Dplyr

Dplyr is a package for data manipulation. It provides some great, easy-to-use functions that are very handy when performing exploratory data analysis and manipulation. Here, I will provide a basic overview of some of the most useful functions contained in the package.

For this article, I will be using the airquality dataset from the datasets package. The airquality dataset contains information about air quality measurements in New York from May 1973 – September 1973.

The head of the dataset looks like this:

```
head(airquality)
```

```
  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
```

Before we dive into the functions, let's load up the two packages:

```
library(datasets)
library(dplyr)
```

Okay, now let's get to the functions.

Filter

The filter function will return all the rows that satisfy a following condition. For example, we can return all rows with Temp greater than 70 as follows:

```
filter(airquality, Temp > 70)
```

```
  Ozone Solar.R Wind Temp Month Day
1    36     118  8.0   72     5   2
2    12     149 12.6   74     5   3
3     7      NA  6.9   74     5  11
4    11     320 16.6   73     5  22
5    45     252 14.9   81     5  29
6   115     223  5.7   79     5  30
...
```

We can specify multiple conditions. The example below will return all rows with Temp larger than 80 and Month higher than 5.

```
filter(airquality, Temp > 80 & Month > 5)
```

```
   Ozone Solar.R Wind Temp Month Day
1     NA     186  9.2   84     6   4
2     NA     220  8.6   85     6   5
3     29     127  9.7   82     6   7
4     NA     273  6.9   87     6   8
5     71     291 13.8   90     6   9
6     39     323 11.5   87     6  10
...
```

Mutate

Mutate is used to add new variables to the data. For example, let's adds a new column that displays the temperature in Celsius.

```
mutate(airquality, TempInC = (Temp - 32) * 5 / 9)
```

```
  Ozone Solar.R Wind Temp Month Day  TempInC
1    41     190  7.4   67     5   1 19.44444
2    36     118  8.0   72     5   2 22.22222
3    12     149 12.6   74     5   3 23.33333
4    18     313 11.5   62     5   4 16.66667
5    NA      NA 14.3   56     5   5 13.33333
...
```

Summarise

The summarise function is used to summarise multiple values into a single value. It is very powerful when used in conjunction with the other functions in the `dplyr` package, as demonstrated below. `na.rm = TRUE` will remove all `NA` values while calculating the mean, so that it doesn't produce spurious results.

```
summarise(airquality, mean(Temp, na.rm = TRUE))
```

```
  mean(Temp)
1   77.88235
```

Group By

The `group_by` function is used to group data by one or more variables. For example, we can group the data together based on the Month, and then use the summarise function to calculate and display the mean temperature for each month.

```
summarise(group_by(airquality, Month), mean(Temp, na.rm = TRUE))
```

```
  Month mean(Temp)
1     5   65.54839
2     6   79.10000
3     7   83.90323
4     8   83.96774
5     9   76.90000
```

Sample

The sample function is used to select random rows from a table. The first line of code randomly selects ten rows from the dataset, and the second line of code randomly selects 15 rows (10% of the original 153 rows) from the dataset.

```
sample_n(airquality, size = 10)
sample_frac(airquality, size = 0.1)
```

Count

The count function tallies observations based on a group. It is slightly similar to the table function in the base package. For example:

```
count(airquality, Month)
```

```
  Month  n
1     5 31
2     6 30
3     7 31
4     8 31
5     9 30
```

This means that there are 31 rows with Month = 5, 30 rows with Month = 6, and so on.

Arrange

The arrange function is used to arrange rows by variables. Currently, the airquality dataset is arranged based on Month, and then Day. We can use the arrange function to arrange the rows in the descending order of Month, and then in the ascending order of Day.

```
arrange(airquality, desc(Month), Day)
```

```
  Ozone Solar.R Wind Temp Month Day
1    96     167  6.9   91     9   1
2    78     197  5.1   92     9   2
3    73     183  2.8   93     9   3
4    91     189  4.6   93     9   4
5    47      95  7.4   87     9   5
6    32      92 15.5   84     9   6
```

Pipe

The pipe operator in R, represented by `%>%` can be used to chain code together. It is very useful when you are performing several operations on data, and don't want to save the output at each intermediate step.

For example, let's say we want to remove all the data corresponding to Month = 5, group the data by month, and then find the mean of the temperature each month. The conventional way to write the code for this would be:

```
filteredData <- filter(airquality, Month != 5)
groupedData <- group_by(filteredData, Month)
summarise(groupedData, mean(Temp, na.rm = TRUE))
```

With piping, the above code can be rewritten as:

```
airquality %>%
    filter(Month != 5) %>%
    group_by(Month) %>%
    summarise(mean(Temp, na.rm = TRUE))
```

This is a very basic example, and the usefulness may not be very apparent, but as the number of operations/functions perfomed on the data increase, the pipe operator becomes more and more useful!

dplyr verbs in action

The two most basic functions are `select()` and `filter()` which selects columns and filters rows, respectively.

Selecting columns using `select()`

Select a set of columns: the name and the sleep_total columns.

```
sleepData <- select(msleep, name, sleep_total)
head(sleepData)
```

```
##                          name sleep_total
## 1                     Cheetah        12.1
## 2                 Owl monkey        17.0
## 3            Mountain beaver        14.4
## 4 Greater short-tailed shrew        14.9
## 5                         Cow         4.0
## 6           Three-toed sloth        14.4
```

To select all the columns *except* a specific column, use the "-" (subtraction) operator (also known as negative indexing)

```
head(select(msleep, -name))
```

```
##        genus  vore        order conservation sleep_total sleep_rem
## 1   Acinonyx carni    Carnivora           lc        12.1        NA
## 2      Aotus  omni     Primates         <NA>        17.0       1.8
## 3 Aplodontia herbi     Rodentia           nt        14.4       2.4
## 4    Blarina  omni Soricomorpha           lc        14.9       2.3
## 5        Bos herbi Artiodactyla domesticated         4.0       0.7
## 6   Bradypus herbi       Pilosa         <NA>        14.4       2.2
##   sleep_cycle awake brainwt  bodywt
## 1          NA  11.9      NA  50.000
## 2          NA   7.0 0.01550   0.480
## 3          NA   9.6      NA   1.350
## 4   0.1333333   9.1 0.00029   0.019
## 5   0.6666667  20.0 0.42300 600.000
## 6   0.7666667   9.6      NA   3.850
```

To select a range of columns by name, use the ":" (colon) operator

```
head(select(msleep, name:order))
```

```
##                         name      genus  vore        order
## 1                    Cheetah   Acinonyx carni    Carnivora
## 2                 Owl monkey      Aotus  omni     Primates
## 3            Mountain beaver Aplodontia herbi     Rodentia
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha
## 5                        Cow        Bos herbi Artiodactyla
## 6           Three-toed sloth   Bradypus herbi       Pilosa
```

To select all columns that start with the character string "sl", use the function `starts_with()`

```
head(select(msleep, starts_with("sl")))
```

```
##   sleep_total sleep_rem sleep_cycle
## 1        12.1        NA          NA
## 2        17.0       1.8          NA
## 3        14.4       2.4          NA
## 4        14.9       2.3   0.1333333
## 5         4.0       0.7   0.6666667
## 6        14.4       2.2   0.7666667
```

Some additional options to select columns based on a specific criteria include

1. `ends_with()` = Select columns that end with a character string
2. `contains()` = Select columns that contain a character string
3. `matches()` = Select columns that match a regular expression
4. `one_of()` = Select columns names that are from a group of names

Selecting rows using `filter()`

Filter the rows for mammals that sleep a total of more than 16 hours.

```
filter(msleep, sleep_total >= 16)
```

```
##                    name        genus    vore           order conservation
## 1            Owl monkey        Aotus    omni        Primates         <NA>
## 2   Long-nosed armadillo      Dasypus   carni       Cingulata           lc
## 3 North American Opossum    Didelphis    omni Didelphimorphia           lc
## 4          Big brown bat    Eptesicus insecti      Chiroptera           lc
## 5   Thick-tailed opposum   Lutreolina   carni Didelphimorphia           lc
## 6       Little brown bat       Myotis insecti      Chiroptera         <NA>
## 7        Giant armadillo   Priodontes insecti       Cingulata           en
## 8 Arctic ground squirrel Spermophilus   herbi        Rodentia           lc
##   sleep_total sleep_rem sleep_cycle awake brainwt bodywt
## 1        17.0       1.8          NA   7.0 0.01550  0.480
## 2        17.4       3.1   0.3833333   6.6 0.01080  3.500
## 3        18.0       4.9   0.3333333   6.0 0.00630  1.700
## 4        19.7       3.9   0.1166667   4.3 0.00030  0.023
## 5        19.4       6.6          NA   4.6      NA  0.370
## 6        19.9       2.0   0.2000000   4.1 0.00025  0.010
## 7        18.1       6.1          NA   5.9 0.08100 60.000
## 8        16.6        NA          NA   7.4 0.00570  0.920
```

Filter the rows for mammals that sleep a total of more than 16 hours *and* have a body weight of greater than 1 kilogram.

```
filter(msleep, sleep_total >= 16, bodywt >= 1)
```

```
##                    name      genus    vore           order conservation
## 1   Long-nosed armadillo    Dasypus   carni       Cingulata           lc
## 2 North American Opossum  Didelphis    omni Didelphimorphia           lc
## 3        Giant armadillo Priodontes insecti       Cingulata           en
##   sleep_total sleep_rem sleep_cycle awake brainwt bodywt
## 1        17.4       3.1   0.3833333   6.6  0.0108    3.5
## 2        18.0       4.9   0.3333333   6.0  0.0063    1.7
## 3        18.1       6.1          NA   5.9  0.0810   60.0
```

Filter the rows for mammals in the Perissodactyla and Primates taxonomic order

```
filter(msleep, order %in% c("Perissodactyla", "Primates"))
```

```
##                name         genus  vore          order conservation
## 1        Owl monkey         Aotus  omni       Primates         <NA>
## 2            Grivet Cercopithecus  omni       Primates           lc
## 3             Horse         Equus herbi Perissodactyla domesticated
## 4            Donkey         Equus herbi Perissodactyla domesticated
## 5      Patas monkey  Erythrocebus  omni       Primates           lc
## 6            Galago        Galago  omni       Primates         <NA>
## 7             Human          Homo  omni       Primates         <NA>
## 8    Mongoose lemur         Lemur herbi       Primates           vu
## 9           Macaque        Macaca  omni       Primates         <NA>
## 10       Slow loris     Nyctibeus carni       Primates         <NA>
## 11       Chimpanzee           Pan  omni       Primates         <NA>
## 12           Baboon         Papio  omni       Primates         <NA>
## 13            Potto  Perodicticus  omni       Primates           lc
## 14 Squirrel monkey       Saimiri  omni       Primates         <NA>
## 15 Brazilian tapir       Tapirus herbi Perissodactyla           vu
```

You can use the boolean operators (e.g. >, <, >=, <=, !=, %in%) to create the logical tests.

Pipe operator: %>%

Before we go any futher, let's introduce the pipe operator: %>%. dplyr imports this operator from another package (magrittr). This operator allows you to pipe the output from one function to the input of another function. Instead of nesting functions (reading from the inside to the outside), the idea of of piping is to read the functions from left to right.

Here's an example you have seen:

```
head(select(msleep, name, sleep_total))
```

```
##                         name sleep_total
## 1                    Cheetah        12.1
## 2                 Owl monkey        17.0
## 3             Mountain beaver        14.4
## 4 Greater short-tailed shrew        14.9
## 5                        Cow         4.0
## 6            Three-toed sloth        14.4
```

Now in this case, we will pipe the msleep data frame to the function that will select two columns (name and sleep_total) and then pipe the new data frame to the function `head()` which will return the head of the new data frame.

```
msleep %>%
    select(name, sleep_total) %>%
    head
```

You will soon see how useful the pipe operator is when we start to combine many functions.

Back to dplyr verbs in action

Now that you know about the pipe operator (%>%), we will use it throughout the rest of this tutorial.

Arrange or re-order rows using `arrange()`

To arrange (or re-order) rows by a particular column such as the taxonomic order, list the name of the column you want to arrange the rows by

```
msleep %>% arrange(order) %>% head
```

```
##       name     genus  vore        order conservation sleep_total sleep_rem
## 1   Tenrec    Tenrec  omni Afrosoricida         <NA>        15.6       2.3
## 2      Cow       Bos herbi Artiodactyla domesticated         4.0       0.7
## 3 Roe deer Capreolus herbi Artiodactyla           lc         3.0        NA
## 4     Goat     Capri herbi Artiodactyla           lc         5.3       0.6
## 5  Giraffe   Giraffa herbi Artiodactyla           cd         1.9       0.4
## 6    Sheep      Ovis herbi Artiodactyla domesticated         3.8       0.6
##   sleep_cycle awake brainwt  bodywt
## 1          NA   8.4  0.0026   0.900
## 2   0.6666667  20.0  0.4230 600.000
## 3          NA  21.0  0.0982  14.800
## 4          NA  18.7  0.1150  33.500
## 5          NA  22.1      NA 899.995
## 6          NA  20.2  0.1750  55.500
```

Now, we will select three columns from msleep, arrange the rows by the taxonomic order and then arrange the rows by sleep_total. Finally show the head of the final data frame

```
msleep %>%
    select(name, order, sleep_total) %>%
    arrange(order, sleep_total) %>%
    head
```

```
##       name        order sleep_total
## 1   Tenrec Afrosoricida        15.6
## 2  Giraffe Artiodactyla         1.9
## 3 Roe deer Artiodactyla         3.0
## 4    Sheep Artiodactyla         3.8
## 5      Cow Artiodactyla         4.0
## 6     Goat Artiodactyla         5.3
```

Same as above, except here we filter the rows for mammals that sleep for 16 or more hours instead of showing the head of the final data frame

```
msleep %>%
    select(name, order, sleep_total) %>%
    arrange(order, sleep_total) %>%
    filter(sleep_total >= 16)
```

```
##                     name           order sleep_total
## 1          Big brown bat      Chiroptera        19.7
## 2       Little brown bat      Chiroptera        19.9
## 3   Long-nosed armadillo       Cingulata        17.4
## 4        Giant armadillo       Cingulata        18.1
## 5 North American Opossum Didelphimorphia        18.0
## 6   Thick-tailed opposum Didelphimorphia        19.4
## 7             Owl monkey        Primates        17.0
## 8 Arctic ground squirrel        Rodentia        16.6
```

Something slightly more complicated: same as above, except arrange the rows in the sleep_total column in a descending order. For this, use the function `desc()`

```
msleep %>%
    select(name, order, sleep_total) %>%
    arrange(order, desc(sleep_total)) %>%
    filter(sleep_total >= 16)
```


```
##                     name           order sleep_total
## 1          Big brown bat      Chiroptera        19.7
## 2       Little brown bat      Chiroptera        19.9
## 3   Long-nosed armadillo       Cingulata        17.4
## 4        Giant armadillo       Cingulata        18.1
## 5 North American Opossum Didelphimorphia        18.0
## 6   Thick-tailed opposum Didelphimorphia        19.4
## 7             Owl monkey        Primates        17.0
## 8 Arctic ground squirrel        Rodentia        16.6
```

Something slightly more complicated: same as above, except arrange the rows in the sleep_total column in a descending order. For this, use the function `desc()`

```
msleep %>%
    select(name, order, sleep_total) %>%
    arrange(order, desc(sleep_total)) %>%
    filter(sleep_total >= 16)
```

```
##                   name          order sleep_total
## 1       Little brown bat      Chiroptera        19.9
## 2          Big brown bat      Chiroptera        19.7
## 3        Giant armadillo       Cingulata        18.1
## 4   Long-nosed armadillo       Cingulata        17.4
## 5   Thick-tailed opposum Didelphimorphia        19.4
## 6 North American Opossum Didelphimorphia        18.0
## 7             Owl monkey        Primates        17.0
## 8 Arctic ground squirrel        Rodentia        16.6
```

Create new columns using `mutate()`

The `mutate()` function will add new columns to the data frame. Create a new column called rem_proportion which is the ratio of rem sleep to total amount of sleep.

```
msleep %>%
    mutate(rem_proportion = sleep_rem / sleep_total) %>%
    head
```

```
##                       name      genus  vore        order conservation
## 1                   Cheetah   Acinonyx carni    Carnivora           lc
## 2                 Owl monkey      Aotus  omni     Primates         <NA>
## 3            Mountain beaver Aplodontia herbi     Rodentia           nt
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha           lc
## 5                       Cow        Bos herbi Artiodactyla domesticated
## 6           Three-toed sloth   Bradypus herbi       Pilosa         <NA>
##   sleep_total sleep_rem sleep_cycle awake brainwt  bodywt rem_proportion
## 1        12.1        NA          NA  11.9      NA  50.000             NA
## 2        17.0       1.8          NA   7.0 0.01550   0.480      0.1058824
## 3        14.4       2.4          NA   9.6      NA   1.350      0.1666667
## 4        14.9       2.3   0.1333333   9.1 0.00029   0.019      0.1543624
## 5         4.0       0.7   0.6666667  20.0 0.42300 600.000      0.1750000
## 6        14.4       2.2   0.7666667   9.6      NA   3.850      0.1527778
```

You can many new columns using mutate (separated by commas). Here we add a second column called bodywt_grams which is the bodywt column in grams.

```
msleep %>%
    mutate(rem_proportion = sleep_rem / sleep_total,
           bodywt_grams = bodywt * 1000) %>%
    head
```

```
##                         name      genus  vore        order conservation
## 1                    Cheetah   Acinonyx carni    Carnivora           lc
## 2                 Owl monkey      Aotus  omni     Primates         <NA>
## 3             Mountain beaver Aplodontia herbi     Rodentia           nt
## 4 Greater short-tailed shrew    Blarina  omni Soricomorpha           lc
## 5                        Cow        Bos herbi Artiodactyla domesticated
## 6            Three-toed sloth   Bradypus herbi       Pilosa         <NA>
##   sleep_total sleep_rem sleep_cycle awake brainwt  bodywt rem_proportion
## 1        12.1        NA          NA  11.9      NA  50.000             NA
## 2        17.0       1.8          NA   7.0 0.01550   0.480      0.1058824
## 3        14.4       2.4          NA   9.6      NA   1.350      0.1666667
## 4        14.9       2.3   0.1333333   9.1 0.00029   0.019      0.1543624
## 5         4.0       0.7   0.6666667  20.0 0.42300 600.000      0.1750000
## 6        14.4       2.2   0.7666667   9.6      NA   3.850      0.1527778
##   bodywt_grams
## 1        50000
## 2          480
## 3         1350
## 4           19
## 5       600000
## 6         3850
```

Create summaries of the data frame using `summarise()`

The `summarise()` function will create summary statistics for a given column in the data frame such as finding the mean. For example, to compute the average number of hours of sleep, apply the `mean()` function to the column sleep_total and call the summary value avg_sleep.

```
msleep %>%
    summarise(avg_sleep = mean(sleep_total))
```

```
##   avg_sleep
## 1  10.43373
```

There are many other summary statistics you could consider such `sd()`, `min()`, `max()`, `median()`, `sum()`, `n()` (returns the length of vector), `first()` (returns first value in vector), `last()` (returns last value in vector) and `n_distinct()` (number of distinct values in vector).

```
msleep %>%
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
##   avg_sleep min_sleep max_sleep total
## 1  10.43373       1.9      19.9    83
```

Group operations using `group_by()`

The `group_by()` verb is an important function in dplyr. As we mentioned before it's related to concept of "split-apply-combine". We literally want to split the data frame by some variable (e.g. taxonomic order), apply a function to the individual data frames and then combine the output.

Let's do that: split the msleep data frame by the taxonomic order, then ask for the same summary statistics as above. We expect a set of summary statistics for each taxonomic order.

```
msleep %>%
    group_by(order) %>%
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
## Source: local data frame [19 x 5]
##
##              order avg_sleep min_sleep max_sleep total
## 1     Afrosoricida 15.600000      15.6      15.6     1
## 2     Artiodactyla  4.516667       1.9       9.1     6
## 3        Carnivora 10.116667       3.5      15.8    12
## 4          Cetacea  4.500000       2.7       5.6     3
## 5       Chiroptera 19.800000      19.7      19.9     2
## 6        Cingulata 17.750000      17.4      18.1     2
## 7  Didelphimorphia 18.700000      18.0      19.4     2
## 8    Diprotodontia 12.400000      11.1      13.7     2
## 9   Erinaceomorpha 10.200000      10.1      10.3     2
## 10      Hyracoidea  5.666667       5.3       6.3     3
## 11      Lagomorpha  8.400000       8.4       8.4     1
## 12     Monotremata  8.600000       8.6       8.6     1
## 13  Perissodactyla  3.466667       2.9       4.4     3
## 14          Pilosa 14.400000      14.4      14.4     1
## 15        Primates 10.500000       8.0      17.0    12
## 16     Proboscidea  3.600000       3.3       3.9     2
## 17        Rodentia 12.468182       7.0      16.6    22
## 18      Scandentia  8.900000       8.9       8.9     1
## 19    Soricomorpha 11.100000       8.4      14.9     5
```

Data : Income Data by States

In this tutorial, we are using the following data which contains income generated by states from year 2002 to 2015. **Note :** This data do not contain actual income figures of the states.

This dataset contains 51 observations (rows) and 16 variables (columns). The snapshot of first 6 rows of the dataset is shown below.

```
  Index      State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387
    Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341
5 1624509 1639670 1921845 1156536 1388461 1644607
6 1913275 1665877 1491604 1178355 1383978 1330736
```

Example 1 : Selecting Random N Rows

The **sample_n** function selects random rows from a data frame (or table). The second parameter of the function tells R the number of rows to select.

```
sample_n(mydata,3)
```

```
   Index    State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009
2      A   Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
8      D Delaware 1330403 1268673 1706751 1403759 1441351 1300836 1762096 1553585
33     N New York 1395149 1611371 1170675 1446810 1426941 1463171 1732098 1426216
     Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
2  1629616 1230866 1512804 1985302 1580394 1979143
8  1370984 1318669 1984027 1671279 1803169 1627508
33 1604531 1683687 1500089 1718837 1619033 1367705
```

Example 2 : Selecting Random Fraction of Rows

The **sample_frac** function returns randomly N% of rows. In the example below, it returns randomly 10% of rows.

```
sample_frac(mydata,0.1)
```

Example 3 : Remove Duplicate Rows based on all the variables (Complete Row)

The **distinct function** is used to eliminate duplicates.

```
x1 = distinct(mydata)
```

In this dataset, there is not a single duplicate row so it returned same number of rows as in mydata.

Example 4 : Remove Duplicate Rows based on a variable

The **.keep_all** function is used to retain all other variables in the output data frame.

```
x2 = distinct(mydata, Index, .keep_all= TRUE)
```

Example 5 : Remove Duplicates Rows based on multiple variables

In the example below, we are using two variables - **Index, Y2010** to determine uniqueness.

```
x2 = distinct(mydata, Index, Y2010, .keep_all= TRUE)
```

select( ) Function

It is used to select only desired variables.

```
select() syntax : select(data , ....)
data : Data Frame
.... : Variables by name or by function
```

Example 6 : Selecting Variables (or Columns)

Suppose you are asked to select only a few variables. The code below selects variables "Index", columns from "State" to "Y2008".

```
mydata2 = select(mydata, Index, State:Y2008)
```

Example 7 : Dropping Variables

The **minus sign** before a variable tells R to drop the variable.

```
mydata = select(mydata, -Index, -State)
```

The above code can also be written like :

```
mydata = select(mydata, -c(Index,State))
```

Example 8 : Selecting or Dropping Variables starts with 'Y'

The **starts_with()** function is used to select variables starts with an alphabet.

```
mydata3 = select(mydata, starts_with("Y"))
```

Adding a negative sign before starts_with() implies dropping the variables starts with 'Y'

```
mydata33 = select(mydata, -starts_with("Y"))
```

The following functions helps you to select variables based on their names.

Helpers	Description
starts_with()	Starts with a prefix
ends_with()	Ends with a prefix
contains()	Contains a literal string
matches()	Matches a regular expression
num_range()	Numerical range like x01, x02, x03.
one_of()	Variables in character vector.
everything()	All variables.

Example 9 : Selecting Variables contain 'I' in their names

```
mydata4 = select(mydata, contains("I"))
```

Example 10 : Reorder Variables

The code below keeps variable **'State'** in the front and the remaining variables follow that.

```
mydata5 = select(mydata, State, everything())
```

New order of variables are displayed below -

```
 [1] "State" "Index" "Y2002" "Y2003" "Y2004" "Y2005" "Y2006" "Y2007" "Y2008" "Y2009"
[11] "Y2010" "Y2011" "Y2012" "Y2013" "Y2014" "Y2015"
```

rename( ) Function

It is used to change variable name.

> ***rename() syntax :*** *rename(data , new_name = old_name)*
> ***data :*** *Data Frame*
> ***new_name :*** *New variable name you want to keep*
> ***old_name :*** *Existing Variable Name*

Example 11 : Rename Variables

The rename function can be used to rename variables.

In the following code, we are renaming **'Index'** variable to **'Index1'**.

> *mydata6 = rename(mydata, Index1=Index)*

```
> names(mydata6)
 [1] "Index1" "State"  "Y2002"

[10] "Y2009"  "Y2010"  "Y2011"
```

Output

filter( ) Function

It is used to subset data with matching logical conditions.

> ***filter() syntax :*** *filter(data , ...)*
> ***data :*** *Data Frame*
> ***.... :*** *Logical Condition*

Example 12 : Filter Rows

Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.

```
mydata7 = filter(mydata, Index == "A")
```

```
 Index    State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009
1     A  Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2     A   Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3     A  Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4     A Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980
    Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341
```

Example 13 : Multiple Selection Criteria

The **%in%** operator can be used to select multiple items. In the following program, we are telling R to select rows against 'A' and 'C' in column 'Index'.

```
mydata7 = filter(mydata6, Index %in% c("A", "C"))
```

Example 14 : 'AND' Condition in Selection Criteria

Suppose you need to apply 'AND' condition. In this case, we are picking data for 'A' and 'C' in the column 'Index' and income greater than 1.3 million in Year 2002.

```
mydata8 = filter(mydata6, Index %in% c("A", "C") & Y2002 >= 1300000 )
```

Example 15 : 'OR' Condition in Selection Criteria

The 'I' denotes OR in the logical condition. It means any of the two conditions.

```
mydata9 = filter(mydata6, Index %in% c("A", "C") | Y2002 >= 1300000)
```

Example 16 : NOT Condition

The "!" sign is used to reverse the logical condition.

```
mydata10 = filter(mydata6, !Index %in% c("A", "C"))
```

Example 17 : CONTAINS Condition

The **grepl function** is used to search for pattern matching. In the following code, we are looking for records wherein column **state** contains **'Ar'** in their name.

```
mydata10 = filter(mydata6, grepl("Ar", State))
```

summarise( ) Function

It is used to summarize data.

summarise() syntax : *summarise(data , ...)*
data : *Data Frame*
..... : *Summary Functions such as mean, median etc*

Example 18 : Summarize selected variables

In the example below, we are calculating mean and median for the variable Y2015.

```
summarise(mydata, Y2015_mean = mean(Y2015),
Y2015_med=median(Y2015))
```

```
Y2015_mean Y2015_med
   1588297    1627508
```

Output

Example 19 : Summarize Multiple Variables

In the following example, we are calculating number of records, mean and median for variables Y2005 and Y2006. The **summarise_at** function allows us to select multiple variables by their names.

```
summarise_at(mydata, vars(Y2005, Y2006), funs(n(), mean, median))
```

```
Y2005_n Y2006_n Y2005_mean Y2006_mean Y2005_median Y2006_median
     51      51    1522064    1530969       1480280       1531641
```

Output

Example 20 : Summarize with Custom Functions

We can also use custom functions in the summarise function. In this case, we are computing the number of records, number of missing values, mean and median for variables Y2011 and Y2012. The **dot (.)** denotes each variables specified in the second argument of the function.

```
summarise_at(mydata, vars(Y2011, Y2012),
funs(n(), missing = sum(is.na(.)), mean(., na.rm = TRUE), median(.,na.rm = TRUE)))
```

```
Y2011_n Y2012_n Y2011_missing Y2012_missing Y2011_mean Y2012_mean Y2011_median
     51      51              0              0    1574968    1591135      1575533
Y2012_median
     1643855
```

Summarize : Output

How to apply Non-Standard Functions

Suppose you want to subtract mean from its original value and then calculate variance of it.

```
set.seed(222)
mydata <- data.frame(X1=sample(1:100,100), X2=runif(100))
summarise_at(mydata,vars(X1,X2), function(x) var(x - mean(x)))
```

```
    X1         X2
1 841.6667 0.08142161
```

Example 21 : Summarize all Numeric Variables

The **summarise_if** function allows you to summarise conditionally.

```
summarise_if(mydata, is.numeric, funs(n(),mean,median))
```

Alternative Method :

First, store data for all the numeric variables

```
numdata = mydata[sapply(mydata,is.numeric)]
```

Second, the **summarise_all** function calculates summary statistics for all the columns in a data frame

```
summarise_all(numdata, funs(n(),mean,median))
```

Example 22 : Summarize Factor Variable

We are checking the **number of levels/categories** and **count of missing observations** in a categorical (factor) variable.

```
summarise_all(mydata["Index"], funs(nlevels(.), nmiss=sum(is.na(.))))
```

```
  nlevels nmiss
1      19   0
```

arrange() function :

Use : Sort data

Syntax

```
arrange(data_frame, variable(s)_to_sort)
or
data_frame %>% arrange(variable(s)_to_sort)
```

To sort a variable in descending order, use **desc(x)**.

Example 23 : Sort Data by Multiple Variables

The default sorting order of **arrange() function** is ascending. In this example, we are sorting data by multiple variables.

> *arrange(mydata, Index, Y2011)*

Suppose you need to sort one variable by descending order and other variable by ascending oder.

> *arrange(mydata, **desc(Index)**, Y2011)*

Pipe Operator %>%

It is important to understand the pipe (%>%) operator before knowing the other functions of dplyr package. dplyr utilizes pipe operator from another package **(magrittr)**.

> *It allows you to write sub-queries like we do it in sql.*

Note : All the functions in dplyr package can be used **without** the pipe operator. The question arises **"Why to use pipe operator %>%". The answer is** it lets to wrap multiple functions together with the use of %>%.

Syntax :

> *filter(data_frame, variable == value)*
> ***or***
> *data_frame **%>%** filter(variable == value)*

The %>% is NOT restricted to filter function. It can be used with any function.

Example :

The code below demonstrates the usage of pipe %>% operator. In this example, we are selecting 10 random observations of two variables "Index" "State" from the data frame "mydata".

```
dt = sample_n(select(mydata, Index, State),10)
or
dt = mydata %>% select(Index, State) %>% sample_n(10)
```

	Index	State
44	T	Texas
32	N	New Mexico
51	W	Wyoming
9	D	District of Columbia
5	C	California
40	R	Rhode Island
22	M	Massachusetts
4	A	Arkansas
42	S	South Dakota
46	V	Vermont

group_by() function :

Use : Group data by categorical variable

Syntax :

```
group_by(data, variables)
or
data %>% group_by(variables)
```

Example 24 : Summarise Data by Categorical Variable

We are calculating count and mean of variables Y2011 and Y2012 by variable Index.

```
t = summarise_at(group_by(mydata, Index), vars(Y2011, Y2012), funs(n(),
mean(., na.rm = TRUE)))
```

The above code can also be written like

```
t = mydata %>% group_by(Index) %>%
  summarise_at(vars(Y2011:Y2015), funs(n(), mean(., na.rm = TRUE)))
```

Index	Y2011_n	Y2012_n	Y2013_n	Y2014_n	Y2015_n	Y2011_mean	Y2012_mean
A	4	4	4	4	4	1432642	1455876
C	3	3	3	3	3	1750357	1547326
D	2	2	2	2	2	1336059	1981868
F	1	1	1	1	1	1497051	1131928
G	1	1	1	1	1	1851245	1850111
H	1	1	1	1	1	1902816	1695126
I	4	4	4	4	4	1690171	1687056
K	2	2	2	2	2	1489353	1899773
L	1	1	1	1	1	1210385	1234234
M	8	8	8	8	8	1582714	1586091
N	8	8	8	8	8	1448351	1470316
O	3	3	3	3	3	1882111	1602463
P	1	1	1	1	1	1483292	1290329
R	1	1	1	1	1	1781016	1909119
S	2	2	2	2	2	1381724	1671744
T	2	2	2	2	2	1724080	1865787
U	1	1	1	1	1	1288285	1108281
V	2	2	2	2	2	1482143	1488651
W	4	4	4	4	4	1711341	1660192

do() function :

Use : Compute within groups

Syntax :

```
do(data_frame, expressions_to_apply_to_each_group)
```

Note : *The **dot (.)** is required to refer to a data frame.*

Example 25 : Filter Data within a Categorical Variable

Suppose you need to pull top 2 rows from 'A', 'C' and 'I' categories of variable Index.

```
t = mydata %>% filter(Index %in% c("A", "C","I")) %>% group_by(Index) %>%
  do(head( . , 2))
```

	Index	State	Y2002	Y2003	Y2004	Y2005	Y2006
1	A	Alabama	1296530	1317711	1118631	1492583	1107408
2	A	Alaska	1170302	1960378	1818085	1447852	1861639
3	C	California	1685349	1675807	1889570	1480280	1735069
4	C	Colorado	1343824	1878473	1886149	1236697	1871471
5	I	Idaho	1353210	1438538	1739154	1541015	1122387
6	I	Illinois	1508356	1527440	1493029	1261353	1540274

Output : do() function

Example 26 : Selecting 3rd Maximum Value by Categorical Variable

We are calculating third maximum value of variable Y2015 by variable Index. The following code first selects only two variables Index and Y2015. Then it filters the variable Index with 'A', 'C' and 'I' and then it groups the same variable and sorts the variable Y2015 in descending order. At last, it selects the third row.

```
t = mydata %>% select(Index, Y2015) %>%
  filter(Index %in% c("A", "C","I")) %>%
  group_by(Index) %>%
  do(arrange(.,desc(Y2015))) %>% slice(3)
```

he **slice() function** is used to select rows by position.

	Index	Y2015
1	A	1647724
2	C	1330736
3	I	1583516

Output

Using Window Functions

Like SQL, dplyr uses window functions that are used to subset data within a group. It returns a vector of values. We could use **min_rank() function** that calculates rank in the preceding example,

```
t = mydata %>% select(Index, Y2015) %>%
  filter(Index %in% c("A", "C","I")) %>%
  group_by(Index) %>%
  filter(min_rank(desc(Y2015)) == 3)
```

```
  Index    Y2015
1     A  1647724
2     C  1330736
3     I  1583516
```

Example 27 : Summarize, Group and Sort Together

In this case, we are computing mean of variables Y2014 and Y2015 by variable Index. Then sort the result by calculated mean variable Y2015.

```
t = mydata %>%
  group_by(Index)%>%
  summarise(Mean_2014 = mean(Y2014, na.rm=TRUE),
            Mean_2015 = mean(Y2015, na.rm=TRUE)) %>%
  arrange(desc(Mean_2015))
```

mutate() function :

Use : Creates new variables

Syntax :

```
mutate(data_frame, expression(s) )
or
data_frame %>% mutate(expression(s))
```

Example 28 : Create a new variable

The following code calculates division of Y2015 by Y2014 and name it "change".

```
mydata1 = mutate(mydata, change=Y2015/Y2014)
```

Example 29 : Multiply all the variables by 1000

It creates new variables and name them with suffix "_new".

```
mydata11 = mutate_all(mydata, funs("new" = .* 1000))
```

Y2002_new	Y2003_new	Y2004_new	Y2005_new
1296530000	1317711000	1118631000	1492583000
1170302000	1960378000	1818085000	1447852000
1742027000	1968140000	1377583000	1782199000
1485531000	1994927000	1119299000	1947979000
1685349000	1675807000	1889570000	1480280000

Output

The output shown in the image above is truncated due to high number of variables.

Note - The above code returns the following error messages -

Warning messages:
1: In Ops.factor(c(1L, 1L, 1L, 1L, 2L, 2L, 2L, 3L, 3L, 4L, 5L, 6L, :
  '*' not meaningful for factors
2: In Ops.factor(1:51, 1000) : '*' not meaningful for factors

It implies you are multiplying 1000 to string(character) values which are stored as factor variables. These variables are 'Index', 'State'. It does not make sense to apply multiplication operation on character variables. For these two variables, it creates newly created variables which contain only NA.

Example 30 : Calculate Rank for Variables

Suppose you need to calculate rank for variables Y2008 to Y2010.

```
mydata12 = mutate_at(mydata, vars(Y2008:Y2010), funs(Rank=min_rank(.)))
```

Y2008_Rank	Y2009_Rank	Y2010_Rank
47	46	8
27	9	38
33	14	12
8	24	40
24	27	36
43	31	47
37	50	48

Output

By default, **min_rank()** assigns 1 to the smallest value and high number to the largest value. In case, you need to assign rank 1 to the largest value of a variable, use **min_rank(desc(.))**

```
mydata13 = mutate_at(mydata, vars(Y2008:Y2010),
funs(Rank=min_rank(desc(.))))
```

Example 31 : Select State that generated highest income among the variable 'Index'

```
out = mydata %>% group_by(Index) %>% filter(min_rank(desc(Y2015)) == 1)
%>%
  select(Index, State, Y2015)
```

```
   Index         State   Y2015
1      A        Alaska 1979143
2      C   Connecticut 1718072
3      D      Delaware 1627508
4      F       Florida 1170389
5      G       Georgia 1725470
6      H        Hawaii 1150882
7      I         Idaho 1757171
8      K      Kentucky 1913350
9      L     Louisiana 1403857
10     M      Missouri 1996005
11     N New Hampshire 1963313
12     O        Oregon 1893515
13     P  Pennsylvania 1668232
14     R  Rhode Island 1611730
15     S  South Dakota 1136443
16     T         Texas 1705322
17     U          Utah 1729273
18     V      Virginia 1850394
19     W       Wyoming 1853858
```

Example 32 : Cumulative Income of 'Index' variable

The **cumsum function** calculates cumulative sum of a variable. With **mutate function,** we insert a new variable called 'Total' which contains values of cumulative income of variable Index.

```
out2 = mydata %>% group_by(Index) %>% mutate(Total=cumsum(Y2015))
%>%
select(Index, Y2015, Total)
```

join() function :

Use : Join two datasets

Syntax :

```
inner_join(x, y, by = )
left_join(x, y, by = )
right_join(x, y, by = )
full_join(x, y, by = )
semi_join(x, y, by = )
anti_join(x, y, by = )
```

x, y - datasets (or tables) to merge / join
by - common variable (primary key) to join by.

Example 33 : Common rows in both the tables

```
df1 = data.frame(ID = c(1, 2, 3, 4, 5),
                 w = c('a', 'b', 'c', 'd', 'e'),
                 x = c(1, 1, 0, 0, 1),
                 y=rnorm(5),
                 z=letters[1:5])

df2 = data.frame(ID = c(1, 7, 3, 6, 8),
                 a = c('z', 'b', 'k', 'd', 'l'),
                 b = c(1, 2, 3, 0, 4),
                 c =rnorm(5),
                 d =letters[2:6])
```

INNER JOIN returns rows when there is a match in both tables. In this example, we are merging df1 and df2 with ID as common variable (primary key).

> *df3 = inner_join(df1, df2, by = "ID")*

```
  ID w x          y z a b          c d
1  1 a 1 -0.9934455 a z 1 -0.6556326 b
2  3 c 0 -1.4342218 c k 3 -1.4055054 d
```

Output : INNER JOIN

If the primary key does not have same name in both the tables, try the following way:

> *inner_join(df1, df2, by = c("ID"="ID1"))*

Example 34 : Applying LEFT JOIN

LEFT JOIN : It returns all rows from the left table, even if there are no matches in the right table.

> *left_join(df1, df2, by = "ID")*

```
  ID w x          y z    a  b          c    d
1  1 a 1 -0.9934455 a    z  1 -0.6556326    b
2  2 b 1 -1.3061685 b <NA> NA         NA <NA>
3  3 c 0 -1.4342218 c    k  3 -1.4055054    d
4  4 d 0 -0.8628479 d <NA> NA         NA <NA>
5  5 e 1  1.7037992 e <NA> NA         NA <NA>
```

Output : LEFT JOIN

Combine Data Vertically

intersect(x, y)
Rows that appear in both x and y.

union(x, y)
Rows that appear in either or both x and y.

setdiff(x, y)
Rows that appear in x but not y.

Example 35 : Applying INTERSECT

Prepare Sample Data for Demonstration

```
mtcars$model <- rownames(mtcars)
first <- mtcars[1:20, ]
second <- mtcars[10:32, ]
```

INTERSECT selects unique rows that are common to both the data frames.

```
intersect(first, second)
```

Example 36 : Applying UNION

UNION displays all rows from both the tables and removes duplicate records from the combined dataset. By using **union_all function**, it allows duplicate rows in the combined dataset.

```
x=data.frame(ID = 1:6, ID1= 1:6)
y=data.frame(ID = 1:6,  ID1 = 1:6)
union(x,y)
union_all(x,y)
```

Example 37 : Rows appear in one table but not in other table

```
setdiff(first, second)
```

Example 38 : IF ELSE Statement

Syntax :

```
if_else(condition, true, false, missing = NULL)
```

true : Value if condition meets
false : Value if condition does not meet
missing : Value if missing cases.It will be used to replace missing values (Default : NULL)

```
df <- c(-10,2, NA)
if_else(df < 0, "negative", "positive", missing = "missing value")
```

Create a new variable with IF_ELSE

If a value is less than 5, add it to 1 and if it is greater than or equal to 5, add it to 2. Otherwise 0.

```
df =data.frame(x = c(1,5,6,NA))
df %>% mutate(newvar=if_else(x<5, x+1, x+2,0))
```

x	newvar
1	2
5	7
6	8
NA	0

Output

Nested IF ELSE

Multiple IF ELSE statement can be written using if_else() function. See the example below -

```
mydf =data.frame(x = c(1:5,NA))
mydf %>% mutate(newvar= if_else(is.na(x),"I am missing",
if_else(x==1,"I am one",
if_else(x==2,"I am two",
if_else(x==3,"I am three","Others")))))
```

Output

```
   x          flag
1  1      I am one
2  2      I am two
3  3    I am three
4  4        Others
5  5        Others
6 NA I am missing
```

SQL-Style CASE WHEN Statement

We can use **case_when()** function to write nested if-else queries. In case_when(), you can use variables directly within case_when() wrapper. **TRUE** refers to ELSE statement.

```
mydf %>% mutate(flag = case_when(is.na(x) ~ "I am missing",
                                 x == 1 ~ "I am one",
                                 x == 2 ~ "I am two",
                                 x == 3 ~ "I am three",
                                TRUE ~ "Others"))
```

Important Point

*Make sure you set **is.na()** condition at the beginning in nested ifelse. Otherwise, it would not be executed.*

Example 39 : Apply ROW WISE Operation

Suppose you want to find maximum value in each row of variables 2012, 2013, 2014, 2015. The **rowwise()** function allows you to apply functions to rows.

```
df = mydata %>%
  rowwise() %>% mutate(Max= max(Y2012,Y2013,Y2014,Y2015)) %>%
  select(Y2012:Y2015,Max)
```

Example 40 : Combine Data Frames

Suppose you are asked to combine two data frames. Let's first create two sample datasets.

```
df1=data.frame(ID = 1:6,  x=letters[1:6])
df2=data.frame(ID = 7:12, x=letters[7:12])
```

df1	
ID	x
1	a
2	b
3	c
4	d
5	e
6	f

df2	
ID	x
7	g
8	h
9	i
10	j
11	k
12	l

The **bind_rows() function** combine two datasets with rows. So combined dataset would contain **12 rows (6+6) and 2 columns.**

```
xy = bind_rows(df1,df2)
```

It is equivalent to base R function rbind.

```
xy = rbind(df1,df2)
```

The **bind_cols() function** combine two datasets with columns. So combined dataset would contain **4 columns and 6 rows.**

```
xy = bind_cols(x,y)
or
xy = cbind(x,y)
```

The output is shown below-

ID	x	ID	x
1	a	7	g
2	b	8	h
3	c	9	i
4	d	10	j
5	e	11	k
6	f	12	l

cbind Output

Example 41 : Calculate Percentile Values

The **quantile()** function is used to determine Nth percentile value. In this example, we are computing percentile values by variable Index.

```
mydata %>% group_by(Index) %>%
  summarise(Pecentile_25=quantile(Y2015, probs=0.25),
            Pecentile_50=quantile(Y2015, probs=0.5),
            Pecentile_75=quantile(Y2015, probs=0.75),
            Pecentile_99=quantile(Y2015, probs=0.99))
```

The **ntile()** function is used to divide the data into N bins.

```
x= data.frame(N= 1:10)
x = mutate(x, pos = ntile(x$N,5))
```

Example 42 : Automate Model Building

This example explains the advanced usage of **do() function**. In this example, we are building linear regression model for each level of a categorical variable. There are 3 levels in variable cyl of dataset mtcars.

```
length(unique(mtcars$cyl))
```

Result : 3

```
by_cyl <- group_by(mtcars, cyl)
models <- by_cyl %>% do(mod = lm(mpg ~ disp, data = .))
summarise(models, rsq = summary(mod)$r.squared)
models %>% do(data.frame(
  var = names(coef(.$mod)),
  coef(summary(.$mod)))
)
```

rsq
0.64840514
0.01062604
0.27015777

Output : R-Squared Values

if() Family of Functions

It includes functions like select_if, mutate_if, summarise_if. They come into action only when logical condition meets. See examples below.

Example 43 : Select only numeric columns

The **select_if()** function returns only those columns where logical condition is TRUE. The **is.numeric** refers to retain only numeric variables.

```
mydata2 = select_if(mydata, is.numeric)
```

Similarly, you can use the following code for selecting factor columns -

```
mydata3 = select_if(mydata, is.factor)
```

Example 44 : Number of levels in factor variables

Like select_if() function, summarise_if() function lets you to summarise only for variables where logical condition holds.

```
summarise_if(mydata, is.factor, funs(nlevels(.)))
```

It returns 19 levels for variable Index and 51 levels for variable State.

Example 45 : Multiply by 1000 to numeric variables

```
mydata11 = mutate_if(mydata, is.numeric, funs("new" = .* 1000))
```

Example 46 : Convert value to NA

In this example, we are converting "" to NA using **na_if()** function.

```
k <- c("a", "b", "", "d")
na_if(k, "")
```

Result : "a" "b" NA "d"

Difference between Correlation and regression

Correlation is described as the analysis which lets us know the association or the absence of the relationship between two variables 'x' and 'y'. On the other end, **Regression** analysis, predicts the value of the dependent variable based on the known value of the independent variable, assuming that average mathematical relationship between two or more variables.

Comparison Chart

BASIS FOR COMPARISON	CORRELATION	REGRESSION
Meaning	Correlation is a statistical measure which determines co-relationship or association of two variables.	Regression describes how an independent variable is numerically related to the dependent variable.
Usage	To represent linear relationship between two variables.	To fit a best line and estimate one variable on the basis of another variable.
Dependent and Independent variables	No difference	Both variables are different.
Indicates	Correlation coefficient indicates the extent to which two variables move together.	Regression indicates the impact of a unit change in the known variable (x) on the estimated variable (y).
Objective	To find a numerical value expressing the relationship between variables.	To estimate values of random variable on the basis of the values of fixed variable.

Key Differences Between Correlation and Regression

The points given below, explains the difference between correlation and regression in detail:

1. A statistical measure which determines the co-relationship or association of two quantities is known as Correlation. Regression describes how an independent variable is numerically related to the dependent variable.
2. Correlation is used to represent the linear relationship between two variables. On the contrary, regression is used to fit the best line and estimate one variable on the basis of another variable.
3. In correlation, there is no difference between dependent and independent variables i.e. correlation between x and y is similar to y and x. Conversely, the regression of y on x is different from x on y.
4. Correlation indicates the strength of association between variables. As opposed to, regression reflects the impact of the unit change in the independent variable on the dependent variable.
5. Correlation aims at finding a numerical value that expresses the relationship between variables. Unlike regression whose goal is to predict values of the random variable on the basis of the values of fixed variable.

Correlation focuses primarily on an association, while regression is designed to help make predictions.

Correlation determines the extent of linear relationship between two variables, and regression is a mathematical model to represent a cause-and-effect relationship, simplest form being a simple linear regression/bi-variate linear regression having one explanatory/predictor (or independent) variable, and one dependent/response variable.

Null Hypothesis for regression

If there is a significant linear relationship between the independent variable *X* and the dependent variable *Y*, the slope will *not* equal zero.

$$H_o: B_1 = 0$$

$$H_a: B_1 \neq 0$$

The null hypothesis states that the slope is equal to zero, and the alternative hypothesis states that the slope is not equal to zero.

Case 1: Estimate the number of cigarettes consumed monthly in India

Solution: A good proxy in such problem is the population of India i.e. 1.2 billion. Following is an effective way to segment this population:

	Population : 1.2 Bn (100%)								
Segment level I	Age above 22 yrs (60%)				Age between 16 & 22 yrs(10%)				Age <16yrs (30%)
Segment level II	Urban (20%)		Rural (40%)		Urban (3%)		Rural (7%)		
Segment level III	Male (11%)	Female (9%)	Male (25%)	Female (15%)	Male (1.5%)	Female (1.5%)	Male (4%)	Female (3%)	
Avg. cigarettes PM	30	15	5	2	20	10	2	1	0
Population	132000000	108000000	300000000	180000000	18000000	18000000	48000000	36000000	360000000
# cigarettes PM	3960000000	1620000000	1500000000	360000000	360000000	180000000	96000000	36000000	0
Total cigarettes	8.1 Trillion								

Following were the key considerations in building the segmentation and the intermediate guesses:

1. The rural population consumes far lesser cigarettes than urban because of the purchasing power difference.

2. Male consume more cigarettes than female in both urban and rural populations.

3. Children below 16 years consume a negligible number of cigarettes.

4. Male to Female ratio in Urban is closer to 1 than that of Rural.

5. Male to Female ratio in younger generations is closer to 1 than that of older. This is because of the increase in awareness level.

6. Bulk of population start smoking after getting into a job and hence the average number cigarettes are higher in older groups.

7. Total number of cigarettes from the supply side also come to around 10 Trillion, which gives a good sense check on the final number.

Case 3: Estimate the number of tennis balls bough in India per month

Solution: A good proxy in this problem is the number of cities in India i.e. ~1700. The catch in this problem is to analyze where all can we use tennis balls. Once we have the number of tennis balls used monthly, we can easily find the number of tennis ball bought in a month using the lifetime of tennis balls.

Following is an effective way to segment this population:

Parameters	Possible Tennis ball usage							
Segment Level I	Tennis				Cricket			
Segment Level II	Urban				Urban			
Segment Level III	Metro	Tier-2	Small towns	Rural	Metro	Tier-2	Small towns	Rural
#cities	5	60	1600	5000	5	60	1600	5000
# sectors/cities	100	50	30	10	100	50	30	10
# grounds/sectors	5	3	2	0	50	40	30	10
# daily balls consumed	5	3	2		2	2	2	2
Total daily balls consumed	12500	27000	192000	0	50000	240000	2880000	1000000
Monthly ball consumption	4.4 Million							

Following were the key considerations in building the segmentation and the intermediate guesses:

1. Rural areas have negligible number of tennis courts.

2. Metro cities have the highest number of sectors.

3. For each sectors in metro cities, the number of grounds for both tennis and cricket is higher. This is both because of the bigger area and the higher buying capacity in metros.

4.Number of balls consumed in metros per ground is higher because of the higher engagement in metros.

How to get 2nd highest salary of each employee in employee table which contains more than one entry in employee table

```
SELECT MAX(T.salery),T.NAME FROM TABLE T
    INNER JOIN (SELECT MAX(salery),NAME FROM TABLE GROUP BY NAME) TT
     ON TT.NAME=T.NAME AND TT.SALERY!= T.SALERY
      GROUP BY T.NAME;
```

example

```
mysql> SELECT * FROM payments;
+----+------------+---------+-------+
| id | date       | user_id | value |
+----+------------+---------+-------+
|  1 | 2016-06-22 |       1 |    10 |
|  2 | 2016-06-22 |       3 |    15 |
|  3 | 2016-06-22 |       4 |    20 |
|  4 | 2016-06-23 |       2 |   100 |
|  5 | 2016-06-23 |       1 |   150 |
|  6 | 2016-06-23 |       2 |   340 |
+----+------------+---------+-------+
```

SQL query for find second highest salary of employee?

```
SELECT salary FROM Employee
ORDER BY salary DESC
LIMIT 1,1;
```

according to documentation first argument is an offset and the second argument specifies a maximum number of rows to return.Remember limit take first entry at 0;
so if u want row number 20
then use
LIMIT 19,1; it will leave 19 rows and take 1 row after that

Difference between One-Way Anova and Two-way Anova

What are the hypotheses of a One-Way ANOVA?

In a one-way ANOVA there are two possible hypotheses.

- The null hypothesis (H0) is that there is no difference between the groups and equality between means. (Walruses weigh the same in different months)
- The alternative hypothesis (H1) is that there is a difference between the means and groups. (Walruses have different weights in different months)

What are the assumptions of a One-Way ANOVA?

- Normality – That each sample is taken from a normally distributed population
- Sample independence – that each sample has been drawn independently of the other samples
- Variance Equality – That the variance of data in the different groups should be the same
- Your dependent variable – here, "weight", should be continuous – that is, measured on a scale which can be subdivided using increments (i.e. grams, milligrams)

What are the assumptions of a Two-Way ANOVA?

- Your dependent variable – here, "weight", should be continuous – that is, measured on a scale which can be subdivided using increments (i.e. grams, milligrams)
- Your two independent variables – here, "month" and "gender", should be in categorical, independent groups.
- Sample independence – that each sample has been drawn independently of the other samples
- Variance Equality – That the variance of data in the different groups should be the same
- Normality – That each sample is taken from a normally distributed population

What are the hypotheses of a Two-Way ANOVA?

Because the two-way ANOVA consider the effect of two categorical factors, and the effect of the categorical factors on each other, there are three pairs of null or alternative hypotheses for the two-way ANOVA. Here, we present them for our walrus experiment, where month of mating season and gender are the two independent variables.

- H0: The means of all month groups are equal
- H1: The mean of at least one month group is different

- H0: The means of the gender groups are equal
- H1: The means of the gender groups are different

- H0: There is no interaction between the month and gender
- H1: There is interaction between the month and gender

Summary: Differences Between One-Way and Two-Way ANOVA

The key differences between one-way and two-way ANOVA are summarized clearly below.

1. A one-way ANOVA is primarily designed to enable the equality testing between three or more means. A two-way ANOVA is designed to assess the interrelationship of two independent variables on a dependent variable.

2. A one-way ANOVA only involves one factor or independent variable, whereas there are two independent variables in a two-way ANOVA.

3. In a one-way ANOVA, the one factor or independent variable analyzed has three or more categorical groups. A two-way ANOVA instead compares multiple groups of two factors.

4. One-way ANOVA need to satisfy only two principles of design of experiments, i.e. replication and randomization. As opposed to Two-way ANOVA, which meets all three principles of design of experiments which are replication, randomization, and local control.

One-Way vs Two-Way ANOVA Differences Chart

	One-Way ANOVA	Two-Way ANOVA
Definition	A test that allows one to make comparisons between the means of three or more groups of data.	A test that allows one to make comparisons between the means of three or more groups of data, where two independent variables are considered.
Number of Independent Variables	One.	Two.
What is Being Compared?	The means of three or more groups of an independent variable on a dependent variable.	The effect of multiple groups of two independent variables on a dependent variable and on each other.
Number of Groups of Samples	Three or more.	Each variable should have multiple samples.

Comparison Chart

BASIS FOR COMPARISON	ONE WAY ANOVA	TWO WAY ANOVA
Meaning	One way ANOVA is a hypothesis test, used to test the equality of three of more population means simultaneously using variance.	Two way ANOVA is a statistical technique wherein, the interaction between factors, influencing variable can be studied.
Independent Variable	One	Two
Compares	Three or more levels of one factor.	Effect of multiple level of two factors.
Number of Observation	Need not to be same in each group.	Need to be equal in each group.
Design of experiments	Need to satisfy only two principles.	All three principles needs to be satisfied.

Analysis of variance (**ANOVA**) is a collection of statistical models and their associated estimation procedures (such as the "variation" among and between groups) used to analyze the differences among group means in a sample. ANOVA was developed by statistician and evolutionary biologist Ronald Fisher. In the ANOVA setting, the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether the population means of several groups are equal, and therefore generalizes the *t*-test to more than two groups. ANOVA is useful for comparing (testing) three or more group means for statistical significance. It is conceptually similar to multiple two-sample t-tests, but is more conservative, resulting in fewer type I errors, and is therefore suited to a wide range of practical problems.