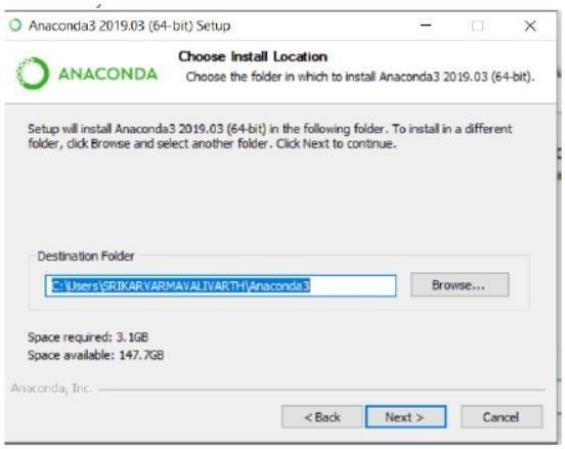
- Read the license agreement and click on I Agree.



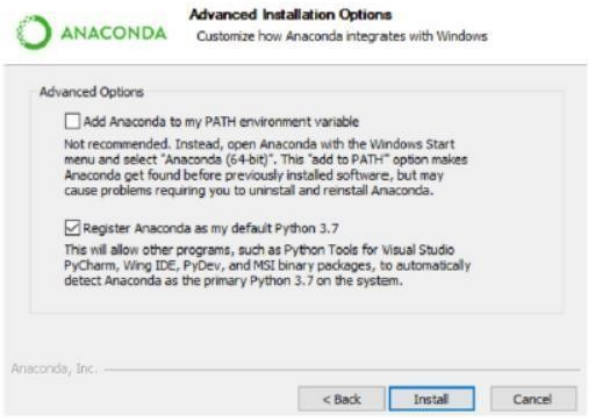
Click on Next.



Note your installation location and then click Next.

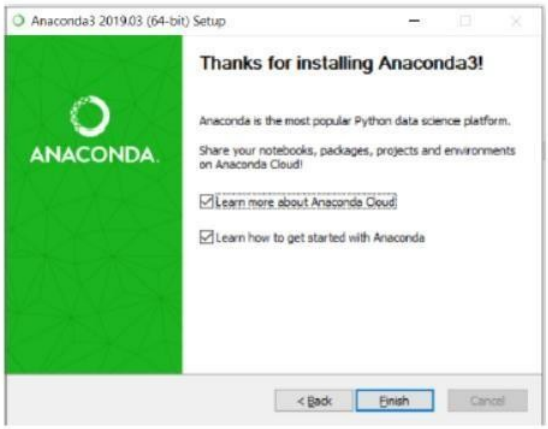


Choose whether to add Anaconda to your PATH environment variable. We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Prompt from the Start Menu.



After that click on next.

Click Finish.



We need to set anaconda path to system environmental variables.

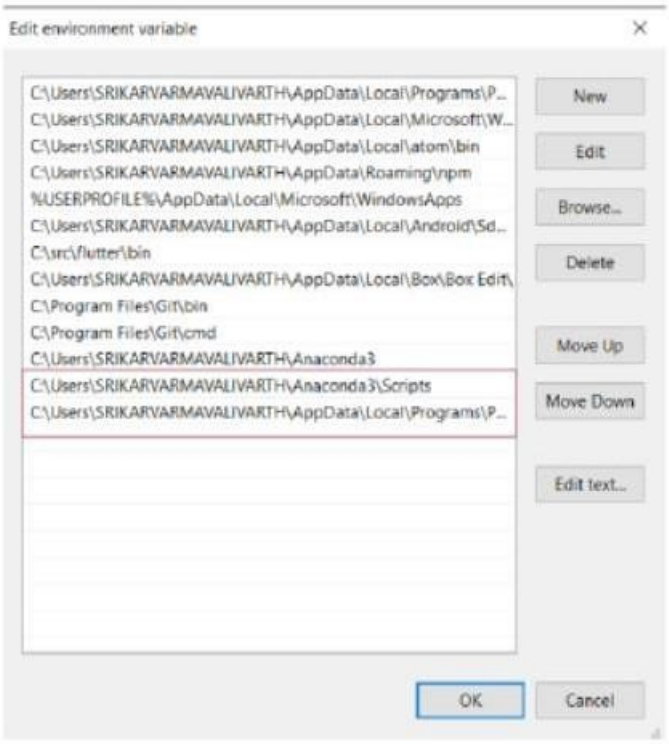
Open a Command Prompt. Check if you already have Anaconda added to your path.
Enter the commands below into your Command Prompt.

Conda –version
Python –version
This is checking if you already have Anaconda added to your path. If you get a command not recognized, then we need to set Anaconda path

If you don't know where your conda and/or python is, open an Anaconda Prompt and type in the following commands. This is telling you where conda and python are located on your computer.

```
(base) C:\Users\SRIKARVARMAVALIVARTH>where conda
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\Library\bin\conda.bat
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\Scripts\conda.exe
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\condabin\conda.bat

(base) C:\Users\SRIKARVARMAVALIVARTH>where python
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\python.exe
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\Programs\Python\Python36\python.exe
```

Add conda and python to your PATH. You can do this by going to your System Environment Variables and adding the output of step 3 (enclosed in the red)



Open a **new Command Prompt.** Try typing conda --version and python --version into the **Command Prompt** to check to see if everything went well.

Conda installation is successful

Introduction to Jupyter Notebook

What is Jupyter

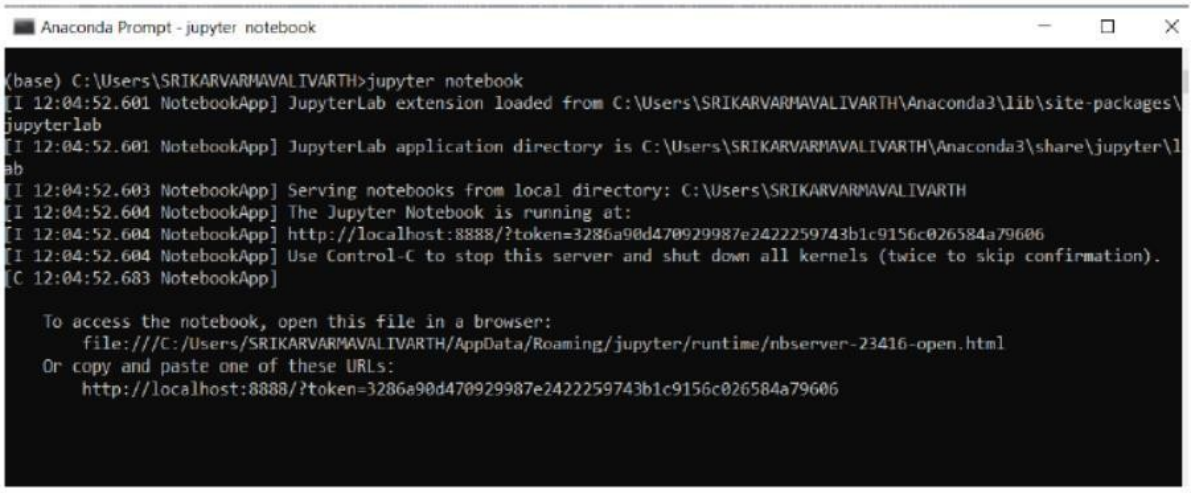
The Jupyter Notebook is an open source web application that you can use to create and share documents that contain live code, equations, visualizations, and text. Jupyter Notebook is maintained by the people at Project Jupyter.

Jupyter Notebooks are a spin-off project from the IPython project, which used to have an IPython Notebook project itself. The name, Jupyter, comes from the core supported programming languages that it supports: Julia, Python, and R. Jupyter ships with the IPython kernel, which allows you to write your programs in Python, but there are currently over 100 other kernels that you can also use.

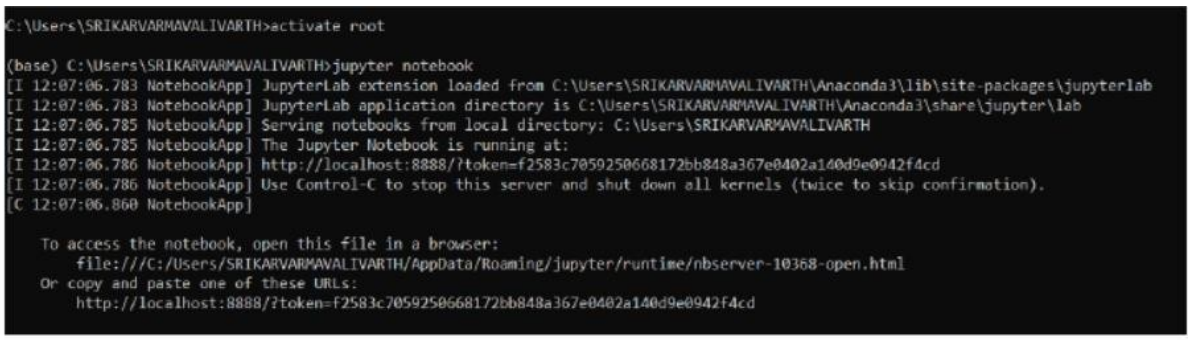
How to access Jupyter Notebook

Installing Anaconda Distribution will also include Jupyter Notebook.

To access the Jupyter Notebook go to anaconda prompt and run below command



Or go to Command Prompt and first activate root before launching jupyter notebook



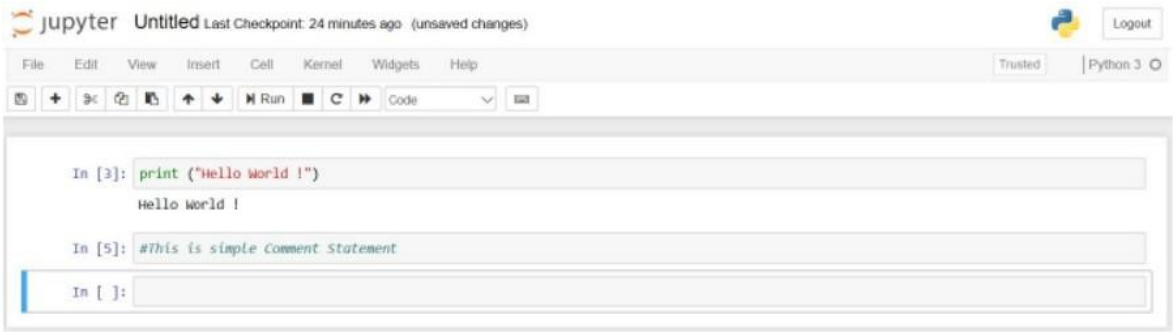
Then you'll see the application opening in the web browser on the following address: http://localhost:8888.

Python Scripting Basics
First Program in Python

```
>>> print ("Hello World !")
Hello World !
>>>
```

A statement or expression is an instruction the computer will run or execute. Perhaps the simplest program you can write is a print statement. When you run the print statement, Python will simply display the value in the parentheses. The value in the parentheses is called the argument.

If you are using a Jupyter notebook, you will see a small rectangle with the statement. This is called a cell. If you select this cell with your mouse, then click the run cell button. The statement will execute. The result will be displayed beneath the cell.



It's customary to comment your code. This tells other people what your code does. You simply put a hash symbol proceeding your comment. When you run the code, Python will ignore the comment.

Data Types

A type is how Python represents different types of data. You can have different types in Python. They can be integers like 11, real numbers like 21.213. They can even be words.

11	int
21.213	float
“Hello Python 101”	str

type(11)	int
type(21.213)	float
type(“Hello Python 101”)	str

The following chart summarizes three data types for the last examples. The first coslumn indicates the expression. The second Column indicates the data type. We can see the actual data type in Python by using the type command. We can have int, which stands for an integer, and float that stands for float, essentially a real number. The type string is a sequence of characters.

Integers can be negative or positive. It should be noted that there is a finite range of integers, but it is quite large. Floats are real numbers; they include the integers but also numbers in between the integers. Consider the numbers between 0 and 1. We can select numbers in between them; these numbers are floats. Similarly, consider the numbers between 0.5 and 0.6. We can select numbers in-between them; these are floats as well.

```
In [6]: float(2)
Out[6]: 2.0

In [7]: int(2.3)
Out[7]: 2

In [8]: int('897')
Out[8]: 897

In [9]: int('ABC')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-9f8eda398053> in <module>
----> 1 int('ABC')

ValueError: invalid literal for int() with base 10: 'ABC'
```

Nothing really changes. If you cast a float to an integer, you must be careful. For example, if you cast the float 1.1 to 1, you will lose some information. If a string contains an integer value, you can convert it to int. If we convert a string that contains a non-integer value, we get an error. You can convert an int to a string or a float to a string.

Boolean is another important type in Python. A Boolean can take on two values. The first value is true, just remember we use an uppercase T. Boolean values can also be false, with an uppercase F. Using the type command on a Boolean value, we obtain the term bool, this is short for Boolean. If we cast a Boolean true to an integer or float, we will get a 1.

If we cast a Boolean false to an integer or float, we get a zero. If you cast a 1 to a Boolean, you get a true. Similarly, if you cast a 0 to a Boolean, you get a false.

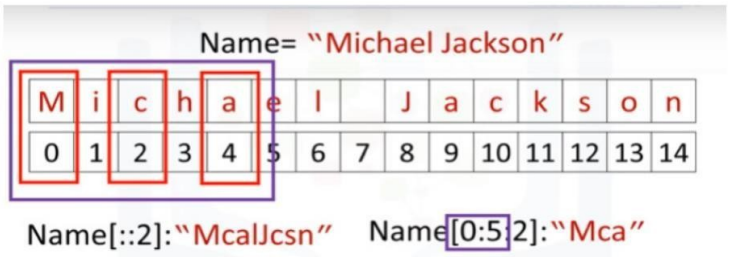
```
In [10]: int(True)
Out[10]: 1

In [11]: int(False)
Out[11]: 0

In [12]: bool(1)
Out[12]: True

In [13]: bool(0)
Out[13]: False
```

String Operations In Python

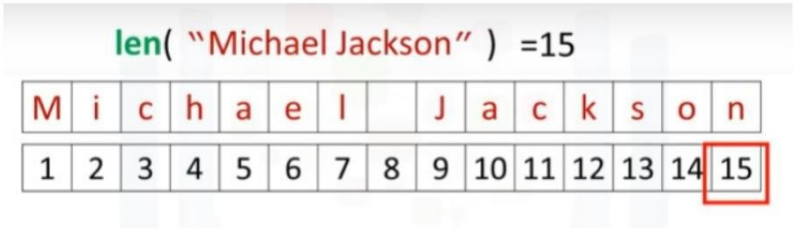
In Python, a string is a sequence of characters. A string is contained within two quotes: You could also use single quotes. A string can be spaces, or digits. A string can also be special characters. We can bind or assign a string to another variable. It is helpful to think of a string as an ordered sequence. Each element in the sequence can be accessed using an index represented by the array of numbers. The first index can be accessed as

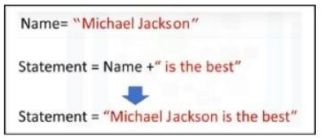
follows. We can access index 6. Moreover, we can access the 13th index. We can also use negative indexing with strings. The last element is given by the index -1. The first element can be obtained by index -15, and so on.

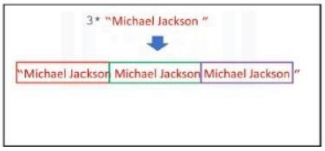
We can bind a string to another variable. It is helpful to think of string as a list or tuple. We can treat the string as a sequence and perform sequence operations. We can also input a string value as follows. The 2 indicates we select every second variable. We can also incorporate slicing.



In this case. we return every second value up to index four. We can use the "Len" command to obtain the length of the string. As there are 15 elements, the result is 15.







We can concatenate or combine strings. We use the addition symbols. The result is a new string that is a combination of both.

We can replicate values of a string. We simply multiply the string by the number of times we would like to replicate it, in this case, three. The result is a new string. The new string consists

of three copies of the original string. This means you cannot change the value of the string, but you can create a new string.

Python COLLECTION (or)Arrays

There are four collection data types in the Python programming language:
Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

List is a collection which is ordered and changeable. Allows duplicate members.

Set is a collection which is unordered and unindexed. No duplicate members.

Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

Tuple:
tuples are expressed as comma-separated elements within parentheses.

```
In [14]: Example_Tuple=(2,4,6,2,24,2345)
In [15]: Example_Tuple_2=("python",32,78.23)
In [16]: type(Example_Tuple_2)
Out[16]: tuple
```

Example_Tuple=(2,4,6,2,24,2345)

Example_Tuple=(2,4,6,2,24,2345)

INDEX	List Elements	negative Index
Example_Tuple[0]	2	Example_Tuple[-6]
Example_Tuple[1]	4	Example_Tuple[-5]
Example_Tuple[2]	6	Example_Tuple[-4]
Example_Tuple[3]	2	Example_Tuple[-3]
Example_Tuple[4]	24	Example_Tuple[-2]
Example_Tuple[5]	2345	Example_Tuple[-1]

```
In [4]: Example_Tuple[0]
Out[4]: 2
In [5]: Example_Tuple[-1]
Out[5]: 2345
In [10]: Example_Tuple[2:5]
Out[10]: (6, 2, 24)
In [8]: Example_Tuple[3:-1]
Out[8]: (2, 24)
```

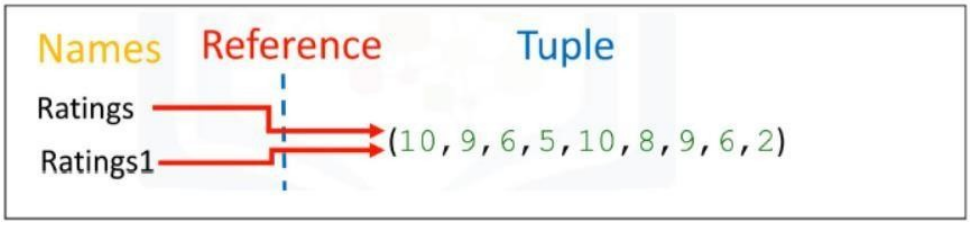
In Python, there are different types: strings, integer, float. They can all be contained in a tuple, but the type of the variable is tuple

Each element of a tuple can be accessed via an index. The element in the tuple can be accessed by the name of the tuple followed by a square bracket with the index number. Use the square brackets for slicing along with the index or indices to obtain value available at that index.

Tuples are immutable, which means we can't change them.

```
In [51]: Ratings=(10,9,6,5,10,8,9,6,2)

In [52]: Ratings1=Ratings

In [53]: print (Ratings1)
         (10, 9, 6, 5, 10, 8, 9, 6, 2)
```

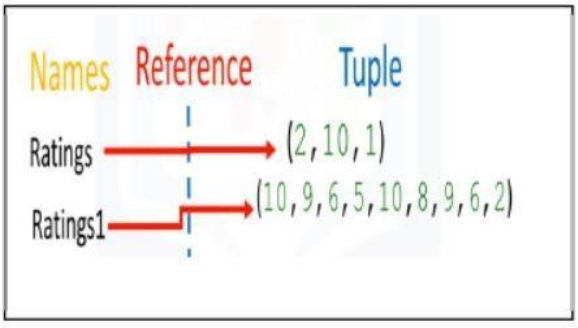
To see why this is important, let's see what happens when we set the variable Ratings 1 to ratings. Each variable does not contain a tuple, but references the same immutable tuple object.



Let's say we want to change the element at index 2. Because tuples are immutable, we can't. Therefore, Ratings 1 will not be affected by a change in Rating because the tuple is immutable i.e., we can't change it.

We can assign a different tuple to the Ratings variable. The variable Ratings now references another tuple.

```
In [51]: Ratings=(10,9,6,5,10,8,9,6,2)

In [52]: Ratings1=Ratings

In [54]: Ratings=(1,2,3,4)

In [57]: print ("Ratings =", Ratings)
         print ("Ratings 2 =", Ratings1)

         Ratings = (1, 2, 3, 4)
         Ratings 2 = (10, 9, 6, 5, 10, 8, 9, 6, 2)
```

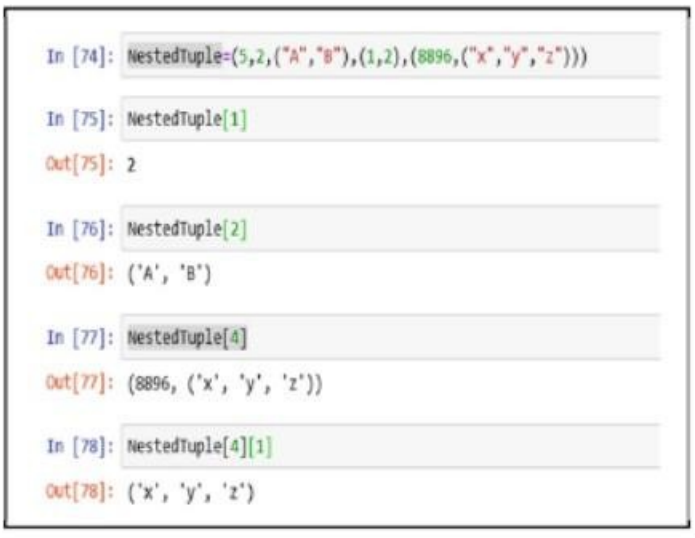


There are many built-in functions that take tuple as a parameter and perform some task. for example, we can find length of the tuple with len () function, minimum value with min () function... etc.

if we would like to sort a tuple, we use the function sorted. The input is the original tuple. The output is a new sorted tuple.

A tuple can contain other tuples as well as other complex data types; this is called nesting.

For Example: NestedTuple = (5,2, ("A","B"),(1,2),(8896,("x","y","z")))
We can access these elements using the standard indexing methods.



For example, we could access the second element. We can apply this indexing directly to the tuple variable NT. It is helpful to visualize this as a tree. We can visualize this nesting as a tree. The tuple has the following indexes. If we consider indexes with other tuples, we see the tuple at index 2 contains a tuple with two elements. We can access those two indexes. The same convention applies to index 3. We can access the elements in those tuples as well. We can

continue the process. We can even access deeper levels of the tree by adding another square bracket like NestedTuple

List:

A list is a collection which is ordered and changeable. A list is represented with square brackets. In many respects' lists are like tuples, one key difference is they are mutable. Lists can contain strings, floats, integers We can nest other lists.

```
In [79]: List = [21,3,"PYTHON",67.2,2.11]

In [80]: print (List)
         [21, 3, 'PYTHON', 67.2, 2.11]
```

We can also nest tuples and other data structures; the same indexing conventions apply for nesting Like tuples, each element of a list can be accessed via an index.

```
In [81]: #NESTED LIST EXAMPLE
         List = [21,3,"PYTHON",67.2,2.11,(121,32),["sublist1",8896]]

In [82]: print (List)
         [21, 3, 'PYTHON', 67.2, 2.11, (121, 32), ['sublist1', 8896]]
```

List =
[21,3,"PYTHON",67.2,2.11,(121,32),["sublist1",8896]]

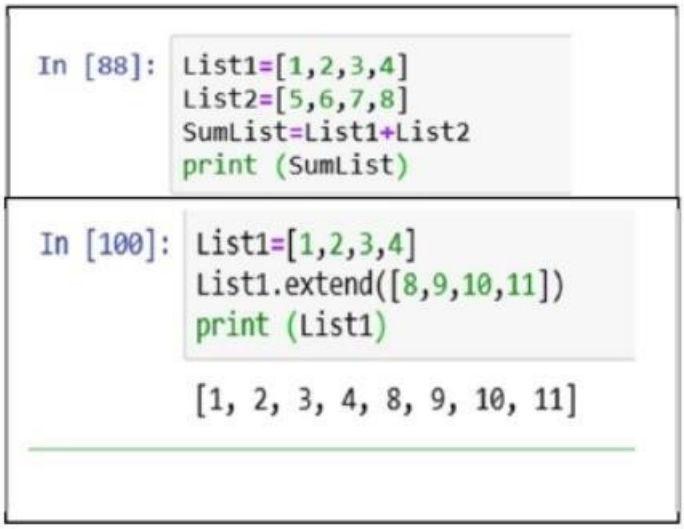
INDEX	List ELEMENTS	Negetive INDEX
List[0]	21	List[-7]
List[1]	3	List[-6]
List[2]	PYTHON	List[-5]
List[3]	67.2	List[-4]
List[4]	2.11	List[-3]
List[5]	(121,32)	List[-2]
List[6]	["subList1",8896]	List[-1]

The following table represents the relationship between the index and the elements in the list. The first element can be accessed by the name of the list followed by a square bracket with the index number, in this case zero. We can access the second element as follows. We can also access the last element. In Python, we can use a negative index.

The index conventions for lists and tuples are identical for accessing and slicing the elements.

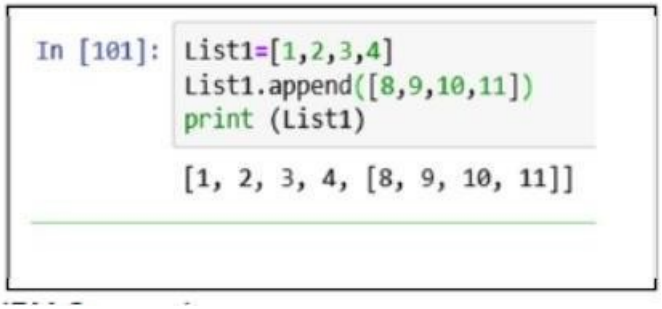
We can concatenate or combine lists by adding them. Lists are mutable; therefore, we can

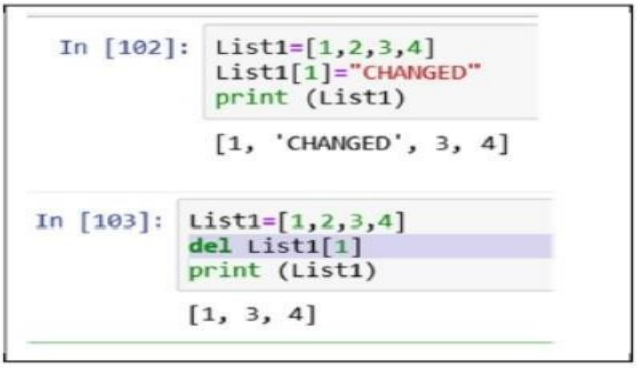
change them. For example, we apply the method Extends by adding a "dot" followed by the name of the method, then parenthesis.



The argument inside the parenthesis is a new list that we are going to concatenate to the original list. In this case, instead of creating a new list, the original list List1 is modified by adding four new elements.

Another similar method is append. If we apply append instead of extended, we add one element to the list. If we look at the index, there is only one more element. Index 4 contains the list we appended.
Every time we apply a method, the lists changes.





As lists are mutable, we can change them. For example, we can change the Second element as

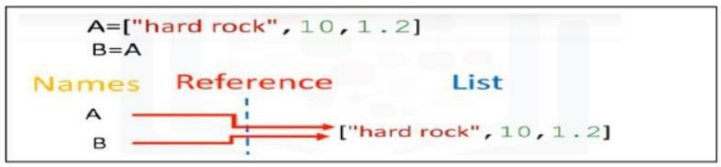
follows. The list now becomes [1," CHANGED",3,4]

We can delete an element of a list using the "del" command; we simply indicate the list item we would like to remove as an argument.

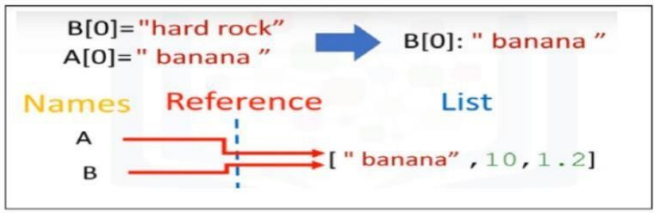
For example, if we would like to remove the Second element, then perform del List [1] command This operation removes the second element of the list then the result becomes [1,3,4]

LISTS: Aliasing

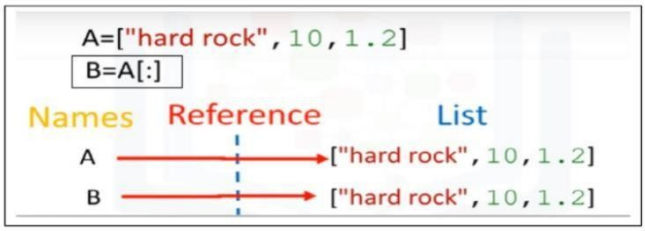
When we set one variable, B equal to A, both A and B are referencing the same list. Multiple names referring to the same object is known as aliasing.



If we change the first element in "A" to "banana" we get a side effect; the value of B will change as a consequence. "A" and "B" are referencing the same list, therefore if we change "A", list "B" also changes. If we check the first element of B after changing list "A" we get banana instead of hard rock



You can clone list "A" by using the following syntax. Variable "A" references one list. Variable "B" references a new copy or clone of the original list.



Now if you change "A", "B" will not change We can get more info on lists, tuples and many other objects in Python using the help command.

Simply pass in the list, tuple or any other Python object example: help(list),help(tuple)..etc.

Set:

Sets are a type of collection. Unlike lists and tuples, they are unordered. You cannot access items in a set by referring to an index, since sets are unordered the items has no index. To define a set, you use curly brackets You place the elements of a set within the curly brackets.

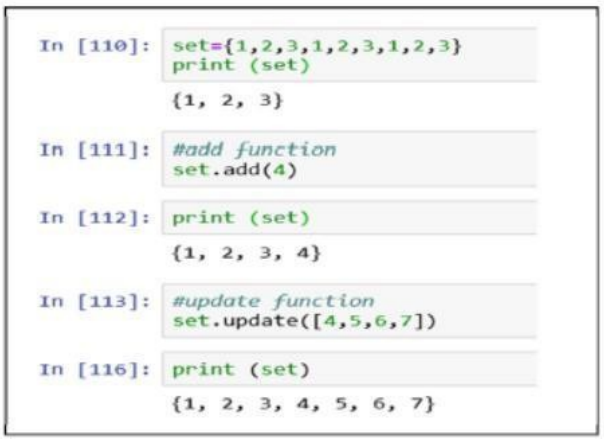
You notice there are duplicate items. When the actual set is created, duplicate items will not be present.

To add one item to a set, use the add () method.

To add more than one item to a set use the update () method with list of values.

To remove an item from the set we can use the pop () method. Remember sets are unordered so it will remove the first item in the set.

To remove an item from the set, use the remove method, we simply indicate the set item we would like to remove as an argument.



```
In [117]: set={1,2,3,1,2,3,1,2,3,4,5,6}
          print (set)

          {1, 2, 3, 4, 5, 6}

In [118]: #pop function
          set.pop()

Out[118]: 1

In [119]: print (set)

          {2, 3, 4, 5, 6}

In [120]: #remove function
          set.remove(5)

In [121]: print (set)

          {2, 3, 4, 6}
```

There are lots of useful mathematical operations we can do between sets. like union, intersection, difference, symmetric difference from two sets.

DICTIONARIES:

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair. Dictionaries are optimized to retrieve values when the key is known. Creating a dictionary is as simple as placing items inside curly braces {} separated by comma. An item has a key and the corresponding value expressed as a pair, key: value. While values can be of any data type and can repeat, keys must be of immutable type (**string, number or tuple** with immutable elements) and must be unique.

```
In [122]: my_dict={"Name":"Srikar","age":22}

In [123]: #my_dict variable is now refering to Dictionary object
          type(my_dict)

Out[123]: dict
```

We can access the elements from the dictionary using keys.

```
In [122]: my_dict={"Name":"Srikar","age":22}

In [125]: print (my_dict["Name"])
          Srikar

In [127]: print (my_dict.get("age"))
          22
```

We can get the value using keys either inside square brackets or with get( ) method.

Dictionary is mutable. We can add new items or change the value of existing items using assignment operator. If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
In [130]: #updates Value when using existing Key
          my_dict["Name"]="Varma"

In [131]: my_dict
Out[131]: {'Name': 'Varma', 'age': 22}

In [132]: #Adds new key:value when new key is used
          my_dict["address"]="High Hill Town"
          print (my_dict)
          {'Name': 'Varma', 'age': 22, 'address': 'High Hill Town'}
```

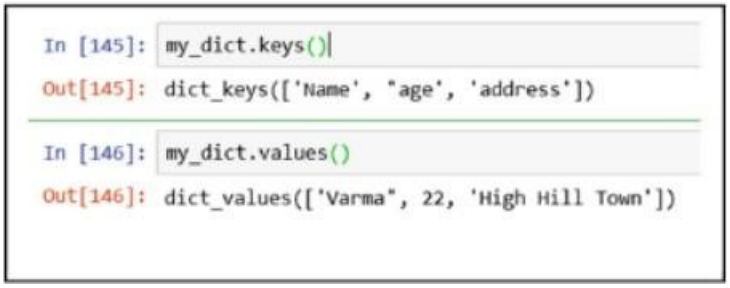
We can delete an entry as follows. This gets rid of the key "address" and its value from my_dict dictionary.

```
In [134]: my_dict={'Name': 'Varma', 'age': 22, 'address': 'High Hill Town'}

In [135]: del(my_dict["address"])

In [136]: print (my_dict)
          {'Name': 'Varma', 'age': 22}
```

We can verify if an element is in the dictionary using the in command as follows.
Syntax: 'KEY_NAME' in DictionaryName
The command checks the keys. If they are in the dictionary, they return a true. If we

try the same command with a key that is not in the dictionary, we get a false. If we try with another key that is not in the dictionary, we get a false.

In order to see all the keys in a dictionary, we can use the method keys to get the keys. The output is a list like object with all keys. In the same way, we can obtain the values.



Conditional Statements

What is Control or Conditional Statements -

In programming languages, most of the time we have to control the flow of execution of your program, you want to execute some set of statements only if the given condition is satisfied, and a different set of statements when it's not satisfied. Which we also call it as control statements or decision-making statements.

Conditional statements are also known as decision-making statements. We use these statements when we want to execute a block of code when the given condition is true or false.

Usually Condition will be in a form of Expression with some relational operators. Refer some below operators mentioned in the chart

Meaning	Operator
Equal to	==
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Not equal	!=
Negation	not

In Python we achieve the decision-making statements by using below statements -

If statements

If-else statements

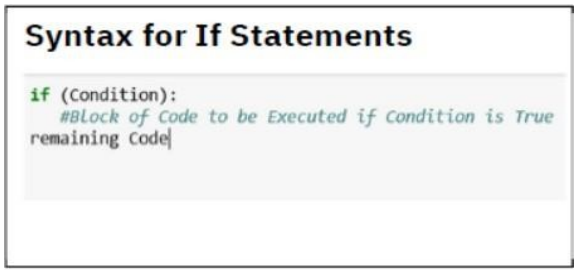
Elif statements

Nested if and if-else statements

If statements -

If statement is one of the most commonly used conditional statement in most of the programming languages. It decides whether certain statements need to be executed or not. If statement checks for a given condition, if the condition is true, then the set of code present inside the if block will be executed.

The If condition evaluates a Boolean expression and executes the block of code only when the Boolean expression becomes TRUE. Check the Syntax first the controller will come to an if condition and evaluate the condition if it is true, then the statements will be executed, otherwise the code present outside the block will be executed.



Let’s take an example to implement the if statement, in this example we have a variable name which stores the string “Srikar” and we also have names list with some names

Example

```
In [151]: name="Srikar"
          names=["Srikar","Harish","Vara"]
          if(name in names):
              print ("your name is present")
          print ("This Statement will always be Executed")

          your name is present
          This Statement will always be Executed
```

We can use if statement to check whether the name is present in the names list or not, if condition is true then it will also print block of statements inside the 'if' block. If condition is false, then it will skip the execution of the 'if' block statements.

If-else statements:

The statement itself tells that if a given condition is true then execute the statements present inside if block and if the condition is false then execute the else block.

Else block will execute only when the condition becomes false, this is the block where you will perform some actions when the condition is not true.

If-else statement evaluates the Boolean expression and executes the block of code present inside the if block if the condition becomes TRUE and executes a block of code present in the else block if the condition becomes FALSE.

Syntax for if-else statement

```
if (Condition):
   #Block of Code to be Executed if Condition is True
else:
    #Block of code to be Executed if condition is false
remaining Code
```

Let's take an example to implement the if-else statement, in this example the if block will get executed if the given condition is true or else it will execute the else block.

Example

```
In [152]: num=4
          if(num>5):
              print("number is greater than 5")
          else:
              print("number is less than 5")

          number is less than 5
```

elif statements:

In python, we have one more conditional statement called elif statements. Elif statement is used to check multiple conditions only if the given if condition false. It's like an if-else statement and the only difference is that in else we will not check the condition but in elif we will do check the condition.

Syntax for elif statement

```
if (condition):
        #Set of statement to execute if condition is true
elif (condition):
        #Set of statements to be executed when if condition is false and elif condition is true
else:
     #Set of statement to be executed when both if and elif conditions are false
```

Elif statements are similar to if-else statements but elif statements evaluate multiple conditions.

Example

```
In [153]: num=4
          if (num==0):
               print("number is zero")
          elif (num > 5):
               print("number is greater than 5")
          else:
              print("number is less than 5")

          number is less than 5
```

Let’s take an example to implement the elif statement, in this example the if block will get executed if the given if-condition is true, or elif block will get executed if the elif-condition is true, or it will execute the else block if both if and elif conditions are false.

Nested if-else statements

Nested if-else statements mean that an if statement or if-else statement is present inside another if or if-else block. Python provides this feature as well, this in turn will help us to check multiple conditions in a given program. An if statement present inside another if statement which is present inside another if statements and so on.

Syntax for nested if-else

```
if(condition):
        #Statements to execute if condition is true
        if(condition):
                #Statements to execute if condition is true
        else:
                #Statements to execute if condition is false
else:
         #Statements to execute if condition is false
```

Example for nested if-else

```
In [164]: num=-5
          if(num!=0):
              print("number not equal to 0")
              if(num>0):
                  print("number is positive")
              else:
                  print("number is negitive")
          else:
              print("number is equal to 0")

          number not equal to 0
          number is negitive
```

Numpy and Pandas > Numpy overview - Creating and Accessing Numpy Arrays > Numpy overview - Creating and Accessing Numpy Arrays

What is numpy ?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms,

basic linear algebra, basic statistical operations, random simulation and much more.

Difference between numpy arrays and lists

There are several important differences between NumPy arrays and the standard Python sequences, NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

CREATING NUMPY 1D ARRAY

A "numpy" array or "ndarray" is similar to a list. It's usually fixed in size and each element is of the same type, we can cast the list to numpy array by first importing the numpy. Or We can also quickly create the numpy array with arange function which creates an array within the range specified.

To verify the dimensionality of this array, use the shape property.

In example Since there is no value after the comma (20,) this is a one-dimensional array.

```
import numpy

#casting list to numpy
a=np.array([1,987,21,872,1])
print ("Numpy array a=",a)
```

```
Numpy array a= [  1 987  21 872   1]
```

```
import numpy

#casting list to numpy
a=np.arange(20)
print ("Numpy array a=",a)

#to check dimensionality of array use shape property
print (a.shape)
```

```
Numpy array a= [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
(20,)
```

Accessing NUMPY 1D ARRAY

```
In [177]: a=np.array([1,987,21,872,1])
          print ("Numpy array a=",a)

          #accessing first element of numpy array
          print ("a[0] -",a[0])

          #change the 2nd value of numpy array to 1000
          a[1]=1000
          print ("After changing 2nd value to 100 - ",a)

          #slicing the first 3 values from numpy array
          print ("After slicing  =",a[0:3])

          Numpy array a= [  1 987  21 872   1]
          a[0] - 1
          After changing 2nd value to 100 =  [   1 1000   21  872    1]
          After slicing  - [   1 1000   21]
```

Accessing and slicing operations for 1D array is same as list. Index values starts from 0 to length of the list.

Creating numpy 2D ARRAY

If you only use the arrange function, it will output a one-dimensional array. To make it a two-dimensional array, chain its output with the reshape function.

In this example first, it will create the 15 integers and then it will convert to two dimensional array with 3 rows and 5 columns.

```
a=np.arange(15).reshape(3,5)
print (a)

print ("SHAPE OF THE NUMPY ARRAY IS =", a.shape)

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
SHAPE OF THE NUMPY ARRAY IS = (3, 5)
```

Accessing NUMPY 2D ARRAY

To access an element in a two-dimensional array, you need to specify an index for both the row and the column.

```
a=np.arange(15).reshape(3,5)
print (a)

print ("Element in 1nd Row and 1 coloumn =", a[0,0])
print ("Element in 2nd Row and 5 coloumn =", a[1,4])
print ("Element in 3nd Row and 5 coloumn =", a[2,4])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
Element in 1nd Row and 1 coloumn = 0
Element in 2nd Row and 5 coloumn = 9
Element in 3nd Row and 5 coloumn = 14
```

Introduction to Pandas

What are pandas ?

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. Pandas is the backbone for most of the data projects.

Through pandas, you get acquainted with your data by cleaning, transforming, and analyzing it. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

We can import the library or a dependency like pandas using the **"import pandas"** command. We now have access to many pre-built classes and functions.

In order to be able to work with the data in Python, we'll need to read the data(csv, excel ,dictionary,..) file into a **Pandas DataFrame**. A DataFrame is a way to represent and work with tabular data. Tabular data has rows and columns, just like our csv file. In order to read in the data, we'll need to use the **pandas.read_csv** function. This function will take in a csv file and return a DataFrame.

What is csv ?

csv stands for comma-separated values, csv file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

How to read the csv file using pandas

Once the pandas library is imported, This assumes the library is installed. Then we can load a csv file using the pandas built-in function "read csv." A csv is a typical file type used to store data.

We simply type the word pandas, then a dot and the name of the function with all the inputs. Typing pandas all the time may get tedious.

We can use the "as" statement to shorten the name of the library; in this case we use the standard abbreviation pd. Now we type pd and a dot followed by the name of the function we would like to use, in this case, read_csv.

```
In [230]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\SAMPLE_DATA.csv")
          df.head(5)
```

Out[230]:

	index	hipi_name	year	hipi	tot_hhs	own	own_wm	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
0	0	Superannuitant	2008	super	300243	263054	15406	87.6	5.1	19.2	70.3	1.6	22367	21538	17203	17211
1	1	All households	2008	allhh	1560859	1087580	574406	69.7	36.8	100.0	35.9	2.7	46704	42394	26869	25132
2	2	Beneficiary	2008	benef	185965	71256	39405	38.3	21.2	11.9	29.9	2.6	23404	25270	14258	15824
3	3	Income quintile 1 (low)	2008	disq1	312376	191470	48424	61.3	15.5	20.0	40.0	2.3	16747	21145	13402	14408
4	4	Income quintile 2	2008	disq2	312333	196203	84171	62.8	26.9	20.0	34.7	2.8	31308	29855	18917	18266

we need to give the path of the csv file as argument to the read_csv function, to read the path string correctly we need to use ‘r’ as a prefix to the command. The result is stored in the variable df. this is short for “dataframe." Now that we have the data in a dataframe, we can work with it. We can use the method head to see the entire data frame or we can pass the number of rows to be checked as an argument to the head method like df.head(5) for 5 rows.

How to Write the csv file using pandas

If you want to write the dataframe to csv we can simple use the “to_csv” function

Syntax:
df.to_csv(EXPORT FILE PATH)

example:
If you see the above data frame example we see the two indexes one is loaded from the csv file and also there is unnamed index which is by default generated by pandas while loading the csv. This problem can be avoided by making sure that the writing of CSV files doesn’t write indexes, because DataFrame will generate it anyway. We can do the same by specifying index = False parameter in to_csv(...) function.

df.to_csv(‘EXPORT FILE PATH’, index=False)

Descriptive statistics using pandas

There are many collective methods to compute descriptive statistics and other related operations on pandas DataFrame.

Steps TO FOLLOW FOR Descriptive Statistics

These are the three steps we should perform to do statistical analysis on pandas dataframe.

collect the data

create the data frame

get the descriptive statistics for pandas dataframe

Collect the data:

To do any statistical analysis, first collection of data is the important task

You can store the collected data in csv, excel, or in dictionary format. For Demo we store the home data in one csv file.

Create the data frame:

we need to create the data frame based on the data collected.
Give the homes csv file path location.

```
In [267]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\homes.csv")
          df.head(5)
```

Out[267]:

	Sell	"Rooms"	"Beds"	"Acres"	"Taxes"
0	142	10.0	5.0	0.28	3167.0
1	175	8.0	4.0	0.43	4033.0
2	129	6.0	3.0	0.33	1471.0
3	138	7.0	3.0	0.46	3204.0
4	232	8.0	4.0	2.05	3613.0

Once you run the above code you will get this DataFrame.

Get the Descriptive Statistics for Pandas DataFrame

Once you have your Data Frame ready, you'll be able to get the Descriptive Statistics. We can calculate the following statistics using the pandas package:

Mean

Total sum

Maximum

Minimum

Count

Median

Standard deviation

Variance

With describe function you will get complete descriptive stats

The syntax is: df.describe()

```
In [320]: df.describe()
```

Out[320]:

	Sell	Rooms	Beds	Acres	Taxes
count	9.000000	9.000000	9.000000	9.000000	9.000000
mean	175.444444	8.000000	4.000000	1.207778	3611.888889
std	50.356507	1.322876	0.707107	1.284132	1247.810327
min	129.000000	6.000000	3.000000	0.280000	1471.000000
25%	138.000000	7.000000	4.000000	0.430000	3131.000000
50%	150.000000	8.000000	4.000000	0.530000	3204.000000
75%	207.000000	8.000000	4.000000	2.050000	4033.000000
max	271.000000	10.000000	5.000000	4.000000	5702.000000

We can also get the descriptive statistics for the 'particular field:

```
In [321]: df["Rooms"].describe()
```

```
Out[321]: count     9.000000
          mean      8.000000
          std       1.322876
          min       6.000000
          25%       7.000000
          50%       8.000000
          75%       8.000000
          max      10.000000
          Name: Rooms, dtype: float64
```

You can further breakdown the descriptive statistics into the following measures:

	Sell	Rooms	Beds	Acres	Taxes
0	142	10	5	0.28	3167
1	175	8	4	0.43	4033
2	129	6	3	0.33	1471
3	138	7	3	0.46	3204
4	232	8	4	2.05	3613
5	135	7	4	0.57	3028
6	150	8	4	4.00	3131
7	207	8	4	2.22	5158
8	271	10	5	0.53	5702

```
In [330]: #To get minimum value for "Sell" Coloumn
          df['Sell'].min()

Out[330]: 129

In [332]: #To get maximum value for "Rooms" Coloumn
          df['Rooms'].max()

Out[332]: 10

In [333]: #To calculate the mean for "Acres" Coloumn
          df['Acres'].mean()

Out[333]: 1.2077777777777778
```

Pandas working with text data and datetime columns

While working with data, it is not an unusual thing to encounter time series data. Working with datetime columns can be quite challenge task. Luckily, pandas are great at handling time series data. Pandas provide a different set of tools using which we can perform all the necessary tasks on date-time data.

Let's see how we can convert a dataframe column of strings (in dd/mm/yyyy format) to datetime format. We cannot perform any time series-based operation on the dates if they are not in the right format. To be able to work with it, we are required to convert the dates into the datetime format.

Convert Pandas dataframe column type from string to datetime format

For any operation we need to first create the data frame based on the data collected, we can load the data either from csv file, or excel file or from any source. Let us use csv file for our

demo.

Follow the below lines of code to load the data and convert that to data Frame.

Once the data frame is ready use df.info( ) to get complete information of dataframe.

```
In [343]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\DateTime.csv")
          df.head(5)
```

Out[343]:

	DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
0	5/23/2018 15:42	6	0	1.38	0.31	2	22.57
1	5/23/2018 15:43	6	0	1.38	0.31	2	22.52
2	5/23/2018 15:44	6	0	1.38	0.30	2	21.51
3	5/23/2018 15:45	6	0	1.38	0.30	2	21.46
4	5/23/2018 15:46	6	0	1.38	0.29	2	21.25

```
In [344]: df.info(
          )
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 46524 entries, 0 to 46523
Data columns (total 7 columns):
DateTime                3778 non-null object
QueueName               46524 non-null int64
Q_Used                  46524 non-null int64
TotalMemoryGB           46524 non-null float64
FreeMemoryGB            46524 non-null float64
Processors              46524 non-null int64
FreeMemoryPercentage    46524 non-null float64
dtypes: float64(3), int64(3), object(1)
memory usage: 2.5+ MB
```

As we can see in the output, the data type of the ‘DateTime’ column is object i.e. string. Now we will convert it to datetime format using **pd.to_datetime()** function.

```
In [345]: df['DateTime']= pd.to_datetime(df['DateTime'])

In [346]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 46524 entries, 0 to 46523
Data columns (total 7 columns):
DateTime                3778 non-null datetime64[ns]
QueueName               46524 non-null int64
Q_Used                  46524 non-null int64
TotalMemoryGB           46524 non-null float64
FreeMemoryGB            46524 non-null float64
Processors              46524 non-null int64
FreeMemoryPercentage    46524 non-null float64
dtypes: datetime64[ns](1), float64(3), int64(3)
memory usage: 2.5 MB
```

After applying the pd.to_datetime () function to DateTime column, we can see in the output, the format of the 'DateTime' column has been changed to the datetime format.

HOW TO CHANGE THE INDEX of the dataframe

Most of the operations related to dateTime requires the DateTime column as the primary index, or else it will throw an error.

We can change the index with set_index() function.it takes two parameters one is column name you want to change as index, and another one is inplace=true. When inplace=True is passed, the data is renamed inplace, when inplace=False is passed (this is the default value, so isn't necessary), performs the operation and returns a copy of the object.

```
In [358]: df.set_index('DateTime',inplace=True)
```

```
In [359]: df
```

Out[359]:

DateTime	QueueNsme	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
2018-05-23 15:42:00	6	0	1.38	0.31	2	22.57
2018-05-23 15:43:00	6	0	1.38	0.31	2	22.52
2018-05-23 15:44:00	6	0	1.38	0.30	2	21.51
2018-05-23 15:45:00	6	0	1.38	0.30	2	21.46
2018-05-23 15:46:00	6	0	1.38	0.29	2	21.25
2018-05-23 15:52:00	6	0	1.38	0.39	2	28.28
2018-05-23 15:53:00	6	0	1.38	0.39	2	28.23
2018-05-23 15:56:00	6	0	1.38	0.43	2	31.52
2018-05-23 15:57:00	6	0	1.38	0.43	2	31.47
2018-05-23 15:58:00	6	0	1.38	0.43	2	31.26
2018-05-23 16:01:00	6	0	1.38	0.32	2	23.46
2018-05-23 16:05:00	6	0	1.38	0.53	2	38.40
2018-05-23 16:06:00	6	0	1.38	0.59	2	42.67
2018-05-23 16:10:00	6	0	1.38	0.57	2	41.47
2018-05-23 16:11:00	6	0	1.38	0.55	2	39.60
2018-05-23 16:13:00	6	0	1.38	0.46	2	33.49
2018-05-23 16:14:00	6	0	1.38	0.38	2	27.60
2018-05-23 16:17:00	6	0	1.38	0.09	2	6.78

Now the DateTime is the index of the dataframe. Now we can perform DateTime operations very easily.

Data Frame Filtering based on index
How to filter data based on particular year.

To Check all the values occurred in a particular year let's say 2018
Run command df['2018'].

```
In [378]: df['2018']
Out[378]:
```

DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
2018-05-23 15:42:00	6	0	1.38	0.31	2	22.57
2018-05-23 15:43:00	6	0	1.38	0.31	2	22.52
2018-05-23 15:44:00	6	0	1.38	0.30	2	21.51
2018-05-23 15:45:00	6	0	1.38	0.30	2	21.46
2018-05-23 15:46:00	6	0	1.38	0.29	2	21.25
2018-05-23 15:52:00	6	0	1.38	0.39	2	28.28
2018-05-23 15:53:00	6	0	1.38	0.39	2	28.23
2018-05-23 15:56:00	6	0	1.38	0.43	2	31.52
2018-05-23 15:57:00	6	0	1.38	0.43	2	31.47
2018-05-23 15:58:00	6	0	1.38	0.43	2	31.26
2018-05-23 16:01:00	6	0	1.38	0.32	2	23.46
2018-05-23 16:05:00	6	0	1.38	0.53	2	38.40
2018-05-23 16:06:00	6	0	1.38	0.59	2	42.67
2018-05-23 16:10:00	6	0	1.38	0.57	2	41.47
2018-05-23 16:11:00	6	0	1.38	0.55	2	39.60
2018-05-23 16:13:00	6	0	1.38	0.46	2	33.49
2018-05-23 16:14:00	6	0	1.38	0.38	2	27.80

Above result gives all the values recorded in year 2018.

How to filter data based on year and month

To View all observations that occurred in June 2018, run the below command

```
In [379]: df['2018-06']
Out[379]:
```

DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
2018-06-24 10:34:00	6	0	1.37	0.08	2	6.87
2018-06-24 10:36:00	6	0	1.37	0.08	2	6.84
2018-06-24 10:36:00	6	0	1.37	0.46	2	33.86
2018-06-24 10:37:00	6	0	1.37	0.46	2	33.70
2018-06-24 10:38:00	6	0	1.37	0.12	2	8.40
2018-06-24 10:39:00	6	0	1.37	0.11	2	8.16
2018-06-24 10:40:00	6	0	1.37	0.10	2	7.16
2018-06-24 10:41:00	6	0	1.37	0.61	2	44.76
2018-06-24 10:42:00	6	4	1.37	0.21	2	16.19
2018-06-24 10:43:00	6	0	1.37	0.66	2	40.20
2018-06-24 10:44:00	6	0	1.37	0.20	2	14.76
2018-06-24 10:46:00	6	0	1.37	0.20	2	14.67
2018-06-24 10:46:00	6	0	1.37	0.17	2	12.67
2018-06-24 10:47:00	6	0	1.37	0.17	2	12.62
2018-06-24 10:48:00	6	3	1.37	0.24	2	17.36
2018-06-24 10:49:00	6	0	1.37	0.23	2	16.43
2018-06-24 11:00:00	6	0	1.37	0.33	2	24.37
2018-06-24 11:01:00	6	0	1.37	0.33	2	24.34
2018-06-24 11:02:00	6	0	1.37	0.32	2	23.17

Similarly, if you want to view the observations after particular year, month and date. then the command is

```
df[datetime(year, month, date):]
```

If you need observations between two dates then command is

```
df['Starting_year, Starting_month, Starting_date':'Ending_year, Ending_month, Ending_date']
```

For ex: if you need Observations between May 3rd and May 4th Of 2018
then command is: df['5/3/2018':'5/4/2018']

Pandas Indexing and Selecting Data

What is Indexing ?

Indexing in pandas means simply selecting particular rows and columns of data from a DataFrame. Indexing could mean selecting all the rows and some of the columns, some of the rows and all the columns, or some of each of the rows and columns. Indexing can also be known as **Subset Selection**.

Let's load one csv file and convert that to data frame to perform the indexing and selection operations.

```
In [465]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\index.csv")
          df
```

Out[465]:

	Name	email	id	team
0	Judith	judith.diaz@sysiphus.com	200000	1
1	Victoria	victoria.price@sysiphus.com	200001	1
2	Leigh	leigh.snyder@sysiphus.com	200002	1
3	Rodney	rodney.barton@sysiphus.com	200003	1
4	Lucy	lucy.garza@sysiphus.com	200004	1
5	Fred	fred.clarke@sysiphus.com	200005	2
6	Jesse	jesse.parks@sysiphus.com	200006	2
7	Lamar	lamar.ruiz@sysiphus.com	200007	2
8	Eric	eric.gonzalez@sysiphus.com	200008	2
9	Arlene	arlene.hughes@sysiphus.com	200009	2

Once the data is loaded into data frame let's make Name as the index of this data frame.

Note: index is the primary key it should not contain duplicates

```
In [466]: df.set_index("Name",inplace=True)

In [467]: df

Out[467]:
```

Name	email	id	team
Judith	judith.diaz@sysiphus.com	200000	1
Victoria	victoria.price@sysiphus.com	200001	1
Leigh	leigh.snyder@sysiphus.com	200002	1
Rodney	rodney.barton@sysiphus.com	200003	1
Lucy	lucy.garza@sysiphus.com	200004	1
Fred	fred.clarke@sysiphus.com	200005	2
Jesse	jesse.parks@sysiphus.com	200006	2
Lamar	lamar.ruiz@sysiphus.com	200007	2
Eric	eric.gonzalez@sysiphus.com	200008	2
Arlene	arlene.hughes@sysiphus.com	200009	2

Selecting Single Column

In order to take single column, we simply put the name of the column in-between the

```
In [468]: # retrieving columns by indexing operator
          Email = df["email"]

In [469]: Email

Out[469]: Name
          Judith         judith.diaz@sysiphus.com
          Victoria     victoria.price@sysiphus.com
          Leigh          leigh.snyder@sysiphus.com
          Rodney        rodney.barton@sysiphus.com
          Lucy             lucy.garza@sysiphus.com
          Fred            fred.clarke@sysiphus.com
          Jesse           jesse.parks@sysiphus.com
          Lamar            lamar.ruiz@sysiphus.com
          Eric          eric.gonzalez@sysiphus.com
          Arlene        arlene.hughes@sysiphus.com
          Name: email, dtype: object
```

brackets.

Selecting Multiple Columns

To select multiple columns, we must pass a list of columns in an indexing operator.

```
In [470]: # retrieving multiple columns by indexing operator
          EmailANDTeam = df[["email","team"]]

In [471]: EmailANDTeam

Out[471]:
```

Name	email	team
Judith	judith.diaz@sysiphus.com	1
Victoria	victoria.price@sysiphus.com	1
Leigh	leigh.snyder@sysiphus.com	1
Rodney	rodney.barton@sysiphus.com	1
Lucy	lucy.garza@sysiphus.com	1
Fred	fred.clarke@sysiphus.com	2
Jesse	jesse.parks@sysiphus.com	2
Lamar	lamar.ruiz@sysiphus.com	2
Eric	eric.gonzalez@sysiphus.com	2
Arlene	arlene.hughes@sysiphus.com	2

Selecting a single row:

In order to select a single row using. loc[], we put a single row label in a .loc function.

```
In [474]: LucyData=df.loc["Lucy"]

In [475]: LucyData

Out[475]: email     lucy.garza@sysiphus.com
          id                         200004
          team                            1
          Name: Lucy, dtype: object
```

Selecting multiple rows:

In order to select multiple rows, we put all the row labels in a list and pass that to .loc function.

```
In [477]: jesseANDEric=df.loc[['Jesse','Eric']]

In [478]: jesseANDEric

Out[478]:
```

Name	email	id	team
Jesse	jesse.parks@sysiphus.com	200006	2
Eric	eric.gonzalez@sysiphus.com	200008	2

Selecting multiple rows and columns:

In order to select two rows and two columns, we select two rows which we want to select and two columns and put it in a separate list:
Dataframe.loc[["row1", "row2"], ["column1", "column2"]]

```
In [482]: selectedDF=df.loc[['Judith','Arlene'],['email','team']]

In [483]: selectedDF

Out[483]:
```

Name	email	team
Judith	judith.diaz@sysiphus.com	1
Arlene	arlene.hughes@sysiphus.com	2

In order to select all the rows and some columns the syntax looks like:

Dataframe.loc [:, ["column1", "column2"]]

```
In [487]: selectedDF=df.loc[: ,['email','team']]

In [488]: selectedDF

Out[488]:
```

Name	email	team
Judith	judith.diaz@sysiphus.com	1
Victoria	victoria.price@sysiphus.com	1
Leigh	leigh.snyder@sysiphus.com	1
Rodney	rodney.barton@sysiphus.com	1
Lucy	lucy.garza@sysiphus.com	1
Fred	fred.clarke@sysiphus.com	2
Jesse	jesse.parks@sysiphus.com	2
Lamar	lamar.ruiz@sysiphus.com	2
Eric	eric.gonzalez@sysiphus.com	2
Arlene	arlene.hughes@sysiphus.com	2

Pandas- groupby

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Let’s load one csv file and convert that to data frame to perform the group-by operations.

```
In [499]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\groups.csv")
          df
```

Out[499]:

	hlpi_name	year	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
0	All households	2008	69.7	36.8	100.0	35.9	2.7	46704	42394	26869	25132
1	Beneficiary	2008	38.3	21.2	11.9	29.9	2.6	23404	25270	14258	15824
2	Income quintile 1 (low)	2008	61.3	15.5	20.0	40.0	2.3	16747	21145	13402	14408
3	Income quintile 2	2008	62.6	26.9	20.0	34.7	2.8	31308	29655	18917	18268
4	Income quintile 3	2008	69.7	45.3	20.0	31.5	3.0	49106	46561	26870	24672
5	Income quintile 4	2008	73.3	47.3	20.0	35.3	2.6	61674	52776	36691	31958
6	Income quintile 5 (high)	2008	81.3	49.1	20.0	39.3	2.5	96861	72822	55637	42932
7	Expenditure quintile 1 (low)	2008	62.1	15.8	20.0	38.7	2.5	23680	16413	15190	11015
8	Expenditure quintile 2	2008	66.1	27.7	20.0	36.1	2.7	34155	29085	20357	18121
9	Expenditure quintile 3	2008	62.3	34.7	20.0	33.0	2.8	49771	42662	27203	25132
10	Expenditure quintile 4	2008	74.2	47.7	20.0	35.1	2.7	60863	59015	34547	34167
11	Expenditure quintile 5 (high)	2008	83.6	58.2	20.0	36.7	2.5	77434	89053	46269	51550
12	Maori	2008	47.4	30.5	16.2	28.9	3.2	42885	35312	23096	19797
13	Superannuitant	2008	87.6	5.1	19.2	70.3	1.6	22367	21536	17203	17211
14	All households	2011	65.2	32.6	100.0	36.3	2.6	53103	46098	30833	27335
15	Beneficiary	2011	28.7	13.8	12.3	28.0	2.7	25902	27605	16097	16685
16	Income quintile 1 (low)	2011	51.7	15.5	20.0	36.3	2.4	19787	24224	15414	16221
17	Income quintile 2	2011	58.2	24.1	20.0	35.0	2.9	37370	34200	21998	20586
18	Income quintile 3	2011	63.8	37.3	20.0	33.4	2.9	54894	49431	30833	28130
19	Income quintile 4	2011	70.6	41.5	20.0	36.8	2.6	69183	55569	42084	33019
20	Income quintile 5 (high)	2011	81.9	44.6	20.0	40.9	2.4	106227	71815	63106	44712
21	Expenditure quintile 1 (low)	2011	53.9	11.2	20.0	37.3	2.6	27501	18877	17612	13077
22	Expenditure quintile 2	2011	55.7	23.9	20.0	35.1	2.7	38932	32790	22895	20168
23	Expenditure quintile 3	2011	65.9	33.8	20.0	35.3	2.8	56117	46651	32053	27335

groupby function based on single category

Now we have data frame ready let’s group the data based on ‘hlpi_name’.

```
In [501]: groups = df.groupby('hlpi_name')

In [502]: groups
Out[502]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000017376CC1F98>
```

Once group by operation is done we get a result as groupby object.

Let’s print the value contained in any one of group. For that use the name of the ‘hlpi_name’. We use the function get_group() to find the entries contained in any of the groups.

```
In [514]: #To know all the unique values(groups) in hlpi_name
          df['hlpi_name'].drop_duplicates()

Out[514]: 0                    All households
          1                       Beneficiary
          2            Income quintile 1 (low)
          3                  Income quintile 2
          4                  Income quintile 3
          5                  Income quintile 4
          6           Income quintile 5 (high)
          7       Expenditure quintile 1 (low)
          8             Expenditure quintile 2
          9             Expenditure quintile 3
          10            Expenditure quintile 4
          11     Expenditure quintile 5 (high)
          12                            Maori
          13                  Superannuitant
          Name: hlpi_name, dtype: object
```

```
In [519]: #To get values contained in any one of group
          groups.get_group('Income quintile 1 (low)')
```

Out[519]:

	year	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
2	2008	61.3	15.5	20.0	40.0	2.3	16747	21145	13402	14408
16	2011	51.7	15.5	20.0	36.3	2.4	19787	24224	15414	16221
30	2014	52.1	16.0	20.0	37.4	2.4	22822	25809	17168	17555
44	2017	54.1	12.9	20.0	40.3	2.3	22733	26775	18859	17850

groupby function based on more than one category

Use groupby() function to form groups based on more than one category (i.e. Use more than one column to perform the splitting).

```
In [526]: group_2 = df.groupby(['hlpi_name', 'year'])
          group_2

Out[526]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000017376DD9080>
```

We got the result as groupby object.

Let’s print the first entries in all the groups formed using first() function.

```
In [526]: group_2 = df.groupby(['hlpi_name', 'year'])
          group_2

Out[526]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000017376DD9080>

In [527]: group_2.first()
```

Out[527]:

hlpi_name	year	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
All households	2008	69.7	36.8	100.0	35.9	2.7	48704	42394	26869	25132
	2011	65.2	32.6	100.0	36.3	2.6	53103	46098	30833	27335
	2014	66.5	33.7	100.0	37.0	2.6	57359	49330	33426	29683
	2017	66.6	33.0	100.0	37.4	2.7	64066	54293	36146	31409
Beneficiary	2008	38.3	21.2	11.9	29.9	2.6	23404	25270	14258	15824
	2011	28.7	13.8	12.3	28.0	2.7	25902	27605	16097	16685
	2014	24.6	14.5	11.4	27.3	2.5	26569	27348	17706	17893
	2017	22.8	10.8	7.8	29.1	2.8	29947	30054	16822	17655
Expenditure quintile 1 (low)	2008	62.1	15.8	20.0	38.7	2.5	23680	16413	15190	11015
	2011	53.9	11.2	20.0	37.3	2.6	27501	18877	17612	13077
	2014	54.9	12.7	20.0	38.5	2.6	30138	19997	19171	13678
	2017	53.8	9.7	20.0	40.0	2.6	33596	20167	20674	14437
Expenditure quintile 2	2008	66.1	27.7	20.0	36.1	2.7	34155	29085	20357	18121
	2011	55.7	23.0	20.0	35.1	2.7	38932	32790	22895	20168
	2014	58.4	24.3	20.0	35.7	2.7	42898	35822	25743	21961
	2017	57.6	23.7	20.0	36.1	2.8	50561	38052	27694	22792

Operations on groups

After splitting a data into a group, we can also apply a function to each group to perform some operations.
Here is the Sample example to get sum of values in particular groups.

```
In [540]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\groups.csv")
          Group = df.groupby(['hlpi_name'])
          Group.sum()
```

Out[540]:

hlpi_name	year	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
All households	8050	268.0	136.1	400.0	146.6	10.6	221232	192115	127274	113559
Beneficiary	8050	114.4	60.5	43.4	114.3	10.6	105822	110277	64883	68057
Expenditure quintile 1 (low)	8050	224.7	49.4	80.0	154.5	10.3	114915	75454	72647	52207
Expenditure quintile 2	8050	237.8	99.6	80.0	143.0	10.9	166546	135749	96689	83042
Expenditure quintile 3	8050	259.0	137.6	80.1	140.4	11.3	231877	193579	128354	113582
Expenditure quintile 4	8050	288.1	177.4	80.0	144.0	10.6	287802	259114	166203	153097
Expenditure quintile 5 (high)	8050	330.4	218.6	80.0	151.9	10.0	371471	392890	228301	226909
Income quintile 1 (low)	8050	219.2	59.9	80.0	154.0	9.4	82089	97953	64843	66034
Income quintile 2	8050	235.1	100.3	80.0	139.1	11.6	155489	146229	90892	85672
Income quintile 3	8050	264.7	158.3	80.0	134.4	11.8	228859	208998	127275	113843
Income quintile 4	8050	290.3	176.8	80.0	147.4	10.5	295602	242338	177082	143343
Income quintile 5 (high)	8050	330.9	187.4	80.0	162.3	9.8	459239	316426	270583	191459
Maori	8050	183.1	118.2	68.6	119.4	12.6	213329	172677	110314	93693
Superannuitant	8050	344.0	32.5	84.1	278.7	6.5	117510	106788	88745	84782

We can also find min, max, average . .etc.

Merge/Join Datasets

Joining and merging DataFrames is the core process to start with data analysis and machine learning tasks. It is one of the toolkits which every Data Analyst or Data Scientist should master because in almost all the cases data comes from multiple source and files. You may need to bring all the data in one place by some sort of join logic and then start your analysis. Thankfully you have the most popular library in python, pandas to your rescue! Pandas provides various facilities for easily combining different datasets.

We can merge two data frames in pandas python by using the merge() function. The different arguments to merge() allow you to perform natural join, left join, right join, and full outer join in pandas.

Understanding the different types of merge:

Before you perform joint operations let's first load the two csv files and convert them into data frames df1 and df2.

```
In [541]: import pandas as pd
          df1=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\joints\df1.csv")
          df1
```

Out[541]:

	customer_id	Product
0	1	Oven
1	2	Television
2	3	AC
3	4	Washing Machine
4	5	AC
5	6	Oven
6	7	Television
7	8	Washing Machine
8	9	Television
9	10	Washing Machine

```
In [542]: df2=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\joints\df2.csv")
          df2
```

Out[542]:

	customer_id	state
0	1	Texas
1	2	California
2	4	Florida
3	7	California
4	10	Florida

Natural join

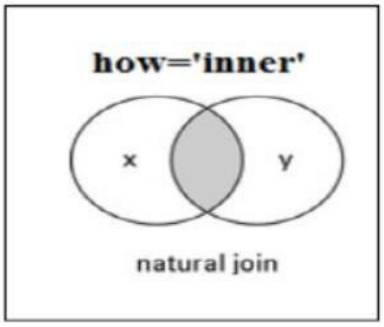
Natural join keeps only rows that match from the data frames(df1 and df2), specify the

argument how=’inner’

Syntax:

pd.merge(df1, df2, on=column', how='inner')

Return only the rows in which the left table have matching keys in the right table



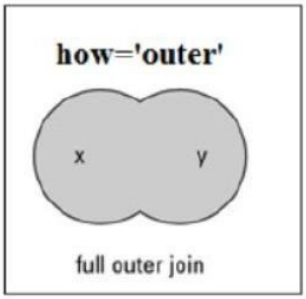

```
In [545]: pd.merge(df1, df2, on='customer_id', how='inner')
```

Out[545]:

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	4	Washing Machine	Florida
3	7	Television	California
4	10	Washing Machine	Florida

Full outer join

Full outer join keeps all rows from both data frames, specify how=‘outer’.



Syntax:

pd.merge(df1, df2, on=column', how=’outer’)

Returns all rows from both tables, join records from the left which have matching keys in the right table.

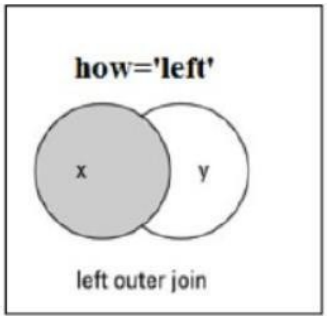
```
In [546]: pd.merge(df1, df2, on='customer_id', how='outer')
```

Out[546]:

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	3	AC	NaN
3	4	Washing Machine	Florida
4	5	AC	NaN
5	6	Oven	NaN
6	7	Television	California
7	8	Washing Machine	NaN
8	9	Television	NaN
9	10	Washing Machine	Florida

Left outer join

Left outer join includes all the rows of your data frame df1 and only those from df2 that match, specify how =‘Left.



Syntax:

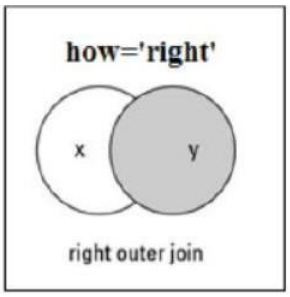
pd.merge(df1, df2, on=column', how=left)
Return all rows from the left table, and any rows with matching keys from the right table.

```
In [547]: pd.merge(df1, df2, on='customer_id', how='left')
Out[547]:
```

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	3	AC	NaN
3	4	Washing Machine	Florida
4	5	AC	NaN
5	6	Oven	NaN
6	7	Television	California
7	8	Washing Machine	NaN
8	9	Television	NaN
9	10	Washing Machine	Florida

Right outer join

Return all rows from the df2 table, and any rows with matching keys from the df1 table, specify how =‘Right’.



Syntax:

pd.merge(df1, df2, on=column', how=right)
Return all rows from the right table, and any rows with matching keys from the left table.

```
In [548]: pd.merge(df1, df2, on='customer_id', how='right')
Out[548]:
```

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	4	Washing Machine	Florida
3	7	Television	California
4	10	Washing Machine	Florida

UNIT – IV

Introduction to Matplotlib

Matplotlib is the most popular plotting library for python which gives control over every aspect of a figure. It was designed to give the end user a similar feeling like MATLAB's graphical plotting. In the coming sections we will learn about Seaborn that is built over matplotlib. The official page of Matplotlib is https://matplotlib.org. You can use this page for official installation instructions and various documentation links. One of the most important section on this page is the gallery section - https://matplotlib.org/gallery.html - it shows all the kind of plots/figures that matplotlib is capable of creating for you. You can select anyone of those, and it takes you the example page having the figure and very well documented code. Another important page is https://matplotlib.org/api/pyplot_summary.html- and it has the documentation functions in it.

Matplotlib's architecture is composed of three main layers: the back-end layer, the artist layer where much of the heavy lifting happens and is usually the appropriate programming paradigm when writing a web application server, or a UI application, or perhaps a script to be shared with other developers, and the scripting layer, which is the appropriate layer for everyday purposes and is considered a lighter scripting interface to simplify common tasks and for a quick and easy generation of graphics and plots.

Now let's go into each layer in a little more detail:

Back-end layer has three built-in abstract interface classes: FigureCanvas, which defines and encompasses the area on which the figure is drawn. Renderer, an instance of the renderer class knows how to draw on the figure canvas. And finally, Event, which handles user inputs such as keyboard strokes and mouse clicks.

Artist layer: It is composed of one main object, which is the Artist. The Artist is the object that knows how to take the Renderer and use it to put ink on the canvas. Everything you see on a Matplotlib figure is an Artist instance. The title, the lines, the tick labels, the images, and so on, all correspond to an individual Artist. There are two types of Artist objects. The first type is the primitive type, such as a line, a rectangle, a circle, or text. And the second type is the composite type, such as the figure or the axes. The top-level Matplotlib object that contains and manages all of the elements in a given graphic is the figure Artist, and the most important composite artist is the axes because it is where most of the Matplotlib API plotting methods are defined, including methods to create and manipulate the ticks, the axis lines, the grid or the plot background. Now it is important to note that each composite artist may contain other composite artists as well as primitive artists. So, a figure artist for example would contain an axis artist as well as a rectangle or text artists.

Scripting layer: it was developed for scientists who are not professional Programmers. The artist layer is syntactically heavy as it is meant for developers and not for individuals whose goal is to perform quick exploratory analysis of some data. Matplotlib's scripting layer is

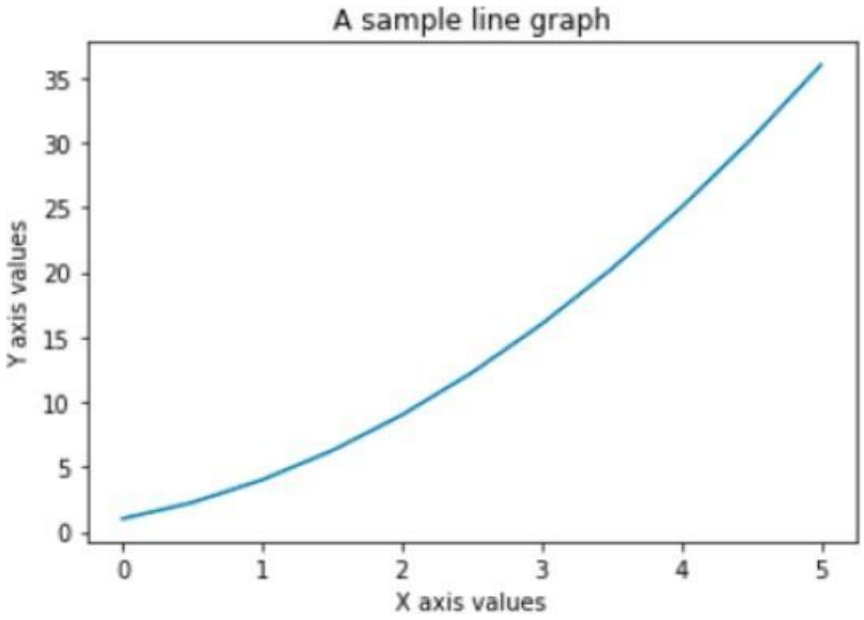
essentially the Matplotlib.pyplot interface, which automates the process of defining a canvas and defining a figure artist instance and connecting them.

Read a CSV and Generate a Line Plot with Matplotlib

A line plot is used to represent quantitative values over a continuous interval or time period. It is generally used to depict trends on how the data has changed over time.

In this sub-section, we will see how to use matplotlib to read a csv file and then generate a plot. We will use jupyter notebook. First, we do a basic example to showcase what a line plot is.

```
In [1]: import matplotlib.pyplot as plt

        #### The below line will allow you to view the plot inside the jupiter notebook

In [2]: %matplotlib inline

In [3]: import numpy as np
        x = np.linspace(0,5,11)
        y = (x+1) **2

In [4]: x

Out[4]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])

In [5]: y

Out[5]: array([ 1.  ,  2.25,  4.  ,  6.25,  9.  , 12.25, 16.  , 20.25, 25.  ,
               30.25, 36.  ])

        #### Now we will plot the graph for these two variables

In [9]: plt.plot(x,y)
        plt.xlabel('X axis values')
        plt.ylabel('Y axis values')
        plt.title('A sample line graph')

Out[9]: Text(0.5, 1.0, 'A sample line graph')
```



Now let us do a small case study using what we just learned now:

Download the dataset from the link: https://www.un.org/en/development/desa/population/migration/data/empirical2/migrationflows.asp

The data set has all the country immigration information. We will use the one for Australia for our case study.

Now let us use this to plot immigration data

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

We need to insall xlrd module that pandas need to read excel files. If you are using anaconda distribution then you can do it by using the command :

conda install -c anaconda xlrd --yes

```
df_aus = pd.read_excel('Australia.xlsx',sheet_name='Australia by Residence',skiprows=range(20),skipfooter=2)
print("pandas dataframe contains this data now")
```

```
pandas dataframe contains this data now
```

```
df_aus.head()
```

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012
0	Emigrants	Both	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	0	...	80	120	70	80	120	..	..	..	..
1	Emigrants	Both	Albania	908	Europe	925	Southern Europe	901	Developed regions	0	...	30	40	30	30	30	..	..	..	..
2	Emigrants	Both	Algeria	903	Africa	912	Northern Africa	902	Developing regions	20	...	20	20	30	40	50	..	..	..	..
3	Emigrants	Both	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	10	...	0	0	0	0	0	..	..	..	..
4	Emigrants	Both	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	0	0	0	..	..	..	..

5 rows × 43 columns

```
df_aus.tail()
```

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012
441	Immigrants	Both	Wallis and Futuna Islands	909	Oceania	957	Polynesia	902	Developing regions	0	...	10	0	0	0	0	..	..	..	..
442	Immigrants	Both	Western Sahara	903	Africa	912	Northern Africa	902	Developing regions	0	...	0	0	0	0	0	..	..	..	..
443	Immigrants	Both	Yemen	935	Asia	922	Western Asia	902	Developing regions	10	...	40	20	10	40	40	..	..	..	..
444	Immigrants	Both	Zambia	903	Africa	910	Eastern Africa	902	Developing regions	150	...	370	240	410	410	400	..	..	..	..
445	Immigrants	Both	Zimbabwe	903	Africa	910	Eastern Africa	902	Developing regions	630	...	1790	1560	1880	2180	2270	..	..	..	..

5 rows × 43 columns

Get the list of column names and the list of indices

```
df_aus.columns.values
```

```
array(['Type', 'Coverage', 'OdName', 'AREA', 'AreaName', 'REG', 'RegName',
       'DEV', 'DevName', 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987,
       1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998,
       1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009,
       2010, 2011, 2012, 2013], dtype=object)
```

```
df_aus.index.values
```

```
array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
        78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
        91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103,
       104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
       117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
       130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142,
       143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155,
       156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
       169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181,
       182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194,
```

Use the tolist() method to get index and columns as lists. View the dimensions of dataframe using ".shape" parameter.

After that let us clean the data set to remove few unnecessary columns.

```
df_aus.columns.tolist()
df_aus.index.tolist()

print (type(df_aus.columns.tolist()))
print (type(df_aus.index.tolist()))
```

```
<class 'list'>
<class 'list'>
```

```
df_aus.shape
```

```
(446, 43)
```

```
#Let's clean the data set to remove a few unnecessary columns. We can use pandas drop() method as follows:
df_aus.drop(['AREA','REG','DEV','Type','Coverage'],axis=1, inplace=True)
df_aus.head(2)
```

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	0	0	0	0	0	0	...	80	120	70	80	120	..	..	..	..	..
1	Albania	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	30	40	30	30	30	..	..	..	..	..

2 rows × 38 columns

```
df_aus.describe()
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989
count	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000
mean	811.547085	850.784753	842.757848	787.825112	767.443946	795.560538	843.520179	920.717489	1024.753363	1060.022422
std	5282.000095	5361.755210	5391.664250	5328.403642	5053.983474	5034.944988	5156.676347	5487.016498	6094.203458	6610.235594
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000
75%	130.000000	140.000000	150.000000	150.000000	160.000000	160.000000	177.500000	190.000000	240.000000	227.500000
max	90860.000000	85600.000000	92340.000000	100510.000000	96360.000000	93440.000000	92450.000000	97770.000000	104770.000000	120040.000000

8 rows × 30 columns

Let us rename the column names so that it makes more sense.

```
df_aus.rename(columns={'OdName':'Country', 'AreaName':'Continent', 'RegName':'Region'}, inplace=True)
df_aus.columns
```

```
Index([  'Country', 'Continent',    'Region',   'DevName',        1980,
             1981,        1982,        1983,        1984,        1985,
             1986,        1987,        1988,        1989,        1990,
             1991,        1992,        1993,        1994,        1995,
             1996,        1997,        1998,        1999,        2000,
             2001,        2002,        2003,        2004,        2005,
             2006,        2007,        2008,        2009,        2010,
             2011,        2012,        2013,     'Total'],
      dtype='object')
```

```
df_aus.describe()
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989
count	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000	446.000000
mean	811.547085	850.784753	842.757848	787.825112	767.443946	795.560538	843.520179	920.717489	1024.753363	1060.022422
std	5282.000096	5361.755210	5391.664250	5328.403642	5053.983474	5034.944088	5156.676347	5487.016498	6094.203458	6810.235594
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000
75%	130.000000	140.000000	150.000000	150.000000	160.000000	160.000000	177.500000	190.000000	240.000000	227.500000
max	90860.000000	85600.000000	92340.000000	100510.000000	96360.000000	93440.000000	92450.000000	97770.000000	104770.000000	120040.000000

8 rows × 30 columns

```
df_aus.head(3)
```

	Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
0	Afghanistan	Asia	Southern Asia	Developing regions	0	0	0	0	0	0	...	120	70	80	120	..	..	..	..	..	1890
1	Albania	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	40	30	30	30	..	..	..	..	..	1410
2	Algeria	Africa	Northern Africa	Developing regions	20	10	10	10	10	10	...	20	30	40	50	..	..	..	..	..	1170

3 rows × 39 columns

Default index is numerical, but it is more convenient to index based on country names.

```
df_aus.set_index('Country', inplace=True)
df_aus.head(3)
```

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Afghanistan	Asia	Southern Asia	Developing regions	0	0	0	0	0	0	0	...	120	70	80	120	..	..	..	..	..	1890
Albania	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	0	...	40	30	30	30	..	..	..	..	..	1410
Algeria	Africa	Northern Africa	Developing regions	20	10	10	10	10	10	0	...	20	30	40	50	..	..	..	..	..	1170

3 rows × 38 columns

Remove the name of the index.

```
df_aus.index.name = None
```

```
df_aus.head(3)
```

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Afghanistan	Asia	Southern Asia	Developing regions	0	0	0	0	0	0	0	...	120	70	80	120	..	..	..	..	..	1890
Albania	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	0	...	40	30	30	30	..	..	..	..	..	1410
Algeria	Africa	Northern Africa	Developing regions	20	10	10	10	10	10	0	...	20	30	40	50	..	..	..	..	..	1170

3 rows × 38 columns

Let us now test it by pulling the data for Bangladesh.

```
print(df_aus[df_aus.index == 'Bangladesh'].T.squeeze())
```

```
                  Bangladesh          Bangladesh
Continent                Asia                Asia
Region          Southern Asia       Southern Asia
DevName    Developing regions  Developing regions
1980                       220                 170
1981                       140                 180
1982                       170                 220
1983                       180                 140
1984                       160                 170
1985                       140                 220
1986                       140                 190
1987                       140                 150
1988                       110                 270
1989                       180                 270
1990                       130                 400
1991                       230                 600
1992                       200                 690
1993                       190                 360
1994                       180                 690
1995                       190                1050
1996                       210                 900
```

Column names as numbers could be confusing. For example: year 1985 could be misunderstood as 1985th column. To avoid ambiguity, let us convert column names to strings and then use that to call full range of years.

```
: df_aus.columns = list(map(str,df_aus.columns))
```

```
: years= list(map(str,range(1980,2014)))
  years
```

```
: ['1980',
   '1981',
   '1982',
   '1983',
   '1984',
   '1985',
   '1986',
   '1987',
   '1988',
   '1989',
   '1990',
   '1991',
   '1992',
   '1993',
   '1994',
   '1995',
   '1996',
   '1997',
```

We can also pass multiple criteria in the same line.

```
df_aus[(df_aus['Continent']=='Asia') & (df_aus['Region']=='Southern Asia')]
```

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	
Afghanistan	Asia	Southern Asia	Developing regions	0	0	0	0	0	0	0	...	120	70	80	120	..	..	..	..	..	
Bangladesh	Asia	Southern Asia	Developing regions	220	140	170	180	160	140	140	...	540	480	500	530	..	..	..	..	..	2
Bhutan	Asia	Southern Asia	Developing regions	10	10	10	0	0	0	10	...	60	80	70	100	..	..	..	..	..	
India	Asia	Southern Asia	Developing regions	740	660	750	780	770	740	760	...	4010	4530	4890	5360	..	..	..	..	..	19
Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	40	30	30	30	20	50	60	...	300	360	380	350	..	..	..	..	..	2
Maldives	Asia	Southern Asia	Developing regions	10	10	10	0	10	20	10	...	80	110	70	70	..	..	..	..	..	
Nepal	Asia	Southern Asia	Developing regions	40	70	50	40	80	100	120	...	220	180	180	280	..	..	..	..	..	1
Pakistan	Asia	Southern Asia	Developing	110	150	150	190	200	170	200	...	550	610	540	580	..	..	..	..	..	3

Let us review the changes we have made to our dataframes.

```
print('data dimensions:', df_aus.shape)
print(df_aus.columns)
df_aus.head(2)
```

```
data dimensions: (446, 38)
Index(['Continent', 'Region', 'DevName', '1980', '1981', '1982', '1983',
       '1984', '1985', '1986', '1987', '1988', '1989', '1990', '1991', '1992',
       '1993', '1994', '1995', '1996', '1997', '1998', '1999', '2000', '2001',
       '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010',
       '2011', '2012', '2013', 'Total'],
      dtype='object')
```

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Afghanistan	Asia	Southern Asia	Developing regions	0	0	0	0	0	0	0	...	120	70	80	120	..	..	..	..	..	1890
Albania	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	0	...	40	30	30	30	..	..	..	..	..	1410

2 rows × 38 columns

Case Study – let us now study the trend of number of immigrants from Bangladesh to Australia.

Case Study - Do a plot of the immigrants from bangladesh

```
bangladesh = df_aus.loc['Bangladesh',years]
bangladesh.head()
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
Bangladesh	220	140	170	180	160	140	140	140	110	180	...	480	540	480	500	530	..	..	..	..	..
Bangladesh	170	180	220	140	170	220	190	150	270	270	...	2970	1860	2570	3680	3160	..	..	..	..	..

2 rows × 34 columns

Since there are two rows of data, let us sum the values of each column and take first 20 years (to eliminate other years for which no values are present).

```
pd_bang=bangladesh.sum(axis=0,skipna=True).head(20)
```

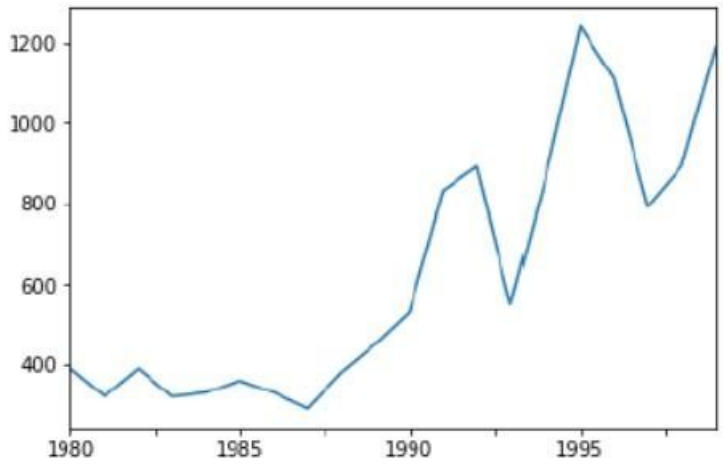
```
pd_bang
```

```
1980     390
1981     320
1982     390
1983     320
1984     330
1985     360
1986     330
1987     290
1988     380
1989     450
1990     530
1991     830
1992     890
1993     550
1994     870
1995    1240
1996    1110
1997     790
1998     890
```

Next, we can plot by using the plot function. Automatically the x-axis is plotted with the index values and y-axis with column values

```
pd_bang.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xc8485c0>
```



Basic Plots using Matplotlib

Area Plot

In the previous module we used line plot to see immigration from Bangladesh to Australia. Now let us try different types of basic plotting using matplotlib.

Area plot

Now let us use area plots to see to visualize cumulative immigration from top 5 countries to Canada. We will use the same process to clean data that we used in the previous section.

```
import numpy as np  # useful for many scientific computing in Python
import pandas as pd # primary data structure library
```

URL - https://s3-api.us-geo.objectstorage.softlayer.net/cf-coursesdata/CognitiveClass/DV0101EN/labs/Data_Files/Canada.xlsx

```
df_can = pd.read_excel('https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data_F
                       sheet_name='Canada by Citizenship',
                       skiprows=range(20),
                       skipfooter=2
                      )

print('Data downloaded and read into a dataframe!')
```

```
Data downloaded and read into a dataframe!
```

```
df_can.head()
```

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	0	1

5 rows × 43 columns

Let's find out how many entries there are in our dataset.

```
# print the dimensions of the dataframe
print(df_can.shape)
```

```
(195, 43)
```

Now clean up data using the same process as the one in the previous section :

```
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)

# let's view the first five elements and see how the dataframe was changed
df_can.head()
```

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450	1223	856	702	560	716	561	539	620	603
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
4	Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 38 columns

```
df_can.rename(columns={'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace=True)

# let's view the first five elements and see how the dataframe was changed
df_can.head()
```

	Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450	1223	856	702	560	716	561	539	620	603
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
4	Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 38 columns

```
# let's examine the types of the column labels
all(isinstance(column, str) for column in df_can.columns)
```

```
False
```

```
df_can.columns = list(map(str, df_can.columns))

# let's check the column labels types now
all(isinstance(column, str) for column in df_can.columns)
```

```
True
```

```
df_can.set_index('Country', inplace=True)

# let's view the first five elements and see how the dataframe was changed
df_can.head()
```

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1450	1223	856	702	560	716	561	539	620	603
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	0	1	1	0	0	0	0	1	1

5 rows × 37 columns

```
df_can['Total'] = df_can.sum(axis=1)

# let's view the first five elements and see how the dataframe was changed
df_can.head()
```

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

5 rows × 38 columns

```
print ('data dimensions:', df_can.shape)
```

```
data dimensions: (195, 38)
```

```
# finally, let's create a list of years from 1980 - 2013
# this will come in handy when we start plotting the data
years = list(map(str, range(1980, 2014)))

years
```

```
['1980',
 '1981',
 '1982',
 '1983',
 '1984',
 '1985',
 '1986',
 '1987',
 '1988'
```

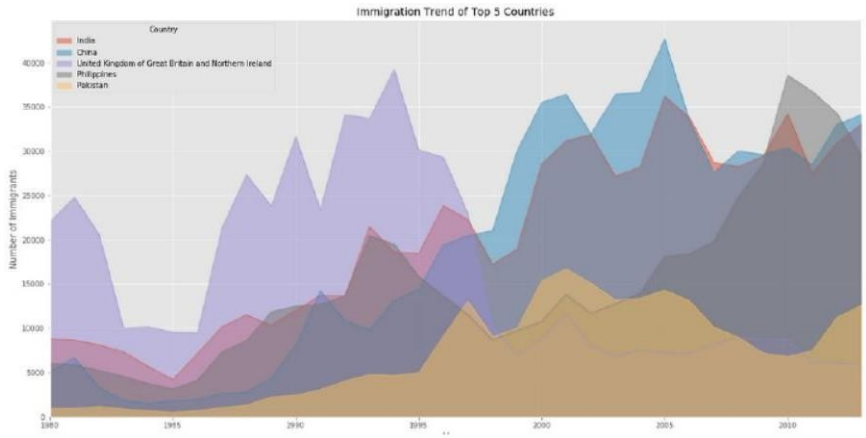
```
# use the inline backend to generate the plots within the browser
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.style.use('ggplot') # optional: for ggplot-like style

# check for latest version of Matplotlib
print ('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

```
Matplotlib version:  3.0.3
```

```
df_can.sort_values(['Total'], ascending=False, axis=0, inplace=True)

# get the top 5 entries
df_top5 = df_can.head()

# transpose the dataframe
df_top5 = df_top5[years].transpose()

df_top5.head()
```

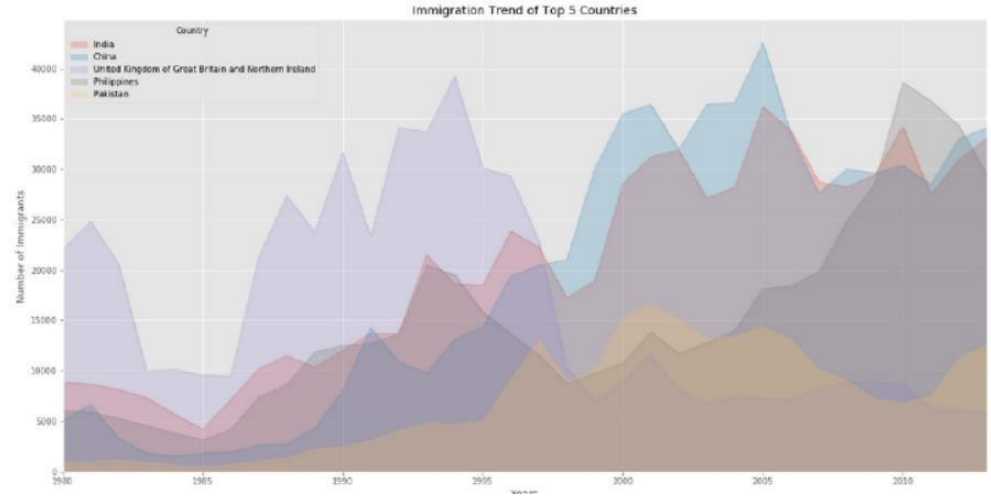
Country	India	China	United Kingdom of Great Britain and Northern Ireland	Philippines	Pakistan
1980	8880	5123	22045	6051	978
1981	8670	6682	24796	5921	972
1982	8147	3308	20620	5249	1201
1983	7338	1863	10015	4562	900
1984	5704	1527	10170	3801	668

```
df_top5.index = df_top5.index.map(int) # let's change the index values of df_top5 to type integer for plotting
df_top5.plot(kind='area',
             stacked=False,
             figsize=(20, 10), # pass a tuple (x, y) size
             )

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



The unstacked plot has a default transparency (alpha value) at 0.5. We can modify this value by passing in the alpha parameter.

```
df_top5.plot(kind='area',
             alpha=0.25, # 0-1, default value a= 0.5
             stacked=False,
             figsize=(20, 10),
            )

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



Bar Chart

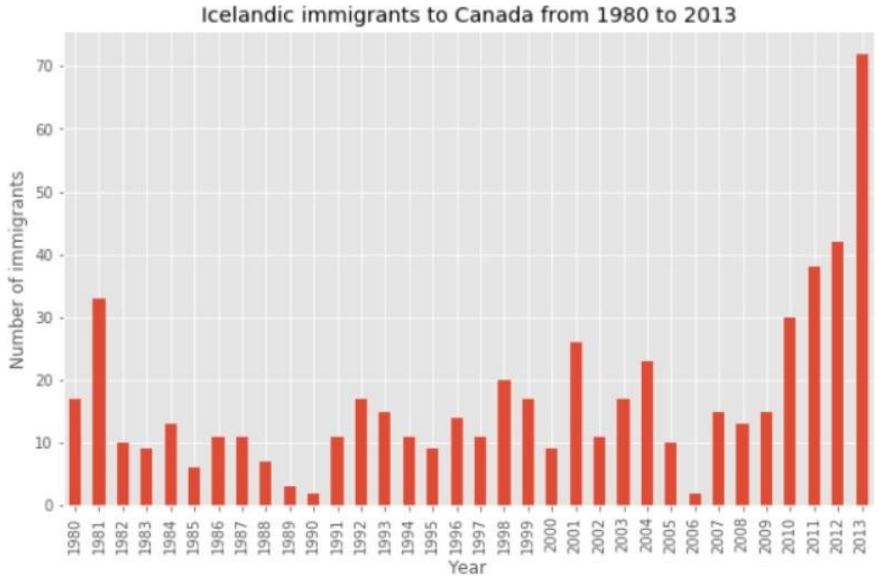
A bar plot is a way of representing data where the length of the bars represents the magnitude/size of the feature/variable. Bar graphs usually represent numerical and categorical variables grouped in intervals.

Let's compare the number of Icelandic immigrants (country = 'Iceland') to Canada from year 1980 to 2013.

```
# step 1: get the data
df_iceland = df_can.loc['Iceland', years]
df_iceland.head()
```

```
1980    17
1981    33
1982    10
1983     9
1984    13
Name: Iceland, dtype: object
```

```
# step 2: plot data
df_iceland.plot(kind='bar', figsize=(10, 6))

plt.xlabel('Year') # add to x-label to the plot
plt.ylabel('Number of immigrants') # add y-label to the plot
plt.title('Icelandic immigrants to Canada from 1980 to 2013') # add title to the plot

plt.show()
```



Histogram

How could you visualize the answer to the following question ?

What is the frequency distribution of the number (population) of new immigrants from the various countries to Canada in 2013 ?

To answer this one would need to plot a histogram - it partitions the x-axis into bins, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin. So, the y-axis is the frequency or the number of data points in each

bin. Note that we can change the bin size and usually one needs to tweak it so that the distribution is displayed nicely.

```
# let's quickly view the 2013 data
df_can['2013'].head()
```

```
Country
India                                                33087
China                                                34129
United Kingdom of Great Britain and Northern Ireland  5827
Philippines                                          29544
Pakistan                                             12603
Name: 2013, dtype: int64
```

```
# np.histogram returns 2 values
count, bin_edges = np.histogram(df_can['2013'])

print(count) # frequency count
print(bin_edges) # bin ranges, default = 10 bins
```

```
[178  11   1   2   0   0   0   0   1   2]
[    0.   3412.9  6825.8 10238.7 13651.6 17064.5 20477.4 23890.3 27303.2
 30716.1 34129. ]
```

By default, the histogram method breaks up the dataset into 10 bins. The figure below summarizes the bin ranges and the frequency distribution of immigration in 2013. We can see that in 2013:

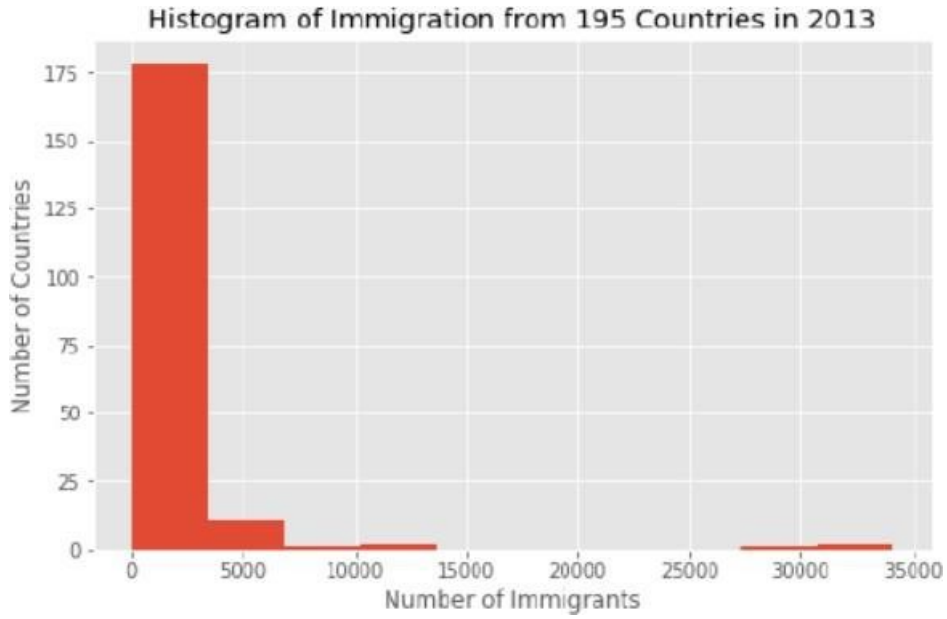
178 Countries contributed between 0 to 3412.9 immigrants

11 Countries contributed between 3412.9 to 6825.8 immigrants

1 Country contributed between 6285.8 to 10238.7 immigrants, and so on.

	Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
Range	0. to 3412.9	3412.9 to 6825.8	6825.8 to 10238.7	10238.7 to 13651.6	13651.6 to 17064.5	17064.5 to 20477.4	20477.4 to 23890.3	23890.3 to 27303.2	27303.2 to 30716.1	30716.1 to 34129.
Frequency	178	11	1	2	0	0	0	0	1	2

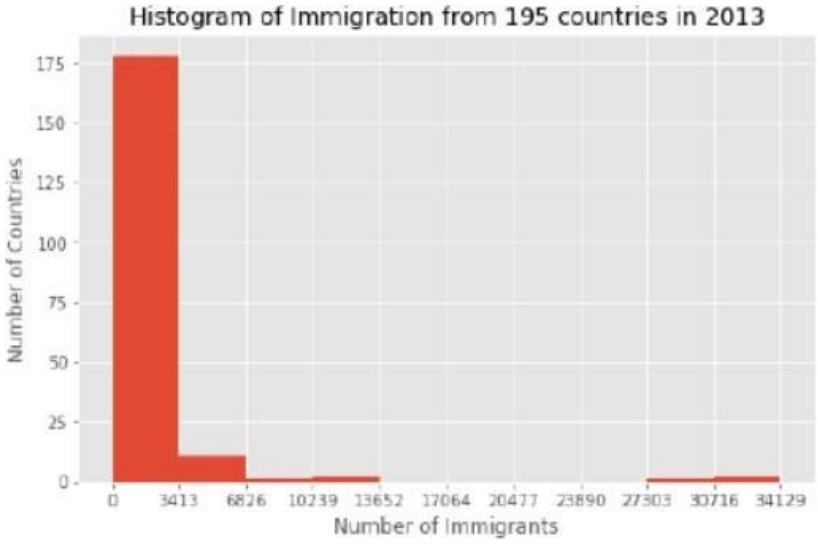
```
df_can['2013'].plot(kind='hist', figsize=(8, 5))

plt.title('Histogram of Immigration from 195 Countries in 2013') # add a title to the histogram
plt.ylabel('Number of Countries') # add y-label
plt.xlabel('Number of Immigrants') # add x-label

plt.show()
```



In the above plot, the x-axis represents the population range of immigrants in intervals of 3412.9. The y-axis represents the number of countries that contributed to the population.

Notice that the x-axis labels do not match with the bin size. This can be fixed by passing in a xticks keyword that contains the list of the bin sizes, as follows:

```
# 'bin_edges' is a list of bin intervals
count, bin_edges = np.histogram(df_can['2013'])

df_can['2013'].plot(kind='hist', figsize=(8, 5), xticks=bin_edges)

plt.title('Histogram of Immigration from 195 countries in 2013') # add a title to the histogram
plt.ylabel('Number of Countries') # add y-label
plt.xlabel('Number of Immigrants') # add x-label

plt.show()
```



Specialized Visualization Tools using Matplotlib

Pie Charts

A pie chart is a circular graphic that displays numeric proportions by dividing a circle (or pie) into proportional slices. You are most likely already familiar with pie charts as it is widely used in business and media. We can create pie charts in Matplotlib by passing in the kind=pie keyword.

Let's use a pie chart to explore the proportion (percentage) of new immigrants grouped by continents for the entire time period from 1980 to 2013. We can continue to use the same dataframe further.

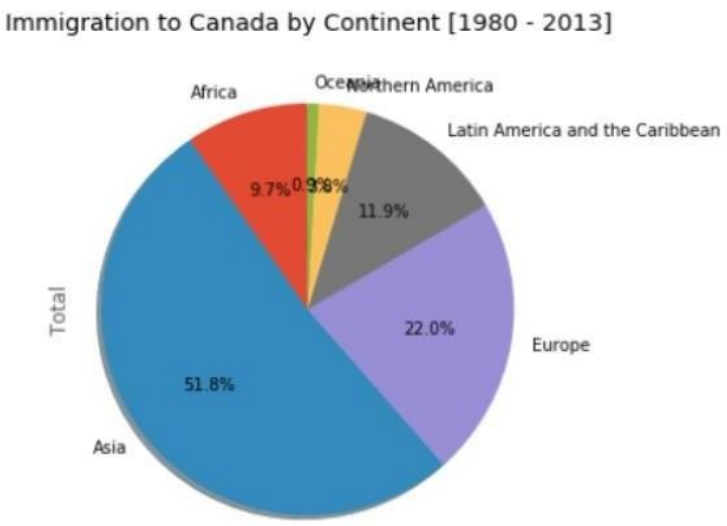
```
# group countries by continents and apply sum() function
df_continents = df_can.groupby('Continent', axis=0).sum()

# note: the output of the groupby method is a `groupby' object.
# we can not use it further until we apply a function (eg .sum())
print(type(df_can.groupby('Continent', axis=0)))

df_continents.head()
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

Continent	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2005	2006	2007	2008	2009	2010	2011	2012
Africa	3951	4363	3819	2671	2639	2650	3782	7494	7552	9894	...	27523	29188	28284	29890	34534	40892	35441	38083
Asia	31025	34314	30214	24696	27274	23850	28739	43203	47454	60256	...	159253	149054	133459	139894	141434	163845	146894	152218
Europe	39760	44802	42720	24638	22287	20844	24370	46698	54726	60893	...	35955	33053	33495	34692	35078	33425	26778	29177
Latin America and the Caribbean	13081	15215	16769	15427	13678	15171	21179	28471	21924	25060	...	24747	24676	26011	26547	26867	28818	27856	27173
Northern America	9378	10030	9074	7100	6661	6543	7074	7705	6469	6790	...	8394	9613	9463	10190	8995	8142	7677	7892

5 rows × 35 columns

```
# autopct create %, start angle represent starting point
df_continents['Total'].plot(kind='pie',
                            figsize=(5, 6),
                            autopct='%1.1f%%', # add in percentages
                            startangle=90,     # start angle 90° (Africa)
                            shadow=True,       # add shadow
                            )

plt.title('Immigration to Canada by Continent [1980 - 2013]')
plt.axis('equal') # Sets the pie chart to look like a circle.

plt.show()
```



The above visual is not very clear, the numbers and text overlap in some instances.
Let's make a few modifications to improve the visuals:

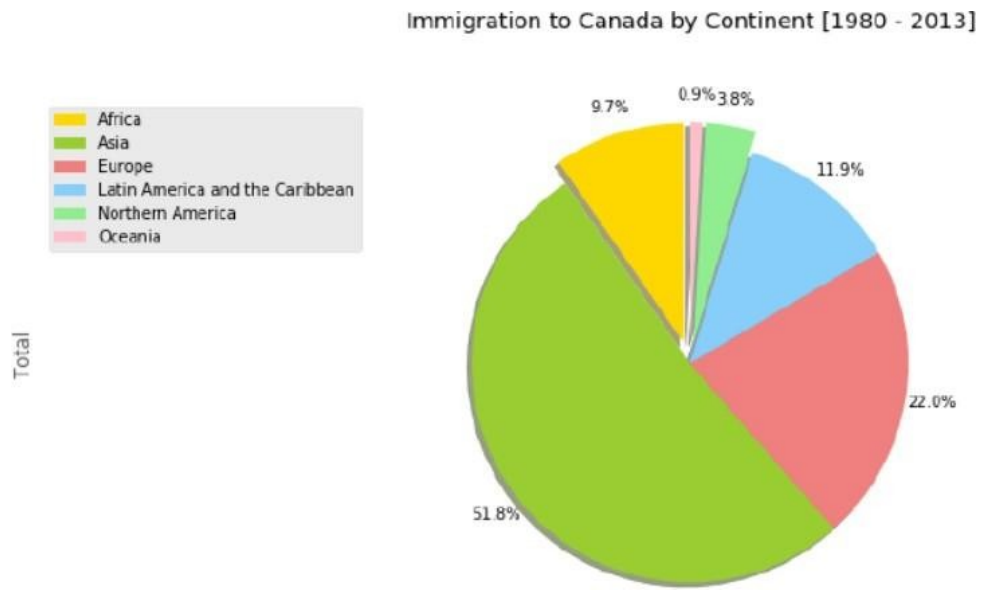
```
colors_list = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'lightgreen', 'pink']
explode_list = [0.1, 0, 0, 0, 0.1, 0.1] # ratio for each continent with which to offset each wedge.

df_continents['Total'].plot(kind='pie',
                            figsize=(15, 6),
                            autopct='%1.1f%%',
                            startangle=90,
                            shadow=True,
                            labels=None,         # turn off labels on pie chart
                            pctdistance=1.12,    # the ratio between the center of each pie slice and the start of the text g
                            colors=colors_list,  # add custom colors
                            explode=explode_list # 'explode' lowest 3 continents
                            )

# scale the title up by 12% to match pctdistance
plt.title('Immigration to Canada by Continent [1980 - 2013]', y=1.12)

plt.axis('equal')

# add legend
plt.legend(labels=df_continents.index, loc='upper left')

plt.show()
```

Raw code :

colors_list = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'lightgreen', 'pink'] explode_list = [0.1, 0, 0, 0, 0.1, 0.1] # ratio for each continent with which to offset each wedge.

df_continents['Total'].plot(kind='pie',
figsize=(15, 6),
autopct='%1.1f%%',
startangle=90,
shadow=True,
labels=None, # turn off labels on pie chart

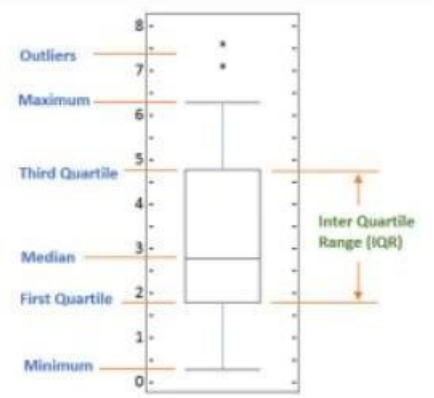
```
pctdistance=1.12, # the ratio between the center of each pie slice and the start of the text generated by autopct
colors=colors_list,  # add custom colors
explode=explode_list # 'explode' lowest 3 continents)

# scale the title up by 12% to match pctdistance
plt.title('Immigration to Canada by Continent [1980 - 2013]', y=1.12)
plt.axis('equal')
# add legend
plt.legend(labels=df_continents.index, loc='upper left')

plt.show()
```



Box Plot

A box plot is a way of statistically representing the distribution of the data through five main dimensions :

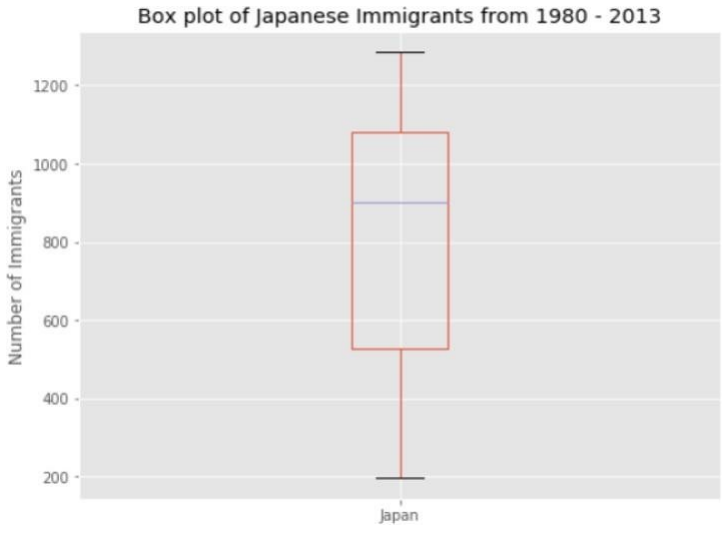
Minimum: Smallest number in the dataset.
First quartile: Middle number between the minimum and the median.
Second quartile (Median): Middle number of the (sorted) dataset.
Third quartile: Middle number between median and maximum.
Maximum: Highest number in the dataset.




```
# to get a dataframe, place extra square brackets around 'Japan'.
df_japan = df_can.loc[['Japan'], years].transpose()
df_japan.head()
```

Country	Japan
1980	701
1981	756
1982	598
1983	309
1984	246

```
df_japan.plot(kind='box', figsize=(8, 6))

plt.title('Box plot of Japanese Immigrants from 1980 - 2013')
plt.ylabel('Number of Immigrants')

plt.show()
```



We can immediately make a few key observations from the plot above:

The minimum number of immigrants is around 200 (min), maximum number is around 1300 (max), and median number of immigrants is around 900 (median).

25% of the years for period 1980 - 2013 had an annual immigrant count of ~500 or fewer (First quartile).

75% of the years for period 1980 - 2013 had an annual immigrant count of ~1100 or fewer (Third quartile).

We can view the actual numbers by calling the describe() method on the dataframe.

```
df_japan.describe()
```

Country	Japan
count	34.000000
mean	814.911765
std	337.219771
min	198.000000
25%	529.000000
50%	902.000000
75%	1079.000000
max	1284.000000

Scatter Plots

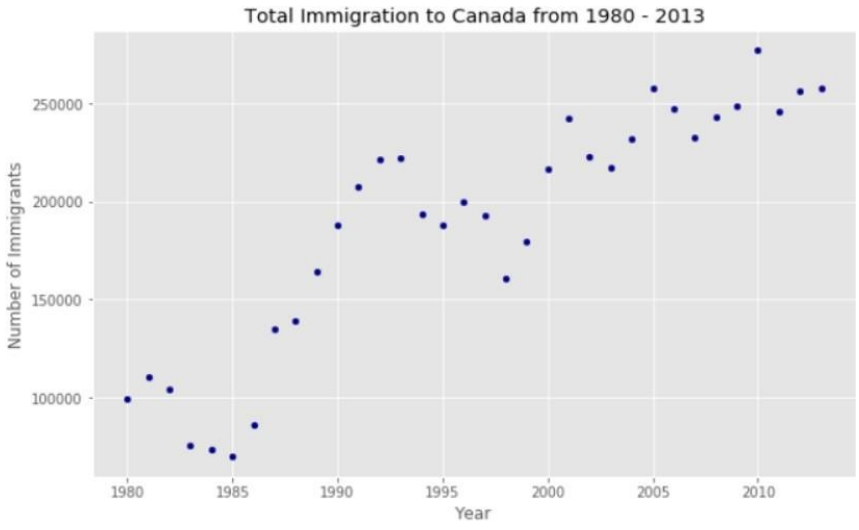
A scatter plot (2D) is a useful method of comparing variables against each other. Scatter plots look similar to line plots in that they both map independent and dependent variables on a 2D graph. While the datapoints are connected by a line in a line plot, they are not connected in a scatter plot. The data in a scatter plot is considered to express a trend. With further analysis using tools like regression, we can mathematically calculate this relationship and use it to predict trends outside the dataset.

Using a scatter plot, let's visualize the trend of total immigration to Canada (all countries combined) for the years 1980 - 2013.

```
# we can use the sum() method to get the total population per year
df_tot = pd.DataFrame(df_can[years].sum(axis=0))

# change the years to type int (useful for regression later on)
df_tot.index = map(int, df_tot.index)

# reset the index to put in back in as a column in the df_tot dataframe
df_tot.reset_index(inplace = True)

# rename columns
df_tot.columns = ['year', 'total']

# view the final dataframe
df_tot.head()
```

	year	total
0	1980	99137
1	1981	110563
2	1982	104271
3	1983	75550
4	1984	73417

```
df_tot.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')

plt.title('Total Immigration to Canada from 1980 - 2013')
plt.xlabel('Year')
plt.ylabel('Number of Immigrants')

plt.show()
```



So, let's try to plot a linear line of best fit, and use it to predict the number of immigrants in 2015.

Step 1: Get the equation of line of best fit. We will use Numpy's polyfit() method by passing in the following:

x: x-coordinates of the data.
y: y-coordinates of the data.

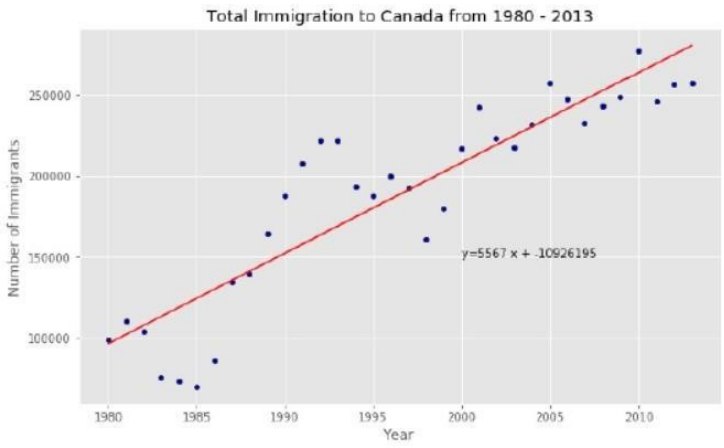
deg: Degree of fitting polynomial. 1 = linear, 2 = quadratic, and so on.

```
x = df_tot['year']      # year on x-axis
y = df_tot['total']     # total on y-axis
fit = np.polyfit(x, y, deg=1)

fit
```

```
array([ 5.56709228e+03, -1.09261952e+07])
```

Plot the regression line on the scatter plot.

```
df_tot.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')

plt.title('Total Immigration to Canada from 1980 - 2013')
plt.xlabel('Year')
plt.ylabel('Number of Immigrants')

# plot line of best fit
plt.plot(x, fit[0] * x + fit[1], color='red') # recall that x is the Years
plt.annotate('y={0:.0f} x + {1:.0f}'.format(fit[0], fit[1]), xy=(2000, 150000))

plt.show()

# print out the line of best fit
'No. Immigrants = {0:.0f} * Year + {1:.0f}'.format(fit[0], fit[1])
```



'No. Immigrants = 5567 * Year + -10926195'

Bubble Plots

A bubble plot is a variation of the scatter plot that displays three dimensions of data (x, y, z). The datapoints are replaced with bubbles, and the size of the bubble is determined by the third variable 'z', also known as the weight. In maplotlib, we can pass in an array or scalar to the keyword s to plot(), that contains the weight of each point.

Let us compare Argentina's immigration to that of its neighbor Brazil. Let's do that using a bubble plot of immigration from Brazil and Argentina for the years 1980 - 2013. We will set the weights for the bubble as the normalized value of the population for each year.

```
df_can_t = df_can[years].transpose() # transposed dataframe

# cast the Years (the index) to type int
df_can_t.index = map(int, df_can_t.index)

# let's label the index. This will automatically be the column name when we reset the index
df_can_t.index.name = 'Year'

# reset index to bring the Year in as a column
df_can_t.reset_index(inplace=True)

# view the changes
df_can_t.head()
```

Country	Year	Afghanistan	Albania	Algeria	American Samoa	Andorra	Angola	Antigua and Barbuda	Argentina	Armenia	...	United States of America	Uruguay	Uzbekistan	Vanuatu	Venezuela (Bolivarian Republic of)
0	1980	16	1	80	0	0	1	0	368	0	...	9378	128	0	0	103
1	1981	39	0	67	1	0	3	0	426	0	...	10030	132	0	0	117
2	1982	39	0	71	0	0	6	0	626	0	...	9074	146	0	0	174
3	1983	47	0	69	0	0	6	0	241	0	...	7100	105	0	0	124
4	1984	71	0	63	0	0	4	42	237	0	...	6661	90	0	0	142

5 rows × 196 columns

Create the normalized weights

There are several methods of normalizations in statistics, each with its own use. In this case, we will use feature scaling to bring all values into the range [0,1]. The general formula is:

where X is an original value, X' is the normalized value. The formula sets the max value in the dataset to 1, and sets the min value to 0. The rest of the datapoints are scaled to a value between 0-1 accordingly.

```
# normalize Brazil data
norm_brazil = (df_can_t['Brazil'] - df_can_t['Brazil'].min()) / (df_can_t['Brazil'].max() - df_can_t['Brazil'].min())

# normalize Argentina data
norm_argentina = (df_can_t['Argentina'] - df_can_t['Argentina'].min()) / (df_can_t['Argentina'].max() - df_can_t['Argentina']
```

Raw Code :

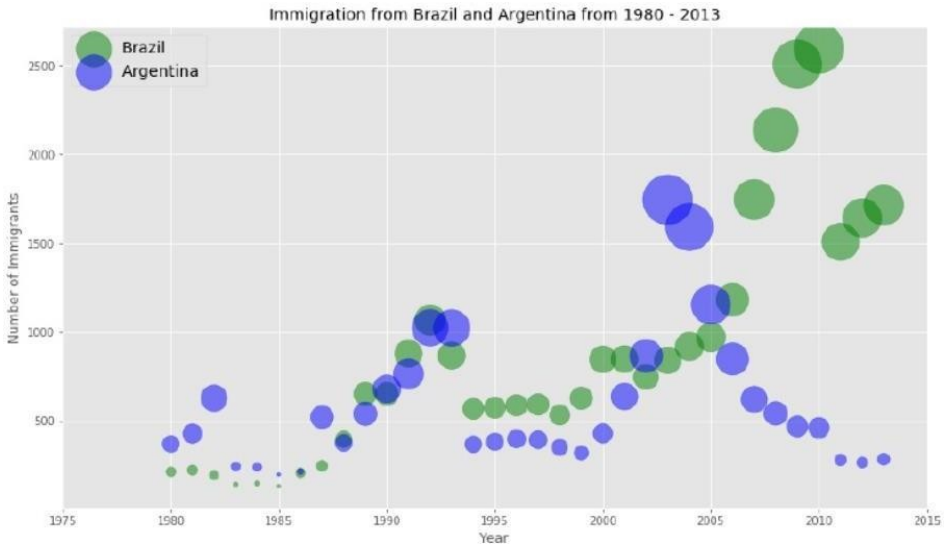
```
# normalize Brazil data
norm_brazil = (df_can_t['Brazil'] - df_can_t['Brazil'].min()) / (df_can_t['Brazil'].max() - df_can_t['Brazil'].min())

# normalize Argentina data
norm_argentina = (df_can_t['Argentina'] - df_can_t['Argentina'].min()) / (df_can_t['Argentina'].max() - df_can_t['Argentina'].min()
```

```
# Brazil
ax0 = df_can_t.plot(kind='scatter',
                    x='Year',
                    y='Brazil',
                    figsize=(14, 8),
                    alpha=0.5,                  # transparency
                    color='green',
                    s=norm_brazil * 2000 + 10,  # pass in weights
                    xlim=(1975, 2015)
                   )

# Argentina
ax1 = df_can_t.plot(kind='scatter',
                    x='Year',
                    y='Argentina',
                    alpha=0.5,
                    color="blue",
                    s=norm_argentina * 2000 + 10,
                    ax = ax0
                   )

ax0.set_ylabel('Number of Immigrants')
ax0.set_title('Immigration from Brazil and Argentina from 1980 - 2013')
ax0.legend(['Brazil', 'Argentina'], loc='upper left', fontsize='x-large')
```

Raw Code :

```
# Brazil
ax0 = df_can_t.plot(kind='scatter',
x='Year',
y='Brazil',
figsize=(14, 8),
alpha=0.5, # transparency
```

```
color='green',
s=norm_brazil * 2000 + 10, # pass in weights
xlim=(1975, 2015)
)

# Argentina
ax1 = df_can_t.plot(kind='scatter',
x='Year',
y='Argentina',

                    alpha=0.5,
color="blue",
s=norm_argentina * 2000 + 10,
ax = ax0
)

ax0.set_ylabel('Number of Immigrants')
ax0.set_title('Immigration from Brazil and Argentina from 1980 - 2013')
ax0.legend(['Brazil', 'Argentina'], loc='upper left', fontsize='x-large')
```



The size of the bubble corresponds to the magnitude of immigrating population for that year, compared to the 1980 - 2013 data. The larger the bubble, the more immigrants in that year.

Waffle Chart

A waffle chart is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an Excel dashboard.

```
# let's create a new dataframe for these three countries
df_dsn = df_can.loc[['Denmark', 'Norway', 'Sweden'], :]

# let's take a look at our dataframe
df_dsn
```

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Denmark	Europe	Northern Europe	Developed regions	272	293	299	106	93	73	93	...	62	101	97	108	81	92	93	94	81	3901
Norway	Europe	Northern Europe	Developed regions	116	77	106	51	31	54	56	...	57	53	73	66	75	46	49	53	59	2327
Sweden	Europe	Northern Europe	Developed regions	281	308	222	176	128	158	187	...	205	139	193	165	167	159	134	140	140	5866

The first step into creating a waffle chart is determing the proportion of each category with respect to the total.

```
# compute the proportion of each category with respect to the total
total_values = sum(df_dsn['Total'])
category_proportions = [(float(value) / total_values) for value in df_dsn['Total']]

# print out proportions
for i, proportion in enumerate(category_proportions):
    print (df_dsn.index.values[i] + ': ' + str(proportion))
```

```
Denmark: 0.32255663965602777
Norway: 0.1924094592359848
Sweden: 0.48503390110798744
```

The second step is defining the overall size of the waffle chart.

```
width = 40 # width of chart
height = 10 # height of chart

total_num_tiles = width * height # total number of tiles

print ('Total number of tiles is ', total_num_tiles)
```

```
Total number of tiles is  400
```

The third step is using the proportion of each category to determe it respective number of tiles

```
# compute the number of tiles for each catagory
tiles_per_category = [round(proportion * total_num_tiles) for proportion in category_proportions]

# print out number of tiles per category
for i, tiles in enumerate(tiles_per_category):
    print (df_dsn.index.values[i] + ': ' + str(tiles))
```

```
Denmark: 129
Norway: 77
Sweden: 194
```

The fourth step is creating a matrix that resembles the waffle chart and populating it.

```
# initialize the waffle chart as an empty matrix
waffle_chart = np.zeros((height, width))

# define indices to loop through waffle chart
category_index = 0
tile_index = 0

# populate the waffle chart
for col in range(width):
    for row in range(height):
        tile_index += 1

        # if the number of tiles populated for the current category is equal to its corresponding alloosted tiles...
        if tile_index > sum(tiles_per_category[0:category_index]):
            # ...proceed to the next category
            category_index += 1

        # set the class value to an integer, which increases with class
        waffle_chart[row, col] = category_index

print ('Waffle chart populated!')
```

```
Waffle chart populated!
```

Raw Code :

```
# initialize the waffle chart as an empty matrix
waffle_chart = np.zeros((height, width))

# define indices to loop through waffle chart
category_index = 0
tile_index = 0

# populate the waffle chart
for col in range(width):
for row in range(height):
tile_index += 1


# if the number of tiles populated for the current category is equal to its
corresponding allocated tiles...
    if tile_index > sum(tiles_per_category[0:category_index]):
```

```
    # ...proceed to the next category
    category_index += 1

# set the class value to an integer, which increases with class
waffle_chart[row, col] = category_index

print ('Waffle chart populated!')
```

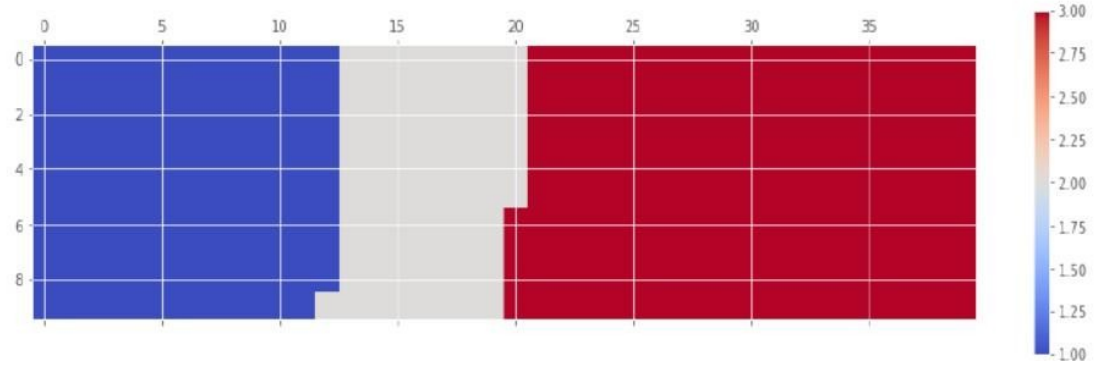
```
waffle_chart
```

```
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2.,
        2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2.,
        2., 2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
        3., 3., 3., 3., 3., 3., 3., 3.]])
```

Map the waffle chart matrix into a visual.

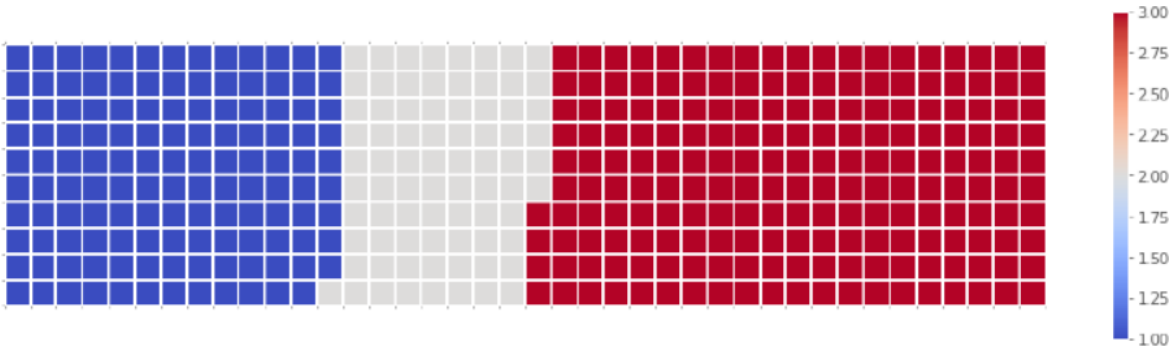
```
# instantiate a new figure object
fig = plt.figure()

# use matshow to display the waffle chart
colormap = plt.cm.coolwarm
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x6076cc0>

<Figure size 432x288 with 0 Axes>
```



Prettify the chart.

```
# instantiate a new figure object
fig = plt.figure()

# use matshow to display the waffle chart
colormap = plt.cm.coolwarm
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()

# get the axis
ax = plt.gca()

# set minor ticks
ax.set_xticks(np.arange(-.5, (width), 1), minor=True)
ax.set_yticks(np.arange(-.5, (height), 1), minor=True)

# add gridlines based on minor ticks
ax.grid(which='minor', color='w', linestyle='-', linewidth=2)

plt.xticks([])
plt.yticks([])
```



Word Clouds

Word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.

```
# import package and its set of stopwords
from wordcloud import WordCloud, STOPWORDS

print ('Wordcloud is installed and imported!')
```

```
Wordcloud is installed and imported!
```

```
# open the file and read it into a variable alice_novel
alice_novel = open('alice_novel.txt', 'r').read()

print ('File downloaded and saved!')
```

```
File downloaded and saved!
```

```
stopwords = set(STOPWORDS)
```

```
# instantiate a word cloud object
alice_wc = WordCloud(
    background_color='white',
    max_words=2000,
    stopwords=stopwords
)

# generate the word cloud
alice_wc.generate(alice_novel)
```

```
# display the word cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Interesting! So, in the first 2000 words in the novel, the most common words are Alice, said, little, Queen, and so on. Let's resize the cloud so that we can see the less frequent words a little better.

```
fig = plt.figure()
fig.set_figwidth(14) # set width
fig.set_figheight(18) # set height

# display the cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Much better! However, said isn't really an informative word. So, let's add it to our stop words and re-generate the cloud.

```
stopwords.add('said') # add the words said to stopwords

# re-generate the word cloud
alice_wc.generate(alice_novel)

# display the cloud
fig = plt.figure()
fig.set_figwidth(14) # set width
fig.set_figheight(18) # set height

plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

UNIT-V

Introduction to Seaborn

Seaborn is a statistical plotting library and is built on top of matplotlib. It has beautiful default styles and is compatible with pandas dataframe objects. In order to install it use the following commands:

Anaconda users: conda install seaborn

Python users: pip install seaborn

Seaborn code is opensource so one can read it at

https://github.com/mwaskom/seaborn. This page has information along with the link to the official documentation page - https://seaborn.pydata.org/. The subsection of this page (https://seaborn.pydata.org/examples/index.html) shows the example visualizations Seaborn is able to work with. The other important section to visit is the one which has API information - https://seaborn.pydata.org/api.html.

Seaborn functionalities and usage

Let us now delve into some of the functionalities Seaborn provides. We will run some snippet of codes in Jupyter notebook (although you can use any other IDE) to exhibit the key features.

Distribution Plots

We will first try to do distribution plots. To get started, we first import one of the standard datasets that comes with Seaborn. The one we choose for our exercise is diamonds.csv. You can pick other datasets from https://github.com/mwaskom/seaborn-data.

```
###### By convention we import Seaborn as sns
In [1]: import seaborn as sns
```

The below command will allow us to see the visualizations inline

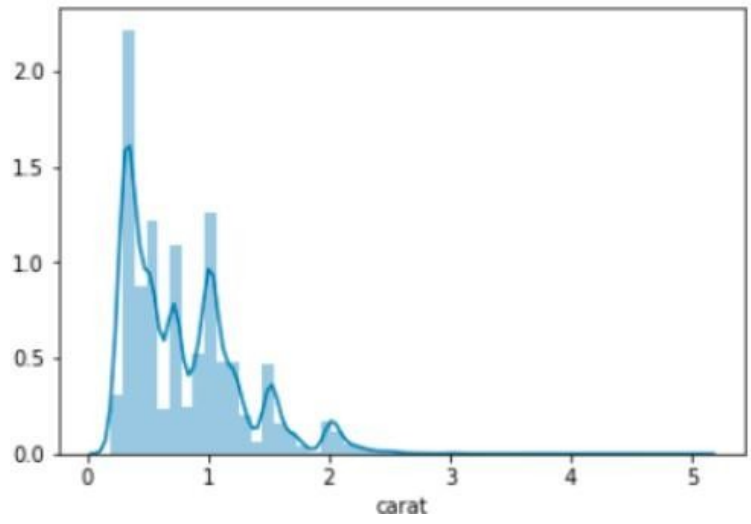
```
In [2]: %matplotlib inline

In [3]: dmd = sns.load_dataset('diamonds')

In [4]: dmd.head()
```

Out[4]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

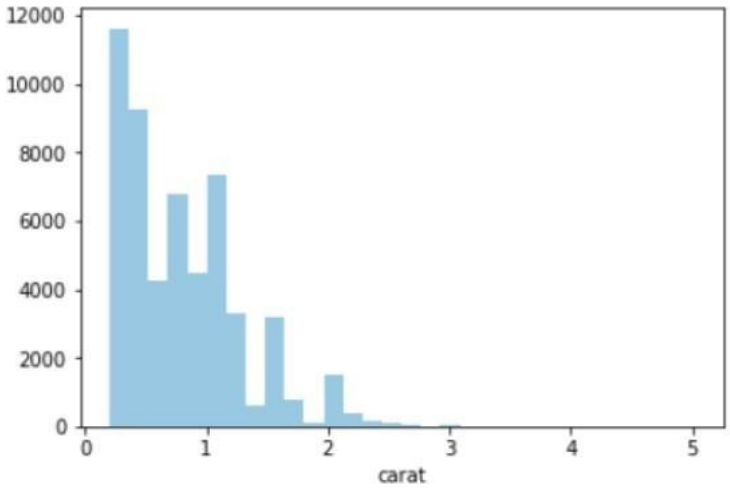
We then use the “displot” function to plot the distribution of a single variable

```
In [6]: sns.distplot(dmd['carat'])
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0xb0112e8>
```



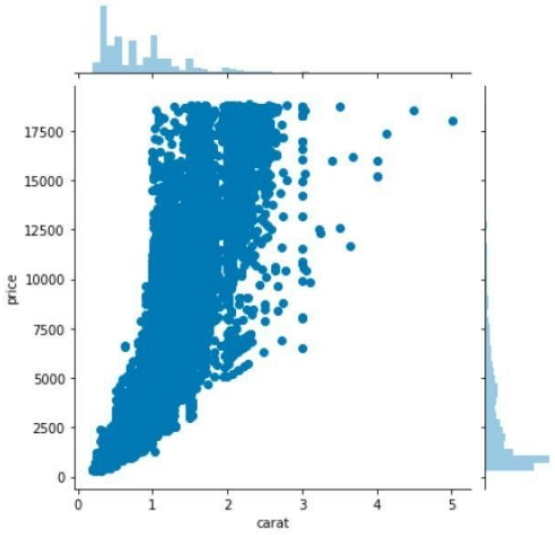
As seen above, we get a histogram and a Kernel Density Estimate (KDE) plot. We can customize it further by removing KDE and specifying the number of bins.

```
In [13]: sns.distplot(dmd['carat'],kde=False,bins=30)
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0xcb67128>
```



Joint plot allows us to plot the relationship represented by bivariate data.

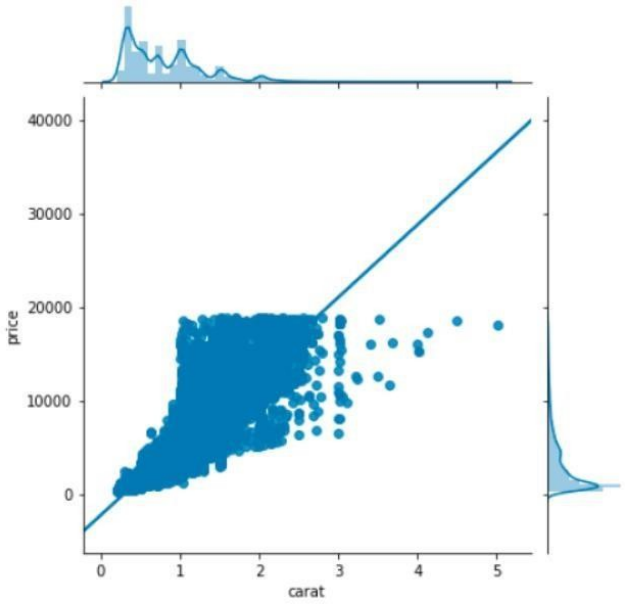
```
In [16]: sns.jointplot(x='carat',y='price',data=dmd,kind='scatter')
Out[16]: <seaborn.axisgrid.JointGrid at 0xcd530b8>
```



The above diagram shows that as the carat approaches the value '5', the value of the diamond also increases which is a phenomenon we also observe. The 'jointplot' function can take multiple values for the parameter 'kind' – scatter, reg, resid, kde, hex.
Let us see the plot using 'reg' which will provide regression and kernel density fits.

```
sns.jointplot(x='carat',y='price',data=dmd,kind='reg')
```

```
<seaborn.axisgrid.JointGrid at 0xd1f2438>
```



Next Steps for students to try :

1. Pairplot function: will plot pairwise relationships across an entire dataframe such that each numerical variable will be depicted in the y-axis across a single row and in the x-axis across a single column. Try the command: sns.pairplot(dmd)
2. Rugplot function: It plots datapoints in an array as sticks on an axis. Try the command: sns.rugplot(dmd['price'])
3. Once you are comfortable with these then you can try out KDE plotting. KDE plotting is used for visualizing the probability density of a continuous variable or a single graph for multiple samples.
   Reference: https://seaborn.pydata.org/generated/seaborn.kdeplot.html

Categorical Plots :

Now let us discuss how to use seaborn to plot categorical data. But let us first understand what categorical variable is. A categorical variable is one that has multiple categories but has no intrinsic ordering specified for categories. For example: Blood type of a person can be any one of A, B, AB or O.

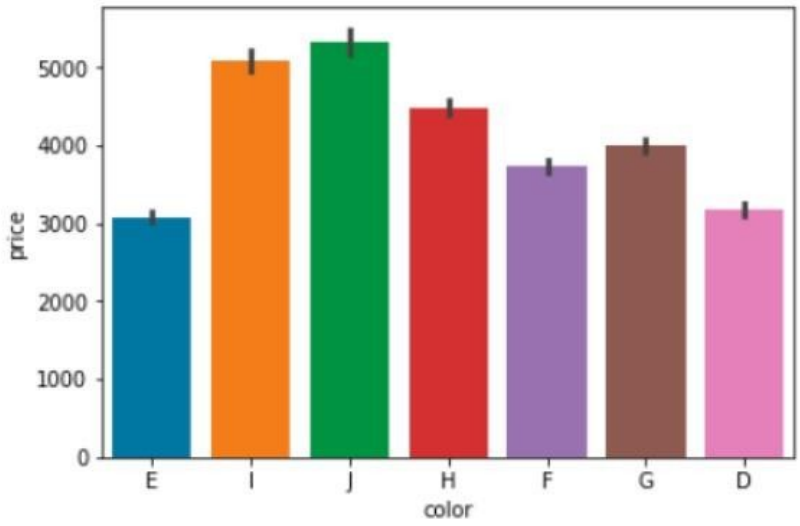
Now let us see examples of the plots:

Barplot and countplot allow you to aggregate data with respective to each category. Barplot

allows to you aggregate around some function but the default is mean.

```
sns.barplot(x='color',y='price',data=dmd)
```

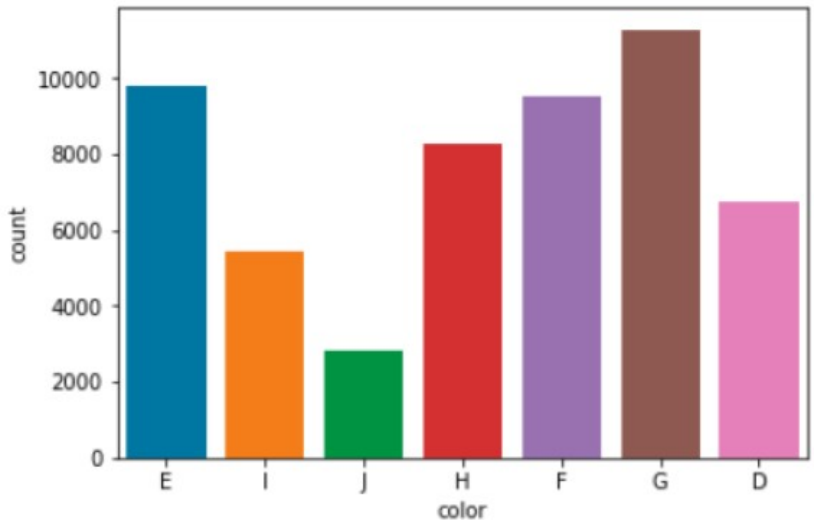
```
<matplotlib.axes._subplots.AxesSubplot at 0x1655af98>
```



The difference between countplot and barplot is that countplot explicitly counts the number of occurrences.

```
sns.countplot(x='color',data=dmd)
```

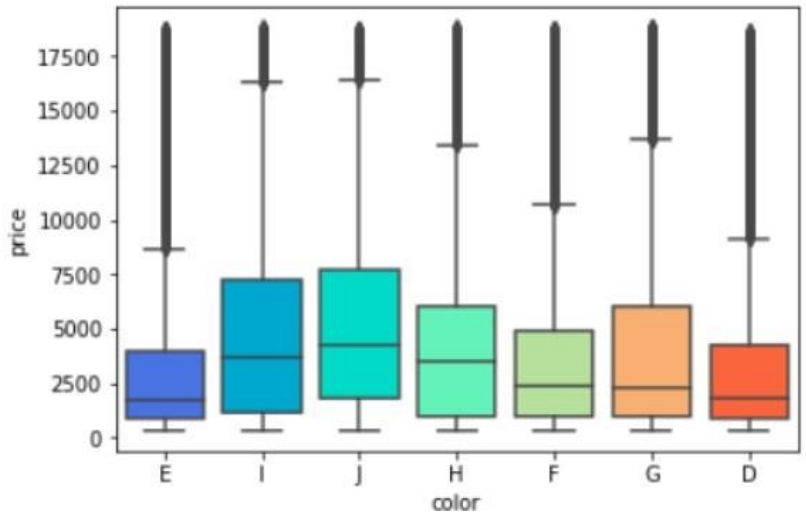
```
<matplotlib.axes._subplots.AxesSubplot at 0x16551860>
```



Boxplot shows the quartiles of the dataset while the whiskers extend encompass the rest of the distribution but leave out the points that are the outliers.

```
sns.boxplot(x="color", y="price", data=dmd,palette='rainbow')
```

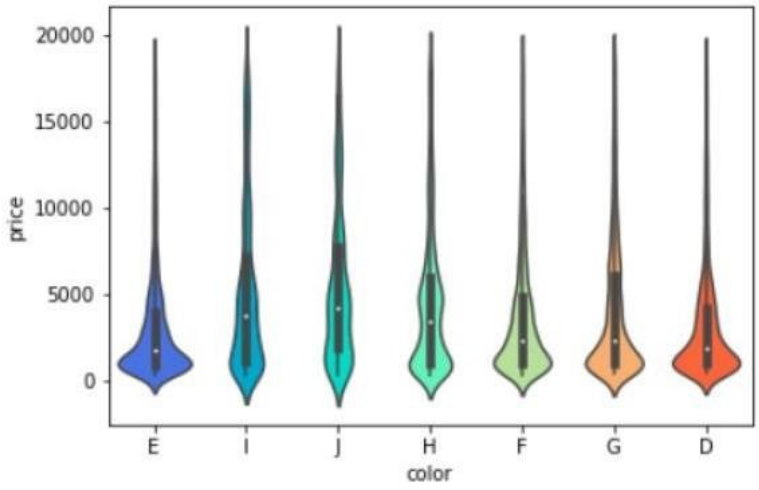
```
<matplotlib.axes._subplots.AxesSubplot at 0x16688358>
```



Violinplot shows the distribution of data across several levels of categorical variable(s) thus helping in comparison of the distribution. Wherever actual datapoints are not present, KDE is used to estimate the remaining points.

```
sns.violinplot(x="color", y="price", data=dmd,palette='rainbow')
```

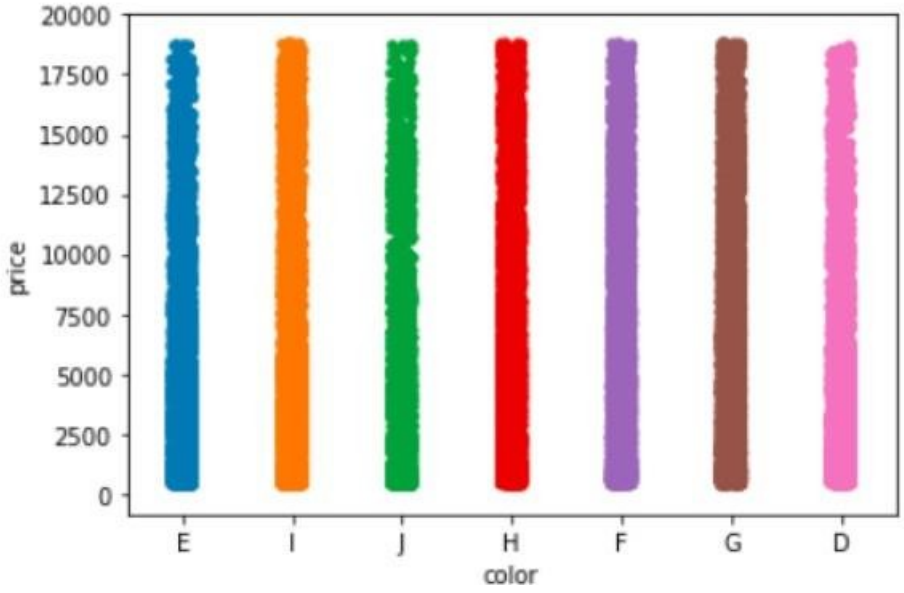
```
<matplotlib.axes._subplots.AxesSubplot at 0x16688470>
```



The stripplot draws a scatterplot where one variable is categorical.

```
sns.stripplot(x="color", y="price", data=dmd)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x177bc828>
```



Matrix Plots :

Now let us delve into matrix plots. It helps to segregate data into color-encoded matrices which can further help in unsupervised learning methods like clustering.

```
import seaborn as sns
```

```
%matplotlib inline
```

```
dmd = sns.load_dataset('diamonds')
```

```
dmd.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

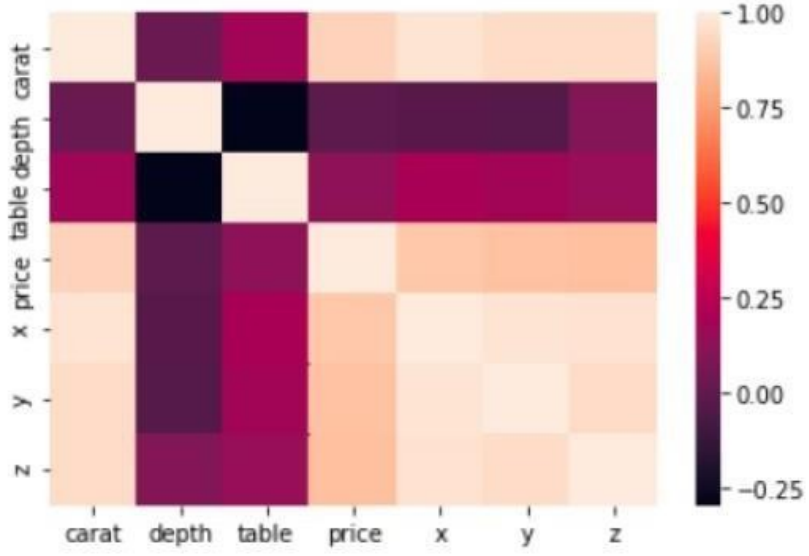
```
dmd.corr()
```

	carat	depth	table	price	x	y	z
carat	1.000000	0.028224	0.181618	0.921591	0.975094	0.951722	0.953387
depth	0.028224	1.000000	-0.295779	-0.010647	-0.025289	-0.029341	0.094924
table	0.181618	-0.295779	1.000000	0.127134	0.195344	0.183760	0.150929
price	0.921591	-0.010647	0.127134	1.000000	0.884435	0.865421	0.861249
x	0.975094	-0.025289	0.195344	0.884435	1.000000	0.974701	0.970772
y	0.951722	-0.029341	0.183760	0.865421	0.974701	1.000000	0.952006
z	0.953387	0.094924	0.150929	0.861249	0.970772	0.952006	1.000000

The corr() function gives the matrix form to correlation data.
Below command generates the heatmap.

```
sns.heatmap(dmd.corr())
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xc6bdef0>
```




```
flightdata = sns.load_dataset('flights')
flightdata.head()
```

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Let us now try the pivot_table formation. Now we need to select the appropriate data for that. Among the available datasets in seaborn, flights data is most suitable to depict this. Let us try to depict the total number of passengers for each month of the year.

```
flightdata = sns.load_dataset('flights')
flightdata.head()
```

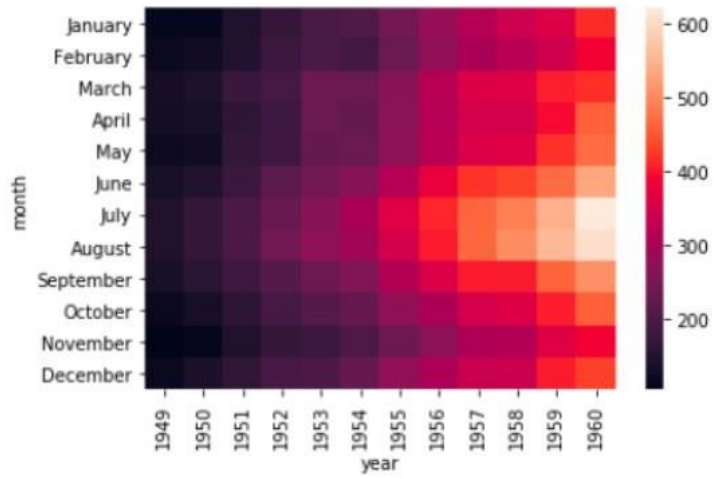
	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

```
flightdata.pivot_table(values='passengers',index='month',columns='year')
```

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
January	112	115	145	171	196	204	242	284	315	340	360	417
February	118	126	150	180	196	188	233	277	301	318	342	391
March	132	141	178	193	236	235	267	317	356	362	406	419
April	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
June	135	149	178	218	243	264	315	374	422	435	472	535
July	148	170	199	230	264	302	364	413	465	491	548	622
August	148	170	199	242	272	293	347	405	467	505	559	606
September	136	158	184	209	237	259	312	355	404	404	463	508
October	119	133	162	191	211	229	274	306	347	359	407	461
November	104	114	146	172	180	203	237	271	305	310	362	390
December	118	140	166	194	201	229	278	306	336	337	405	432

```
sns.heatmap(flightdata.pivot_table(values='passengers',index='month',columns='year'))
```

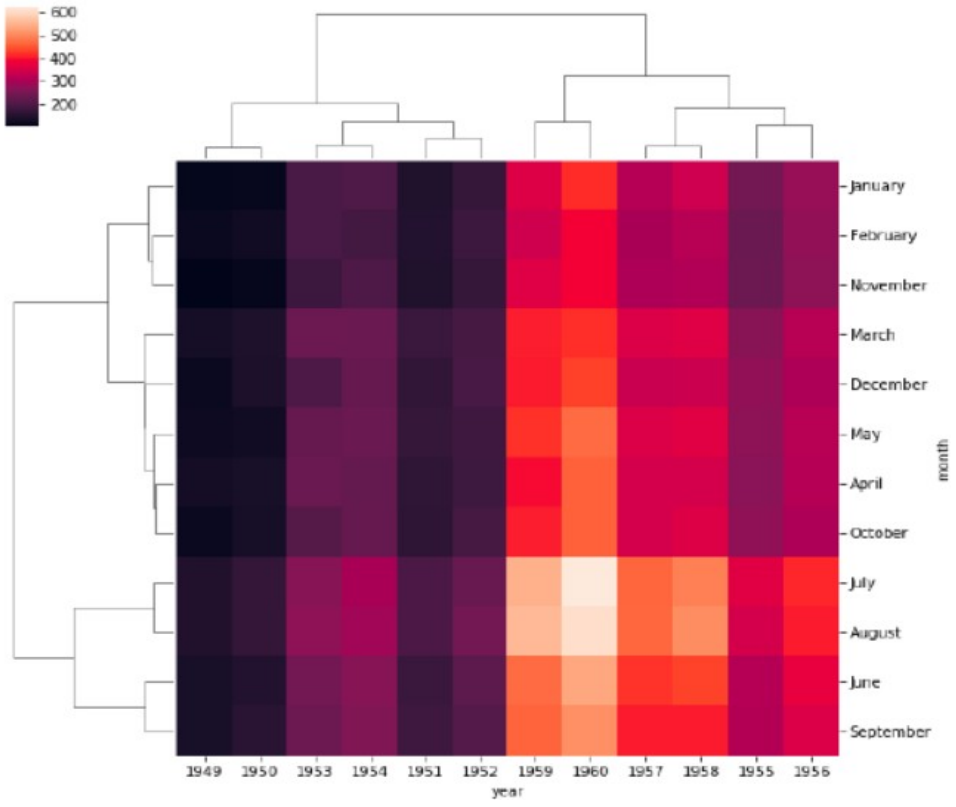
```
<matplotlib.axes._subplots.AxesSubplot at 0x83a84e0>
```



The cluster map uses hierarchal clustering. It no more depicts months and years in order but groups them similarity in the passenger count. So, it can be inferred that April and May are similar in passenger volume.

```
sns.clustermap(flightdata.pivot_table(values='passengers',index='month',columns='year'))
```

```
<seaborn.matrix.ClusterGrid at 0xb0dccc0>
```



Regression plot :

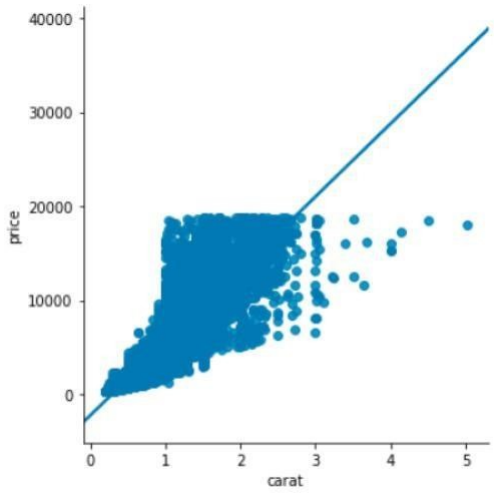
In this final section, we will explore seaborn and see how efficient it is to create regression lines and fits using this library. Implot function allows you to display linear models.

```
dmd.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

```
sns.lmplot(x='carat',y='price',data=dmd)
```

```
<seaborn.axisgrid.FacetGrid at 0xc5354a8>
```



Spatial Visualizations and Analysis in Python with Folium

Folium is a powerful data visualization library in Python that was built primarily to help people visualize geospatial data. With Folium, you can create a map of any location in the world if you know its latitude and longitude values. You can also create a map and superimpose markers as well as clusters of markers on top of the map for cool and very interesting visualizations. You can also create maps of different styles such as street level map, stamen map.

Folium is not available by default. So, we first need to install it before we can import it. We can use the command : conda install -c conda-forge folium=0.5.0 --yes

It is not available via default conda channel. Try using conda-forge channel to install folium as shown: conda install -c conda-forge folium

Generating the world map is straightforward in Folium. You simply create a Folium Map object and then you display it. What is attractive about Folium maps is that they are interactive, so you can zoom into any region of interest despite the initial zoom level.

```
import folium
print('Folium installed and imported!')
```

```
Folium installed and imported!
```

```
# define the world map
world_map = folium.Map()

# display world map
world_map
```

Go ahead. Try zooming in and out of the rendered map above. You can customize this default definition of the world map by specifying the center of your map and the initial zoom level. All locations on a map are defined by their respective Latitude and Longitude values. So, you can create a map and pass in a center of Latitude and Longitude values of [0, 0]. For a defined center, you can also define the initial zoom level into that location when the map is rendered. The higher the zoom level the more the map is zoomed into the center. Let's create a map centered around Canada and play with the zoom level to see how it affects the rendered map.

```
# define the world map centered around Canada with a low zoom level
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4)

# display world map
world_map
```

Let's create the map again with a higher zoom level

```
# define the world map centered around Canada with a higher zoom level
world_map = folium.Map(location=[56.130, -106.35], zoom_start=8)

# display world map
world_map
```

As you can see, the higher the zoom level the more the map is zoomed into the given center.

Stamen Toner Maps

These are high-contrast B+W (black and white) maps. They are perfect for data mashups and exploring river meanders and coastal zones. Let's create a Stamen Toner map of Canada with a zoom level of 4.

```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Toner')

# display map
world_map
```

Stamen Terrain Maps

These are maps that feature hill shading and natural vegetation colors. They showcase advanced labeling and linework generalization of dual-carriageway roads. Let's create a Stamen Terrain map of Canada with zoom level 4.

```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Terrain')

# display map
world_map
```

Mapbox Bright Maps

These are maps that are quite like the default style, except that the borders are not visible with a low zoom level. Furthermore, unlike the default style where country names are displayed in each country's native language, Mapbox Bright style displays all country names in English. Let's create a world map with this style.

Case Study

Now that you are familiar with folium, let us use it for our next case study which is as mentioned below:

Case Study: An e-commerce company ‘ wants to get into logistics “Deliver4U” . It wants to know the pattern for maximum pickup calls from different areas of the city throughout the day. This will result in:

i) Build optimum number of stations where its pickup delivery personnel will be located.
ii) Ensure pickup personnel reaches the pickup location at the earliest possible time.

For this the company uses its existing customer data in Delhi to find the highest density of probable pickup locations in the future.

Solution:

Pre-requisites : Python, Jupyter Notebooks, Pandas

Data set : Please download the following from the location specified by the trainer.

The dataset contains two separate data files – train_del.csv and test_del.csv. The difference is that train_del.csv contains additional column which is trip_duration which we will not be needed for our present analysis.

Importing and pre-processing data:

a) Import libraries – Pandas and Folium. Drop the trip_duration column and combine the 2 different files as one dataframe.

```
import pandas as pd
import folium
```

```
df_train = pd.read_csv('train_del.csv').drop(columns=['trip_duration', 'dropoff_datetime'])
```

```
df_train.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id2875421	2	2016-03-14 17:24:55	1	77.207845	28.637937	77.225370	28.635602	N
1	id2377394	1	2016-06-12 00:43:35	1	77.209585	28.608564	77.190519	28.601152	N
2	id3858529	2	2016-01-19 11:35:24	1	77.210973	28.633939	77.184667	28.580087	N
3	id3504673	2	2016-04-06 19:32:31	1	77.179960	28.589971	77.177732	28.576718	N
4	id2181028	2	2016-03-26 13:30:55	1	77.216947	28.663209	77.217077	28.652520	N

```
df_test=pd.read_csv('test_del.csv')
```

```
df_test.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id3004672	1	2016-06-30 23:59:58	1	77.201871	28.602029	77.199827	28.626680	N
1	id3505355	1	2016-06-30 23:59:53	1	77.225797	28.549993	77.230192	28.525403	N
2	id1217141	1	2016-06-30 23:59:47	1	77.192563	28.607583	77.203840	28.599523	N
3	id2150126	2	2016-06-30 23:59:41	1	77.233930	28.641900	77.203573	28.600469	N
4	id1598245	1	2016-06-30 23:59:33	1	77.219785	28.631475	77.228490	28.625890	N

```
df=pd.concat([df_train,df_test],sort=False,ignore_index=True)
```

```
df.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag
0	id2875421	2	2016-03-14 17:24:55	1	77.207845	28.637937	77.225370	28.635602	N
1	id2377394	1	2016-06-12 00:43:35	1	77.209585	28.608564	77.190519	28.601152	N
2	id3858529	2	2016-01-19 11:35:24	1	77.210973	28.633939	77.184667	28.580087	N
3	id3504673	2	2016-04-06 19:32:31	1	77.179960	28.589971	77.177732	28.576718	N
4	id2181028	2	2016-03-26 13:30:55	1	77.216947	28.663209	77.217077	28.652520	N

preview of the data we will be working with

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2083778 entries, 0 to 2083777
Data columns (total 9 columns):
id                  object
vendor_id           int64
pickup_datetime     object
passenger_count     int64
pickup_longitude    float64
pickup_latitude     float64
dropoff_longitude   float64
dropoff_latitude    float64
store_and_fwd_flag  object
dtypes: float64(4), int64(2), object(3)
memory usage: 143.1+ MB
```

```
df.pickup_datetime = pd.to_datetime(df.pickup_datetime, format='%Y-%m-%d %H:%M:%S')
```

```
pd.to_datetime(df.pickup_datetime)
```

```
0         2016-03-14 17:24:55
1         2016-06-12 00:43:35
2         2016-01-19 11:35:24
3         2016-04-06 19:32:31
4         2016-03-26 13:30:55
5         2016-01-30 22:01:40
6         2016-06-17 22:34:59
7         2016-05-21 07:54:58
8         2016-05-27 23:12:23
9         2016-03-10 21:45:01
10        2016-05-10 22:08:41
11        2016-05-15 11:16:11
12        2016-02-19 09:52:46
13        2016-06-01 20:58:29
14        2016-05-27 00:43:36
15        2016-05-16 15:29:02
```

We will need to generate some columns such as month or other time features using Datetime package of python. Let us then use it with Folium

```
df['month'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.month)
df['week'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.week)
df['day'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.day)
df['hour'] = pd.to_datetime(df.pickup_datetime).apply(lambda x: x.hour)
```

```
df.head()
```

	id	vendor_id	pickup_datetime	passenger_count	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	store_and_fwd_flag	month	w
0	id2875421	2	2016-03-14 17:24:55	1	77.207845	28.637937	77.225370	28.635602	N	3	
1	id2377394	1	2016-06-12 00:43:35	1	77.209585	28.608564	77.190519	28.681152	N	6	
2	id3858529	2	2016-01-19 11:35:24	1	77.210973	28.633939	77.184667	28.580087	N	1	
3	id3504673	2	2016-04-06 19:32:31	1	77.179960	28.589971	77.177732	28.576718	N	4	
4	id2181028	2	2016-03-26 13:30:55	1	77.216947	28.663209	77.217077	28.652520	N	3	

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2083778 entries, 0 to 2083777
Data columns (total 13 columns):
id                  object
vendor_id           int64
pickup_datetime     datetime64[ns]
passenger_count     int64
pickup_longitude    float64
pickup_latitude     float64
dropoff_longitude   float64
dropoff_latitude    float64
store_and_fwd_flag  object
month               int64
week                int64
day                 int64
hour                int64
dtypes: datetime64[ns](1), float64(4), int64(6), object(2)
memory usage: 206.7+ MB
```

Please note that month, week, day, hour columns will be used next for our analysis
Note the following regarding visualizing spatial data with Folium:
• Maps are defined as folium.Map object. We will need to add other objects on top of this before rendering

• Different map tiles for map rendered by Folium can be seen at

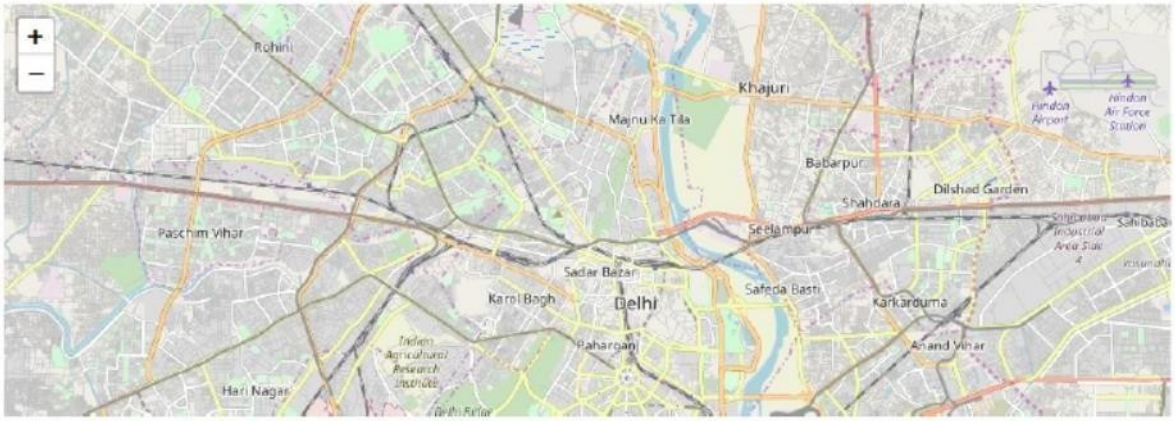
: https://github.com/pythonvisualization/folium/tree/master/folium/templates/tiles

• Folium.Map() : First thing to be executed when you work with Folium.

Let us define the default map object:

```
def generateBaseMap(default_location=[28.637937, 77.207845], default_zoom_start=12):
    base_map = folium.Map(location=default_location, control_scale=True, zoom_start=default_zoom_start)
    return base_map
```

```
base_map=generateBaseMap()
base_map
```



Let us now visualize the rides data using a class method called Heatmap()

Using data from May to June 2016 to generate the heat map

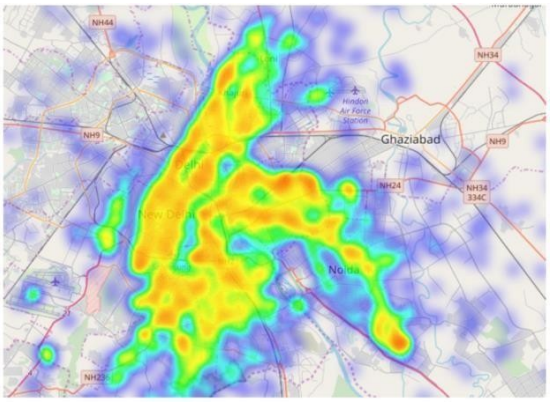
```
from folium.plugins import HeatMap
```

```
df_copy = df[df.month>4].copy()
df_copy['count'] = 1
base_map = generateBaseMap()
```

```
###HeatMap(data=df_copy[['pickup_latitude', 'pickup_longitude', 'count']].groupby(['pickup_latitude', 'pickup_longitude']).su
HeatMap(data=df_copy[['pickup_latitude', 'pickup_longitude', 'count']].groupby(['pickup_latitude', 'pickup_longitude']).sum()
```

```
<folium.plugins.heat_map.HeatMap at 0x25bcb2b0>
```

```
base_map
```



Code for reference:

```
from folium.plugins import HeatMap
df_copy = df[df.month>4].copy()
df_copy['count'] = 1
base_map = generateBaseMap()
HeatMap(data=df_copy[['pickup_latitude', 'pickup_longitude',
'count']].groupby(['pickup_latitude',
'pickup_longitude']).sum().reset_index().values.tolist(), radius=8,
max_zoom=13).add_to(base_map)
```

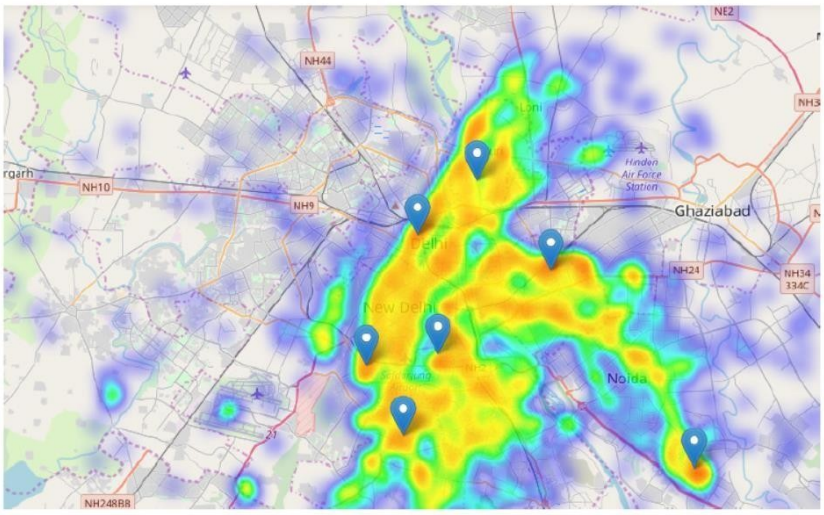
Interpretation of the output:

There is high demand for cabs in areas marked by the heat map which is central Delhi most probably and other surrounding areas.

Now let us add functionality to add markers to the map by using the folium.ClickForMarker() object.

After adding the below line of code, we can add markers on the map to recommends points where logistic pickup stops can be built

```
base_map.add_child(folium.ClickForMarker(popup='Potential Location'))
```



We can also animate our heat maps to dynamically change the data on timely basis based on a certain dimension of time. This can be done using HeatMapWithTime(). Use the following code :

df_hour_list = []

for hour in df_copy.hour.sort_values().unique():

df_hour_list.append(df_copy.loc[df_copy.hour == hour, ['pickup_latitude', 'pickup_longitude', 'count']].groupby(['pickup_latitude', 'pickup_longitude']).sum().reset_index().values.tolist())

from folium.plugins import HeatMapWithTime

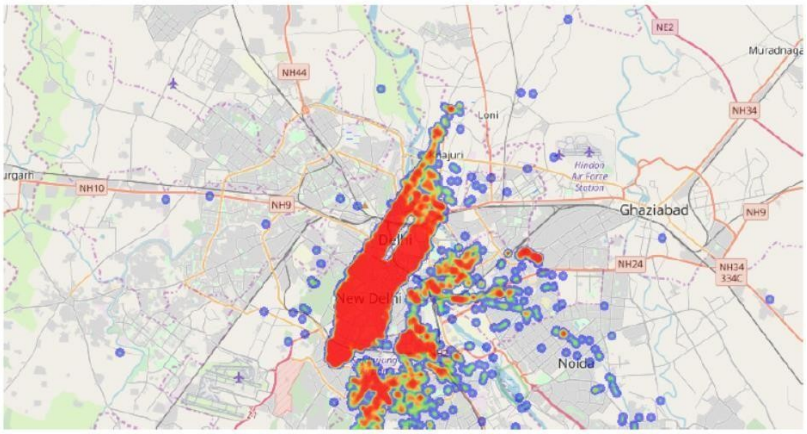
base_map = generateBaseMap(default_zoom_start=11)

HeatMapWithTime(df_hour_list, radius=5, gradient={0.2: 'blue', 0.4: 'lime', 0.6: 'orange', 1: 'red'}, min_opacity=0.5, max_opacity=0.8, use_local_extrema=True).add_to(base_map)

base_map

```
df_hour_list = []
for hour in df_copy.hour.sort_values().unique():
    df_hour_list.append(df_copy.loc[df_copy.hour == hour, ['pickup_latitude', 'pickup_longitude', 'count']].groupby(['pickup_
```

```
from folium.plugins import HeatMapWithTime
base_map = generateBaseMap(default_zoom_start=11)
HeatMapWithTime(df_hour_list, radius=5, gradient={0.2: 'blue', 0.4: 'lime', 0.6: 'orange', 1: 'red'}, min_opacity=0.5, max_op
base_map
```



Conclusion

Throughout the city, pickups are more probable from central area so better to set lot of pickup stops at these locations

Therefore, by using maps we can highlight trends and uncover patterns and derive insights from the data.