DIGITAL NOTES ON DATA VISUALIZATION

B.TECH II YEAR - II SEM
(2020-2021)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SYLLABUS

DATA VISUALIZATION

Course Objectives:

- To learn different statistical methods for Data visualization.
- To learn basics of R and Python.
- To learn usage of Watson studio.
- To learn about packages Numpy, pandas and matplotlib.
- To learn functionalities and usages of Seaborn.

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, Data visualization with R .

Data visualization in Watson Studio: Adding data to data refinery, 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.

UNIT IV

Data Visualization Tools in Python- Introduction to Matplotlib, Basic plots using matplotlib, Specialized Visualization Tools using Matplotlib, 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, Dreamtech Press.
2. R Graphics Essentials for Great Data Visualization by Alboukadel Kassambara

Course Outcomes:

At Completion of this course, students would be able to -

- Apply statistical methods for Data visualization.
- Gain knowledge on R and Python
- Understand usage of various packages in R and Python.
- Demonstrate knowledge of Watson studio.
- Apply data visualization tools on various data sets.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDEX

S. No	Unit	Topic	Page no
1	I	Introduction to Statistics, Difference between inferential statistics and descriptive statistics	1
2	I	Inferential Statistics- Drawing Inferences from Data, Random Variables, Normal Probability Distribution	4
3	I	Sampling, Sample Statistics and Sampling Distributions	9
4	I	Overview and About R, R and R studio Installation, Descriptive Data analysis using R	12
5	I	Description of basic functions used to describe data in R.	12

S. No	Unit	Topic	Page no
1	II	Data manipulation with R: Data manipulation packages	30
2	II	Data visualization with R	48
3	II	Data visualization in Watson Studio: Adding data to data refinery	55
4	II	Visualization of Data on Watson Studio	58

S. No	Unit	Topic	Page no
1	III	Python: Introduction to Python,	62
2	III	How to Install, Introduction to Jupyter Notebook,	66
3	III	Python scripting basics	68
4	III	Numpy	85
5	III	Pandas	88

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

S. No	Unit	Topic	Page no
1	IV	Data Visualization Tools in Python- Introduction to Matplotlib	107
2	IV	Basic plots using matplotlib	109
3	IV	Specialized Visualization Tools using Matplotlib	110
4	IV	Advanced Visualization Tools using Matplotlib- Waffle Charts, Word Clouds	114

S. No	Unit	Topic	Page no
1	V	Introduction to Seaborn: Seaborn functionalities and usage	116
2	V	Spatial Visualizations and Analysis in Python with Folium	118
3	V	Case Study	119

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.

Introduction to Statistics

Statistics is a mathematical science including methods of collecting, organizing and analyzing data in such a way that meaningful conclusions can be drawn from them.

Data can be defined as groups of information that represent the qualitative or quantitative attributes of a variable or set of variables. In layman's terms, data in statistics can be any set of information that describes a given entity. An example of data can be the ages of the students in a given class. When you collect those ages, that becomes your data.

As we have seen in the definition of statistics, data collection is a fundamental aspect and therefore there are different methods of collecting data which when used on one set will result in different kinds of data. Let's move on to look at these individual methods of collection to better understand the types of data that will result.

1. Census Data Collection

Census data collection is a method of collecting data whereby all the data from every member of the population is collected.

2. Sample Data Collection

Sample data collection, which is commonly just referred to as sampling, is a method which collects data from only a chosen portion of the population.

Sampling assumes that the portion that is chosen to be sampled is a good estimate of the entire population. Thus, one can save resources and time by only collecting data from a small part of the population. But this raises the question of whether sampling is accurate or not. The answer is that for the most part, sampling is approximately accurate. This is only true if you choose your sample carefully to be able to closely approximate what the true population consists of.

Sampling is used commonly in everyday life, for example, all the different research polls that are conducted before elections. Pollsters don't ask all the people in a given state who they'll vote for, but they choose a small sample and assume that these people represent how the entire population of the state is likely to vote. History has shown that these polls are almost always close to accuracy, and as such sampling is a very powerful tool in statistics.

3. Experimental Data Collection

Experimental data collection involves one performing an experiment and then collecting the data to be further analysed. Experiments involve tests and the results of these tests are your data. An example of experimental data collection is rolling a die one hundred times while recording the outcomes. Your data would be the results you get in each roll. The experiment could involve rolling the die in different ways and recording the results for each of those different ways.

4. Observational Data Collection

Observational data collection method involves not carrying out an experiment but observing without influencing the population at all. Observational data collection is popular in studying trends and behaviors of society where, for example, the lives of a bunch of people are observed and data is collected for the different aspects of their lives. Analysis of data collected in such ways can broadly categorized into 2 categories called descriptive and inferential statistics.

Descriptive vs Inferential Statistics

Descriptive statistics deals with the processing of data without attempting to draw any inferences from it. The data are presented in the form of tables and graphs. The characteristics of the data are described in simple terms. Events that are dealt with include everyday happenings such as accidents, prices of goods, business, incomes, epidemics, sports data, population data. Inferential statistics is a scientific discipline that uses mathematical tools to make forecasts and projections by analyzing the given data. This is of use to people employed in such fields as engineering, economics, biology, the social sciences, business, agriculture and communications.

Descriptive Statistics

Descriptive statistics is the term given to the analysis of data that helps describe, show or summarize data in a meaningful way such that, for example, patterns might emerge from the data. Descriptive statistics do not, however, allow us to make conclusions beyond the data we have analyzed or reach conclusions regarding any hypotheses we might have made. They are simply a way to describe our data.

Descriptive statistics are very important because if we simply presented our raw data it would be hard to visualize what the data was showing, especially if there was a lot of it. Descriptive statistics therefore enables us to present the data in a more meaningful way, which allows simpler interpretation of the data. For example, if we had the results of 100 records of students' marks, we may be interested in the overall performance of those students. We would also be interested in the distribution or spread of the marks. Descriptive statistics allow us to do this. Typically, there are two general types of statistic that are used to describe data:

1. Measures of Central Tendency:

These are ways of describing the central position of a frequency distribution for a group of data. In this case, the frequency distribution is simply the distribution and pattern of marks scored by the 100 students from the lowest to the highest. We can describe this central position using the mode, median, and mean.

2. Measures of Spread:

These are ways of summarizing a group of data by describing how spread out the scores is. For example, the mean score of our 100 students may be 65 out of 100. However, not all students will have scored 65 marks. Rather, their scores will be spread out. Some will be lower and others higher. Measures of spread help us to summarize how spread out these scores is. To describe this spread, a number of statistics are available to us, including the range, quartiles, absolute deviation, variance and standard deviation.

Measures of Central Tendencies:

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. As such, measures of central tendency are sometimes called measures of central location. They are also classed as summary statistics. The mean (often called the average) is most likely the measure of central tendency that you are most familiar with, but there are others, such as the median and the mode.
The mean, median and mode are all valid measures of central tendency, but under different conditions, some measures of central tendency become more appropriate to use than others.

Mean:
The mean is the average of all numbers and is sometimes called the arithmetic. To calculate mean, add together all of the numbers in a set and then divide the sum by the total count of numbers.

$$Mean = \sum_{i=0}^{n} \frac{x_i}{n}$$

Where (x1, x2, x3,…..xn) are all the elements in the sample and n is the size of the sample.

For example, in a data centre rack, five servers consume 100 watts, 98 watts, 105 watts, 90 watts and 102 watts of power, respectively. The mean power use of that rack is calculated as (100 + 98 + 105 + 90 + 102 W)/5 servers = a calculated mean of 99 W per server.

Median
In a data center, means and medians are often tracked over time to spot trends, which inform capacity planning or power cost predictions. The statistical median is the middle number in a sequence of numbers. To find the median, organize each number in order by size; the number in the middle is the median.

For the five servers in the rack, arrange the power consumption figures from lowest to highest: 90 W, 98 W, 100 W, 102 W and 105 W. The median power consumption of the rack is 100 W. If there is an even set of numbers, average the two middle numbers. For example, if the rack had a sixth server that used 110 W, the new number set would be 90 W, 98 W, 100 W, 102 W, 105 W and 110 W. Find the median by averaging the two middle numbers: (100 + 102)/2 = 101 W.

Mode

The mode is the number that occurs most often within a set of numbers. For the server power consumption examples above, there is no mode because each element is different. But suppose the administrator measured the power consumption of an entire network operations centre (NOC) and the set of numbers is 90 W, 104 W, 98 W, 98 W, 105 W, 92 W, 102 W, 100 W, 110 W, 98 W, 210 W and 115 W. The mode is 98 W since that power consumption measurement occurs most often amongst the 12 servers. Mode helps identify the most common or frequent occurrence of a characteristic. It is possible to have two modes (bimodal), three modes (tri modal) or more modes within larger sets of numbers.

Measures of spread

A measure of spread, sometimes also called a measure of dispersion, is used to describe the variability in a sample or population. It is usually used in conjunction with a measure of central tendency, such as the mean or median, to provide an overall description of a set of data. The range, the variance, and the standard deviation are the most common measures of spread or variation. The range is the length of the smallest interval which contains all the data. It is calculated by subtracting the smallest observation (sample minimum) from the greatest (sample maximum). Alternatively, the range can be articulated as simply listing the lowest to highest value (i.e. range of 2 – 10).

The variance is the average difference of each value in the sample from the mean.

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2$$

The standard deviation is simply the square root of the variance.

Inferential Statistics > Drawing Inferences from Data

The objective of making inference from data is to make **intelligent assertion** like -

1. People who don’t smoke live longer than people who smoke
2. 80% of all vehicles in USA are 4 wheelers

Why making inference from data is important?

In our professional life, we make decision driven by data. It is always a better idea to have data to back our decision. In case if we don’t have data to back our decision, it can be easy that we can make wrong conclusion. It is a tangible way which you can use to defend yourself from the consequence of a decision which was correct based on information available at the point when decision was made, and which then went wrong later.

How to make assert

Let’s take the above example statement, People who don’t smoke live longer than people who smoke.

We all know how long a person lives is subject to chance. No one will know how he or she will live. Any variable which is subjected to chance is called **random variable.** A random variable could take any one of different value. And the behavior of random variable is governed by their **probability distributions**.

Usually we will study a small group of people who smoke daily and then compare them with another small group of people who don't smoke. This is called **sampling**.

For each of these 2 small groups, we will try and form a **Hypothesis** what might be true for all people and see if this Hypothesis is supported or not supported by statistics.

Our findings about the sample would allow us to test our Hypothesis. And this testing would rely on probability distributions of the underlying random variable.

This process of testing hypothesis is key, to make defensible assertion which are backed by data.

Random variables are variable

-Whose value cannot be determined before an event happens.

- Whose value is subject to variation due to chance, but we know the value is restricted to a finite set of values.

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.

Types of random variable

- Discrete, which can take only integer values (like 0,1,2,…)
- Continuous, which can take any value from a range of values
- Categorical, which can take one of a limited, fixed set of values. Eg., red, blue

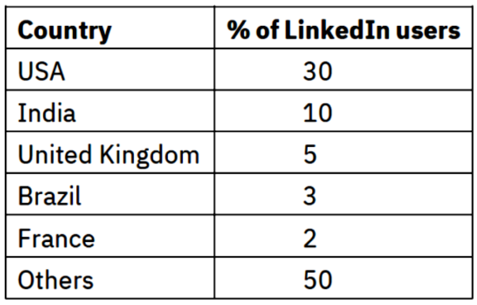
Now let's take a problem like fraud detection. To detect any fraudulent card transaction, we need to identify those variables which are related to a card transaction and can influence the problem. Below are few such variables and their types.
1. Amount spent (0 to ∞) – Continuous
2. IP address (set of all IP address in the world) – Categorical
3. Number of failed attempts on using the card (0, 1, 2..) – Discrete
4. Time since last transactions (0 … ∞) – Continuous
5. Location of transaction (Austin, Dallas, New York) – Categorical

As we see, each of these variables can have some influence on the transaction. We will not know the value of these variables before the transaction occur but will know the range for their values.

And for each transaction, each variable's value will be different, but it has to be from with in this set of range.

Statistical Experiment

Below statistic report showing LinkedIn user's top 10 geographical distribution. Let say we need to pick a user at random from the entire group of LinkedIn user, and tell what country they are likely to be from. This is a statistical experiment – whose set of outcomes can be specified beforehand, but the actual outcome of the experiment is subject to chance.



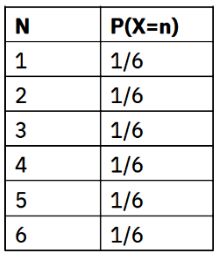
Country	% of LinkedIn users
USA	30
India	10
United Kingdom	5
Brazil	3
France	2
Others	50

Please note here the country of the user will be random variable which usually will be represented as X. Probability distribution is a table or function that links each outcome of a statistical experiment with its probability of occurrence.

P (the person picked is from USA) = P(X) = 0.3 (from above table)

Tossing a Die

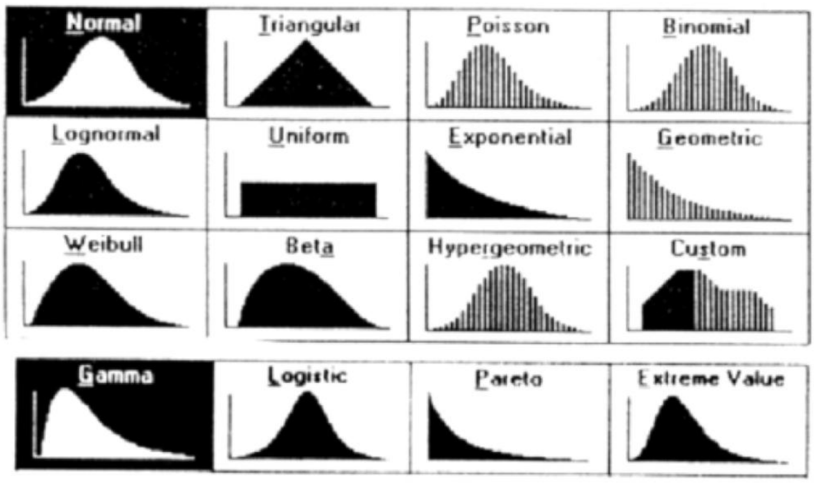
Another statistical experiment is tossing a die. When we toss a die, we will not know which value will come up, but we will know it has to be one of the values from 1 to 6. The outcome of a statistical experiment is represented by random variable.
In this case let say the outcome of the toss is X. Now X will be a discrete variable as it can take value from 1 to 6 all integers. Below table links each possible value of X with its probability.



N	P(X=n)
1	1/6
2	1/6
3	1/6
4	1/6
5	1/6
6	1/6

As we see, all of the outcomes have equal probability and hence it is called uniform probability distribution. A Uniform distribution is a distribution that has a constant probability or function.

So as random variable can be Discrete or Continuous, so can their probability distribution also. Statisticians and Mathematicians have studied a lot of different random variables in nature and realized that there are some recurring themes. They have defined some standard distribution and most random variables that you would ever encounter would fall into one of these below distributions.



Normal Distribution

Let's talk about the normal distribution which is very commonly seen in many instances of machine learning and statistical problems. Normal distribution is a distribution pattern which happens to occur lot in many natural phenomena. Below are some examples of normally distributed random variables which when plotted on a curve will results in an inverted curved which is a normal distribution.

- Height of a person

- Blood pressure

- Performance of students in a class

In normal distribution, most of the measurement (say height of a person) will be concentrated in the central peak. And there will be very few measurements that are very far off from the central point. Now these measurements are drawn from probability distribution which is basically a normal distribution.
- X axis is the value of the random variable.
- Y axis is the probability that it can take.
- The peak is the mean or the average value.
- Most of the measurement will be concentrated around the mean or average value.
- The spread of the distributions is described by the standard deviation.
- Mean and standard deviation are sufficient to completely describe the normal distribution.

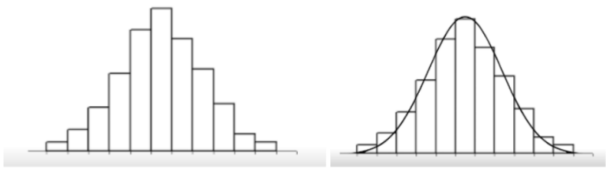
Normal Distribution is also called as gaussian distribution or bell curve. It plays a special role in statistic as many phenomena in natural life just follows the normal curve. Below are few data sets whose values are distributed normally.

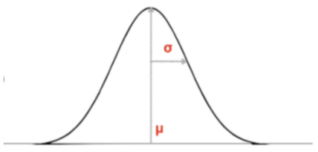
- Weights of a group of football players

-The sizes of houses in a neighbourhood

- The IQs of a group of students.

If we took all of the values of any of these data set and plot a histogram, then the resulting histogram will look very much similar like below. Then if we draw a smooth curve through the histogram, we will get a normal curve as shown. This normal curve is mathematically significant.





Mean – The peak of this curve occurs at the MEAN and it is represented as μ. When a variable is normal, its value will most likely be close to the mean.

Standard deviation - The spread of this curve is defined by the standard deviation represented as σ

The normal distribution is entirely defined by 2 parameters μ & σ with the following formula. In other word, given the mean and SD, we can tell the probability of any value.

$$f(x) = \frac{e^{-(x-\mu)^2/(2\sigma^2)}}{\sigma\sqrt{2\pi}}$$

Or simply we can define as f(x) = F(x, μ, σ)

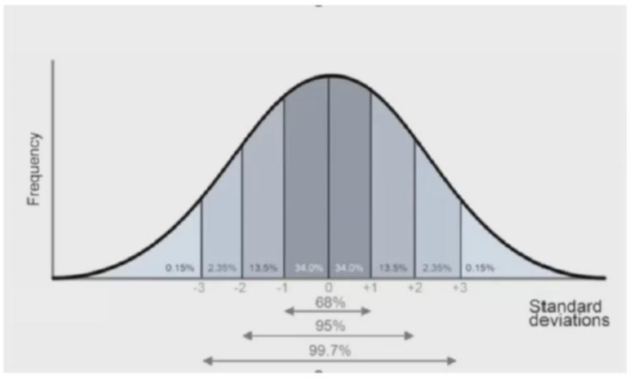
If we know the mean, SD you can tell the probability of any value that can occur in the normal distribution.

- Regardless of the actual values of mean and SD some characteristic remains same.
- Probability that a value lies within 1 SD either direction from the mean is always 68%
- Probability that a value lies within 2 SD in either direction from the mean is always 95%
- Probability that a value lies within 3 SD in either direction from the mean is always 99.7%

Above rule is very useful for

1.Testing whether a distribution is NORMAL

2. Finding outliners: Any values more than 3 σ away from the mean can be treated as outliers.



Sampling

Let’s take a Microbiologist who learn about Fish. Normally he does the following to learn about all fishes in the sea.

1. Catch **some fish**. We refer this some fish as **Sample** and the process of catching some fish is called **Sampling.**

2. Then he studies the caught fish.

3. And finally draw conclusions about all the fish in the sea. We call all the fish in the sea as **population** and the process of drawing conclusions about the population by observing the sample is called **Generalization.**

In simple word, Sampling means drawing conclusions about the population by observing the sample and generalizing. The conclusion which we draw is called inference. To draw meaning inference, we learn few techniques and hypothesis testing.

Below are few sampling use cases which are becoming very common across industry.

- Phycological studies

- Polling

- Drug Trails

- A/B Tests

- Market Research Surveys

Sample statistics – We describe the sample which is subset of the entire population using sample statistics. The sample statistics usually characterize the sample and not the population.

Sample statistics can broadly classified into two type -

1. Sample Mean - used when the variable is continuous like

• Height of a group of people

• User engagement on a website

2. Sample percentage – normally used when the variable is Binary (yes/no) like
• Do you support this candidate?

• Is this drug an effective treatment?

Standard deviation (SD) of the sample is critical as it will allow us to determine the confidence interval around the assertion which we will manner.

The manner in which we calculate the standard deviation of the sample is different for sample mean and sample percentage.

Below shows standard deviation of the same when we are interested in sample mean

$$SD=\sqrt{\frac{\sum(x-\bar{x})^2}{n}}$$

Below shows standard deviation of the same when we are interested in sample percentage

where p is the % of YESes

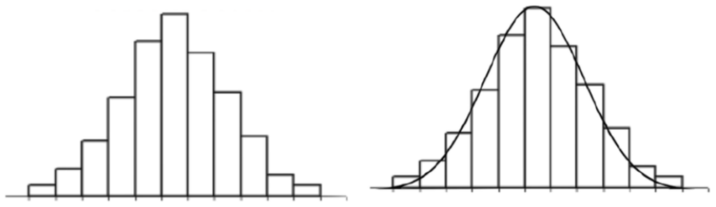
$$\text{s.d. }(p)=\sqrt{\frac{p\,(1-p)}{n}}$$

Sample Statistics and Sampling Distribution

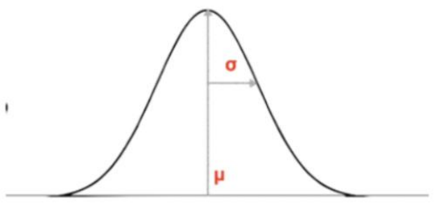
Let us pick 100 different samples of any dataset from any population. For each of these samples, you then compute the sample mean or % as shown below. These computed sample mean or % will vary as it is different for each sample and hence, we call it as random variable.

Sample 1	53%
Sample 2	54%
…………	…………
Sample 99	53%
Sample 100	52%

If we plot the histogram of these values, that will represent its probability distribution. It turns out whenever you take a large number of samples, the sample % or sample mean of those samples follows a normal distribution. This normal distribution is also known as the sampling distribution.



Now we will see how to calculate sample mean and SD of the sampling distribution with one sample, as in near life we usually have limited samples. Sampling distribution of the standard deviation is termed as Standard error σ.



The best estimate of a sample distribution's mean is its sample average ie, $\mu = \bar{x}$

For sample mean, σ = SD / SQRT (N), where N is the number of points in our datasets

For Sample percentage, σ = SD

So as of now, we have

1. Sample mean
2. Standard deviation of the sample
3. Sample standard error

Using these 3 numbers, we can now make assertion or inference about the population. Most of the inference falls under few specific types and there are standard procedures involved for each type to draw inference. Below are the inference types and an example for each types

1. Identify the population mean

- Indian Police are on average 80KG +/- 5 KG with 95% confidence

2. Identify the population percentage

- 20% +/- 2% of software engineers in a given city goes for morning walk

3. Verify if population mean is equal to a certain value

- Is the average life expectancy of Indians is 70 years?

4. Verify if population percentage is equal to a certain value

- 20% of people who took the drug has a side effect

5. Verify if 2 population means are different

- Indians are on average taller than Chinese

6. Verify if 2 population percentage are different

- Only 10% of people who don't take the drug get better, but 80% of people who take the drug better.

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

Basic Commands:

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.
options()	# Set options to control how R computes & displays re-sults
?NA	# Help page on handling of missing data values
abs(x)	# The absolute value of "x"
append()	# Add elements to a vector
c(x)	# A generic function which combines its arguments
cat(x)	# Prints the arguments
cbind()	# Combine vectors by row/column (cf. "paste" in Unix)
diff(x)	# Returns suitably lagged and iterated differences

gl()	# Generate factors with the pattern of their levels
grep()	# Pattern matching
identical()	# Test if 2 objects are *exactly* equal
jitter()	# Add a small amount of noise to a numeric vector
julian()	# Return Julian date
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
system("cmd")	# Execute "cmd" in operating system (outside of R)
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.getenv(x)	# Get the value of the environment variable "x"
Sys.putenv(x)	# Set the value of the environment variable "x"
Sys.time()	# Return system time
Sys.Date()	# Return system date
getwd()	# Return working directory
setwd()	# Set working directory
?files	# Help on low-level interface to file system
list.files()	# List files in a give directory
file.info()	# Get information about files
#	Built-in constants:

pi,letters,LETTERS	# Pi, lower & uppercase letters, e.g. letters[7] = "g"
month.abb,month.name	# Abbreviated & full names for months

Mathematics:

log(x),logb(),log10(),log2(),exp(),expm1(),log1p(),sqrt()	# Fairly obvious
cos(),sin(),tan(),acos(),asin(),atan(),atan2()	# Usual stuff
cosh(),sinh(),tanh(),acosh(),asinh(),atanh()	# Hyperbolic functions
union(),intersect(),setdiff(),setequal()	# Set operations
+,-,*,/,^,%%,%/%	# Arithmetic operators
<,>,<=,>=,==,!=	# Comparison operators
eigen()	# Computes eigenvalues and eigenvectors
deriv()	# Symbolic and algorithmic derivatives of

	simple expressions
integrate()	# Adaptive quadrature over a finite or infinite interval.
sqrt(),sum()?Control	# Help on control flow statements (e.g. if, for, while)
?Extract	# Help on operators acting to extract or re-place subsets of vectors
?Logic	# Help on logical operators
?Mod	# Help on functions which support complex arithmetic in R
?Paren	# Help on parentheses
?regex	# Help on regular expressions used in R
?Syntax	# Help on R syntax and giving the precedence of operators
?Special	# Help on special functions related to beta

	and gamma functions

Graphical:

help(package=graphics)	# List all graphics functions
plot()	# Generic function for plotting of R objects
par()	# Set or query graphical parameters
curve(5*x^3,add=T)	# Plot an equation as a curve
points(x,y)	# Add another set of points to an existing graph
arrows()	# Draw arrows [see errorbar script]
abline()	# Adds a straight line to an existing graph
lines()	# Join specified points with line segments
segments()	# Draw line segments between pairs of points

hist(x)	# Plot a histogram of x
pairs()	# Plot matrix of scatter plots
matplot()	# Plot columns of matrices
?device	# Help page on available graphical devices
postscript()	# Plot to postscript file
pdf()	# Plot to pdf file
png()	# Plot to PNG file
jpeg()	# Plot to JPEG file
X11()	# Plot to X window
persp()	# Draws perspective plot
contour()	# Contour plot

image()	# Plot an image

Fitting/ Regression/ Optimization:

lm	# Fit liner model
glm	# Fit generalised linear model
nls	# non-linear (weighted) least-squares fitting
lqs	# "library(MASS)" resistant regression
optim	# general-purpose optimisation
optimize	# 1-dimensional optimisation
constrOptim	# Constrained optimisation
nlm	# Non-linear minimisation
nlminb	# More robust (non-)constrained non-linear minimisation

Statistical:

help(package=stats)	# List all stats functions
?Chisquare	# Help on chi-squared distribution functions
?Poisson	# Help on Poisson distribution functions
help(package=survival)	# Survival analysis
cor.test()	# Perform correlation test
cumsum(); cumprod(); cummin(); cummax()	# Cumuluative functions for vectors
density(x)	# Compute kernel density estimates
ks.test()	# Performs one or two sample Kolmogorov-Smirnov tests
loess(), lowess()	# Scatter plot smoothing
mad()	# Calculate median absolute deviation
mean(x), weighted.mean(x), median(x), min(x),	# Generate random data with Gaussian/uniform dis-

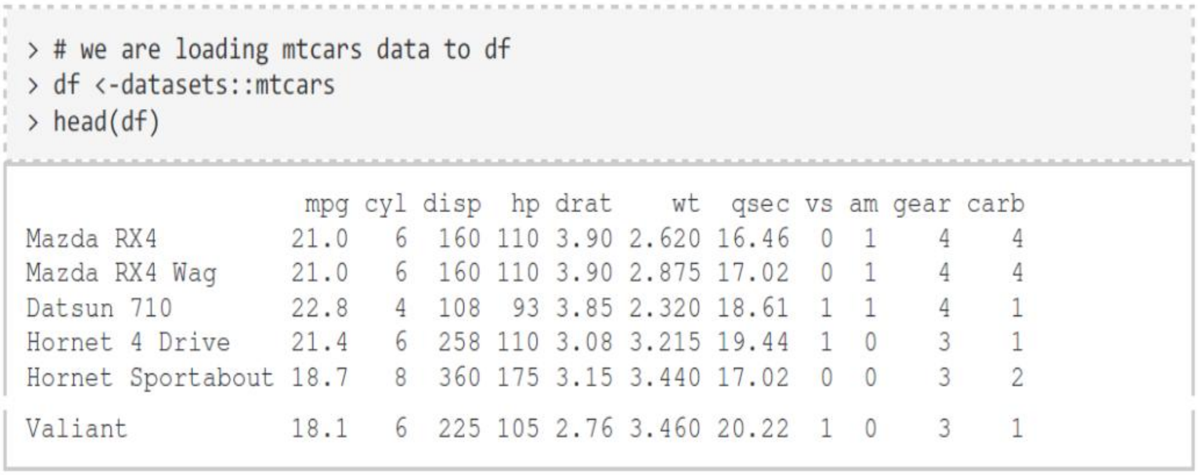
max(x), quantile(x) rnorm(), runif()	tribution
splinefun()	# Perform spline interpolation
smooth.spline()	# Fits a cubic smoothing spline
sd()	# Calculate standard deviation

The above screenshot shows some basic commands that you run in the console. You can run some simple arithmetic operations directly, assign values to variables, print them. Note that unlike other programming languages, R's assignment operator is "<-".

This is your very first step in learning R. Additionally you can learn how to install packages, load them and execute code/functions that are already defined in these functions. R has over 10000+ packages, each tailored for some specific use!

As the name indicates, Descriptive Analysis is used to find and define the basic features of the data that is being studied. Descriptive Analysis provides simple summary about the sample data and the measures. Descriptive Analysis and Graphical Analysis are used together to form the basic virtual of any quantitative analysis of data. With descriptive analysis, you can describe what is the data or what the data is depicting. Description of data is needed to determine the normality of the distribution and because the techniques that need to be applied to infer data depends on the characteristics of the data.

The dataset used for this analysis is mtcars. To start, install the required packages for the analysis.



```
> # we are loading mtcars data to df
> df <-datasets::mtcars
> head(df)
```

```
                   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
```

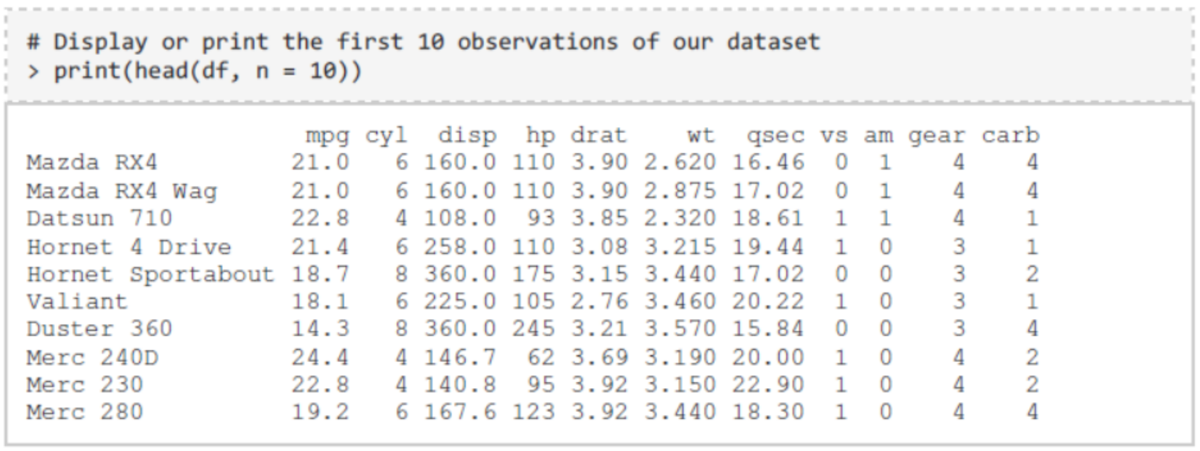
The above data was obtained from the 1974 Motor Trend US magazine. It gives details about fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

This data set consists of 32 observations on 11 (numeric) variables, and the variables are:

1.	mpg	Miles/(US) gallon
2.	cyl	Number of cylinders
3.	disp	Displacement (cu.in.)
4.	hp	Gross horsepower
5.	drat	Rear axle ratio
6.	wt	Weight (1000 lbs)
7.	qsec	1/4 mile time
8.	vs	Engine (0 = V-shaped, 1 = straight)
9.	am	Transmission (0 = automatic, 1 = manual)
10.	gear	Number of forward gears
11.	carb	Number of carburetors

Summary and Descriptive Statistics

Descriptive (Summary) Analysis is the first figure used to represent nearly every dataset. This forms the foundation for further complicated computation and analysis. Therefore, it is essential to the analysis process. In this chapter, we will use R to calculate summary statistics, including mean, standard deviation, range, and percentile. We have also included the summary function, which is one of the most useful tools in the R set of commands. To start with, let us inspect the mtcars dataset.



```
# Display or print the first 10 observations of our dataset
> print(head(df, n = 10))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

We observe that the rows of our dataset refer to mtcars.

ls(DATAVAR) or names(DATAVAR)

To see the nature of variables/columns in the mtcars dataset, you can use the ls(DATAVAR) Or names(DATAVAR). See the following code for the sample.

```
# To see the variable in the dataset; Use names(dataset) or ls(dataset)
> ls(df)
```

```
[1] "am"   "carb" "cyl"  "disp" "drat" "gear" "hp"   "mpg"  "qsec" "vs"   "wt"
```

```
# To see the number of columns and number of rows in the Iris dataset; use ncol(datas
et) and nrow(dataset)
> ncol(df)
```

```
[1] 11
```

```
> nrow(df)
```

```
[1] 32
```

```
# More advanced and complete way to see the structure of our dataset
> str(df)
```

```
'data.frame':    32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Mean of each variable (mean(DATAVAR)

To calculate the mean of on an isolated variable, use the mean(VAR) command. VAR is the name of the variable whose mean you want to compute. You can also calculate, the mean for all the variables in the dataset using the mean(DATAVAR) function, where DATAVAR is the name of the dataset containing the variables.
For analysis purposes, we will exclude the variables census and type from the descriptive statis-tics.

To select a subset of a dataset, use the subset(DATAVAR, select = c(“VAR1”, “VAR2”, “VAR3”….“VARi”)) command.

NOTE: To know how to subset() vectors, matrices, and dataframes, type ?subset() in your R console and press ENTER.
The code below demonstrates how to select a subset of Iris dataset.

```
# Subsetting Miles/(US) gallon, Number of cylinders, Displacement (cu.in.),Gross hors
epower, and 1/4 mile time from the dataset mtcars

> subset.data <- subset(df, select = c("mpg","cyl","disp","hp","qsec"))
```

```
# Checking subset.data to make sure we have the needed subset
> str(subset.data)
```

```
'data.frame':    32 obs. of  5 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
```

```
# Calculate the mean of a variable with mean(DATAVAR$VAR); mean of variable mpg
> mean(subset.data$mpg)
```

```
[1] 20.09062
```

```
# Mean of 1/4 mile time variable
> mean(subset.data$qsec)
```

```
[1] 17.84875
```

```
# Mean of Gross horsepower variable
> mean(subset.data$hp)
```

```
[1] 146.6875
```

Standard Deviation of each variable (sd(VAR))

Standard deviation is used to tell how measurements for a group are spread out from the average (mean), or expected value. There two types of Standard Deviations, A low standard deviation means that most of the numbers are close to the average. A high standard deviation means that the numbers are more spread out.

To compute the standard deviation in R , use the command sd(VAR). Standard deviation measures how spread the data is. The following code shows the use of the standard deviation function:

```
# Calculate the standard deviation of a variable with sd(DATAVAR$VAR); standard devia
tion of variable mpg
> sd(subset.data$mpg)
```

```
[1] 6.026948
```

```
> sd(subset.data$qsec)
```

```
[1] 1.786943
```

```
# Standard Deviation of Gross horsepower variable
> sd(subset.data$hp)
```

```
[1] 68.56287
```

Range of each variable: MINIMUM & MAXIMUM (min(VAR) & max(VAR))

Minimum can be computed on a single variable using the min(VAR) formula. Minimum and maximum give the min and max of individual variables in the dataset. The following code shows how to calculate minimums and maximums.

```
# Minimum and Maximum value of mpg
> min(subset.data$mpg); max(subset.data$mpg)
```

```
[1] 10.4
[1] 33.9
```

```
# Minimum and Maximum value of 1/4 mile time variable
> min(subset.data$qsec); max(subset.data$qsec)
```

```
[1] 14.5
[1] 22.9
```

```
# Minimum and Maximum value of Gross horsepower variable
> min(subset.data$hp); max(subset.data$hp)
```

```
[1] 52
[1] 335
```

PERCENTILES: VALUES from PERCENTILES (QUANTILE)

A percentile (is also known as centile) is a measure in statistics. It shows the value below which a given percentage of observations falls.

To understand the distribution of a set of observations, you must verify the quantiles. A quantile is a value computed from a collection of numeric measurements that indicates the rank of an observation when compared to all other observations. Quantile can be expressed as a percentile, this is identical but on a percent scale of 0 to 100.

To obtain quantiles and percentile in R use the quantile() function. The command is written as follows:

quantile(VAR, c(PROB1, PROB2, PROB3,….PROBi)) or quantile(VAR, prob = c(prob value1, prob value 2, prob value 3…prob valuei)).

```
# Calculate the 25th, 50th, 75th, and 95th percentiles for mpg in the subset dataset
> quantile(subset.data$mpg, prob = c(.25, .50, .75, .95))
```

```
          25%    50%    75%    95%
       15.425 19.200 22.800 31.300
```

```
> quantile(subset.data$qsec, prob = c(.25, .50, .75, .95))
```

```
    25%     50%     75%     95%
16.8925 17.7100 18.9000 20.1045
```

```
> quantile(subset.data$hp, prob = c(.25, .50, .75, .95))
```

```
   25%    50%    75%    95%
 96.50 123.00 180.00 253.55
```

PERCENTILES FROM VALUES (PERCENTILE RANK)

To find a percentile rank corresponding to a given value, use must use a custom method. Following are the steps for computing a percentile rank:
Count the number of data points that are at or below the given value Divide by the total number of data points multiple by 100

To derive the formula to calculate a percentile rank, use the following command:

percentile rank = length(VAR[VAR <= VALUE]) / length(VAR) * 100. length(VAR[VAR <= VALUE])

The command counts the number of data points in a variable that are below the given value. NOTE: You can replace '<=' with other operators, such as '<', '>', and =.length(VAR) counts the number of data points in the variable.
The final step is to multiply the result by 100 to transform the decimal value into a percentage.

Let us apply these steps in the following examples:

```
# In the sample, less than 5.50 qsec is at what percentile rank?
> length(subset.data$qsec[subset.data$qsec <=15.60])/length(subset.data$qsec)*100
```

```
[1] 12.5
```

Only 12.5% of cars in the data take less than 15.60 sec's to complete 1/4 mile.

Summary Function (summary(x))

Summary functions is used to produce a summary of all records in the found set, or subsummary values for records in different groups summary(x) is a very useful and multipurpose function in R. x can be any dataset, variable, linear model, and so on. The command provides summary data related to the object that was queried. The output of the function depends on the type of object that was queried. It is very useful because it can be used to summarize the previous actions. The result caters to the requirements of summary statistics. In the following examples, we have applied the command to the sample dataset.

```
# Summarize a variable using summary(VAR). Summary statistics of Miles/(US) gallon
> print(summary(subset.data$mpg))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10.40   15.42   19.20   20.09   22.80   33.90
```

```
> print(summary(subset.data$hp))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   52.0    96.5   123.0   146.7   180.0   335.0
```

```
> print(summary(subset.data$qsec))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.50   16.89   17.71   17.85   18.90   22.90
```

UNIT – II

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

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

Data Manipulation with R > Introduction to dplyr (filter, select, arrange, mutate, summarize) > Introduction to dplyr (filter, select, arrange, mutate, summarize)

When it comes to Predictive Modeling, data Manipulation is an important and unavoidable phase. Machine learning algorithms are just not sufficient to build a robust predictive model. The approach must be to understand the business problem, the data, performing data manipulations, and then extracting business insights. Majority of time is spent in understanding the data and manipulating data as required. In this chapter we shall look at the details of Data Manipulation.

Data Manipulation is also called as Data Exploration (also known as Data Wrangling or Data Cleaning). Data Manipulation is done to improve data accuracy and precision. Data Manipulation is a mandatory step when it comes to predictive modeling because of the many faults in data collection process, because of many uncontrollable factors involved in data collection.

In reality, there is no right or wrong way to do Data Manipulation. However, you have to understand the data and must take necessary steps to improve the accuracy. Following are some of the points you need to consider for Data Manipulation:

- You can use the inbuilt functions in R to manipulate data. Though it is a good step to start with initially, it is not very efficient, because you must be repeating the process and it is also time consuming.
- You can use the packages available in CRAN. As these packages are tried and tested, they are more efficient. Using the CRAN packages is the most widely accepted industry way of doing Data Manipulation.
- You can also use ML algorithms. For example, tree based boosting algorithms take care of missing data and outliers. Though time-efficient, you will need to have a very thorough understanding of data.

dplyr Package

dplyr is a powerful R-package which transforms and summarizes tabular data with rows and columns. It is best known for data exploration and transformation. Its chaining syntax makes it highly adaptive to use. It includes 5 major data manipulation commands:

- filter – filters the data based on a condition
- select –used to select columns of interest from a data set

- arrange –used to arrange data set values on ascending or descending order
- mutate – used to create new variables from existing variables
- summarise (with group_by) – used to perform analysis by commonly used operations such as min, max, mean count etc.

Filter rows with filter()

You can use the filter() command to select a subset of rows in a data frame. Similar to all single verbs, the first argument is the tibble (or data frame). The subsequent arguments refer to variables within that data frame, selecting rows where the expression is TRUE.

In this example, we will be using another dataset, this dataset is a public data set. As mentioned in the Wikipedia, The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. It is sometimes called Anderson's Iris data set because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of three related species. Two of the three species were collected in the Gaspé Peninsula "all from the same pasture, and picked on the same day and measured at the same time by the same person with the same apparatus".

The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

To install dplyr, use the below command

install.packages("dplyr")

To load dplyr, use the below command

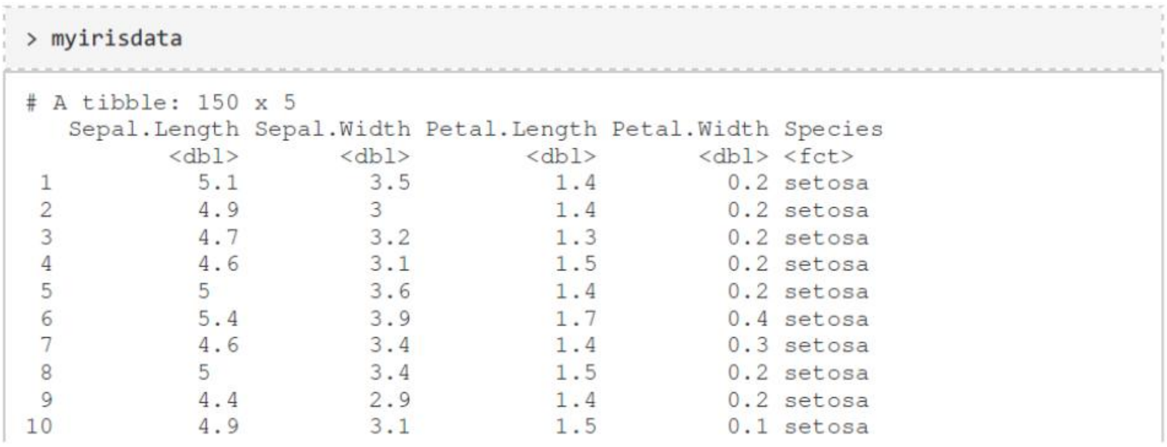
library(dplyr)

```
> 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
```

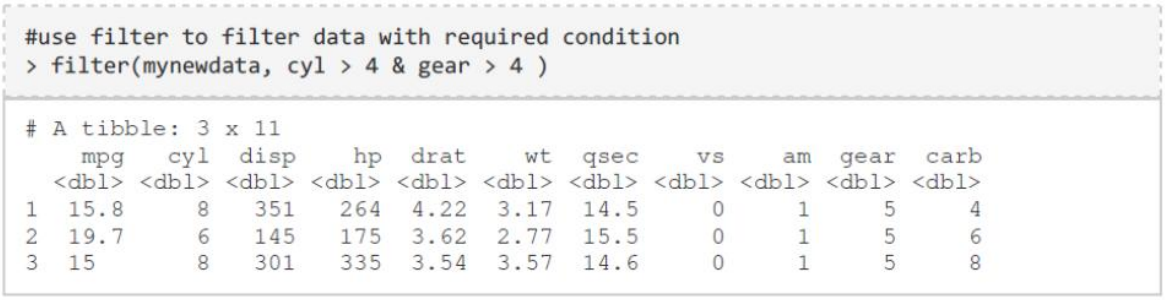
```
 # A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
```



```
> myirisdata
```

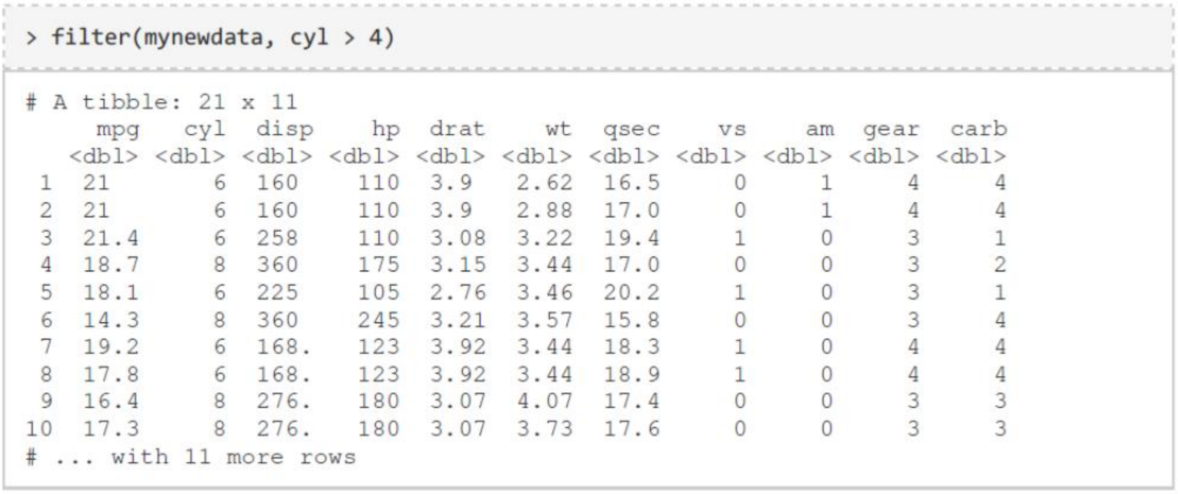
```
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            1.4         0.2 setosa
 3          4.7         3.2          1.3         0.2 setosa
 4          4.6         3.1          1.5         0.2 setosa
 5          5           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           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
```

```
# ... with 140 more rows
```



```
#use filter to filter data with required condition
> filter(mynewdata, cyl > 4 & gear > 4 )
```

```
# A tibble: 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       8   301   335  3.54  3.57  14.6     0     1     5     8
```



```
> filter(mynewdata, cyl > 4)
```

```
# A tibble: 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       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 4  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 5  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 6  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 7  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
 8  17.8     6  168.   123  3.92  3.44  18.9     1     0     4     4
 9  16.4     8  276.   180  3.07  4.07  17.4     0     0     3     3
10  17.3     8  276.   180  3.07  3.73  17.6     0     0     3     3
# ... with 11 more rows
```

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

```
# A tibble: 100 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            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           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           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
# ... with 90 more rows
```

Select columns with select()

When you are working with large datasets with many columns, but you are actually interested in a few, select() allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
# use select to pick columns by name
> select(mynewdata, cyl,mpg,hp)
```

```
# A tibble: 32 x 3
     cyl   mpg    hp
 * <dbl> <dbl> <dbl>
 1     6  21     110
 2     6  21     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
# ... with 22 more rows
# here you can use (-) to hide columns
> select(mynewdata, -cyl, -mpg )
```

```
# A tibble: 32 x 9
    disp    hp  drat    wt  qsec    vs    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  160    110  3.9   2.62  16.5     0     1     4     4
 2  160    110  3.9   2.88  17.0     0     1     4     4
 3  108     93  3.85  2.32  18.6     1     1     4     1
 4  258    110  3.08  3.22  19.4     1     0     3     1
 5  360    175  3.15  3.44  17.0     0     0     3     2
 6  225    105  2.76  3.46  20.2     1     0     3     1
 7  360    245  3.21  3.57  15.8     0     0     3     4
 8  147.    62  3.69  3.19  20       1     0     4     2
 9  141.    95  3.92  3.15  22.9     1     0     4     2
10  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
```

```
# hide a range of columns
> select(mynewdata, -c(cyl,mpg))
```

```
# A tibble: 32 x 9
    disp    hp  drat    wt  qsec    vs    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  160    110  3.9   2.62  16.5     0     1     4     4
 2  160    110  3.9   2.88  17.0     0     1     4     4
 3  108     93  3.85  2.32  18.6     1     1     4     1
 4  258    110  3.08  3.22  19.4     1     0     3     1
 5  360    175  3.15  3.44  17.0     0     0     3     2
 6  225    105  2.76  3.46  20.2     1     0     3     1
 7  360    245  3.21  3.57  15.8     0     0     3     4
 8  147.    62  3.69  3.19  20       1     0     4     2
 9  141.    95  3.92  3.15  22.9     1     0     4     2
10  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
```

```
# select series of columns
> select(mynewdata, cyl:gear)
```

```
# A tibble: 32 x 9
     cyl  disp    hp  drat    wt  qsec    vs    am  gear
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1     6  160    110  3.9   2.62  16.5     0     1     4
 2     6  160    110  3.9   2.88  17.0     0     1     4
 3     4  108     93  3.85  2.32  18.6     1     1     4
 4     6  258    110  3.08  3.22  19.4     1     0     3
 5     8  360    175  3.15  3.44  17.0     0     0     3
 6     6  225    105  2.76  3.46  20.2     1     0     3
 7     8  360    245  3.21  3.57  15.8     0     0     3
 8     4  147.    62  3.69  3.19  20       1     0     4
 9     4  141.    95  3.92  3.15  22.9     1     0     4
10     6  168.   123  3.92  3.44  18.3     1     0     4
# ... with 22 more rows
# chaining or pipelining - a way to perform multiple operations
#in one line
> mynewdata %>%
     select(cyl, wt, gear)%>%
     filter(wt > 2)
```

```
# A tibble: 28 x 3
     cyl    wt  gear
   <dbl> <dbl> <dbl>
 1     6  2.62     4
 2     6  2.88     4
 3     4  2.32     4
 4     6  3.22     3
 5     8  3.44     3
 6     6  3.46     3
 7     8  3.57     3
 8     4  3.19     4
 9     4  3.15     4
10     6  3.44     4
# ... with 18 more rows
```

Arrange rows with arrange()

arrange() re-orders the rows. It takes a data frame, and a set of column names (or more complicated expressions) to order the rows. If you provide more than one column name, each additional column is used to break ties in the values of preceding columns.

```
# arrange can be used to reorder rows
> mynewdata%>%
     select(cyl, wt, gear)%>%
     arrange(wt)
```

```
# A tibble: 32 x 3
     cyl    wt  gear
   <dbl> <dbl> <dbl>
 1     4  1.51     5
 2     4  1.62     4
 3     4  1.84     4
 4     4  1.94     4
 5     4  2.14     5
 6     4  2.2      4
 7     4  2.32     4
 8     4  2.46     3
 9     6  2.62     4
10     6  2.77     5
# ... with 22 more rows
```

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

```
# A tibble: 32 x 3
     cyl    wt  gear
   <dbl> <dbl> <dbl>
 1     8  5.42     3
 2     8  5.34     3
 3     8  5.25     3
 4     8  4.07     3
 5     8  3.84     3
 6     8  3.84     3
 7     8  3.78     3
 8     8  3.73     3
 9     8  3.57     3
10     8  3.57     5
# ... with 22 more rows
```

<u>Add new columns with mutate()</u>

This function, mutate() adds new variables while preserving the existing ones. mutate() is used to select sets of existing columns and add new columns that are functions of existing columns. Following is the example:

```
#mutate - create new variables

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

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

```
# A tibble: 32 x 3
     mpg   cyl newvariable
   <dbl> <dbl>       <dbl>
 1  21       6       126
 2  21       6       126
 3  22.8     4        91.2
 4  21.4     6       128.
 5  18.7     8       150.
 6  18.1     6       109.
 7  14.3     8       114.
 8  24.4     4        97.6
 9  22.8     4        91.2
10  19.2     6       115.
# ... with 22 more rows
```

<u>Summarise values with summarise()</u>

The summarise() collapses a data frame to a single row.

```
#summarise - this is used to find insights from data
> myirisdata%>%
       group_by(Species)%>%
       summarise(Average = mean(Sepal.Length, na.rm = TRUE))
```

```
# A tibble: 3 x 2
  Species    Average
  <fct>        <dbl>
1 setosa        5.01
2 versicolor    5.94
3 virginica     6.59
```

```
#or use summarise each
> myirisdata%>%
      group_by(Species)%>%
      summarise_each(funs(mean, n()), Sepal.Length, Sepal.Width)
```

```
# A tibble: 3 x 5
  Species    Sepal.Length_mean Sepal.Width_mean Sepal.Length_n Sepal.Width_n
  <fct>                  <dbl>            <dbl>          <int>         <int>
1 setosa                  5.01             3.43             50            50
2 versicolor              5.94             2.77             50            50
3 virginica               6.59             2.97             50            50
```

```
#You can create complex chain commands using these 5 verbs.
#you can rename the variables using rename command
> mynewdata %>% rename(miles = mpg)
```

```
# A tibble: 32 x 11
   miles   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
```

data.table Package

A data table is nothing but a group of related facts arranged in labeled rows and columns and is used to record information. data.table can be used to perform faster manipulation in a data set. Using data.table reduces computing time when compared to data.frame. A data table has 3 parts namely DT[i,j,by]. Here, we are instructing R to subset the rows using 'i', to calculate 'j' which is grouped by 'by'. Most of the times, 'by' relates to categorical variable. In the code below, we have used 2 data sets (airquality and iris).

```
# Loading Airquality 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
```

```
> data(iris)
>
> myiris <- iris
>
> #load package
>
> library(data.table)
```

```
     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
```

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

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

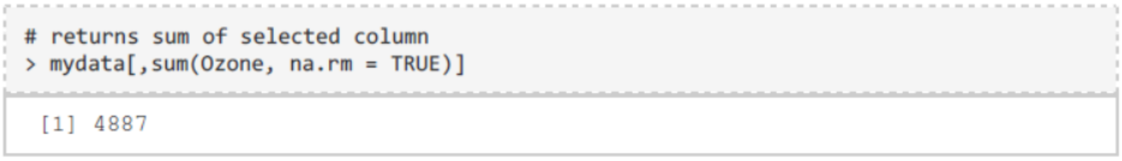
```
 47:          5.1         3.8          1.6         0.2    setosa
 48:          4.6         3.2          1.4         0.2    setosa
 49:          5.3         3.7          1.5         0.2    setosa
 50:          5.0         3.3          1.4         0.2    setosa
 51:          6.3         3.3          6.0         2.5 virginica
 52:          5.8         2.7          5.1         1.9 virginica
 53:          7.1         3.0          5.9         2.1 virginica
 54:          6.3         2.9          5.6         1.8 virginica
 55:          6.5         3.0          5.8         2.2 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 67
 [29] 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 76 75
 [57] 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 74 81 82
 [85] 86 85 82 86 88 86 83 81 81 81 82 86 85 87 89 90 90 92 86 86 82 80 79 77 79 76 78 78
[113] 77 72 75 79 81 86 88 97 94 96 94 91 92 93 93 87 84 80 78 75 73 81 76 77 71 71 78 67
[141] 76 68 82 64 71 81 69 63 70 77 75 76 68
```

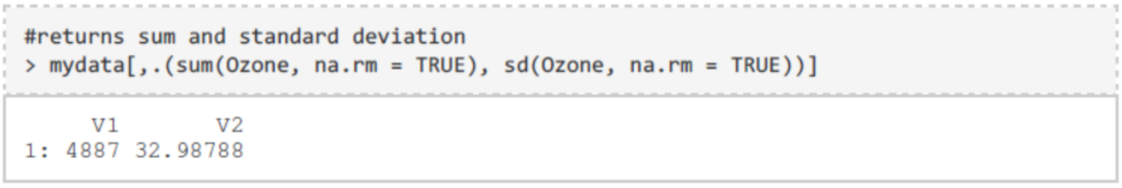
```
> mydata[,.(Temp,Month)]
```

```
     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
```



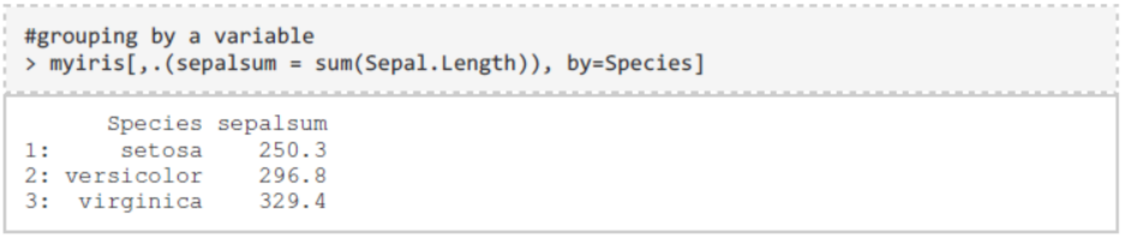
```
# returns sum of selected column
> mydata[,sum(Ozone, na.rm = TRUE)]
```

```
 [1] 4887
```



```
#returns sum and standard deviation
> mydata[,.(sum(Ozone, na.rm = TRUE), sd(Ozone, na.rm = TRUE))]
```

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

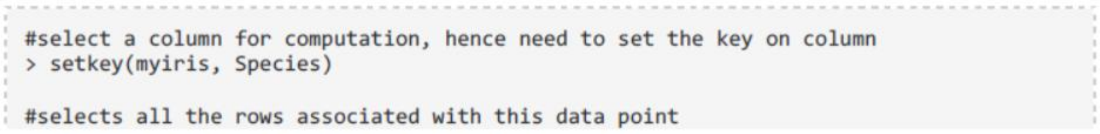


```
#grouping by a variable
> myiris[,.(sepalsum = sum(Sepal.Length)), by=Species]
```

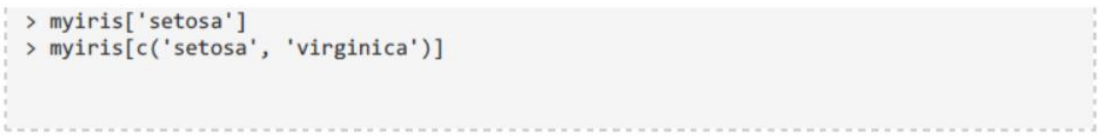
```
      Species sepalsum
1:     setosa    250.3
2: versicolor    296.8
3:  virginica    329.4
```

```
#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')]
```

reshape2 Package

reshape2 is an R package, was written by Hadley Wickham which makes it easy to transform data between wide and long formats. Use the reshape2 package to reshape your data. Using the reshape2 package, we can combine features that have unique values. It has 2 functions namely melt and cast.

- **melt:** Converts data from wide format to long format. It is a form of restructuring where multiple categorical columns are 'melted' into unique rows. Let us understand it using the code below.

```
#create a new 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
```

```
#loading package
> install.packages('reshape2')
```

```
> library(reshape2)

#working with 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:** converts data from long format to wide format. It starts with melted data and re-shapes into long format. It's the reverse of melt function. It has two functions namely, **dcast** and **acast**.

-dcast returns a data frame as output.

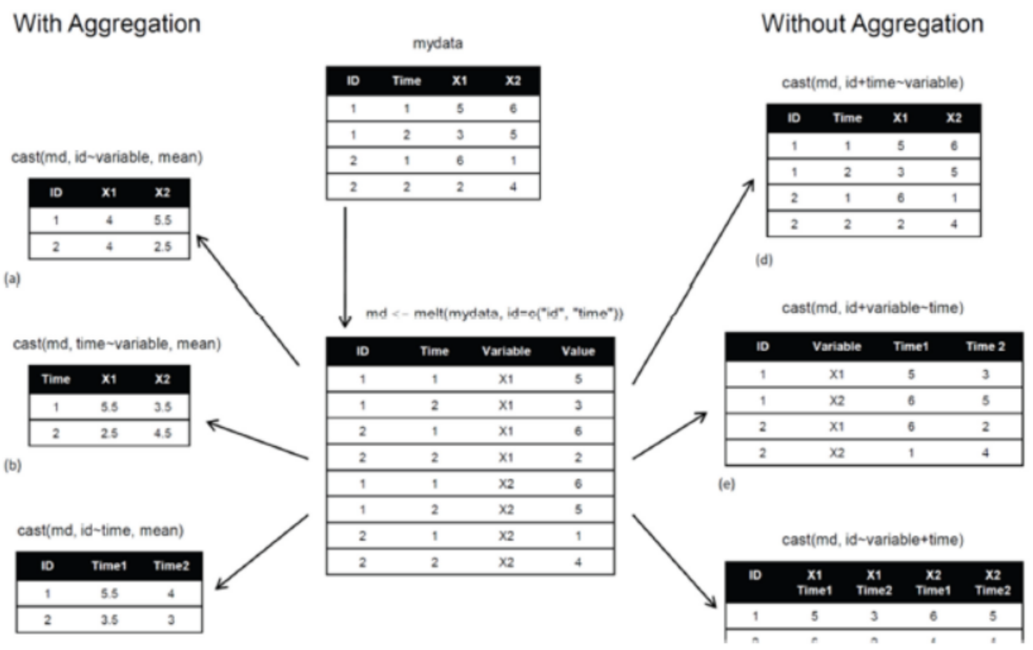
-acast returns a vector/matrix/array as the output.

Let's understand it using the code below.

```
#working with cast
> 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
```

This image would help us understand how reshape package works



mydata

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

md <- melt(mydata, id=c("id", "time"))

ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

cast(md, id~variable, mean)

ID	X1	X2
1	4	5.5
2	4	2.5

cast(md, time~variable, mean)

Time	X1	X2
1	5.5	3.5
2	2.5	4.5

cast(md, id~time, mean)

ID	Time1	Time2
1	5.5	4
2	3.5	3

cast(md, id+time~variable)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

cast(md, id+variable~time)

ID	Variable	Time1	Time 2
1	X1	5	3
1	X2	6	5
2	X1	6	2
2	X2	1	4

cast(md, id~variable+time)

ID	X1 Time1	X1 Time2	X2 Time1	X2 Time2
1	5	3	6	5
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

readr Package

The readr package is also developed by Hadley Wickham to deal with reading in large flat files quickly.

'readr' is used to read various forms of data in R. It is very fast. The characters are not converted to factors. Therefore, you don't set stringAsFactors = FALSE. 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 displays a progress bar. Let's look at the code below:

```
> install.packages('readr')
> library(readr)
> read_csv('test.csv',col_names = TRUE)
```

You can also specify the data type of every column loaded in data using the code below:

```
> 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"))
))
```

You can also omit unimportant columns. So, the code above can also be re-written as:

```
> read_csv("iris.csv", col_types = list(Species = col_factor(c("setosa",
"versicolor", "virginica")))
```

readr has many helper functions. When you write a csv file, use write_csv instead of write.csv. The later is very fast.

tidyr Package

tidyr is a package which was developed by Hadley Wickham which makes it easy to tidy your data.

To make the data look neat and tidy, use the tidyr package. The package has 4 major functions. You can use these functions if you are stuck in the data exploration phase, along with dplyr.

- gather() – 'gathers' multiple columns and converts them into key:value pairs. This function transforms wide form of data to long form. It can be used as an alternative to 'melt' in reshape package.

- spread() – Does reverse of gather. It accepts a key:value pair and converts it into separate columns.

- separate() – Splits a column into multiple columns.
- unite() – Does reverse of separate. It unites multiple columns into single column

```
# Loading the package
```

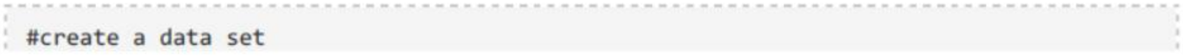
```
> library(tidyr)

#create a dummy data set
> names <- c('Akanksh','Bhanu','Susheel','Vinay','Varun','Prashanth','Manohar')
> 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   Akanksh  21     55     Maths
2     Bhanu  20     49   Science
3   Susheel  25     76    Social
4     Vinay  29     71   Physics
5     Varun  33     65   Biology
6 Prashanth  32     44 Economics
7   Manohar  38     34  Accounts
```

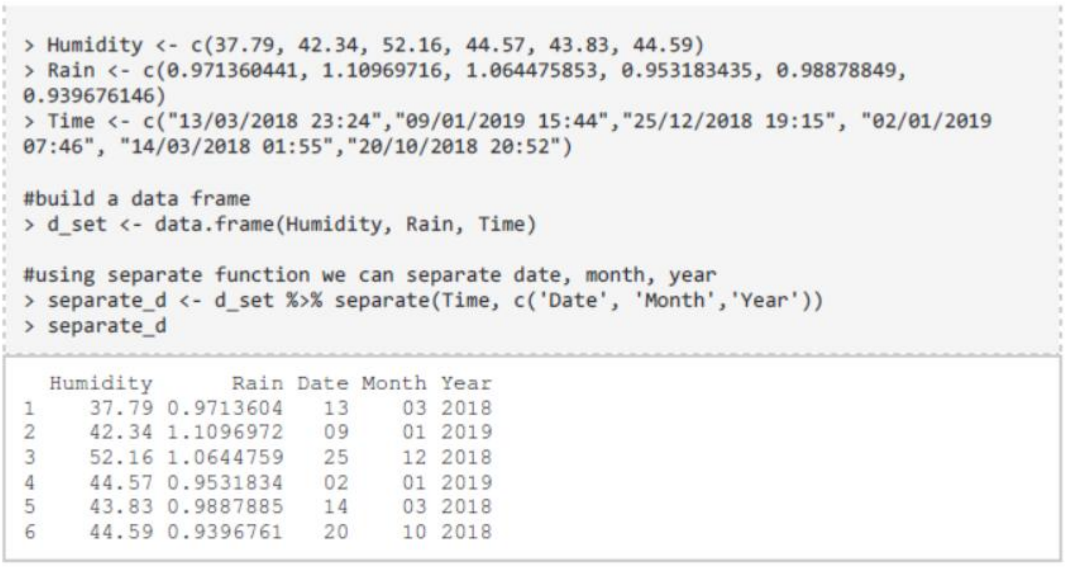
```
#using gather function
> long_t <- tdata %>% gather(Key, Value, weight:Class)
> long_t
```

```
       names age    Key     Value
1    Akanksh  21 weight        55
2      Bhanu  20 weight        49
3    Susheel  25 weight        76
4      Vinay  29 weight        71
5      Varun  33 weight        65
6  Prashanth  32 weight        44
7    Manohar  38 weight        34
8    Akanksh  21  Class     Maths
9      Bhanu  20  Class   Science
10   Susheel  25  Class    Social
11     Vinay  29  Class   Physics
12     Varun  33  Class   Biology
13 Prashanth  32  Class Economics
14   Manohar  38  Class  Accounts
```

Use the Separate function when you have date time variable in the data set. Because a column contains multiple information, it makes sense to split it and use those values individually. The following code shows the usage of the Separate function:

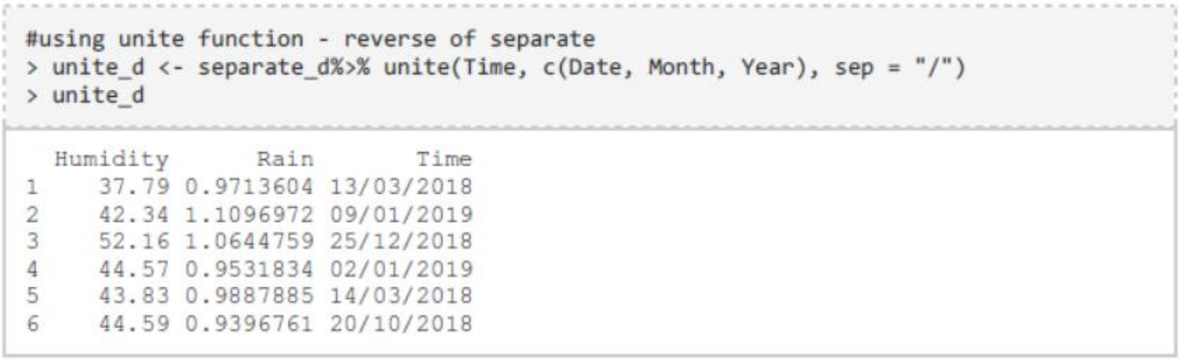


```
#create a data set
```



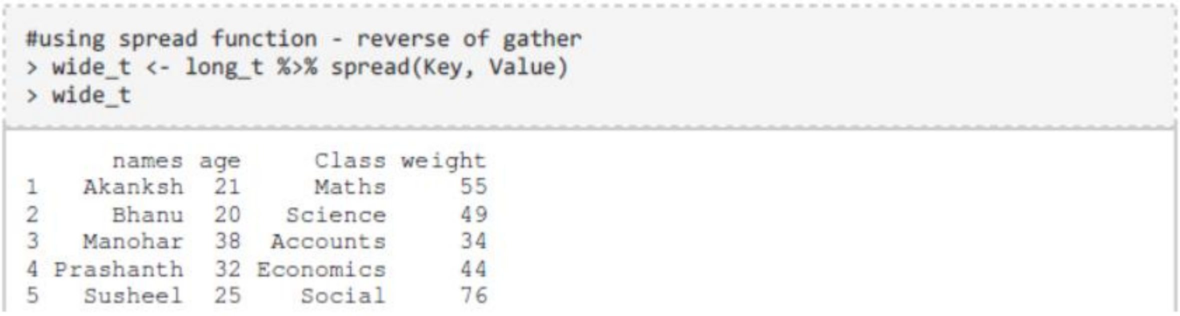
```
> Humidity <- c(37.79, 42.34, 52.16, 44.57, 43.83, 44.59)
> Rain <- c(0.971360441, 1.10969716, 1.064475853, 0.953183435, 0.98878849,
0.939676146)
> Time <- c("13/03/2018 23:24","09/01/2019 15:44","25/12/2018 19:15", "02/01/2019
07:46", "14/03/2018 01:55","20/10/2018 20:52")

#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   13    03 2018
2    42.34 1.1096972   09    01 2019
3    52.16 1.0644759   25    12 2018
4    44.57 0.9531834   02    01 2019
5    43.83 0.9887885   14    03 2018
6    44.59 0.9396761   20    10 2018
```



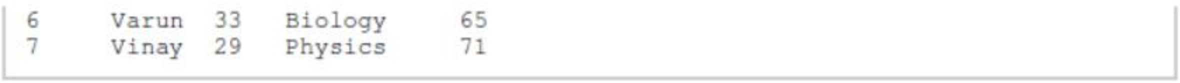
```
#using unite function - reverse of separate
> unite_d <- separate_d%>% unite(Time, c(Date, Month, Year), sep = "/")
> unite_d
```

```
  Humidity      Rain       Time
1    37.79 0.9713604 13/03/2018
2    42.34 1.1096972 09/01/2019
3    52.16 1.0644759 25/12/2018
4    44.57 0.9531834 02/01/2019
5    43.83 0.9887885 14/03/2018
6    44.59 0.9396761 20/10/2018
```



```
#using spread function - reverse of gather
> wide_t <- long_t %>% spread(Key, Value)
> wide_t
```

```
      names age     Class weight
1   Akanksh  21     Maths     55
2     Bhanu  20   Science     49
3   Manohar  38  Accounts     34
4 Prashanth  32 Economics     44
5   Susheel  25    Social     76
```



```
6     Varun  33   Biology     65
7     Vinay  29   Physics     71
```

Lubridate Package

Lubridate package, makes it easier to work with dates and times.

Use the Lubridate package to reduce the issues related to working of data time variable in R. The inbuilt function of this package helps in easy parsing in dates and times. Lubridate is used with data comprising of timely data.

Following are three basic tasks that are accomplished using Lubridate – The update, duration function, and data extraction functions.

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

#current date and time
> now()
```

```
[1] "2019-01-15 22:24:25 IST"
```

```
#assigning current date and time to variable n_time
> n_time <- now()

#using update function
> n_update <- update(n_time, year = 2018, month = 10)
> n_update
```

```
[1] "2018-10-15 22:24:39 IST"
```

```
#add days, months, year, seconds
> d_time <- now()
> d_time + ddays(10)
```

```
[1] "2019-01-25 22:25:15 IST"
```

```
> d_time + dweeks(2)
```

```
[1] "2019-01-29 22:25:15 IST"
```

```
> d_time + dhours(2)
```

```
[1] "2019-01-16 00:25:15 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 n_time.year
1          22            26      33.22655            1        2019
```

Working with Base R Graphics (Scatter Plot, Bar Plot, and Histogram)

ggplot2 Package

ggplot2 offers a wide range of colors and patterns. To understand what is necessary to get started, follow the codes below. You must be proficient with plotting at least 3 graphs – Scatter Plot, Bar Plot, and Histogram.

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.

lements of ggplot2

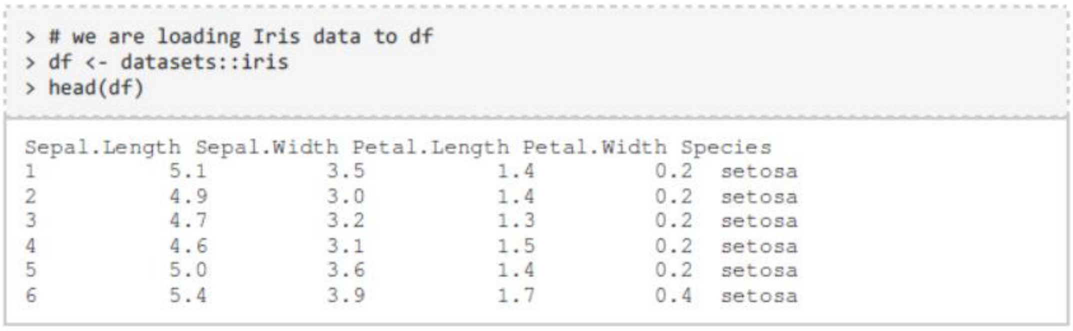
Data: The data-set for which we would want to plot a graph.

Aesthetics: The metrics onto which we plot our data, we can map xaxis, yaxis, fill, col, shape, size.

Geometry: Visual Elements to plot the data.

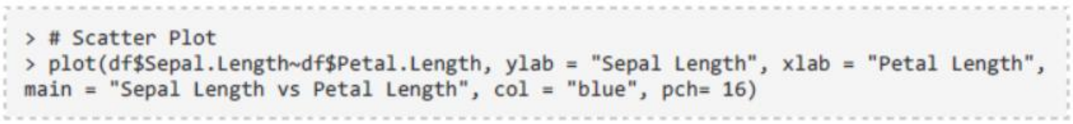
Facet: Groups by which we divide the data.

Working with Base R Graphics

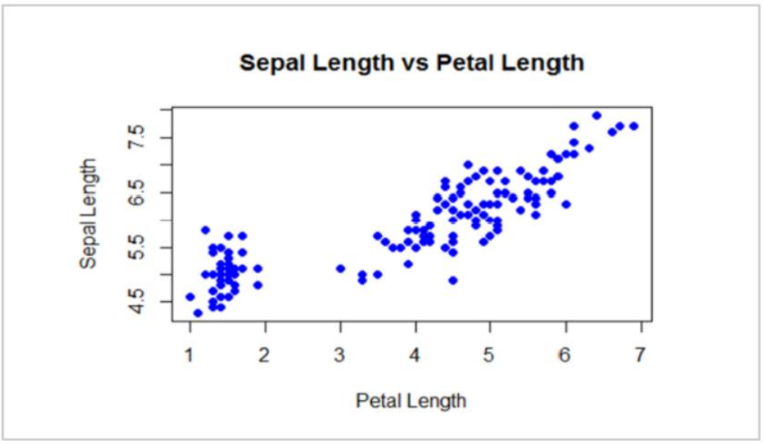


```
> # we are loading Iris data to df
> df <- datasets::iris
> head(df)
```

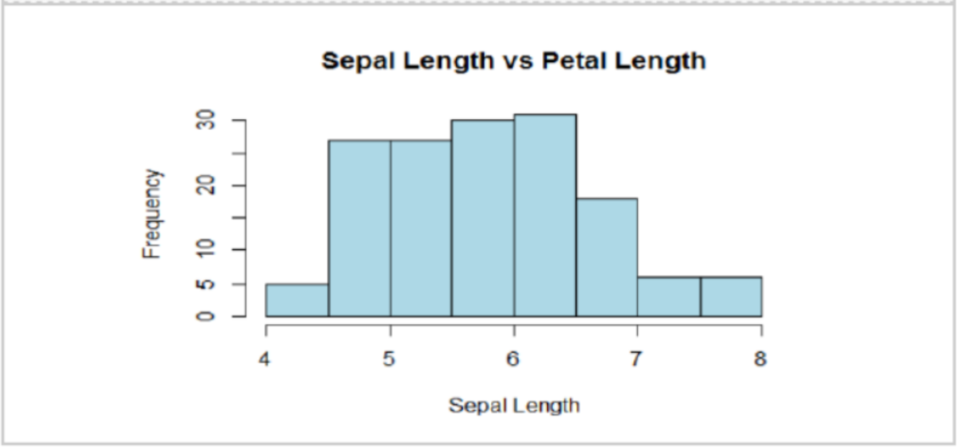
```
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa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



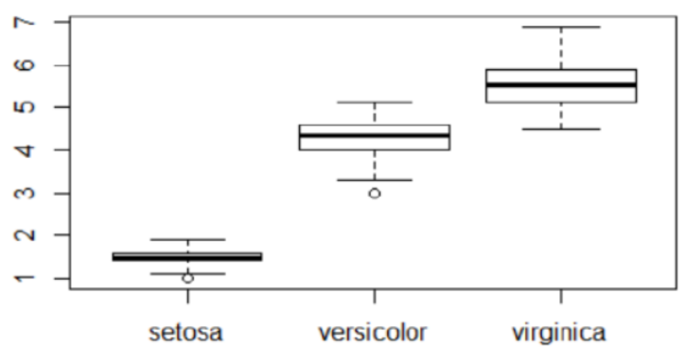
```
> # Scatter Plot
> plot(df$Sepal.Length~df$Petal.Length, ylab = "Sepal Length", xlab = "Petal Length",
main = "Sepal Length vs Petal Length", col = "blue", pch= 16)
```



```
# Histogram
> hist(df$Sepal.Length,xlab = "Sepal Length", main = "Sepal Length vs Petal Length",
col = "light blue", pch= 16)
```



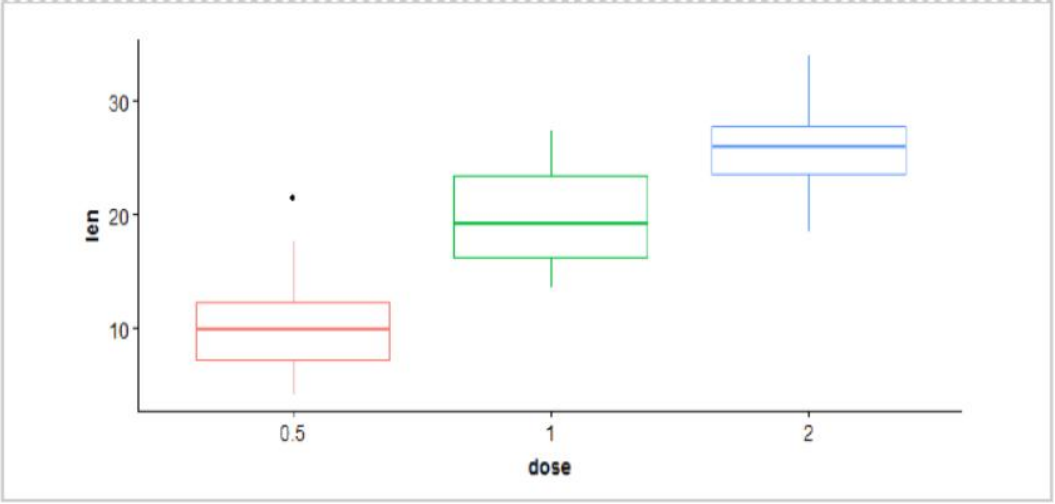
```
# Boxplot
> boxplot(df$Petal.Length~df$Species)
```



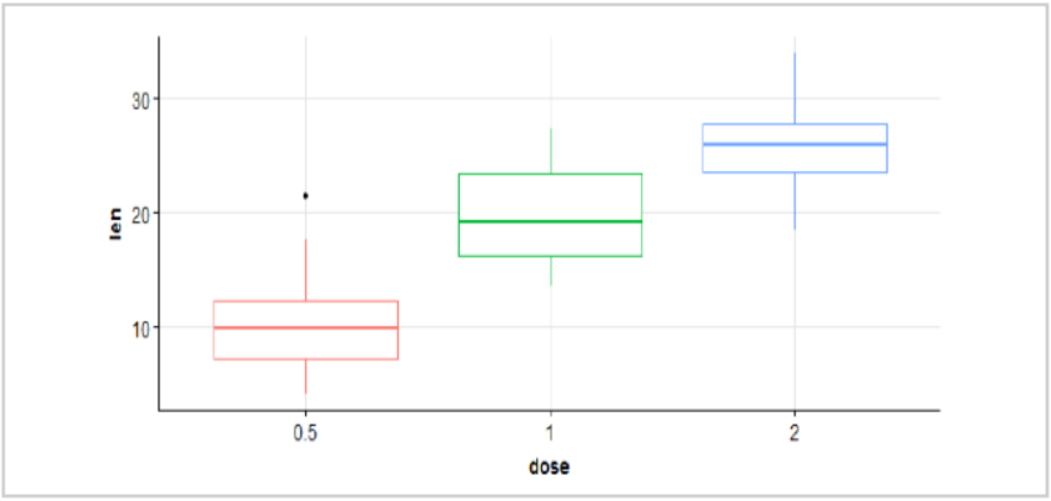
```
> library(ggplot2) # Loading ggplot2 package
> library(grid)

# Loading Tooth Growth data
> df <- datasets::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
```

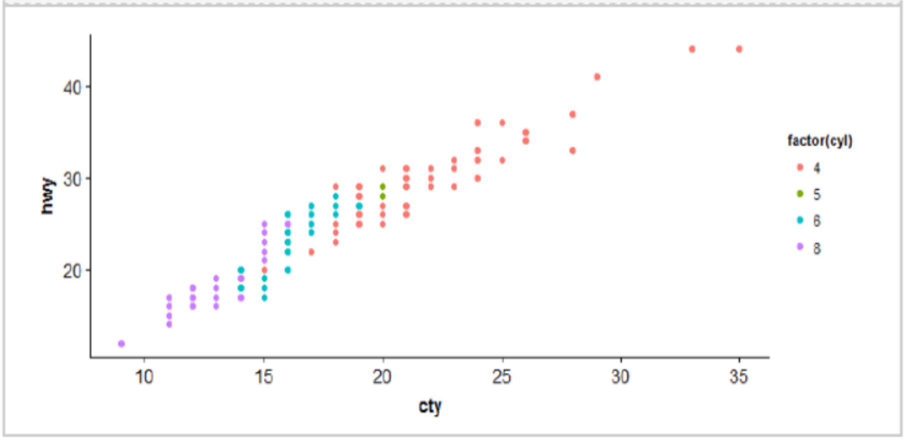
```
# Boxplot
> bp <- ggplot(df, aes(x = dose, y = len, color = dose)) + geom_boxplot() + theme(leg
end.position = 'none')

> bp
```



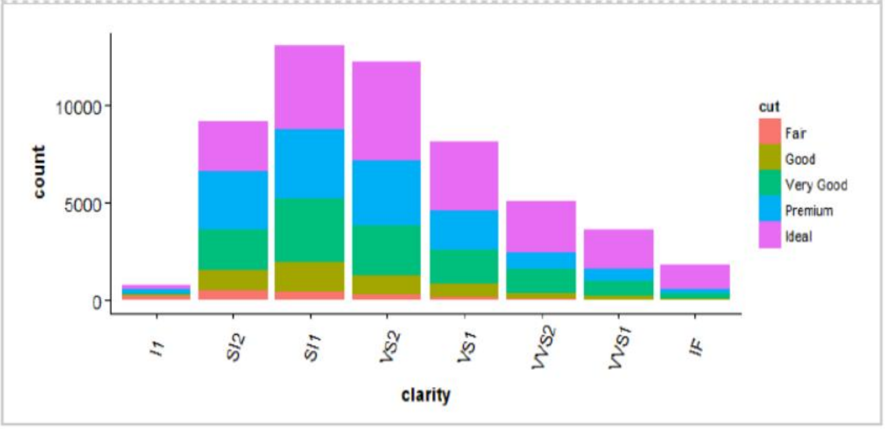
```
#add gridlines
> bp + background_grid(major = "xy", minor = 'none')
```



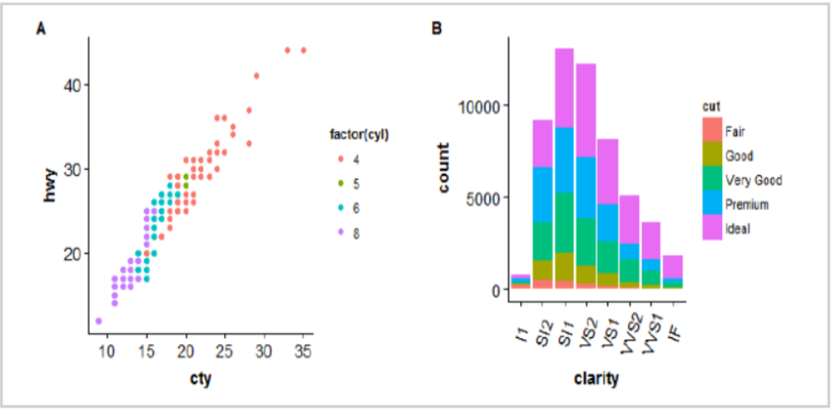
```
# Scatterplot
> sp <- ggplot(mpg, aes(x = cty, y = hwy, color = factor(cyl)))+geom_point(size = 2.5
)
> sp
```

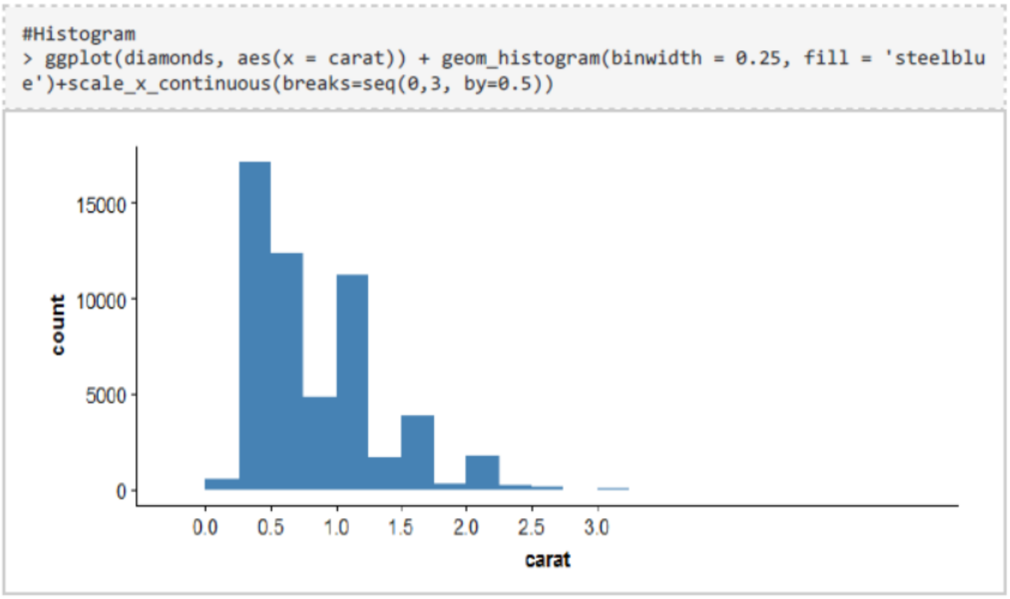


```
#Barplot
> bp <- ggplot(diamonds, aes(clarity, fill = cut)) + geom_bar() +theme(axis.text.x =
element_text(angle = 70, vjust = 0.5))

> bp
```



```
# Compare two plots
> plot_grid(sp, bp, labels = c("A","B"), ncol = 2, nrow = 1)
```



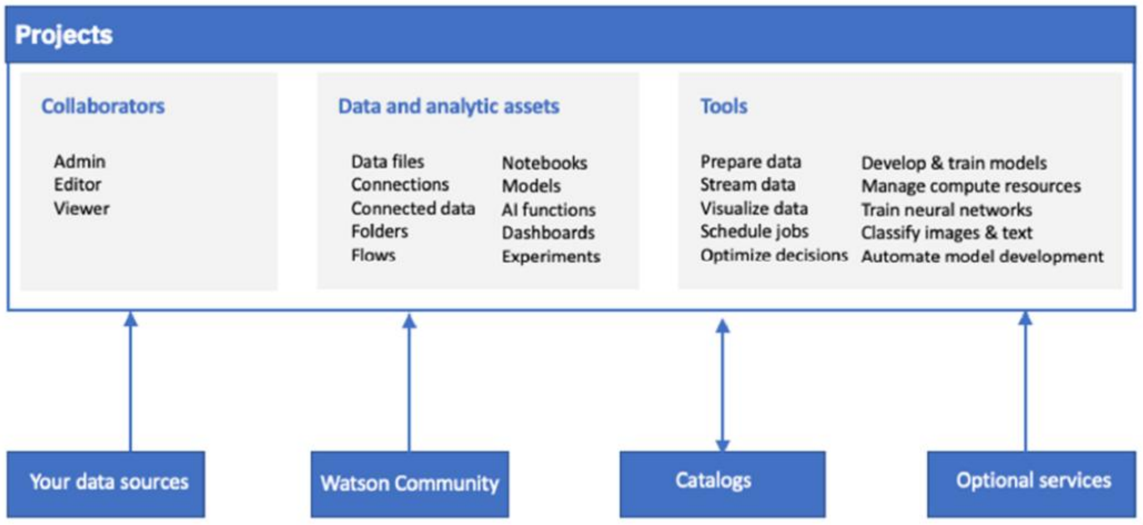


```
#Histogram
> ggplot(diamonds, aes(x = carat)) + geom_histogram(binwidth = 0.25, fill = 'steelblu
e')+scale_x_continuous(breaks=seq(0,3, by=0.5))
```

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.



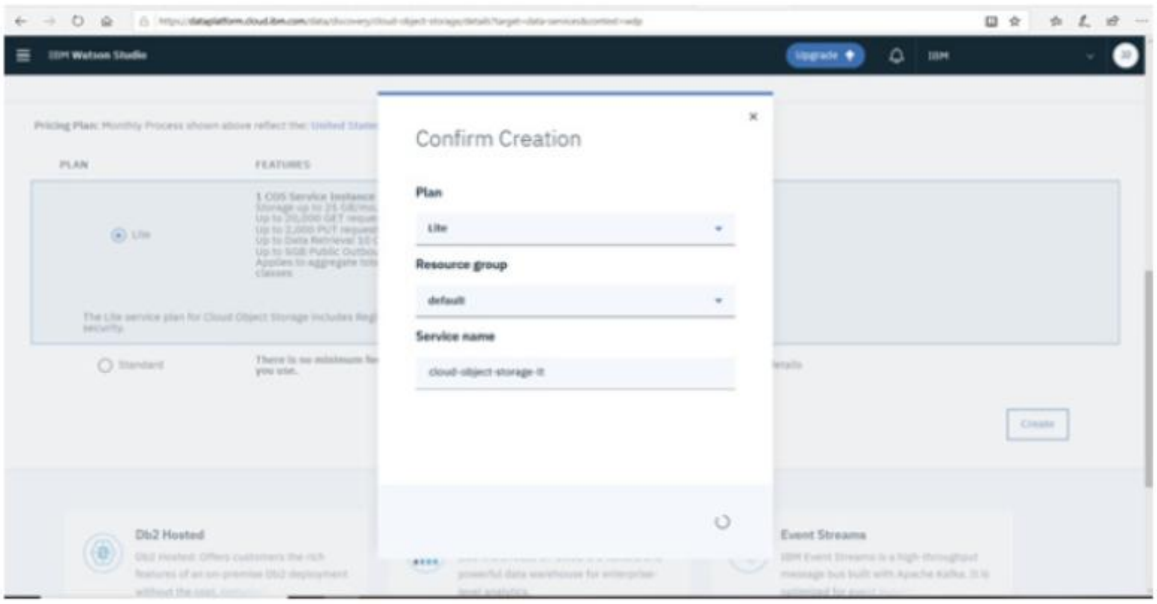
Visualizing information in graphical ways can give you insights into your data. By enabling you to look at and explore data from different perspectives, visualizations can help you identify patterns, connections, and relationships within that data as well as understand large amounts of information very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


Confirm Creation

Plan: Lite

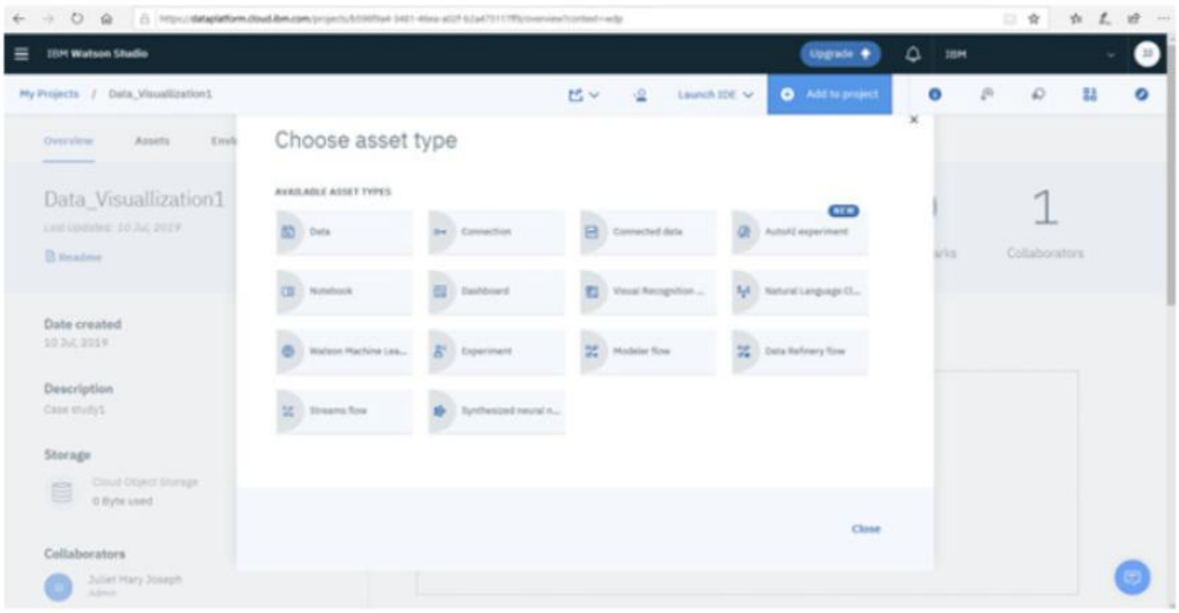
Resource group: default

Service name: cloud-object-storage-lt

To add data files to a project:

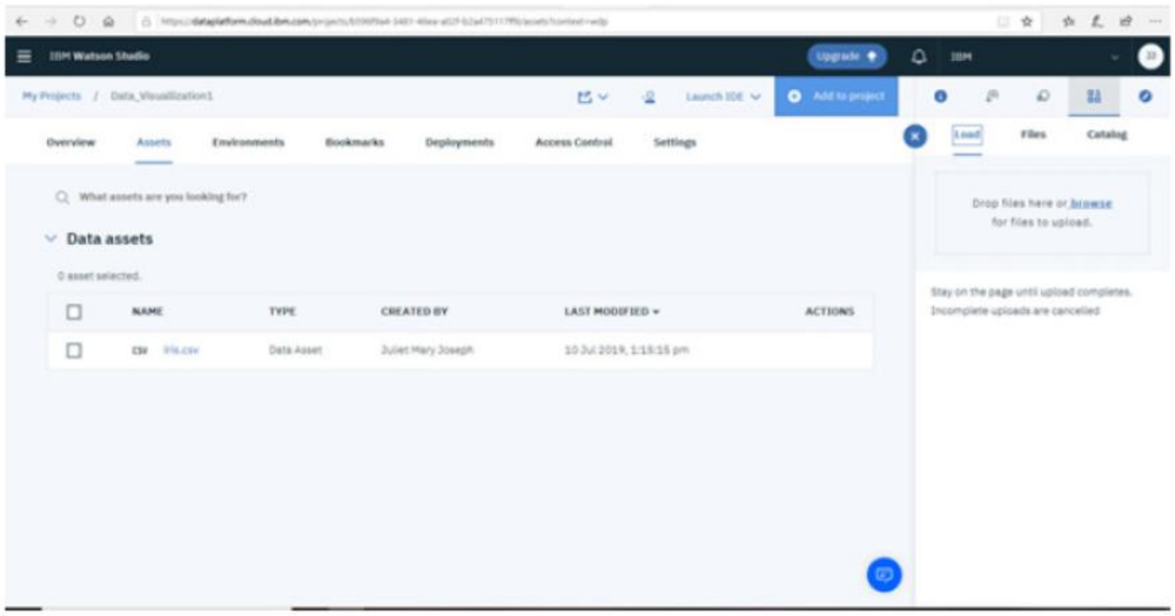
- From your project's Assets page, click Add to project > Data or click the Find and add data icon ().You can also click the Find and add data icon from within a notebook or canvas.

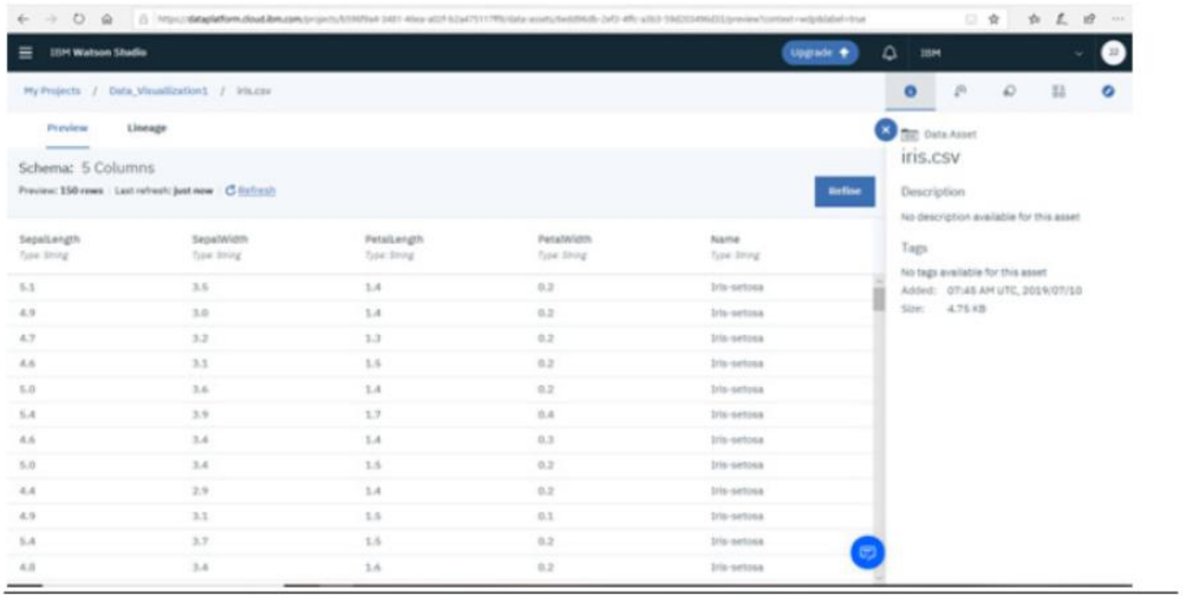
- In the Load pane that opens, browse for the files or drag them onto the pane. You must stay on the page until the load is complete. You can cancel an ongoing load process if you want to stop loading a file.



Case Study:

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





IBM Watson Studio

My Projects / Data_Visualization1 / iris.csv

Preview Lineage

Schema: 5 Columns

Preview: 150 rows | Last refresh: just now Refresh

SepalLength Type: String	SepalWidth Type: String	PetalLength Type: String	PetalWidth Type: String	Name Type: String
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5.0	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa

Data Asset

iris.csv

Description

No description available for this asset

Tags

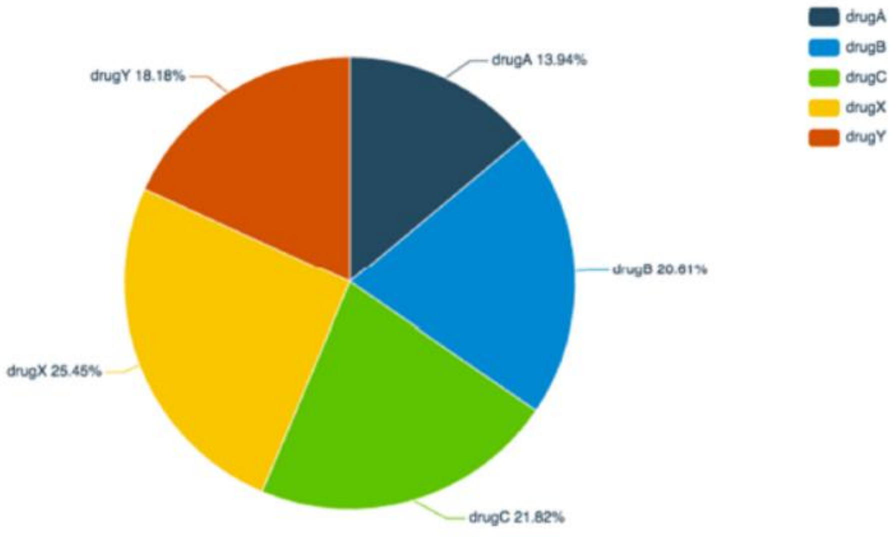
No tags available for this asset

Added: 07:45 AM UTC, 2019/07/10

Size: 4.75 KB

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.**



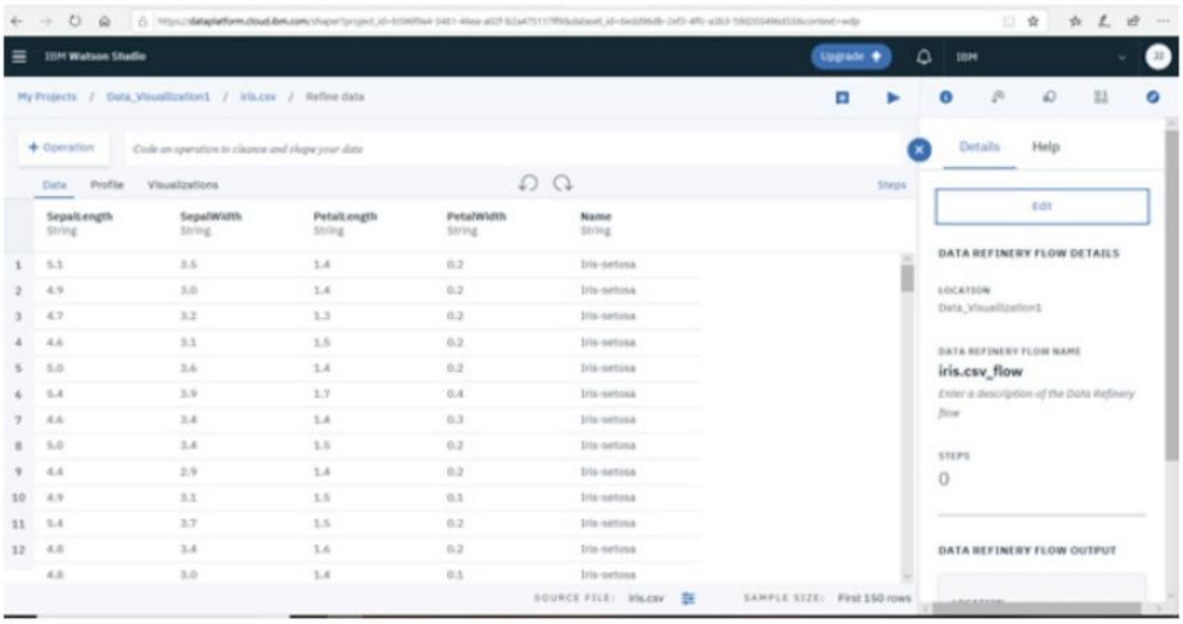
To visualize your data:

- 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.

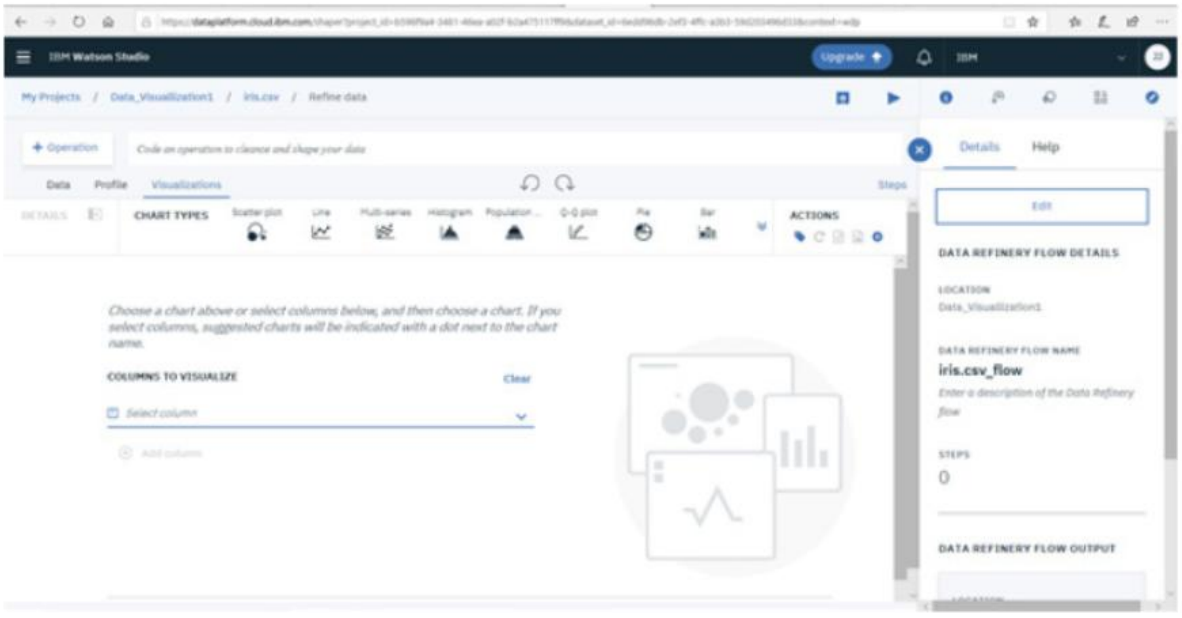
2. Select the columns that you want to work with. Suggested charts will be indicated with a dot next to the chart name. Click a chart to visualize your data.

Click on refine


IBM Watson Studio
Upgrade
IBM
My Projects / Data_Visualization1 / iris.csv / Refine data
+ Operation
Code an operation to cleanse and shape your data
Data
Profile
Visualizations
Steps
Details
Help
Edit
DATA REFINERY FLOW DETAILS
LOCATION
Data_Visualization1
DATA REFINERY FLOW NAME
iris.csv_flow
Enter a description of the Data Refinery flow
STEPS
0
DATA REFINERY FLOW OUTPUT
SOURCE FILE: iris.csv
SAMPLE SIZE: First 150 rows

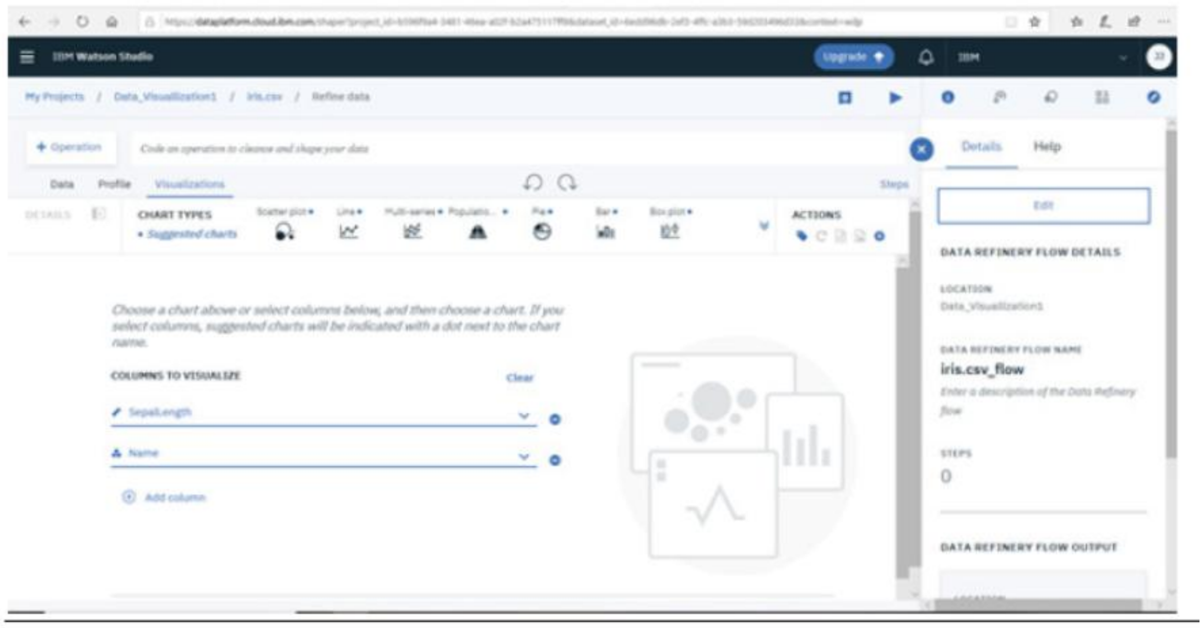
	SepalLength String	SepalWidth String	PetalLength String	PetalWidth String	Name String
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa
7	4.6	3.4	1.4	0.3	Iris-setosa
8	5.0	3.4	1.5	0.2	Iris-setosa
9	4.4	2.9	1.4	0.2	Iris-setosa
10	4.9	3.1	1.5	0.1	Iris-setosa
11	5.4	3.7	1.5	0.2	Iris-setosa
12	4.8	3.4	1.6	0.2	Iris-setosa
	4.8	3.0	1.4	0.1	Iris-setosa

Click on Visualization tab:



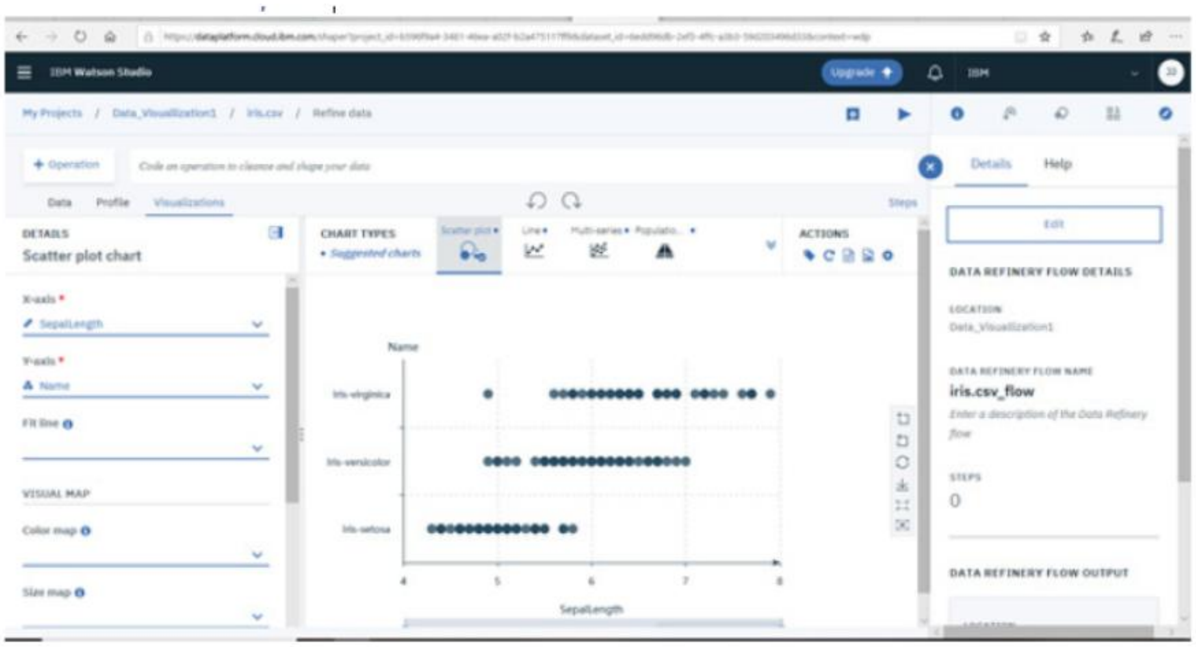
Choose a chart above or select columns below, and then choose a chart. If you select columns, suggested charts will be indicated with a dot next to the chart name.

Add the columns by selecting.

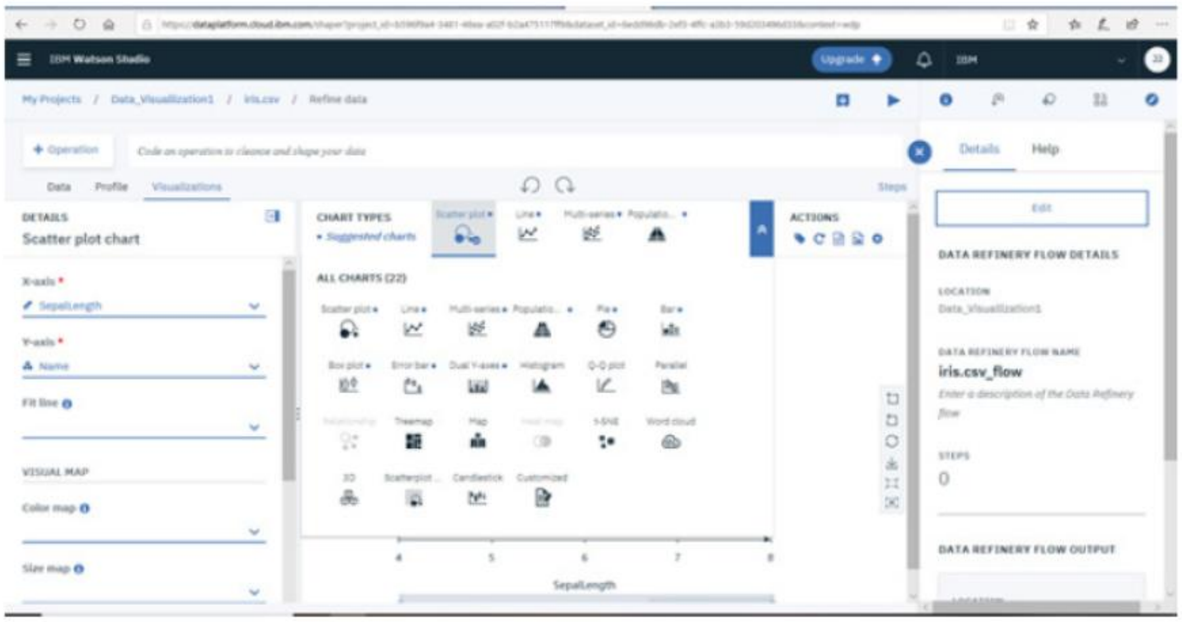


Visualization of Data on Watson Studio

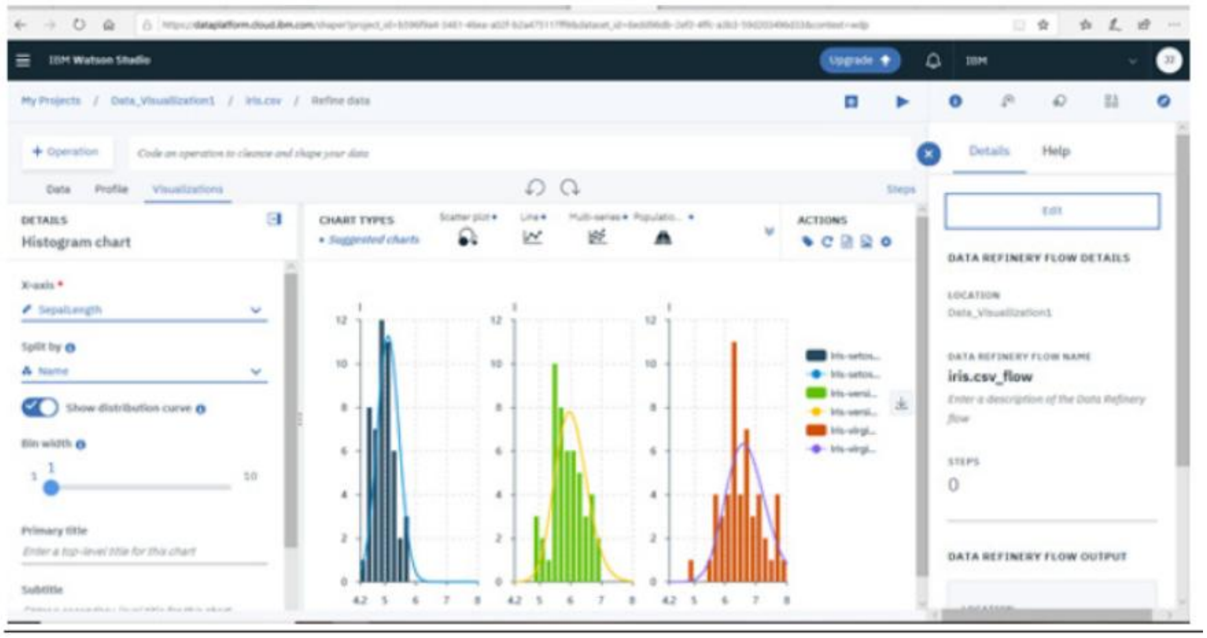
Select Scatter plot:



Various types of option to visualize the data:



Select Histogram and select the x axis and y axis :



UNIT – III

Python: Introduction toPython, How toInstall, Introduction to JupyterNotebook, Python scriptingbasics, Numpy andPandas-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 / Join datasets

Python and Anaconda Installation

Introduction to Anaconda -

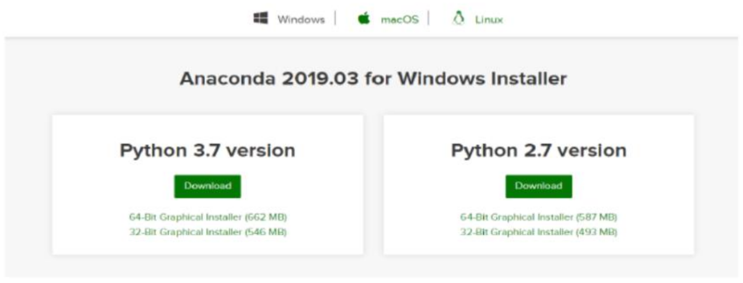
Anaconda is a package manager, an environment manager, and Python distribution that contains a collection of many open source packages.

This is advantageous as when you are working on a data science project, you will find that you need many different packages (NumPy, scikit-learn, SciPy, pandas to name a few), which an installation of Anaconda comes preinstalled with. If you need additional packages after installing Anaconda, you can use Anaconda's package manager, conda, or pip to install those packages.

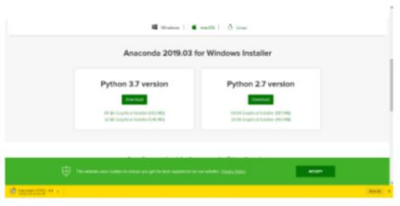
This is highly advantageous as you don't have to manage dependencies between multiple packages yourself. Conda even makes it easy to switch between Python 2 and 3.

Anaconda Installation -

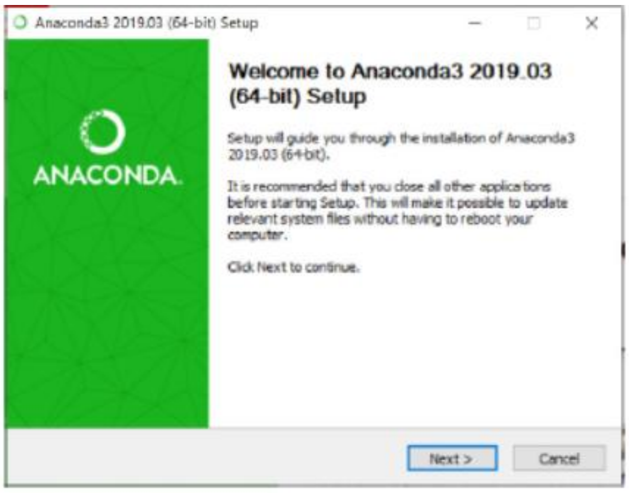
- Go to the Anaconda Website and choose a Python 3.x graphical installer (A) or a Python 2.x graphical installer.


Windows
macOS
Linux
Anaconda 2019.03 for Windows Installer
Python 3.7 version
Download
64-Bit Graphical Installer (662 MB)
32-Bit Graphical Installer (546 MB)
Python 2.7 version
Download
64-Bit Graphical Installer (587 MB)
32-Bit Graphical Installer (493 MB)

- Locate your download and double click it.



When the screen below appears, click on Next


Anaconda3 2019.03 (64-bit) Setup

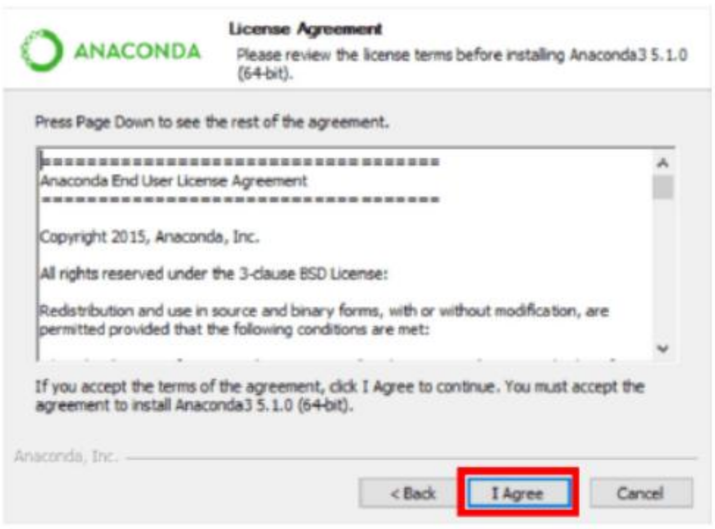
Welcome to Anaconda3 2019.03 (64-bit) Setup

Setup will guide you through the installation of Anaconda3 2019.03 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

- Read the license agreement and click on I Agree.


License Agreement

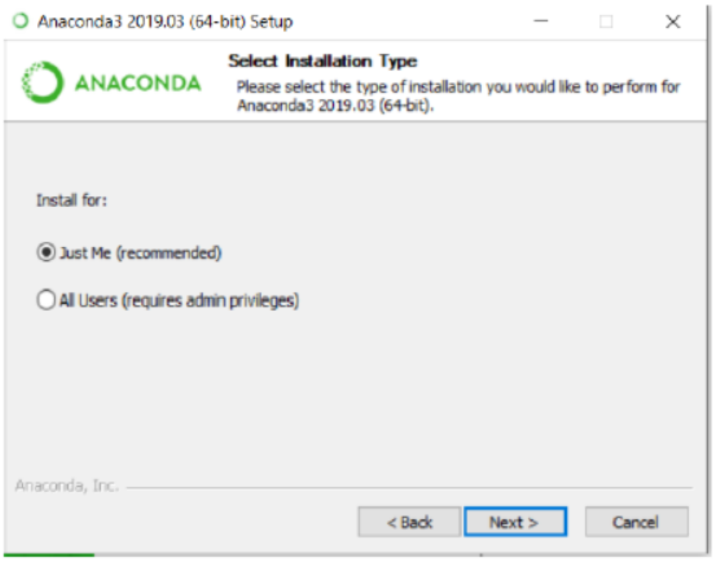
Please review the license terms before installing Anaconda3 5.1.0 (64-bit).

Press Page Down to see the rest of the agreement.

```
===================================
Anaconda End User License Agreement
===================================

Copyright 2015, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 5.1.0 (64-bit).

Click on Next.


Anaconda3 2019.03 (64-bit) Setup

Select Installation Type

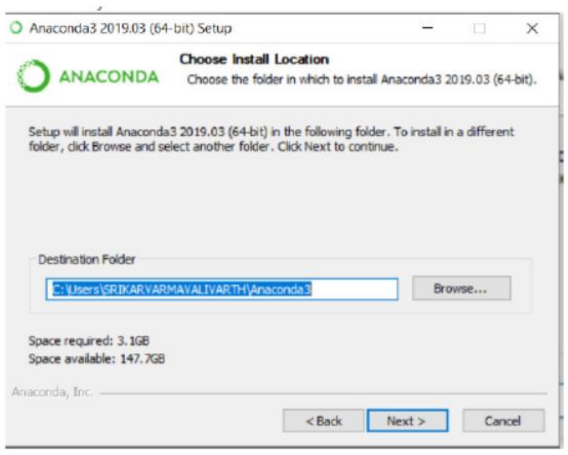
Please select the type of installation you would like to perform for Anaconda3 2019.03 (64-bit).

Install for:

[✓ Just Me (recommended)]

All Users (requires admin privileges)

Note your installation location and then click Next.



Anaconda3 2019.03 (64-bit) Setup

Choose Install Location

Choose the folder in which to install Anaconda3 2019.03 (64-bit).

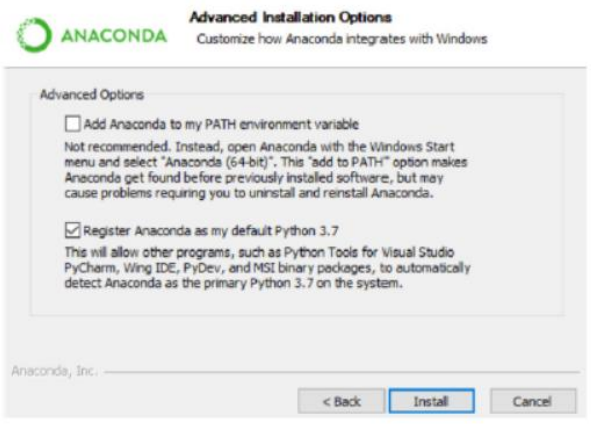
Setup will install Anaconda3 2019.03 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

C:\Users\SRIKARVARMAVALIVARTH\Anaconda3

Space required: 3.1GB
Space available: 147.7GB

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.



Advanced Installation Options

Customize how Anaconda integrates with Windows

Advanced Options

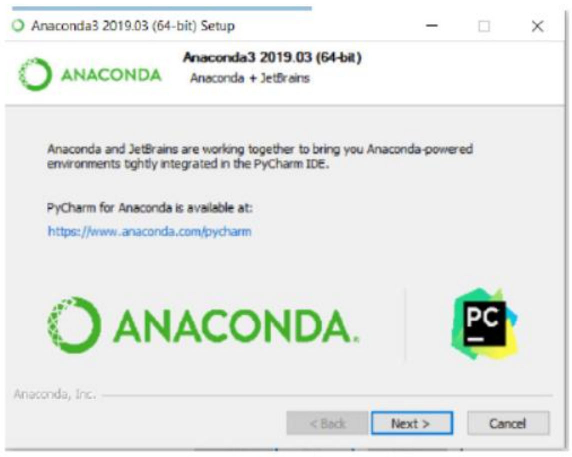
☐ Add Anaconda to my PATH environment variable

Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

☑ Register Anaconda as my default Python 3.7

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.7 on the system.

After that click on next.



Anaconda3 2019.03 (64-bit) Setup

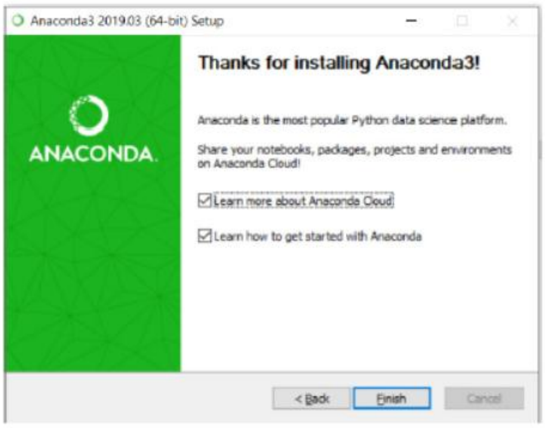
Anaconda3 2019.03 (64-bit)

Anaconda + JetBrains

Anaconda and JetBrains are working together to bring you Anaconda-powered environments tightly integrated in the PyCharm IDE.

PyCharm for Anaconda is available at:
https://www.anaconda.com/pycharm

Click Finish.


Anaconda3 2019.03 (64-bit) Setup

Thanks for installing Anaconda3!

Anaconda is the most popular Python data science platform.

Share your notebooks, packages, projects and environments on Anaconda Cloud!

☑ Learn more about Anaconda Cloud

☑ Learn how to get started with Anaconda

We need to set anaconda path to system environmental variables.

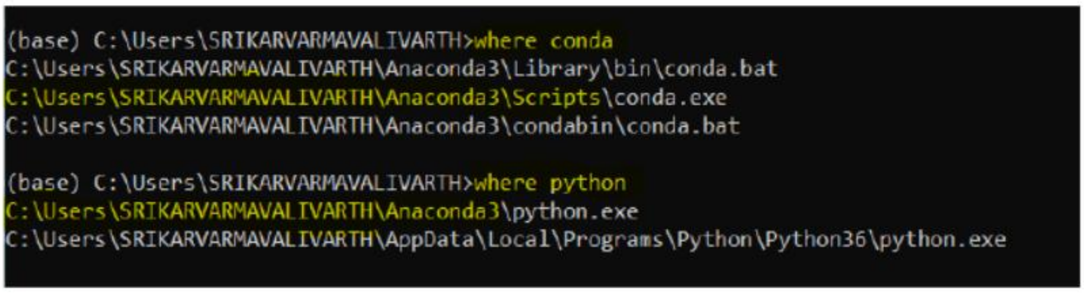
Open a Command Prompt. Check if you already have Anaconda added to your path. Enter the commands below into your Command Prompt.

Conda –version

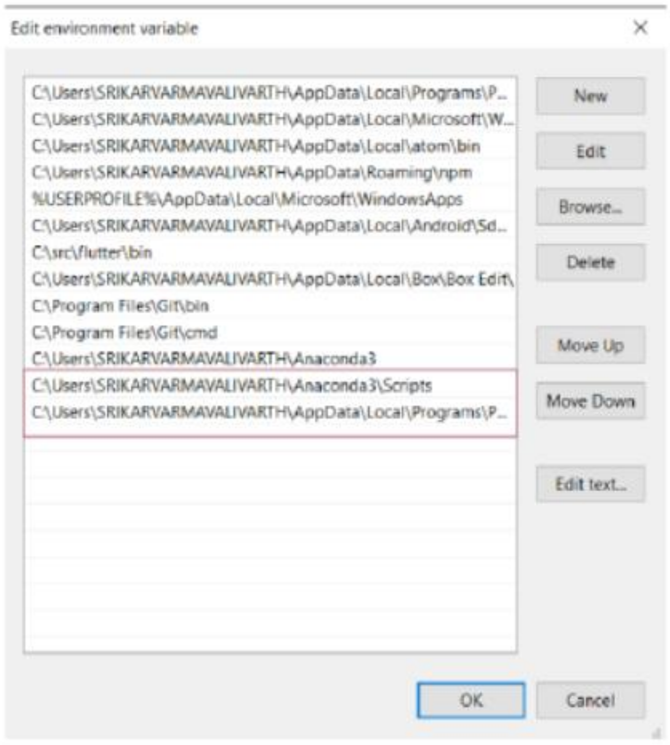
Python –version

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.

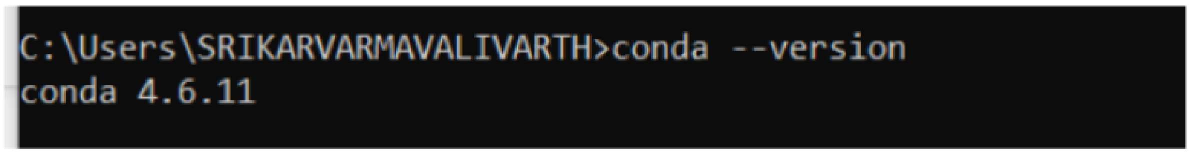

```
(base) C:\Users\SRIKARVARMAVALIVARTH>where conda
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\Library\bin\conda.bat
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\Scripts\conda.exe
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\condabin\conda.bat

(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)



```
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\Programs\P...
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\Microsoft\W...
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\atom\bin
C:\Users\SRIKARVARMAVALIVARTH\AppData\Roaming\npm
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\Android\Sd...
C:\src\flutter\bin
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\Box\Box Edit\
C:\Program Files\Git\bin
C:\Program Files\Git\cmd
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3
C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\Scripts
C:\Users\SRIKARVARMAVALIVARTH\AppData\Local\Programs\P...
```

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



```
C:\Users\SRIKARVARMAVALIVARTH>conda --version
conda 4.6.11
```

Conda installation is successful

Introduction to Jupyter Notebook

What is Jupyter

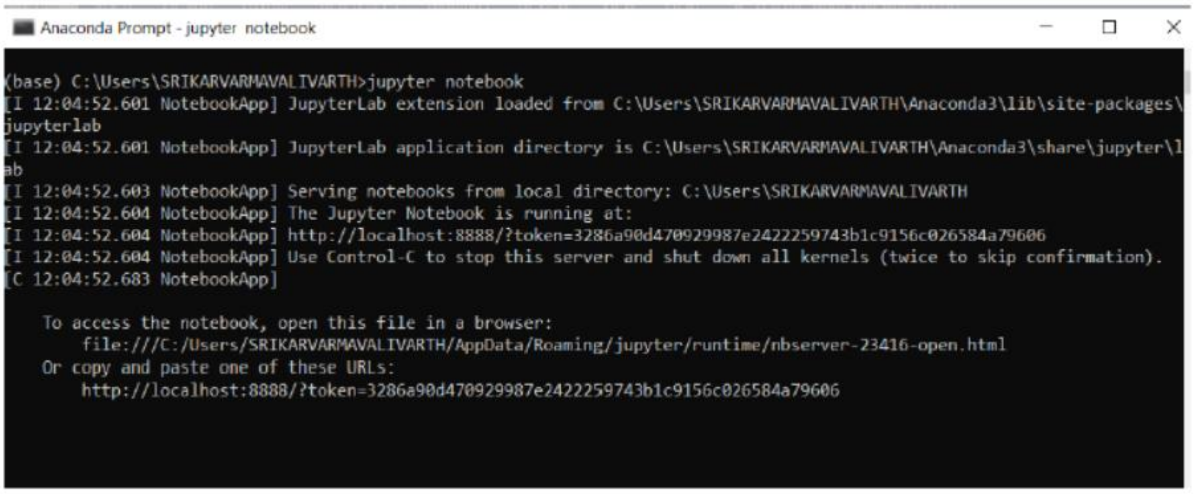
The Jupyter Notebook is an open source web application that you can use to create and share documents that contain live code, equations, visualizations, and text. Jupyter Notebook is maintained by the people at Project Jupyter.

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

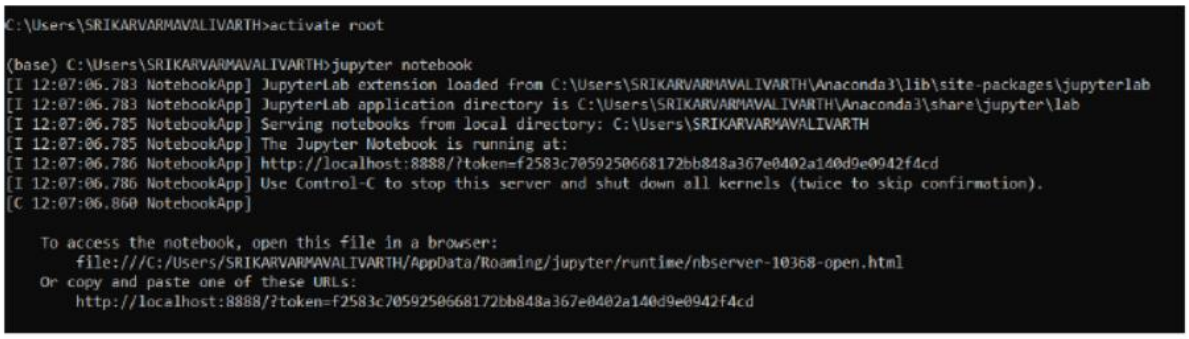
Installing Anaconda Distribution will also include Jupyter Notebook.

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



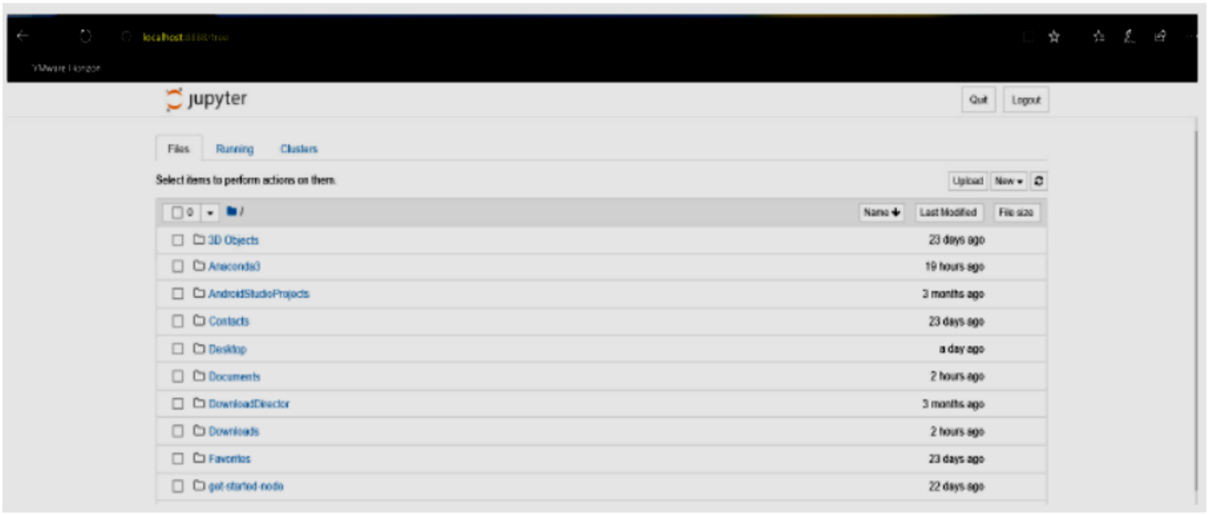
```
Anaconda Prompt - jupyter notebook

(base) C:\Users\SRIKARVARMAVALIVARTH>jupyter notebook
[I 12:04:52.601 NotebookApp] JupyterLab extension loaded from C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\lib\site-packages\
jupyterlab
[I 12:04:52.601 NotebookApp] JupyterLab application directory is C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\share\jupyter\l
ab
[I 12:04:52.603 NotebookApp] Serving notebooks from local directory: C:\Users\SRIKARVARMAVALIVARTH
[I 12:04:52.604 NotebookApp] The Jupyter Notebook is running at:
[I 12:04:52.604 NotebookApp] http://localhost:8888/?token=3286a90d470929987e2422259743b1c9156c026584a79606
[I 12:04:52.604 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 12:04:52.683 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///C:/Users/SRIKARVARMAVALIVARTH/AppData/Roaming/jupyter/runtime/nbserver-23416-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3286a90d470929987e2422259743b1c9156c026584a79606
```

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



```
C:\Users\SRIKARVARMAVALIVARTH>activate root

(base) C:\Users\SRIKARVARMAVALIVARTH>jupyter notebook
[I 12:07:06.783 NotebookApp] JupyterLab extension loaded from C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\lib\site-packages\jupyterlab
[I 12:07:06.783 NotebookApp] JupyterLab application directory is C:\Users\SRIKARVARMAVALIVARTH\Anaconda3\share\jupyter\lab
[I 12:07:06.785 NotebookApp] Serving notebooks from local directory: C:\Users\SRIKARVARMAVALIVARTH
[I 12:07:06.785 NotebookApp] The Jupyter Notebook is running at:
[I 12:07:06.786 NotebookApp] http://localhost:8888/?token=f2583c7059250668172bb848a367e0402a140d9e0942f4cd
[I 12:07:06.786 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 12:07:06.860 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///C:/Users/SRIKARVARMAVALIVARTH/AppData/Roaming/jupyter/runtime/nbserver-10368-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=f2583c7059250668172bb848a367e0402a140d9e0942f4cd
```

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



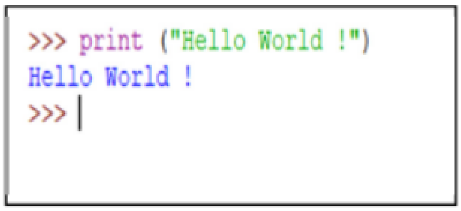
Select items to perform actions on them.

/	Name	Last Modified	File size
3D Objects		23 days ago	
Anaconda3		19 hours ago	
AndroidStudioProjects		3 months ago	
Contacts		23 days ago	
Desktop		a day ago	
Documents		2 hours ago	
DownloadDirector		3 months ago	
Downloads		2 hours ago	
Favorites		23 days ago	
get-started-node		22 days ago	

Python Scripting Basics

First Program in Python

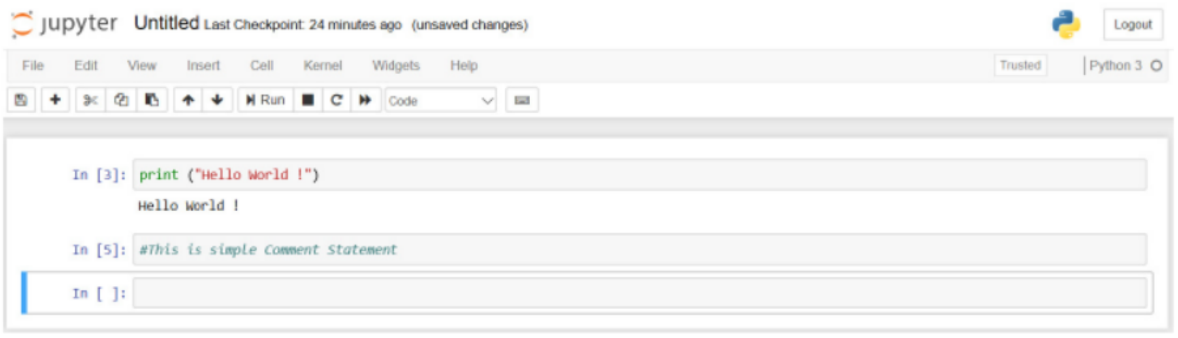
Let’s try your first program in Python.



```
>>> 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 Jupiter notebook in this course, 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.

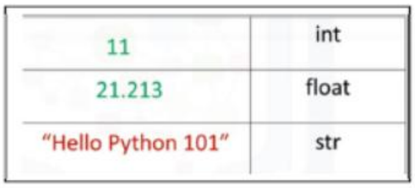

jupyter Untitled Last Checkpoint: 24 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [3]: print ("Hello World !")
        Hello World !

In [5]: #This is simple Comment Statement

In [ ]:
```

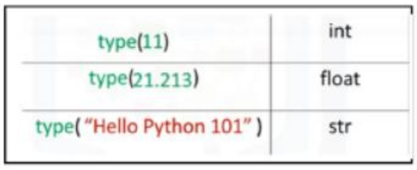
It’s customary to comment your code. This tells other people what your code does. You simply put a hash symbol preceding your comment. When you run the code, Python will ignore the comment.

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

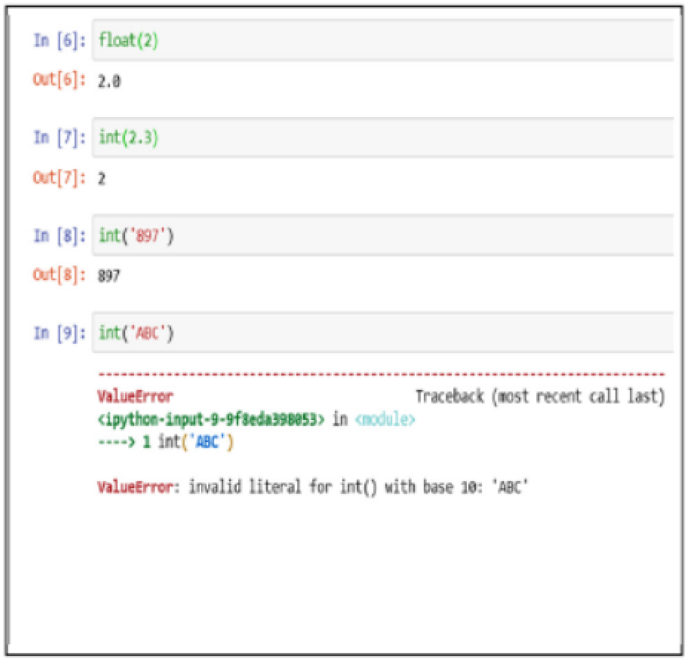


type(11)	int
type(21.213)	float
type("Hello Python 101")	str

The following chart summarizes three data types for the last examples. The first column 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.

We can continue the process, zooming in for different numbers. Of course, there is a limit, but it is quite small. You can change the type of the expression in Python; this is called type-casting. You can convert an int to a float. For example, you can convert or cast the integer 2 to a float 2.

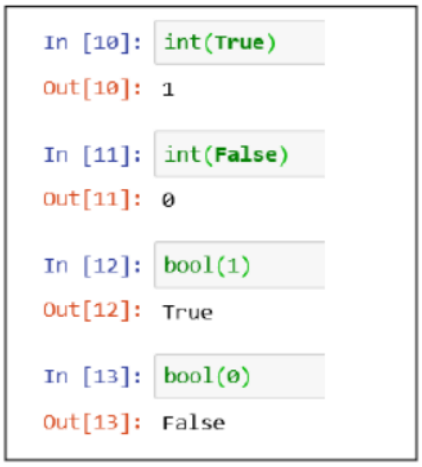


```
In [6]: float(2)
Out[6]: 2.0

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.

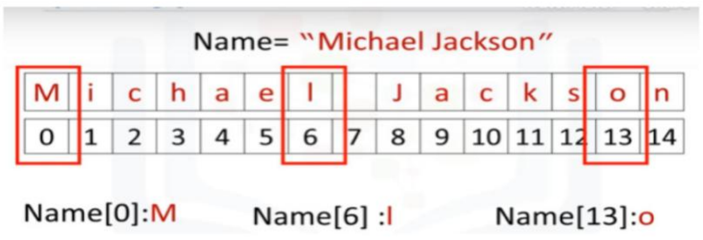
If we cast a Boolean false to an integer or float, we get a zero. If you cast a 1 to a Boolean, you get a true. Similarly, if you cast a 0 to a Boolean, you get a false.



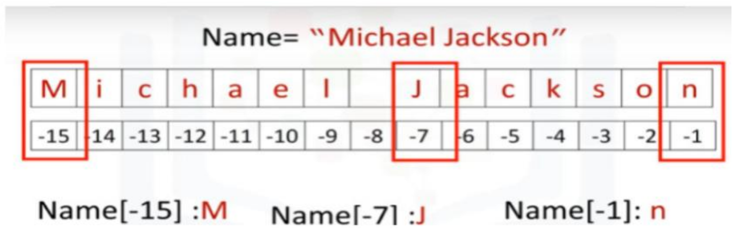
```
In [10]: int(True)
Out[10]: 1

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

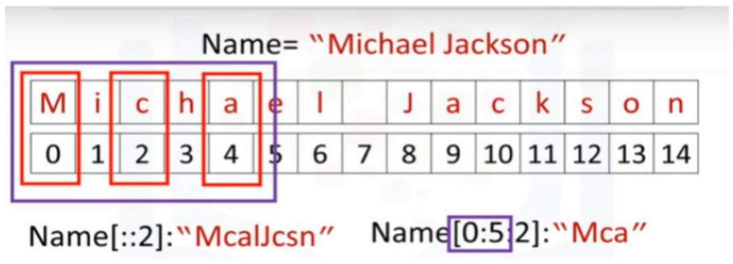
In Python, a string is a sequence of characters. A string is contained within two quotes: You could also use single quotes. A string can be spaces, or digits. A string can also be special characters. We can bind or assign a string to another variable. It is helpful to think of a string as an ordered 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



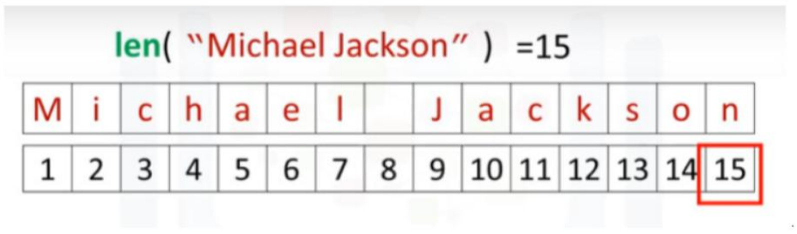
follows. We can access index 6. Moreover, we can access the 13th index. We can also use negative indexing with strings. The last element is given by the index -1. The first element can be obtained by index -15, and so on.

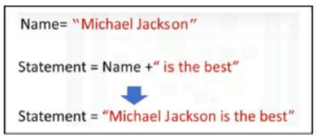


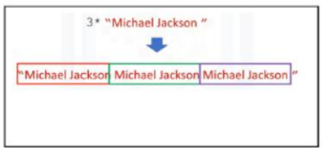
We can bind a string to another variable. It is helpful to think of string as a list or tuple. We can treat the string as a sequence and perform sequence operations. We can also input a string value as follows. The 2 indicates we select every second variable. We can also incorporate slicing.



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.




Name= "Michael Jackson"
Statement = Name +" is the best"
Statement = "Michael Jackson is the best"



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.

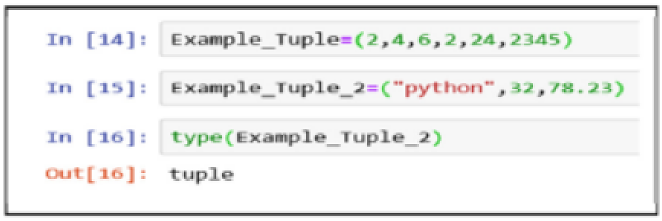
Python COLLECTION (or)Arrays

There are four collection data types in the Python programming language:

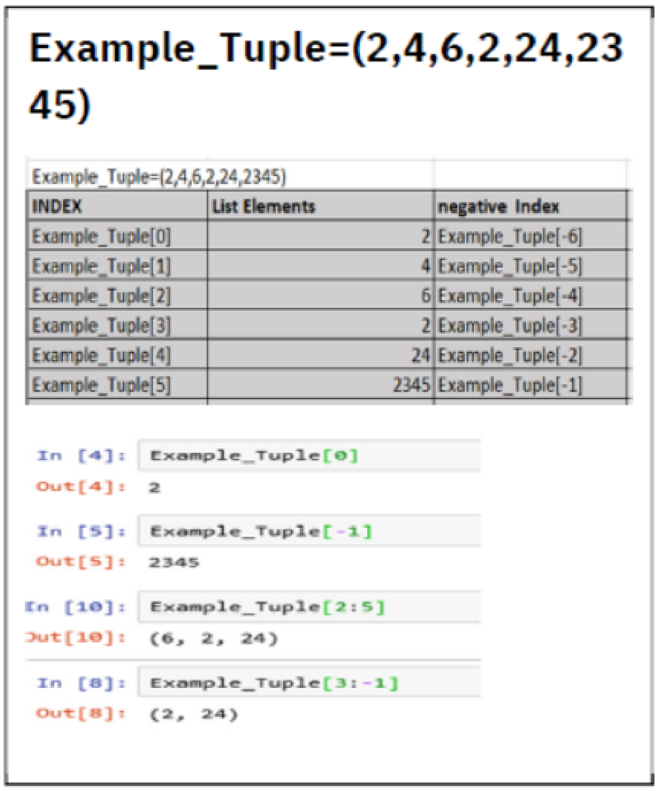
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.

- **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)

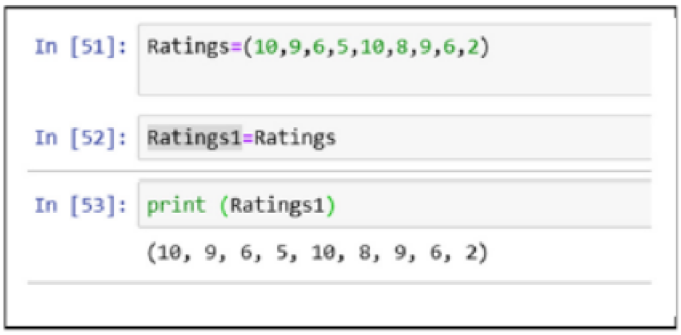
Example_Tuple=(2,4,6,2,24,2345)		
INDEX	List Elements	negative Index
Example_Tuple[0]	2	Example_Tuple[-6]
Example_Tuple[1]	4	Example_Tuple[-5]
Example_Tuple[2]	6	Example_Tuple[-4]
Example_Tuple[3]	2	Example_Tuple[-3]
Example_Tuple[4]	24	Example_Tuple[-2]
Example_Tuple[5]	2345	Example_Tuple[-1]

```
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

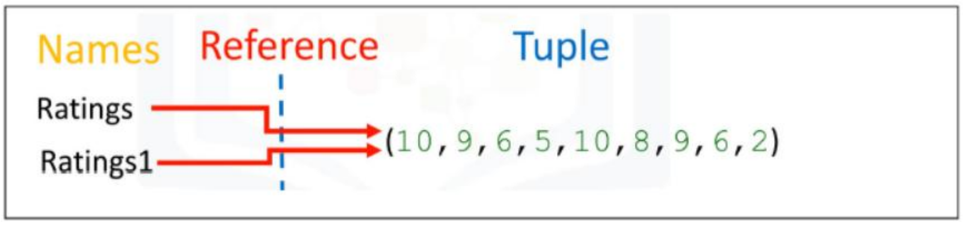
Each element of a tuple can be accessed via an index. The element in the tuple can be accessed by the name of the tuple followed by a square bracket with the index number. Use the square brackets for slicing along with the index or indices to obtain value available at that index.

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



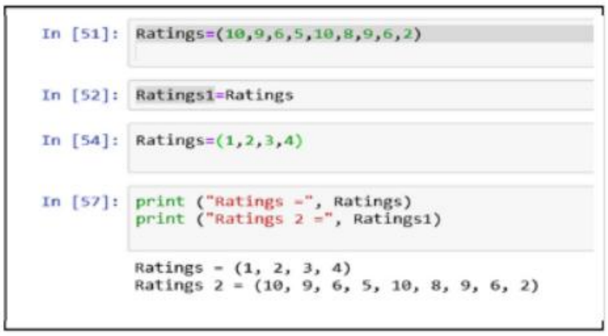
```
In [51]: Ratings=(10,9,6,5,10,8,9,6,2)

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.

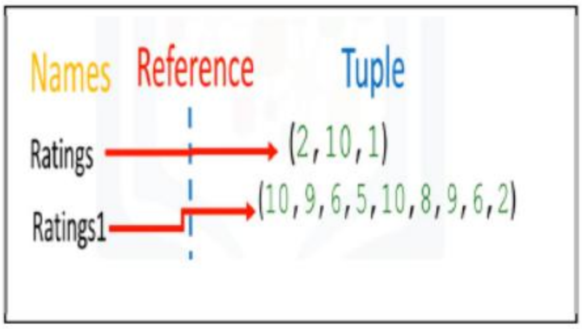


Let's say we want to change the element at index 2. Because tuples are immutable, we can't. Therefore, Ratings 1 will not be affected by a change in Rating because the tuple is immutable i.e., we can't change it.

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



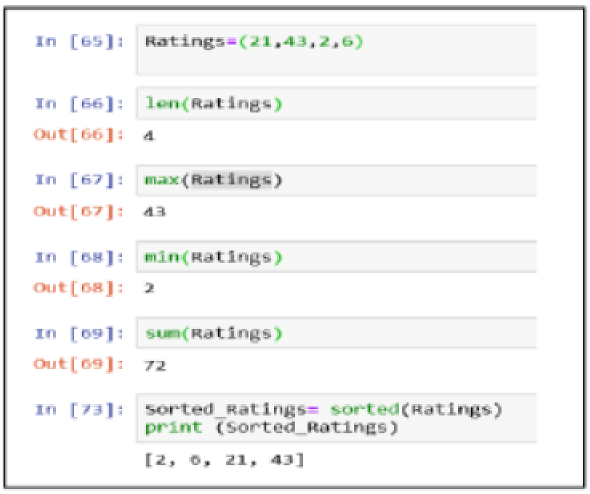
```
In [51]: Ratings=(10,9,6,5,10,8,9,6,2)

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.

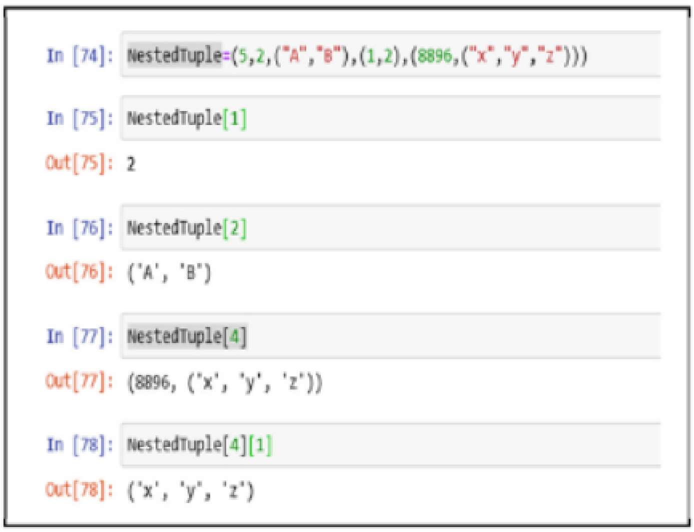
A tuple can contain other tuples as well as other complex data types; this is called nesting.



```
In [65]: Ratings=(21,43,2,6)

In [66]: len(Ratings)
Out[66]: 4

In [67]: max(Ratings)
Out[67]: 43

In [68]: min(Ratings)
Out[68]: 2

In [69]: sum(Ratings)
Out[69]: 72

In [73]: sorted_Ratings= sorted(Ratings)
         print (sorted_Ratings)
         [2, 6, 21, 43]
```

For Example: NestedTuple = (5,2, ("A","B"),(1,2),(8896,("x","y","z")))

We can access these elements using the standard indexing methods.

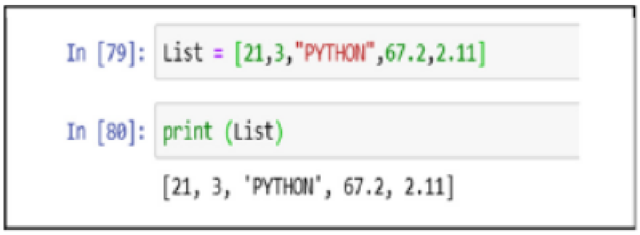


```
In [74]: NestedTuple=(5,2,("A","B"),(1,2),(8896,("x","y","z")))

In [75]: NestedTuple[1]
Out[75]: 2

In [76]: NestedTuple[2]
Out[76]: ('A', 'B')

In [77]: NestedTuple[4]
Out[77]: (8896, ('x', 'y', 'z'))

In [78]: NestedTuple[4][1]
Out[78]: ('x', 'y', 'z')
```

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 [4][1].

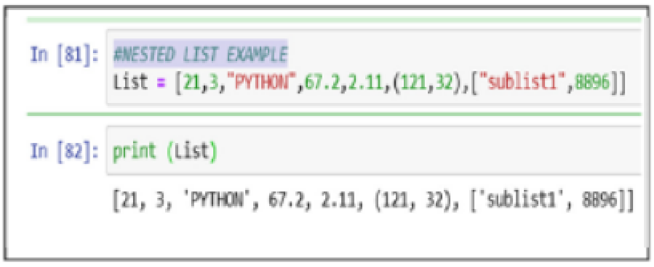
List:

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.

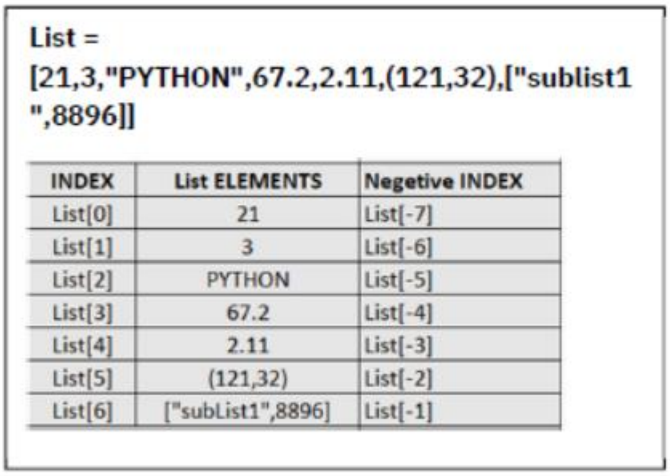


```
In [79]: List = [21,3,"PYTHON",67.2,2.11]

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]]
```



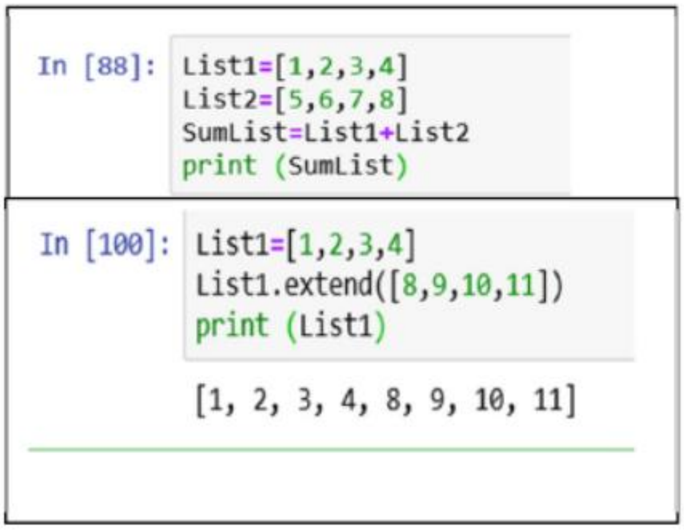
List = [21,3,"PYTHON",67.2,2.11,(121,32),["sublist1",8896]]

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.



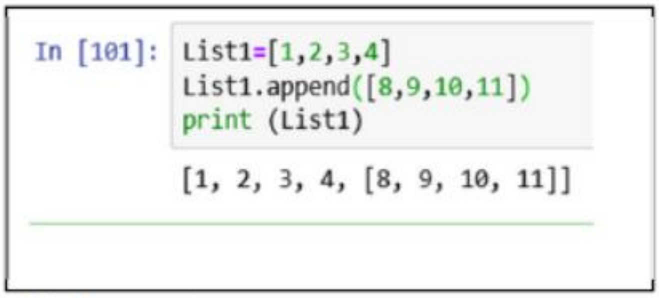
```
In [88]: List1=[1,2,3,4]
         List2=[5,6,7,8]
         SumList=List1+List2
         print (SumList)
```

```
In [100]: List1=[1,2,3,4]
          List1.extend([8,9,10,11])
          print (List1)

          [1, 2, 3, 4, 8, 9, 10, 11]
```

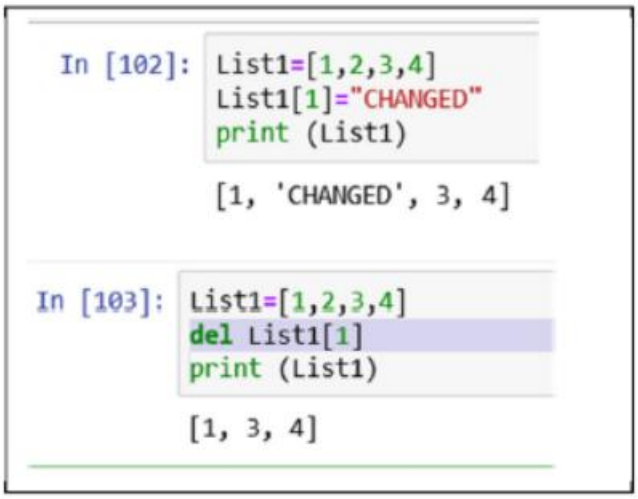
The argument inside the parenthesis is a new list that we are going to concatenate to the original list. In this case, instead of creating a new list, the original list List1 is modified by adding four new elements.

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.



```
In [101]: List1=[1,2,3,4]
          List1.append([8,9,10,11])
          print (List1)

          [1, 2, 3, 4, [8, 9, 10, 11]]
```



```
In [102]: List1=[1,2,3,4]
          List1[1]="CHANGED"
          print (List1)

          [1, 'CHANGED', 3, 4]

In [103]: List1=[1,2,3,4]
          del List1[1]
          print (List1)

          [1, 3, 4]
```

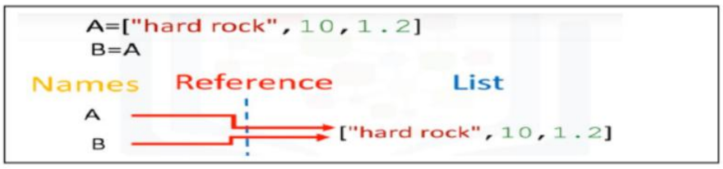
As lists are mutable, we can change them. For example, we can change the Second element as follows. The list now becomes [1,” CHANGED”,3,4]

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.

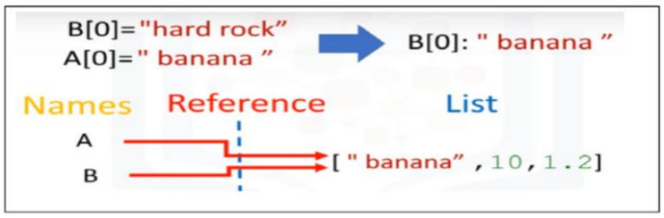
For example, if we would like to remove the Second element, then perform del List [1] command This operation removes the second element of the list then the result becomes [1,3,4]

LISTS: Aliasing

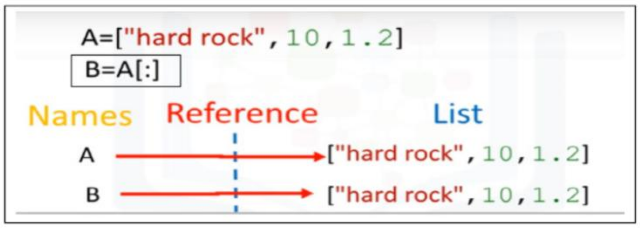
When we set one variable, B equal to A, both A and B are referencing the same list. Multiple names referring to the same object is known as aliasing.



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.

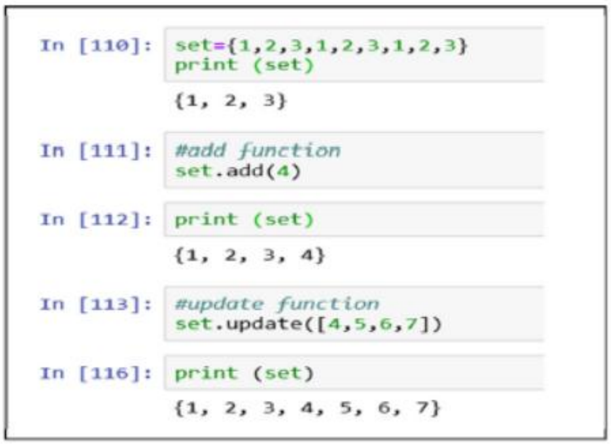
You notice there are duplicate items. When the actual set is created, duplicate items will not be present.

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

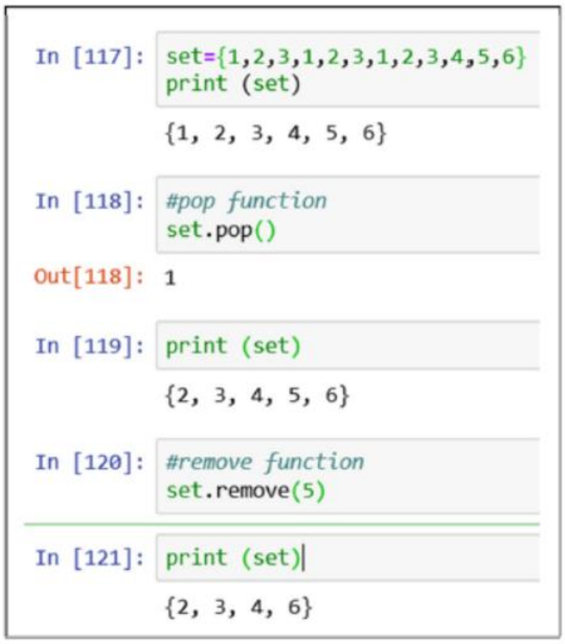
To add more than one item to a set use the update () method with list of values.

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 [110]: set={1,2,3,1,2,3,1,2,3}
          print (set)
          {1, 2, 3}

In [111]: #add function
          set.add(4)

In [112]: print (set)
          {1, 2, 3, 4}

In [113]: #update function
          set.update([4,5,6,7])

In [116]: print (set)
          {1, 2, 3, 4, 5, 6, 7}
```

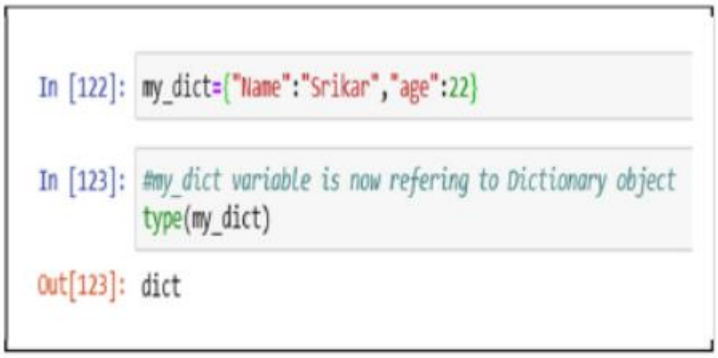


```
In [117]: set={1,2,3,1,2,3,1,2,3,4,5,6}
          print (set)

          {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.

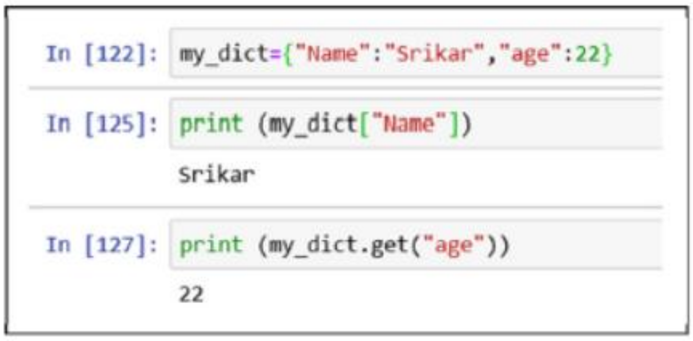
DICTIONARIES:

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
```

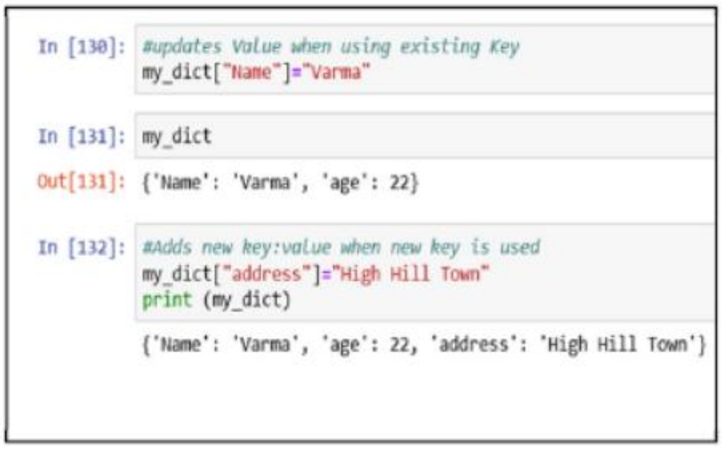
We can access the elements from the dictionary using keys.



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

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

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

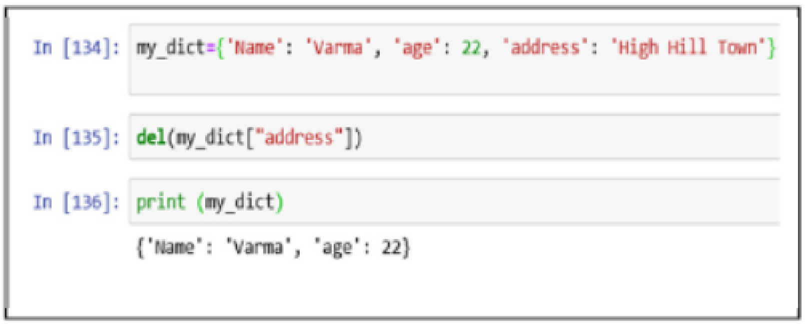
We can get the value using keys either inside square brackets or with get( ) method.

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.

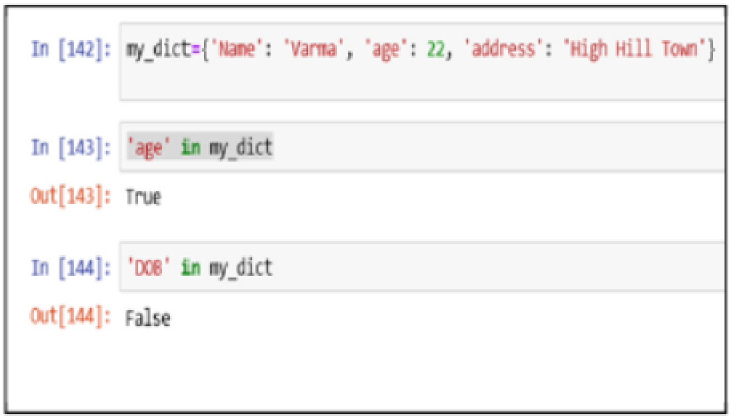


```
In [134]: my_dict={'Name': 'Varma', 'age': 22, 'address': 'High Hill Town'}

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.

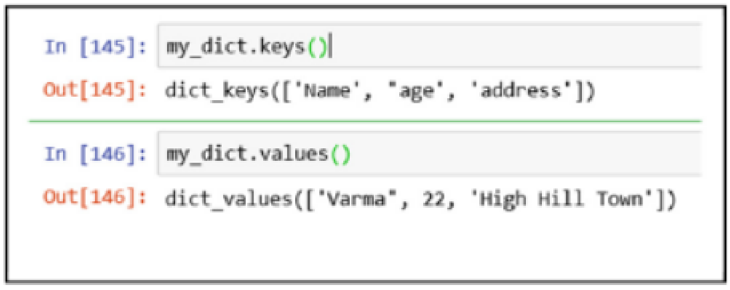
Syntax: ‘KEY_NAME’ in DictionaryName

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

In [143]: 'age' in my_dict
Out[143]: True

In [144]: 'DOB' in my_dict
Out[144]: 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.



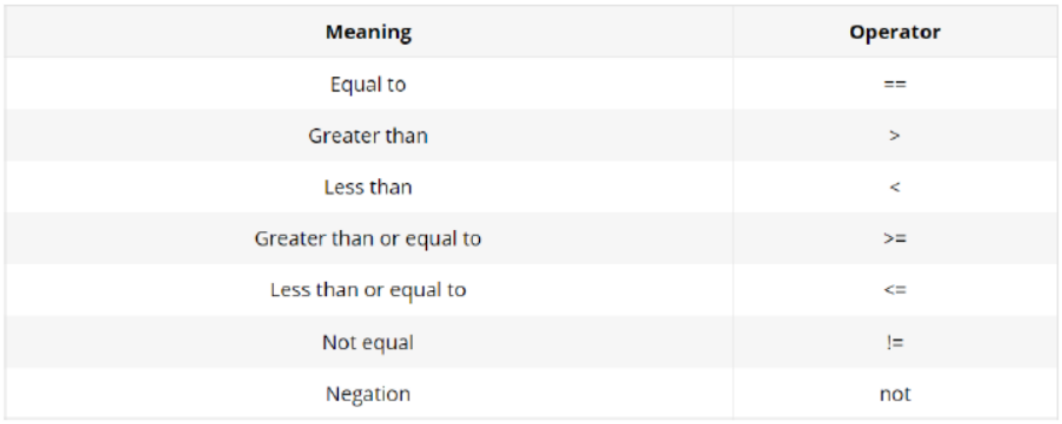
```
In [145]: my_dict.keys()
Out[145]: dict_keys(['Name', 'age', 'address'])

In [146]: my_dict.values()
Out[146]: dict_values(['Varma', 22, 'High Hill Town'])
```

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

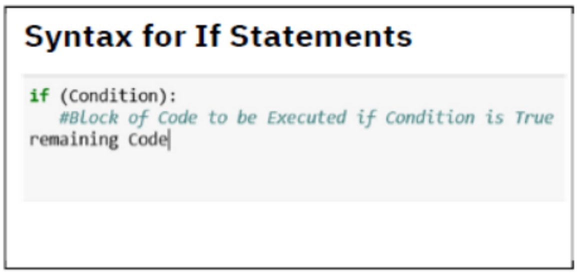
In Python we achieve the decision-making statements by using below statements -

- 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.

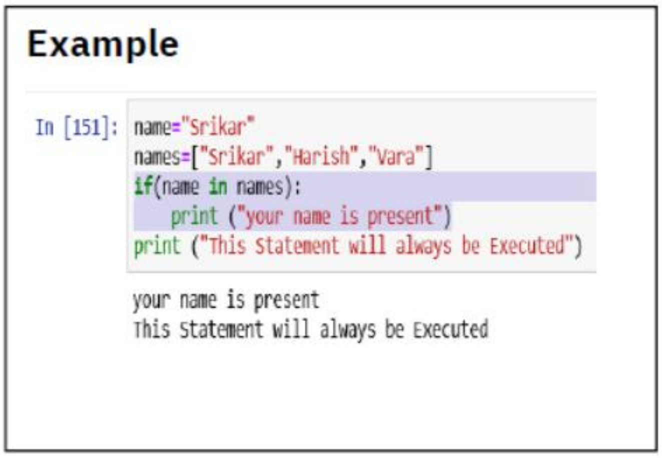
The If condition evaluates a Boolean expression and executes the block of code only when the Boolean expression becomes TRUE. Check the Syntax first the controller will come to an if condition and evaluate the condition if it is true, then the statements will be executed, otherwise the code present outside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



Syntax for If Statements

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

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
```

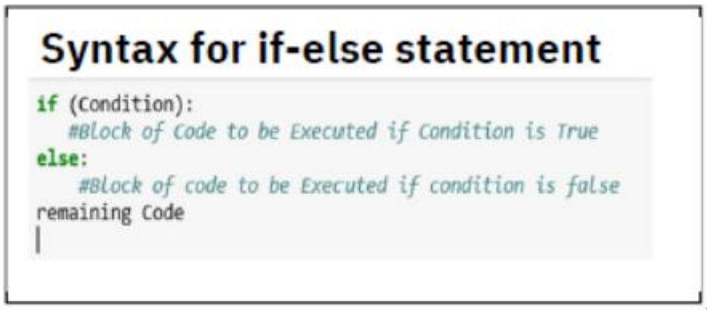
We can use if statement to check whether the name is present in the names list or not, if condition is true then it will also print block of statements inside the 'if' block. If condition is false, then it will skip the execution of the 'if' block statements.

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.

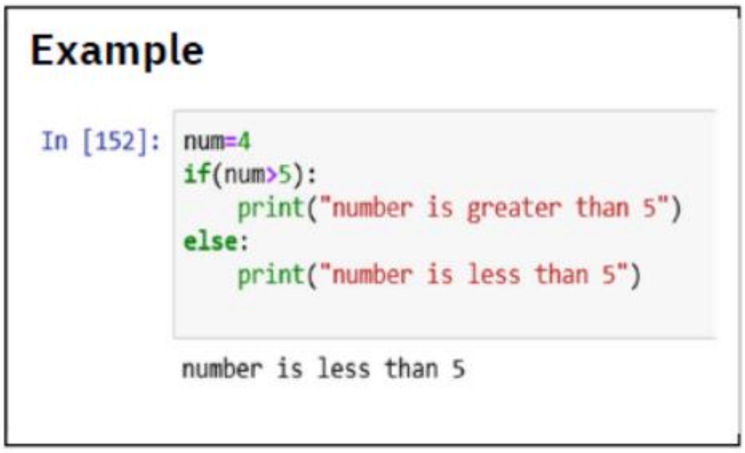
If-else statement evaluates the Boolean expression and executes the block of code present inside the if block if the condition becomes TRUE and executes a block of code present in the else block if the condition becomes FALSE.



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
```

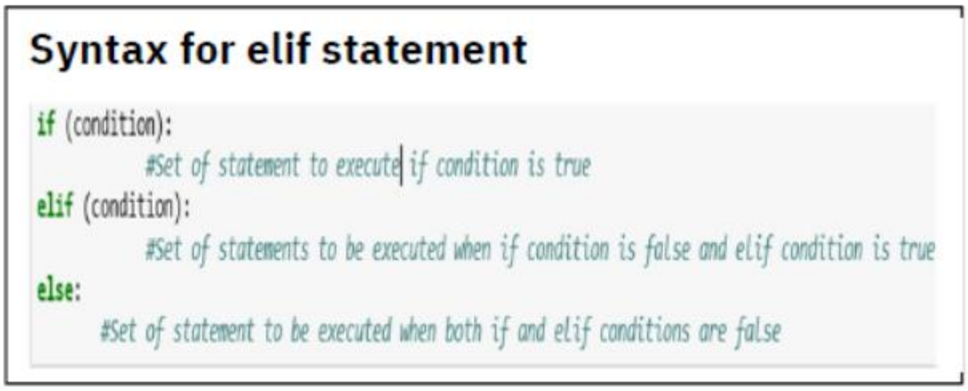
Let's take an example to implement the if-else statement, in this example the if block will get executed if the given condition is true or else it will execute the else block.


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
```

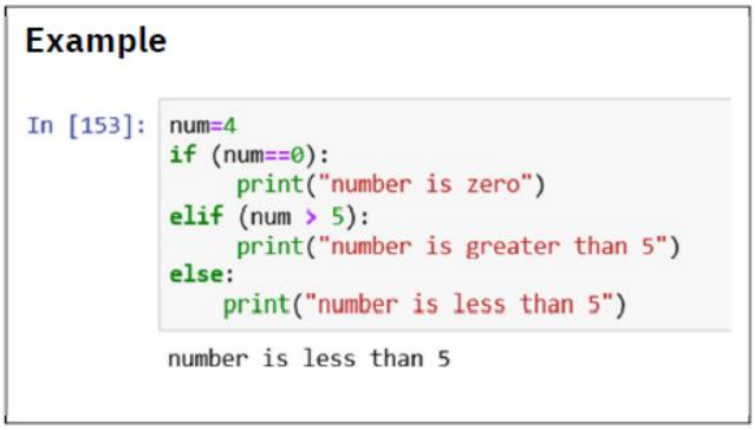
elif statements:

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.

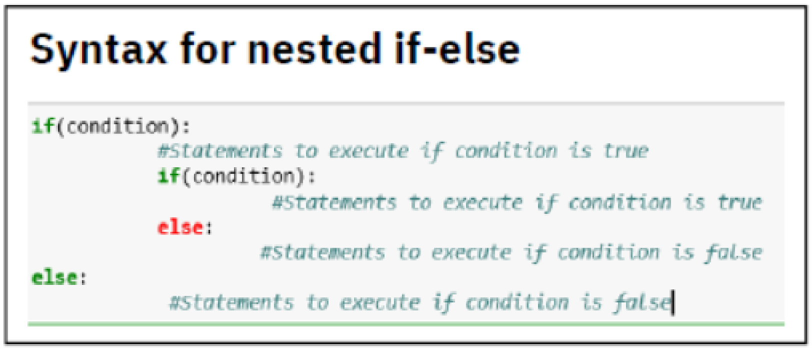

Example

```
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.

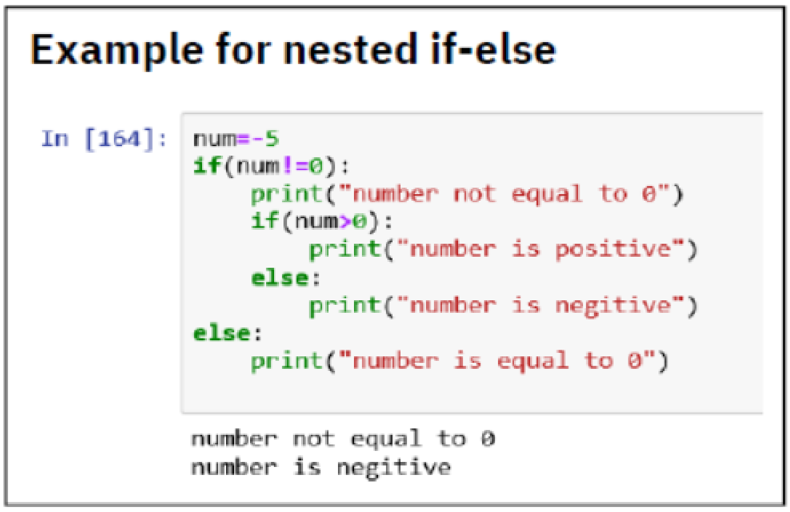
Nested if-else statements

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

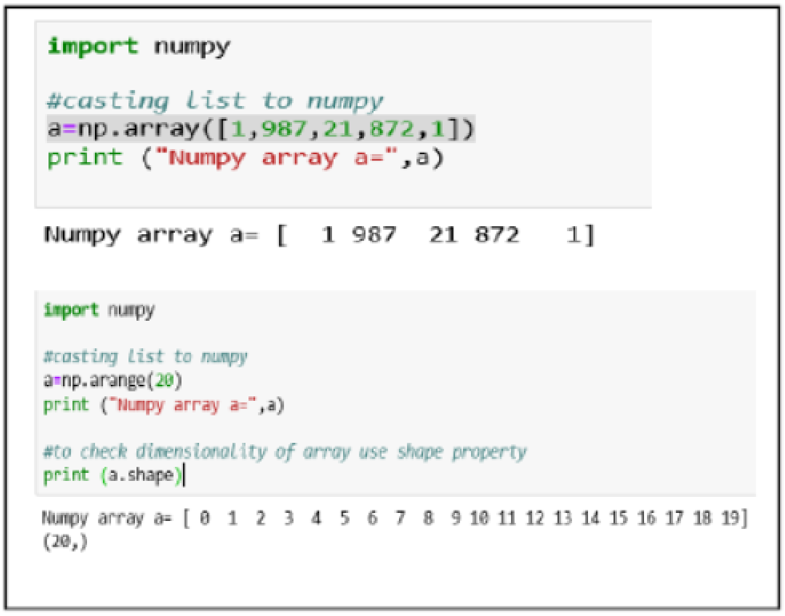
- There are several important differences between NumPy arrays and the standard Python sequences, NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size 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.



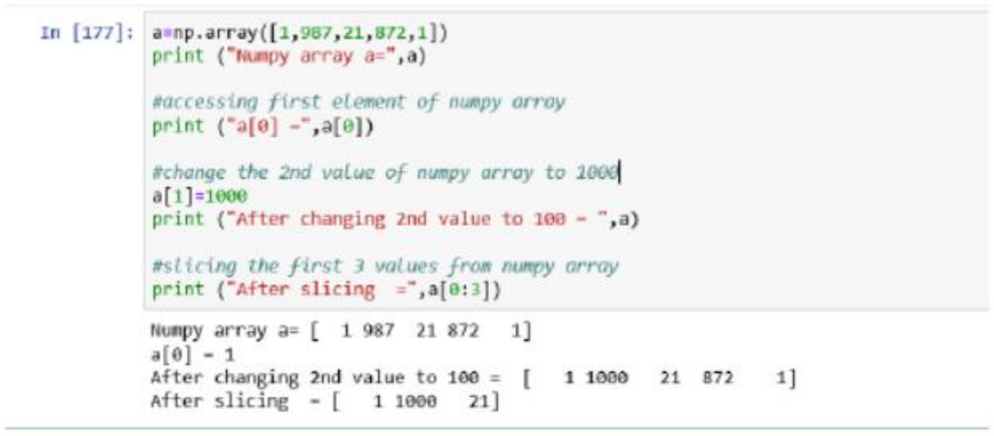
```
import numpy

#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



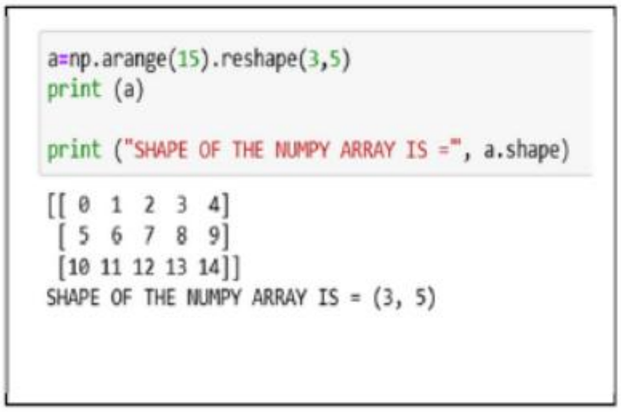
```
In [177]: a=np.array([1,987,21,872,1])
          print ("Numpy array a=",a)

          #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.

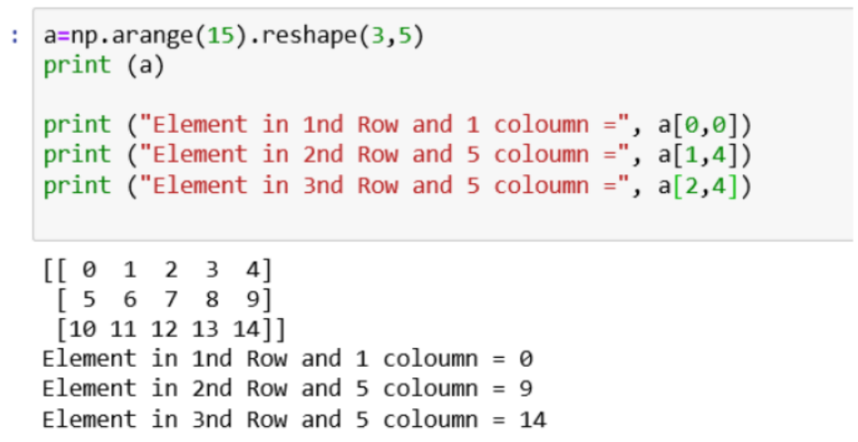
In this example first, it will create the 15 integers and then it will convert to two dimensional array with 3 rows and 5 columns.



```
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 ?

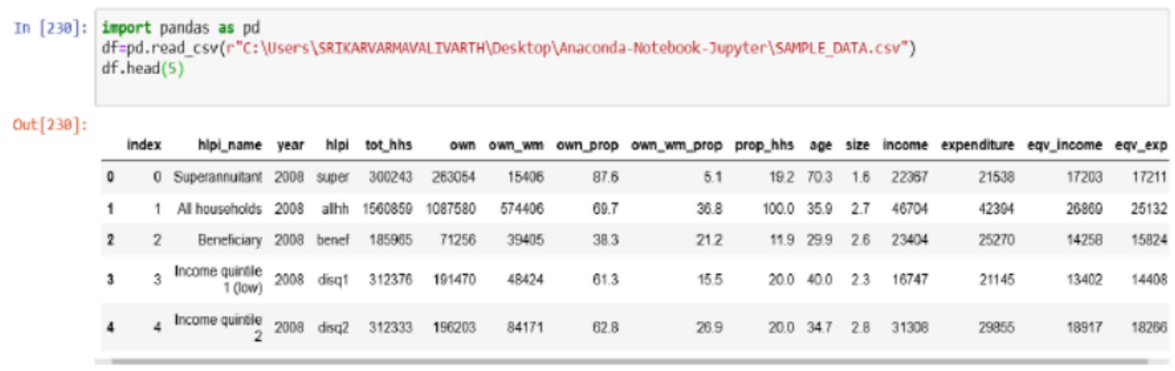
csv stands for comma-separated values, csv file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

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	hlpi_name	year	hlpi	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:

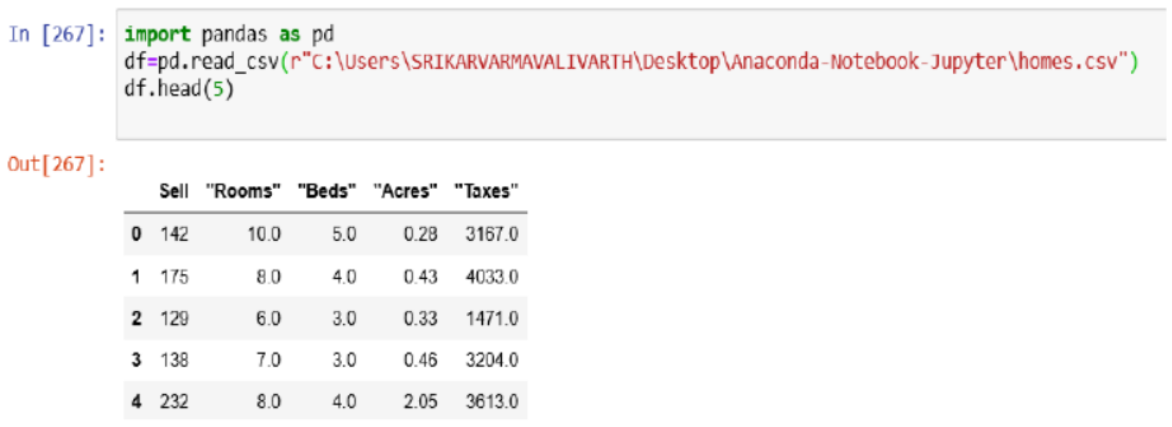
To do any statistical analysis, first collection of data is the important task

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

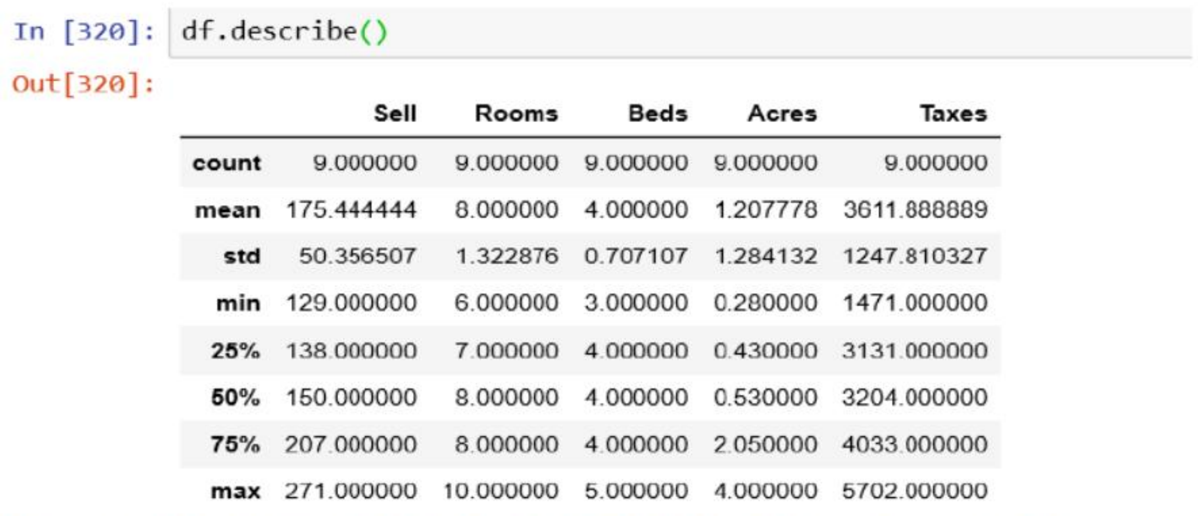
Once you have your Data Frame ready, you'll be able to get the Descriptive Statistics. We can calculate the following statistics using the pandas package:

- Mean
- Total sum
- Maximum
- Minimum
- Count
- Median

- Standard deviation
- Variance

With describe function you will get complete descriptive stats

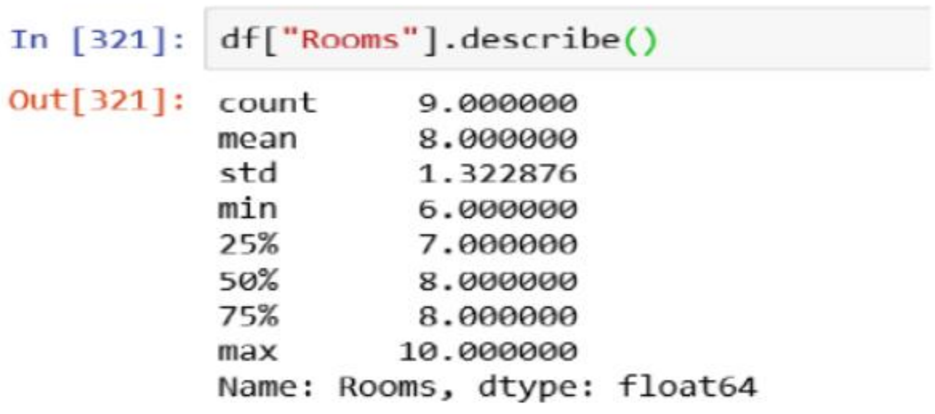
The syntax is: df.describe()



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

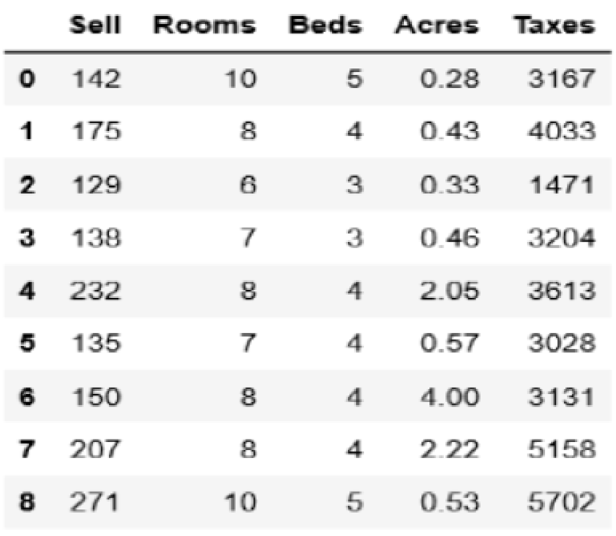
	Sell	Rooms	Beds	Acres	Taxes
count	9.000000	9.000000	9.000000	9.000000	9.000000
mean	175.444444	8.000000	4.000000	1.207778	3611.888889
std	50.356507	1.322876	0.707107	1.284132	1247.810327
min	129.000000	6.000000	3.000000	0.280000	1471.000000
25%	138.000000	7.000000	4.000000	0.430000	3131.000000
50%	150.000000	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:

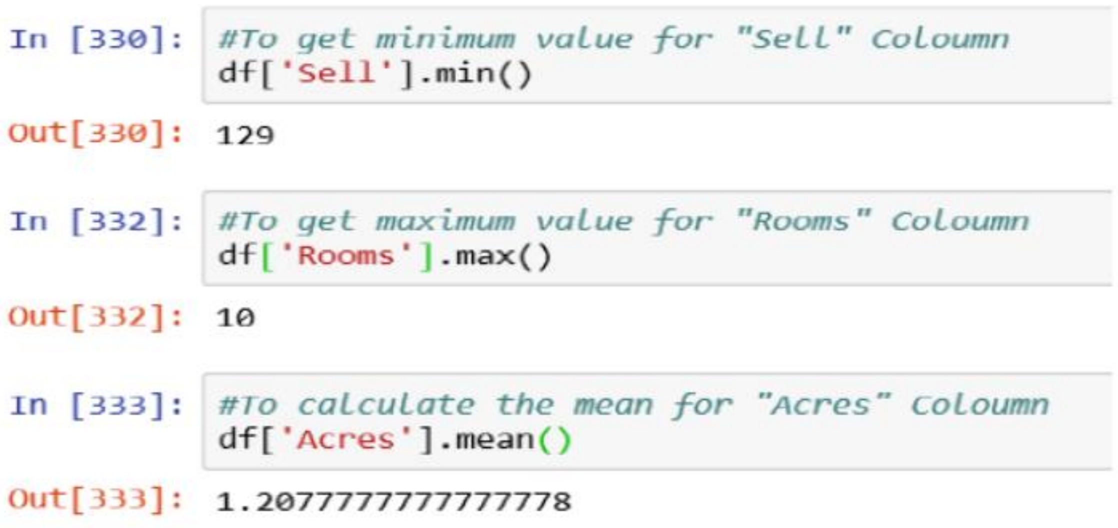


```
In [321]: df["Rooms"].describe()
Out[321]: count     9.000000
          mean      8.000000
          std       1.322876
          min       6.000000
          25%       7.000000
          50%       8.000000
          75%       8.000000
          max      10.000000
          Name: Rooms, dtype: float64
```

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.

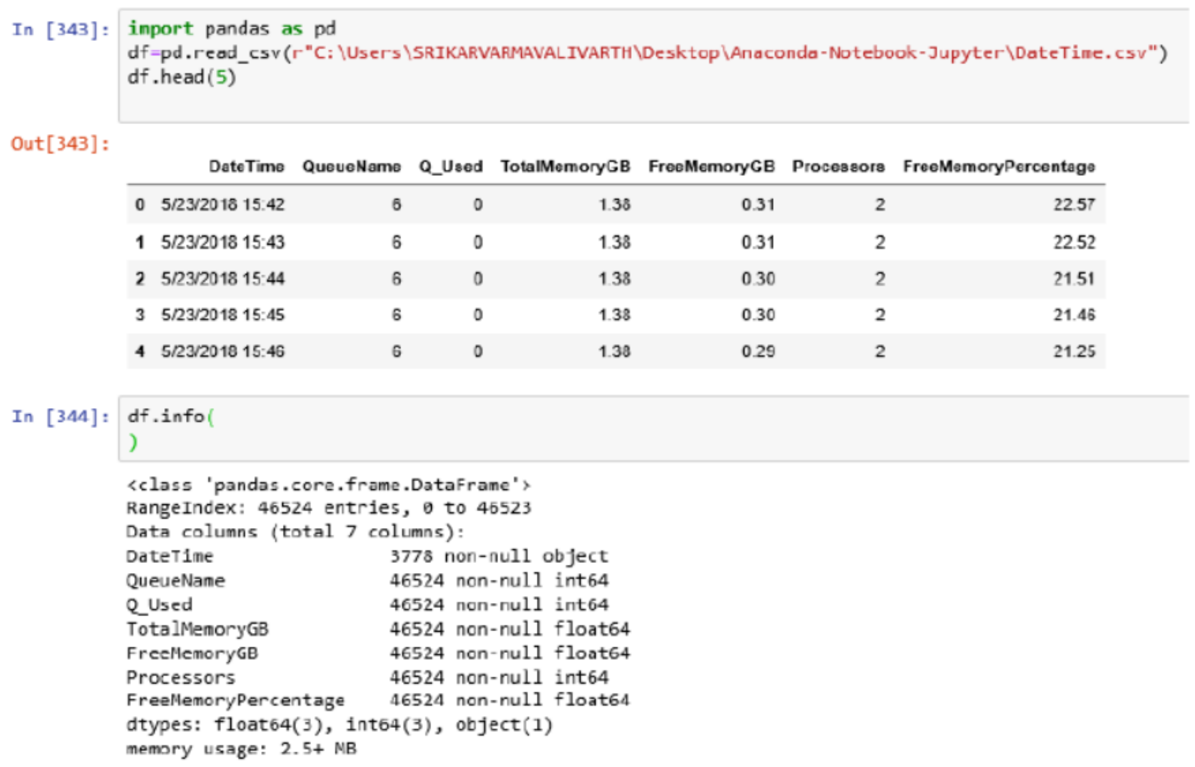
Let’s see how we can convert a dataframe column of strings (in dd/mm/yyyy format) to datetime format. We cannot perform any time series-based operation on the dates if they are not in the right format. To be able to work with it, we are required to convert the dates into the datetime format.

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.



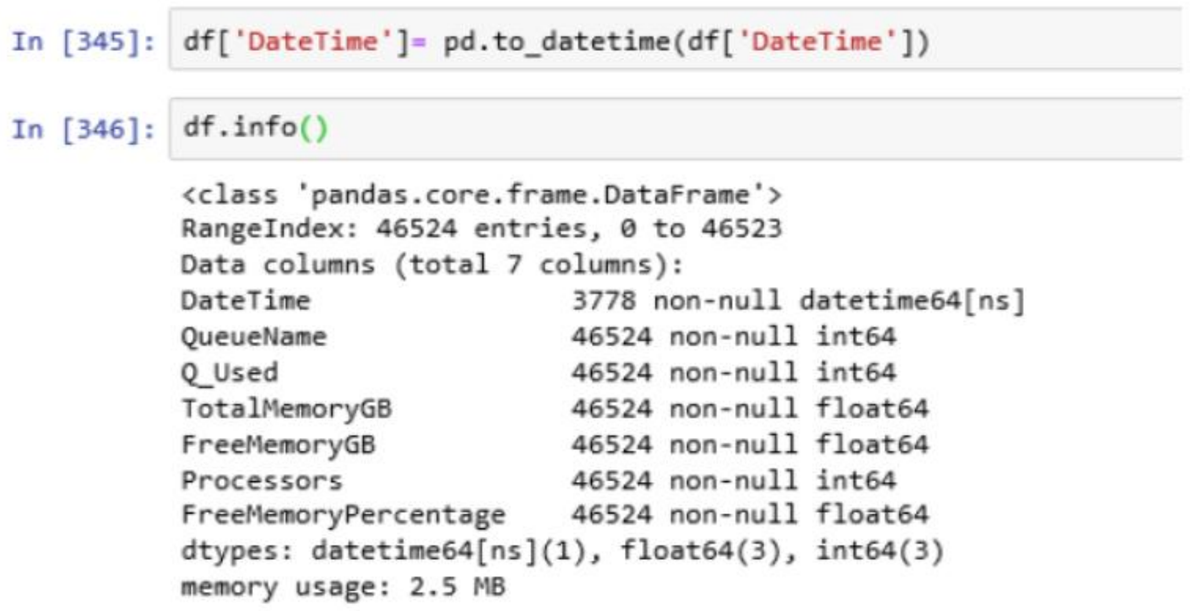
```
In [343]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\DateTime.csv")
          df.head(5)
```

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.

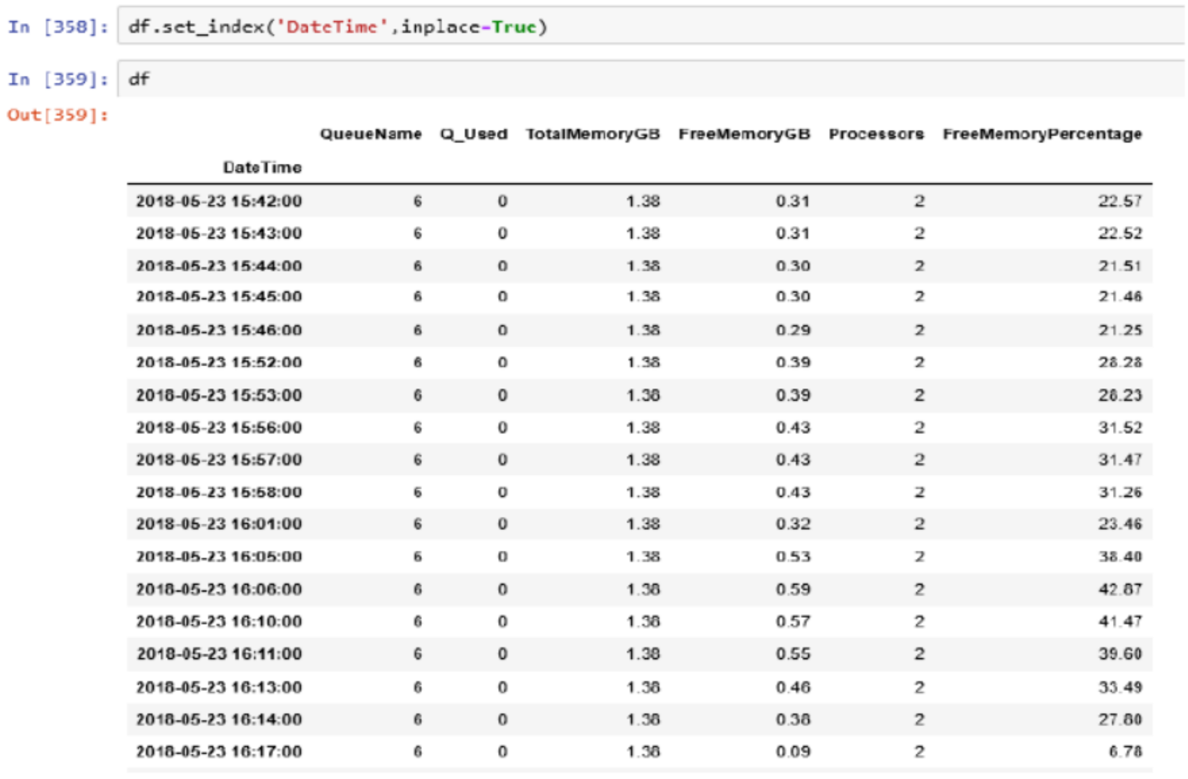


```
In [345]: df['DateTime']= pd.to_datetime(df['DateTime'])

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 require 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.



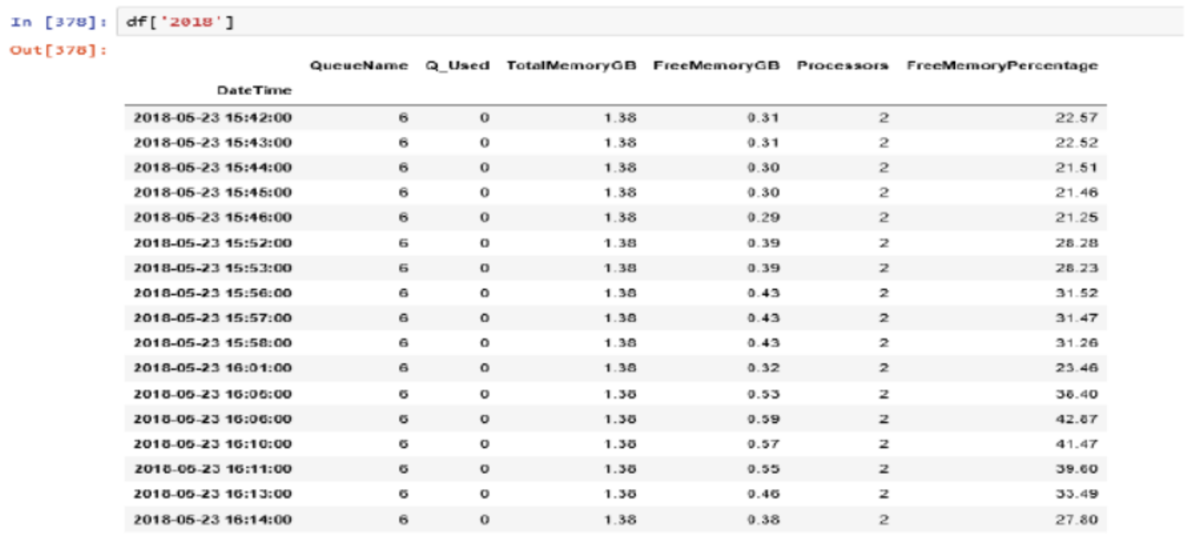
```
In [358]: df.set_index('DateTime',inplace-True)

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

	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
DateTime						
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	42.87
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
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.

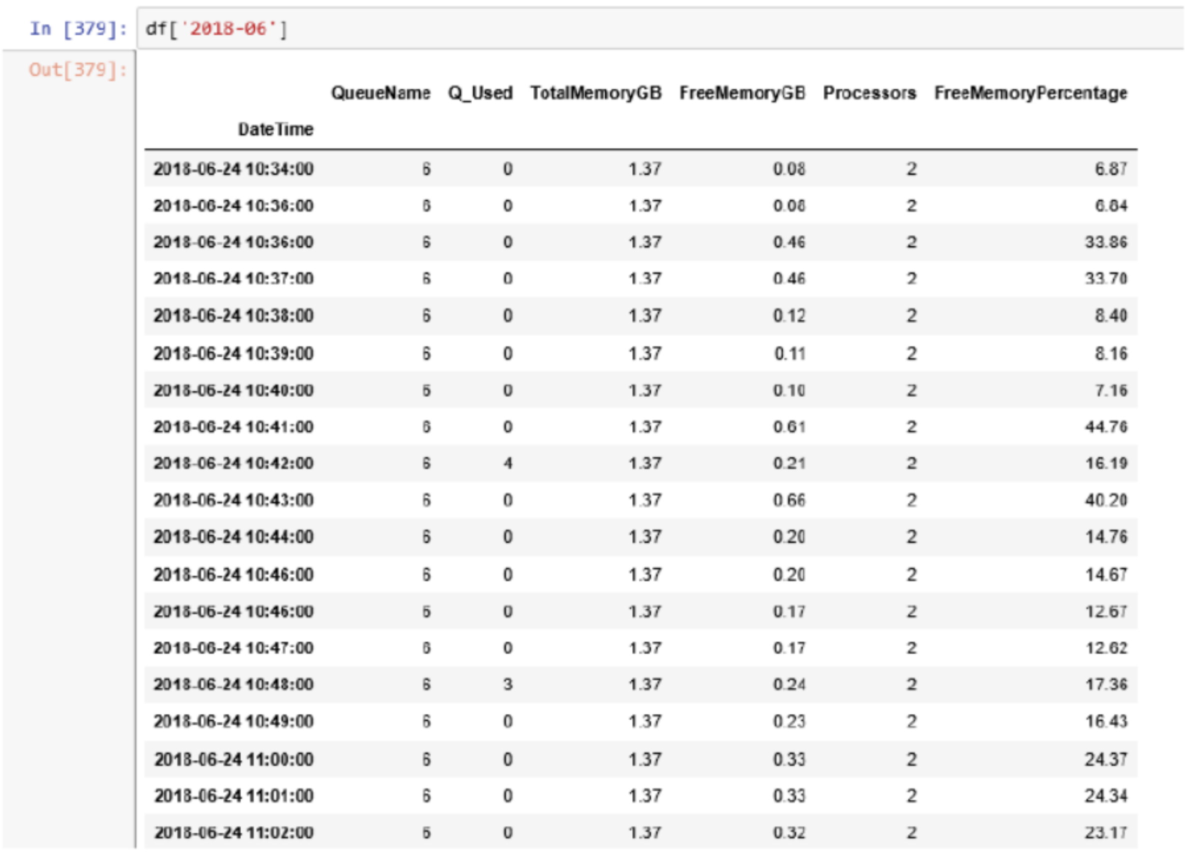
To Check all the values occurred in a particular year let's say 2018 Run command df['2018'].



```
In [378]: df['2018']
Out[378]:
                     QueueName  Q_Used  TotalMemoryGB  FreeMemoryGB  Processors  FreeMemoryPercentage
           DateTime
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                 42.87
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
```

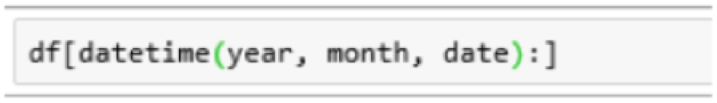
Above result gives all the values recorded in year 2018.

How to filter data based on year and month To View all observations that occurred in June 2018, run the below command



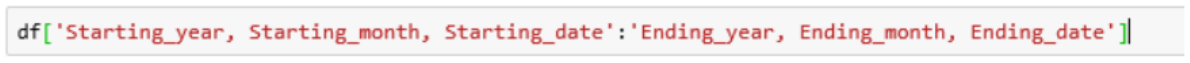
```
In [379]: df['2018-06']
Out[379]:
                     QueueName  Q_Used  TotalMemoryGB  FreeMemoryGB  Processors  FreeMemoryPercentage
           DateTime
2018-06-24 10:34:00          6       0           1.37          0.08           2                  6.87
2018-06-24 10:35:00          6       0           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 10:45:00          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

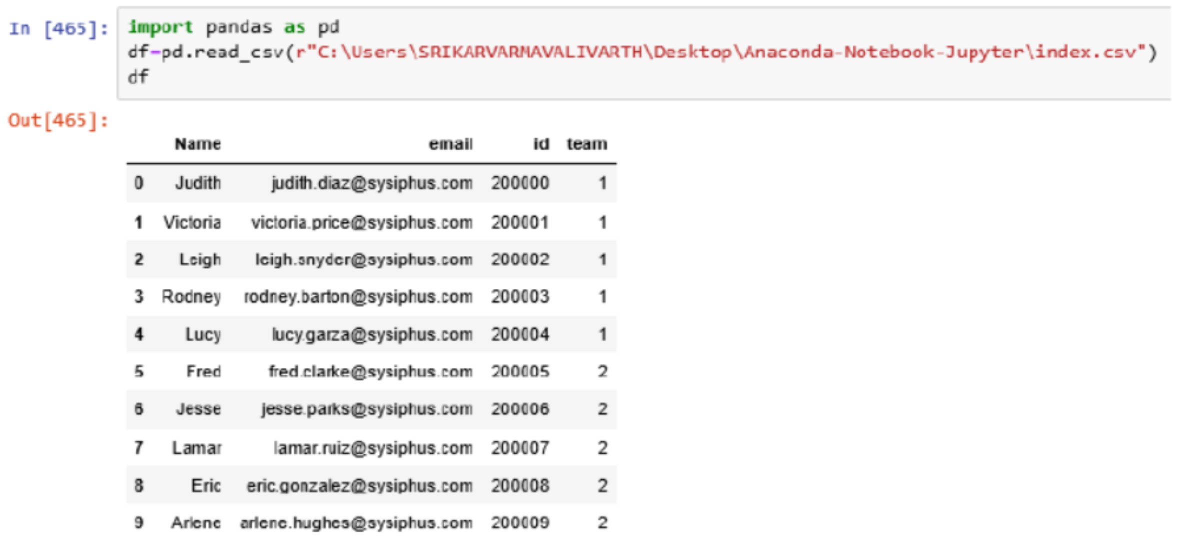
then command is: df['5/3/2018':'5/4/2018']

Pandas Indexing and Selecting Data

What is Indexing ?

Indexing in pandas means simply selecting particular rows and columns of data from a Data-Frame. 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.

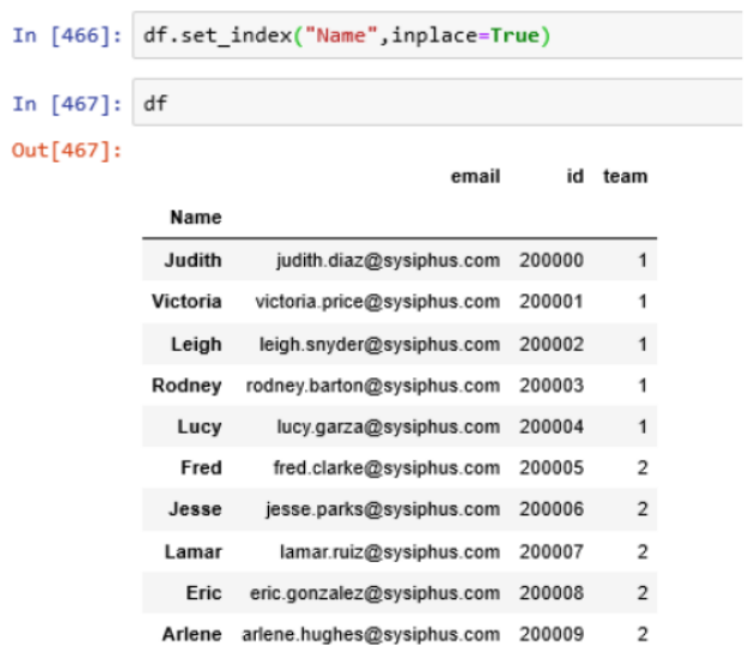


```
In [465]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\index.csv")
          df
Out[465]:
```

	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

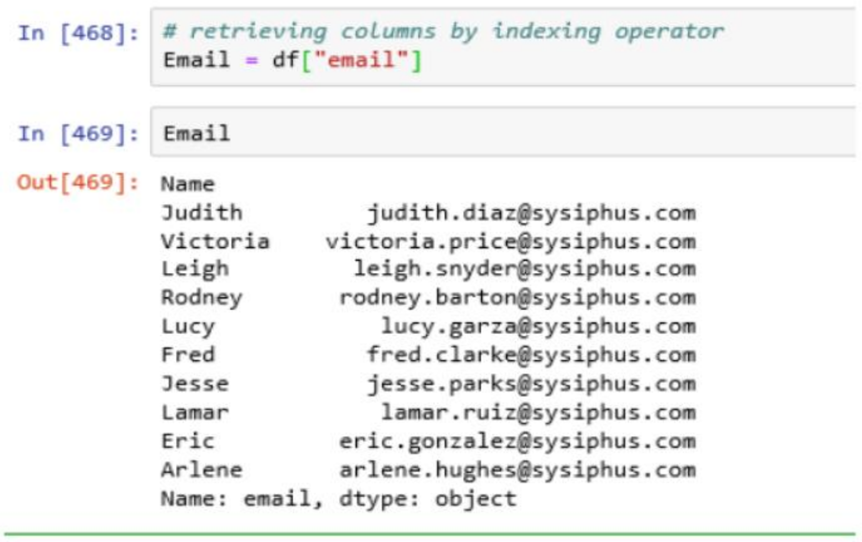


```
In [466]: df.set_index("Name",inplace=True)

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

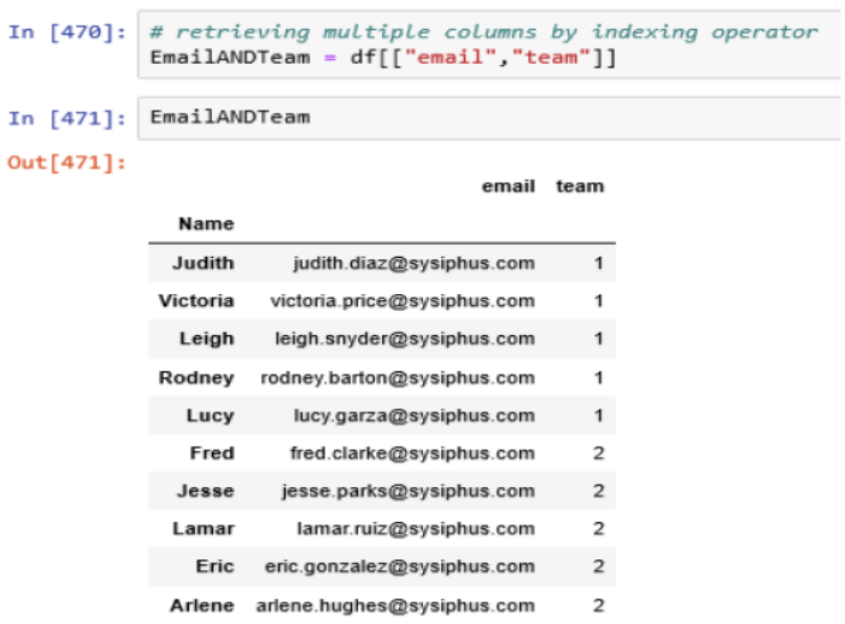
In order to take single column, we simply put the name of the column in-between the brackets.



```
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
```

Selecting Multiple Columns

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

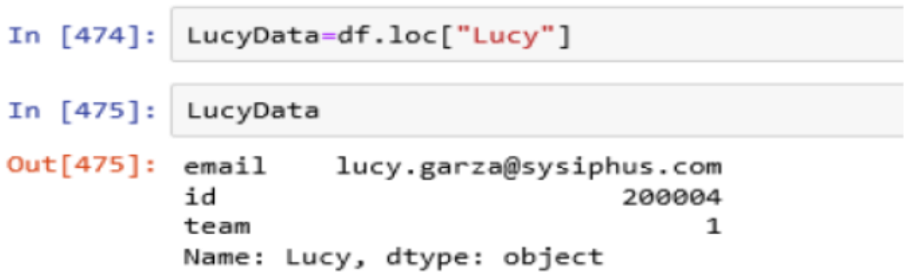


```
In [470]: # retrieving multiple columns by indexing operator
          EmailANDTeam = df[["email","team"]]

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:

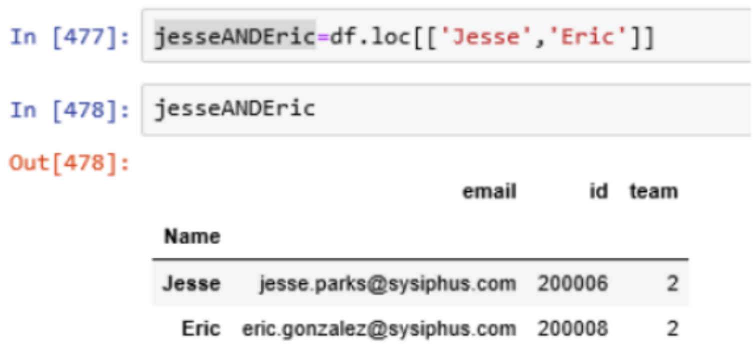
In order to select a single row using. loc[], we put a single row label in a .loc function.



```
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.

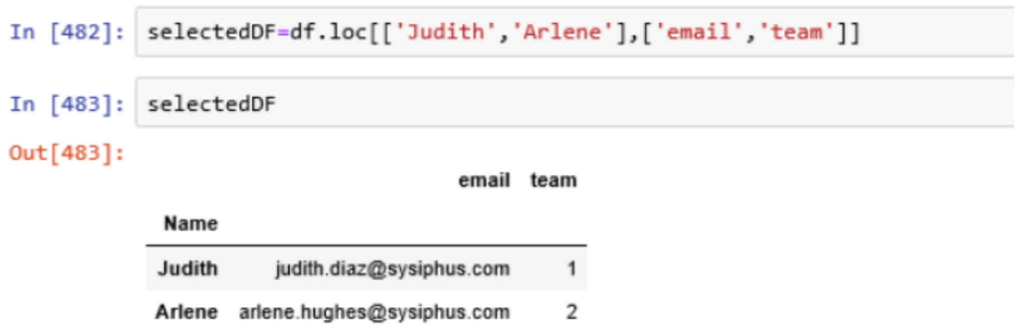


```
In [477]: jesseANDEric=df.loc[['Jesse','Eric']]

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:

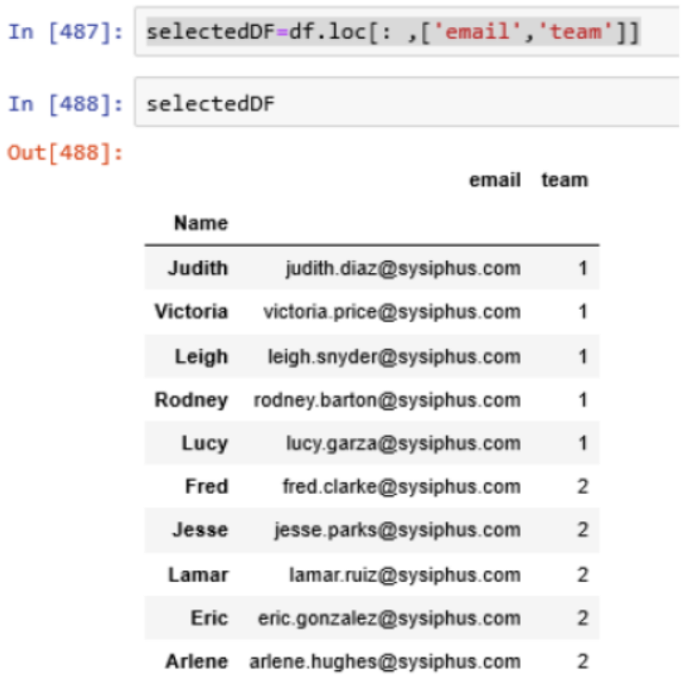
In order to select two rows and two columns, we select two rows which we want to select and two columns and put it in a separate list: **Dataframe.loc[["row1", "row2"], ["column1", "column2"]]**



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

In [483]: selectedDF

Out[483]:
                                  email   team
          Name
          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"]]

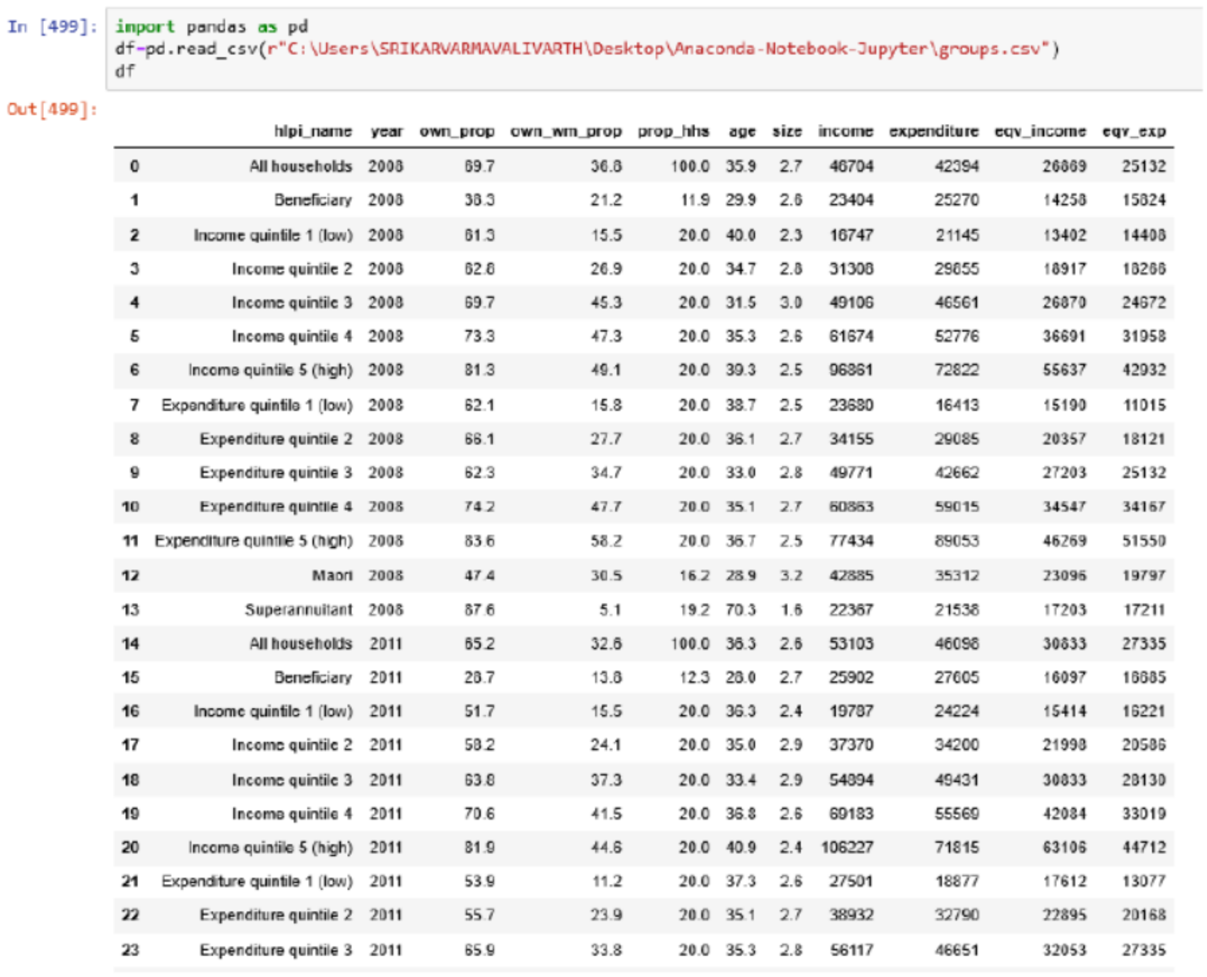


```
In [487]: selectedDF=df.loc[: ,['email','team']]

In [488]: selectedDF

Out[488]:
                                    email   team
           Name
           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.

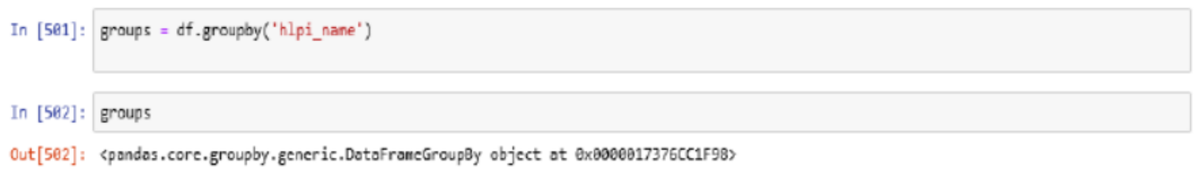


```
In [499]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\groups.csv")
          df
Out[499]:
```

	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	62.8	26.9	20.0	34.7	2.8	31308	29855	18917	18266
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	21538	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

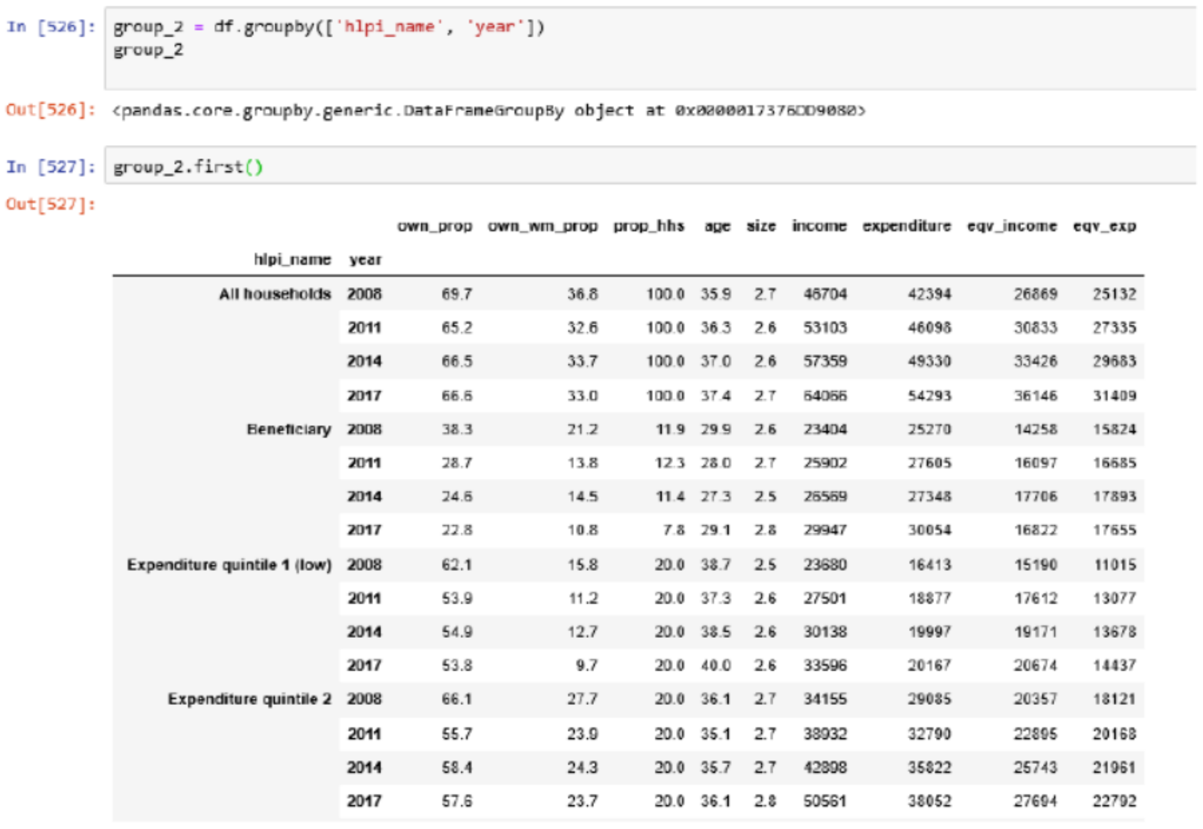
groupby function based on more than one category

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.

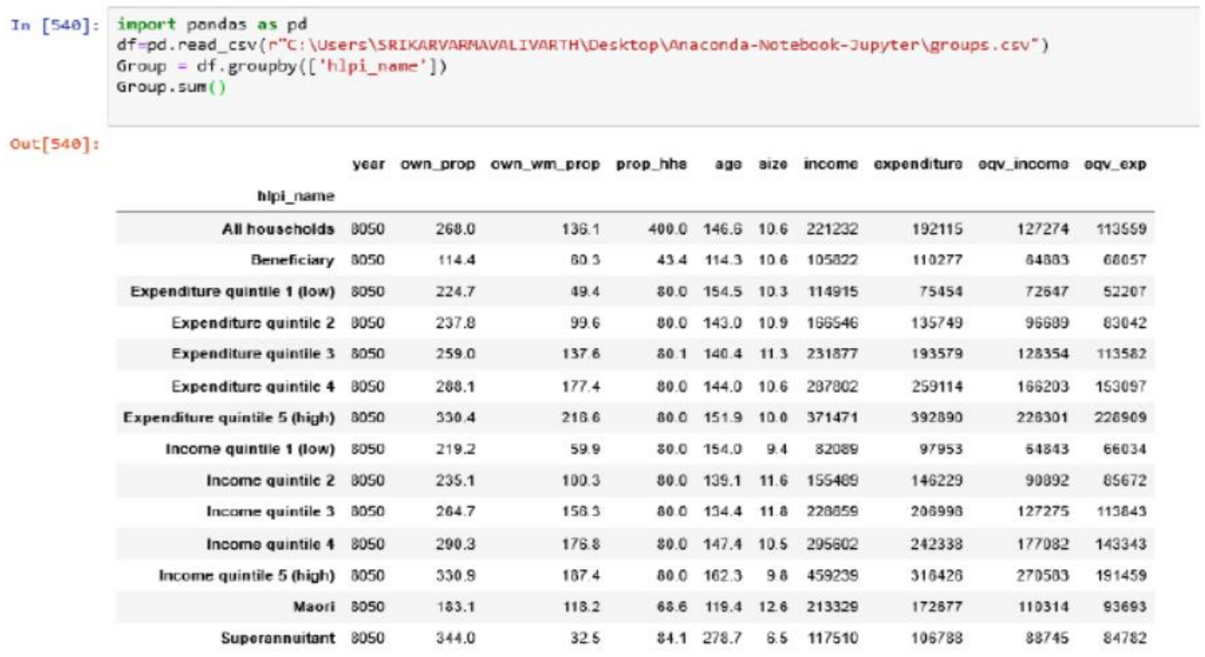


```
In [526]: group_2 = df.groupby(['hlpi_name', 'year'])
          group_2

Out[526]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000173760D9080>

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	46704	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	23.9	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	60.3	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	228909
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	318428	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.

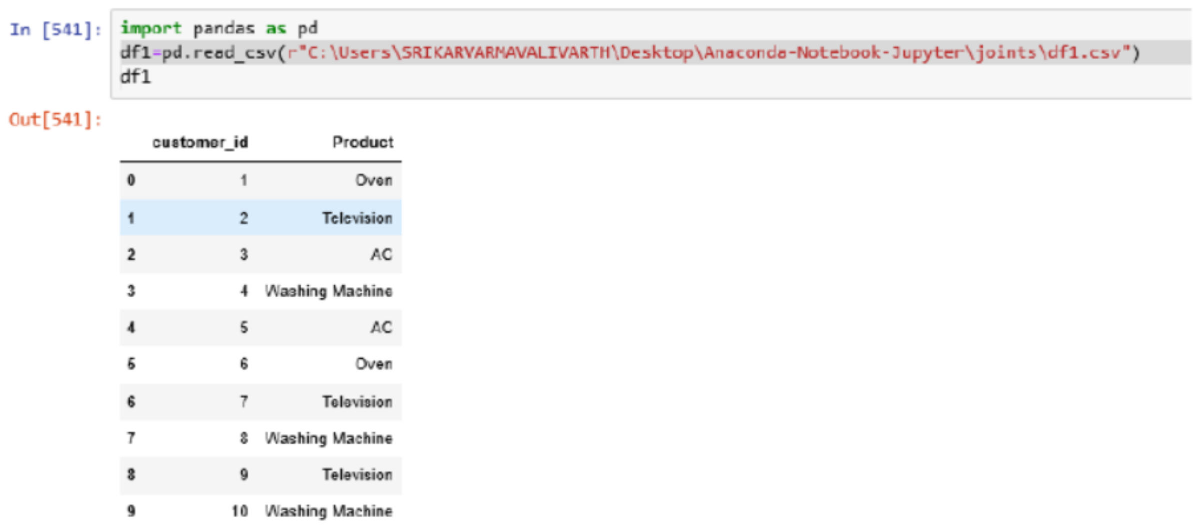
Merge/Join Datasets

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.

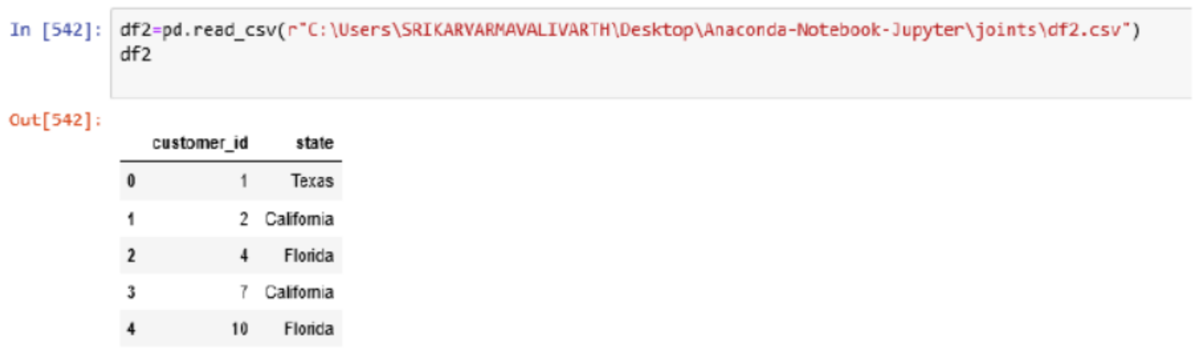
We can merge two data frames in pandas python by using the merge() function. The different arguments to merge() allow you to perform natural join, left join, right join, and full outer join in pandas.

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
```



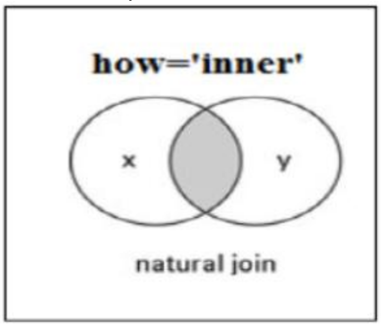
```
In [542]: df2=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\joints\df2.csv")
          df2

Out[542]:
     customer_id       state
  0            1       Texas
  1            2  California
  2            4     Florida
  3            7  California
  4           10     Florida
```

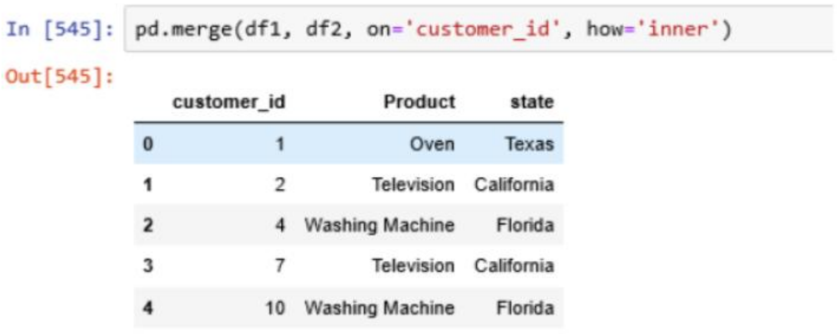
Natural join

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



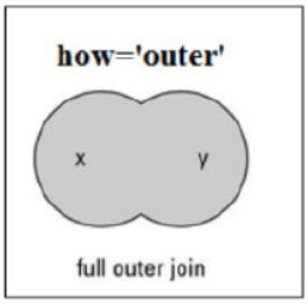


```
In [545]: pd.merge(df1, df2, on='customer_id', how='inner')
Out[545]:
```

	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

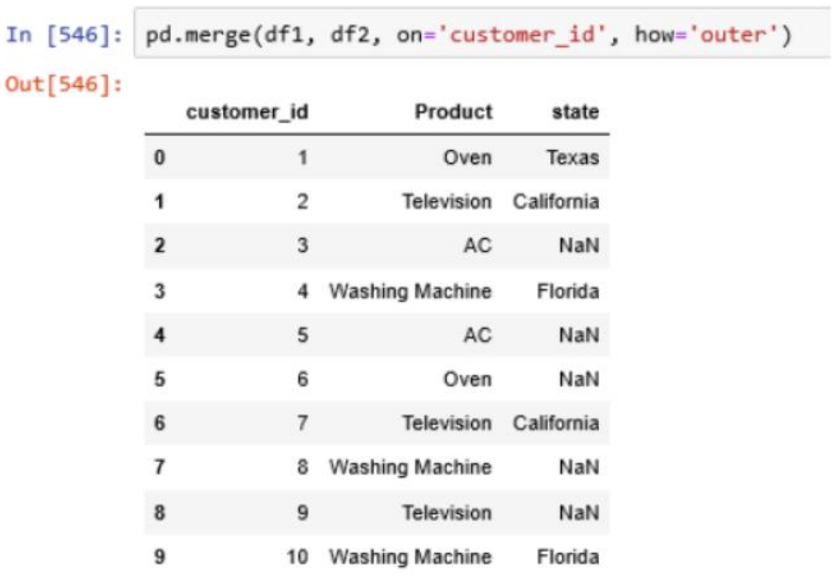
Full outer join

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.

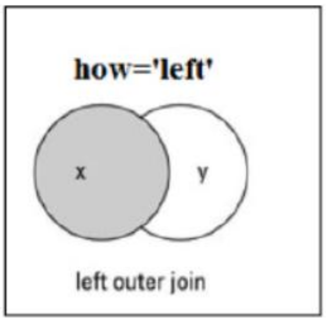


```
In [546]: pd.merge(df1, df2, on='customer_id', how='outer')
Out[546]:
```

	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

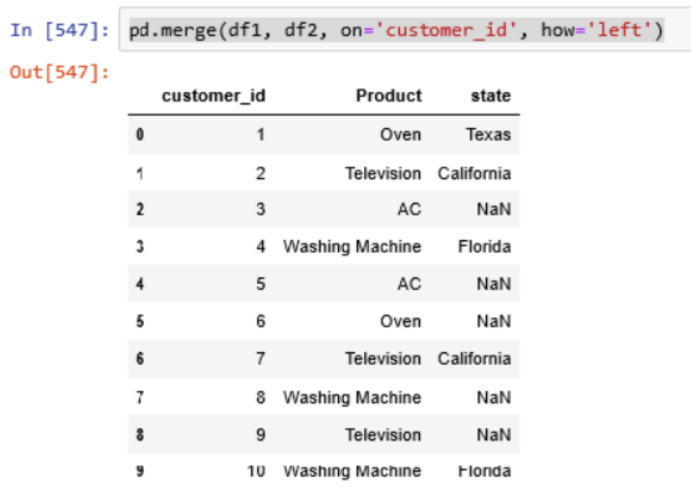
Left outer join

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



Syntax:

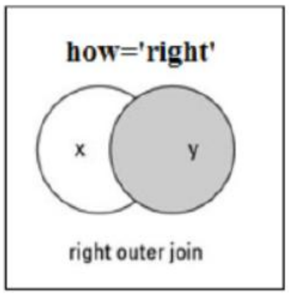
pd.merge(df1, df2, on=column', how=left)
Return all rows from the left table, and any rows with matching keys from the right table.



```
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
```

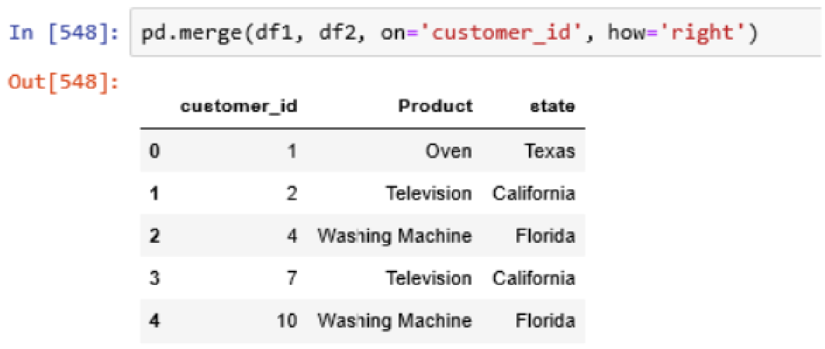
Right outer join

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

Data Visualization Tools inPython- Introduction to Matplotlib, Basic plots using matplotlib, Specialized Visualization Tools usingMatplotlib, Advanced Visualization Tools usingMatplotlib- WaffleCharts, WordClouds.

Introduction to Data Visualization Tools in Python

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.

As for the scripting layer, it was developed for scientists who are not professional programmers and I'm sure you agree with me based on the histogram that we just created that 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. For more details, please refer:

http://aosabook.org/en/matplotlib.html

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.

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/empirical 2/migrationflows.asp

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

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.

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

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

Remove the name of the index.

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

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.

We can also pass multiple criteria in the same line.

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

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

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).

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

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.

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

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

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

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.

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.

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.

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:

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.

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:

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.

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.

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.

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.

Plot the regression line on the scatter plot.

'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 data points 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 neighbour 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.

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.

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()
```

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.

The first step into creating a waffle chart is determing the proportion of each category with respect to the total.

The second step is defining the overall size of the waffle chart.

The third step is using the proportion of each category to determe it respective number of tiles

The fourth step is creating a matrix that resembles the waffle chart and populating it.

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!')
```

Map the waffle chart matrix into a visual.

Prettify the chart.

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.

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.

Much better! However, said isn't really an informative word. So, let's add it to our stop words and re-generate the cloud.

UNIT – V

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

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.
We then use the "displot" function to plot the distribution of a single variable

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.

Joint plot allows us to plot the relationship represented by bivariate data.

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.

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.

The difference between countplot and barplot is that countplot explicitly counts the number of occurrences.

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.

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.

The stripplot draws a scatterplot where one variable is categorical.

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.

The corr() function gives the matrix form to correlation data.

Below command generates the heatmap.

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.

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.

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.

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.

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.

Let's create the map again with a higher zoom level

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.

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.

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.

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

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:

Let us now visualize the rides data using a class method called Heatmap()

Code for reference:

```
from folium.plugins import HeatMap

df_copy = df[df.month>4].copy()

df_copy['count'] = 1

base_map = generateBaseMap()

Heat-
Map(data=df_copy[['pickup_latitude','pickup_longitude','count']].groupby(['pickup_latitude','pick
up_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

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

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.