Session - 3

Data Frame

Data Frame

- Probably the most important data type you’ll use.
  - All external data (from excel, csv, tables, webpages etc) is read as data frame
  - It’s a list where each element of list must have the same length.
  - Think of it like a matrix but with the flexibility that each column can have different data type. E.g. set of Names, weights and heights
  - Example:

    -
      ```
      d = data.frame(name = c("a", "b"), weight = c(70, 75),
                     height = c(1.78, 1.82));
      ```
    - `d; d$name; d[1,]; d$weight; d[,3];`
    - `d$bmi = d$weight / (d$height^2); # new row`
    - `nrow(d); ncol(d); dim(d);`
    - `colnames(d)[1] = "names";`
    - `rownames(d) = c("I", "II");`

Reading Data

Reading Data

- Download some stock data from NSE
  - https://www.nseindia.com/products/content/equities/indices/historical_index_data.htm
  - Save the CSV file as data.csv

Reading Data

- Download some stock data from NSE
  - https://www.nseindia.com/products/content/equities/indices/historical_index_data.htm
  - Save the CSV file as data.csv
- From CSV (most common)

  -
    ```
    setwd("C:/Users/nikhi/Downloads/");
     nifty = read.csv("data.csv");
    ```
  - Alternatively: `nifty = read.csv("C:/Users/nikhi/Downloads/data.csv");`

Reading Data

- Download some stock data from NSE
  - https://www.nseindia.com/products/content/equities/indices/historical_index_data.htm
  - Save the CSV file as data.csv
- From CSV (most common)

  -
    ```
    setwd("C:/Users/nikhi/Downloads/");
     nifty = read.csv("data.csv");
    ```
  - Alternatively: `nifty = read.csv("C:/Users/nikhi/Downloads/data.csv");`
- From Excel
  - Search it yourself! It is not recommended btw.

Reading Data

- Download some stock data from NSE
  - https://www.nseindia.com/products/content/equities/indices/historical_index_data.htm
  - Save the CSV file as data.csv
- From CSV (most common)

  -
    ```
    setwd("C:/Users/nikhi/Downloads/");
     nifty = read.csv("data.csv");
    ```
  - Alternatively: `nifty = read.csv("C:/Users/nikhi/Downloads/data.csv");`
- From Excel
  - Search it yourself! It is not recommended btw.
- From clipboard
  - `read.table("clipboard");`
  - This is quick fix for small data transfer between R and excel. Use `read.csv()` as your primary method for data reading!

Reading Data (cont.)

Reading Data (cont.)

- Viewing data
  - `View(nifty);`

Reading Data (cont.)

- Viewing data
  - View(nifty);
- Date

```
nifty$Date = as.Date(nifty$Date, format = "%d-%b-%Y");
n = nrow(nifty);
d = nifty$Date[1];
format(d, format = "%D");            #  10/01/20
format(d, format = "%d-%m-%y");      #  01-10-20
format(d, format = "%d.%b.%Y");      #  01.Oct.2020
format(d, format = "%A, %B %d, %Y") #  Thursday, October 01, 2020
```

Reading Data (cont.)

- Viewing data
  - `View(nifty);`
- Date

```
nifty$Date = as.Date(nifty$Date, format = "%d-%b-%Y");
n = nrow(nifty);
d = nifty$Date[1];
format(d, format = "%D");          #  10/01/20
format(d, format = "%d-%m-%y");    #  01-10-20
format(d, format = "%d.%b.%Y");    #  01.Oct.2020
format(d, format = "%A, %B %d, %Y") #  Thursday, October 01, 2020
```

- Alternatively,
  - `read.table("data.csv", header = T, sep = ",", nrows = 5);`

if-else

if-else

-
  ```
  if(<COND_1>) {
    # do something!
  }
  ```

-
  ```
  if(<COND_1>) {
    # do something!
  } else {
    # ...
  }
  ```

-
  ```
  if(<COND_1>) {
    # do something!
  } else if(<COND_2>) {
    # ...
  } else {
    # ...
  }
  ```

if-else

-
  ```
  if(<COND_1>) {
    # do something!
  }
  ```

-
  ```
  if(<COND_1>) {
    # do something!
  } else {
    # ...
  }
  ```

-
  ```
  if(<COND_1>) {
    # do something!
  } else if(<COND_2>) {
    # ...
  } else {
    # ...
  }
  ```

```
if(nifty$Close[2] > nifty$Close[1]) {
  str = paste("Stock market closed green on", nifty$Date[2]);
} else if(nifty$Close[2] > nifty$Open[2]) {
  str = paste("Stock market closed above opening on", nifty$Date[2]);
} else {
  str = paste("Stock market was red and closed below opening on", nifty$Date[2]);
}
print(str);
```

for loop

for loop

- Looping is used to perform similar set of tasks repetitively

  -
    ```
    for(i in n:1) {
      print(nifty$Date[i]);
    }
    ```
  - `n:1;` is same as `seq(n,1,1);` i.e. backwards counting!
  - Alternatively, you can execute: `rev(nifty$Date);` or `nifty$Date[n:1];`

for loop

- Looping is used to perform similar set of tasks repetitively

  -
    ```
    for(i in n:1) {
      print(nifty$Date[i]);
    }
    ```
  - `n:1;` is same as `seq(n,1,1);` i.e. backwards counting!
  - Alternatively, you can execute: `rev(nifty$Date);` or `nifty$Date[n:1];`

- Try avoiding loops if you can!
  - Increasing all dates by a week: `nifty$Date + 7`
  - Finding Daily growth: `nifty$Close[-1] / nifty$Close[-n]`
  - Daily diff. b/w high and low prices: `nifty$High - nifty$Low`
  - Question: find % growth in daily volatility
    - $G = \frac{(Value_{t+1} - Value_t)}{Value_t} * 100$

Nested if-else and for loop

Nested if-else and for loop

```
for(i in 2:n) {
  if(nifty$Close[i] > 1.01 * nifty$Close[i-1]) {
    # market gained more than 1%
    for(j in 1:ncol(nifty)) {
      print( paste("Gain", i, colnames(nifty)[j], nifty[i,j], sep =":") );
    } # end for(j)
  } else if(nifty$Close[i] < 0.99 * nifty$Close[i-1]) {
    # market lost more than 1%
    for(j in 1:ncol(nifty)) {
      print( paste("Loss", i, colnames(nifty)[j], nifty[i,j], sep =":") );
    }
  } else {
    print(paste("Market movement was within 1% for i =", i));
  } # end if()
} # end for(i)
```

Jumping

Jumping

- Till now all our commands executed sequentially
  - There may be circumstances when we need to jump

Jumping

- Till now all our commands executed sequentially
  - There may be circumstances when we need to jump
- Next and Break
  - next is used to skip an iteration, while break exits the loop entirely.
  -
    ```
    for(i in 1:10) {
      if(i <= 3) {
        next;
      }
      if(i > 6) {
        break;
      }
      print(i);
    }
    i;
    ```

Jumping

- Till now all our commands executed sequentially
  - There may be circumstances when we need to jump
- Next and Break
  - `next` is used to skip an iteration, while `break` exits the loop entirely.
  -
    ```
    for(i in 1:10) {
      if(i <= 3) {
        next;
      }
      if(i > 6) {
        break;
      }
      print(i);
    }
    i;
    ```
- `return()` is used to exit a function with a value

Function

Function

- Organize often-used set of instructions separately in a “function”
- Calling a function will execute all the commands in the body of function

Function

- Organize often-used set of instructions separately in a “function”
- Calling a function will execute all the commands in the body of function
- We have used many functions till now
  - They end with parenthesis: `()`
    - Not square or curly braces
  - E.g. `sum(); rbind(); vector(); format(); read.csv();` etc
  - Note that curly braces `{}` are used for if-else, for and function body, square braces `[]` for vector/matrix indexing and parenthesis `()` for grouping, if-else condition, for condition and functions arguments.

Function

- Organize often-used set of instructions separately in a “function”
- Calling a function will execute all the commands in the body of function
- We have used many functions till now
  - They end with parenthesis: `()`
    - Not square or curly braces
  - E.g. `sum(); rbind(); vector(); format(); read.csv();` etc
  - Note that curly braces `{}` are used for if-else, for and function body, square braces `[]` for vector/matrix indexing and parenthesis `()` for grouping, if-else condition, for condition and functions arguments.
- A function has
  - A name by which we call them, e.g. `sum`
  - A set of inputs to be put within parenthesis like numbers `1:10` in `sum()`
    - A function can have no input: `getwd()`
  - Return value which is the output of the function like the sum of numbers in `sum()`

Function Example

```
my_mean = function(x) {
  n = length(x);
  mean = sum(x) / n;
  return(mean);
}
```

Function Example

```
my_mean = function(x) {
  n = length(x);
  mean = sum(x) / n;
  return(mean);
}
```

- Name of the function is: `my_mean`
- Input is: `x`
- Output is: `mean`
  - Note that the mean here is just a name, we could well have used any other name without changing anything about our function

Function (Example) Cont.

- Alternate ways to write the same function

  -
    ```
    my_mean = function(x) {
      return( sum(x) / length(x) );
    }
    ```
    - No need to store sum and length. We can directly divide them!

  -
    ```
    my_mean = function(x) {
      sum(x) / length(x);
    }
    ```
    - No need for an explicit return. The last statement is returned by default.

Function (Example) Cont.

- Alternate ways to write the same function

  -
    ```
    my_mean = function(x) {
      return( sum(x) / length(x) );
    }
    ```
    - No need to store sum and length. We can directly divide them!

  -
    ```
    my_mean = function(x) {
      sum(x) / length(x);
    }
    ```
    - No need for an explicit return. The last statement is returned by default.
- Try various value with `my_mean()` and the inbuilt `mean()`. See that the answers are exactly the same.

Function (Example) Cont.

- Alternate ways to write the same function

  -
    ```
    my_mean = function(x) {
      return( sum(x) / length(x) );
    }
    ```
    - No need to store sum and length. We can directly divide them!

  -
    ```
    my_mean = function(x) {
      sum(x) / length(x);
    }
    ```
    - No need for an explicit return. The last statement is returned by default.
- Try various value with `my_mean()` and the inbuilt `mean()`. See that the answers are exactly the same.
- Exercise: Write your own version of variance function
  - $Var(x) = mean([x - mean(x)]^2)$
  - Compare it with the inbuilt `var()` function in R

Multiple conditions & which() function

Multiple conditions & which() function

- The arguments to `if()` and `which()` and the output of `is.xx()` family of functions is a logical object, i.e. either `TRUE` or `FALSE`.

Multiple conditions & which() function

- The arguments to `if()` and `which()` and the output of `is.xx()` family of functions is a logical object, i.e. either `TRUE` or `FALSE`.

- A valid combination of logical objects is also a logical object. E.g.
  - Logical AND: `TRUE & FALSE` is `FALSE`
  - Logical OR: `TRUE | FALSE` is `TRUE`
  - Logical NOT: `! FALSE` is `TRUE`

Multiple conditions & which() function

- The arguments to `if()` and `which()` and the output of `is.xx()` family of functions is a logical object, i.e. either `TRUE` or `FALSE`.

- A valid combination of logical objects is also a logical object. E.g.
  - Logical AND: `TRUE & FALSE` is `FALSE`
  - Logical OR: `TRUE | FALSE` is `TRUE`
  - Logical NOT: `! FALSE` is `TRUE`

- De Morgan’s Law
  - $!\,(A \,\&\, B) = (!\,A) \mid (!\,B)$

- The below two indexes are one and same (by De Morgan Law),
    - `day = as.numeric( substr(nifty$Date, 9, 10) );`
        - OR `day = as.numeric( format(df[,1], format = "%d") );`

```
idx_1 = which( nifty$Close > nifty$Open    & (day < 5)   );
idx_2 = which(!(nifty$Close <= nifty$Open  | (day >= 5)) );
```

- The below two indexes are one and same (by De Morgan Law),
  - `day = as.numeric( substr(nifty$Date, 9, 10) );`
    - OR `day = as.numeric( format(df[,1], format = "%d") );`

```
idx_1 = which( nifty$Close > nifty$Open     & (day < 5)    );
idx_2 = which(!(nifty$Close <= nifty$Open  | (day >= 5)) );
```

- `which()` gives the indexes matching the criterion. E.g. out of `101:200` which numbers are multiples of 2,3 and 5 ?
  - `count = 101:200;`
  - `which( count %% 2 == 0 & count %% 3 == 0 & count %% 5 == 0 );`
  - `count[count %% 2 == 0 & count %% 3 == 0 & count %% 5 == 0];`

- The below two indexes are one and same (by De Morgan Law),
  - `day = as.numeric( substr(nifty$Date, 9, 10) );`
    - OR `day = as.numeric( format(df[,1], format = "%d") );`

```
idx_1 = which( nifty$Close > nifty$Open     & (day < 5)    );
idx_2 = which(!(nifty$Close <= nifty$Open   | (day >= 5)) );
```

- `which()` gives the indexes matching the criterion. E.g. out of `101:200` which numbers are multiples of 2,3 and 5 ?
  - `count = 101:200;`
  - `which( count %% 2 == 0 & count %% 3 == 0 & count %% 5 == 0 );`
  - `count[count %% 2 == 0 & count %% 3 == 0 & count %% 5 == 0];`

- We can do multi-way match using `%in%`
  - `mult_17 = seq(17,300,17);`
  - `which(count %in% mult_17);`
  - `which(mult_17 %in% count);`
  - `which(!(count %in% mult_17));`
  - `which(!(mult_17 %in% count));`

Some Useful functions

Some Useful functions

- `unique(), duplicated()`

Some Useful functions

- `unique(), duplicated()`
- `list.files()`
    - Pattern matching using regex
    - All files starting from “s”: `"^s"`
    - All files starting with “b” or “d”: `"^(b|d)“`
    - All CSV files: `".*.csv"`

Some Useful functions

- `unique(), duplicated()`
- `list.files()`
  - Pattern matching using regex
  - All files starting from “s”: `"^s"`
  - All files starting with “b” or “d”: `"^(b|d)“`
  - All CSV files: `".*.csv"`
- `order()`
  - Sort data/dataframes
  - It gives the sequence of ordered indexes NOT the ordered numbers
  - Can do 2-way and 3-way sorts

Some Useful functions

- `unique(), duplicated()`
- `list.files()`
  - Pattern matching using regex
  - All files starting from “s”: `"^s"`
  - All files starting with “b” or “d”: `"^(b|d)“`
  - All CSV files: `".*.csv"`
- `order()`
  - Sort data/dataframes
  - It gives the sequence of ordered indexes NOT the ordered numbers
  - Can do 2-way and 3-way sorts
- `union(), intersect()`

Some Useful functions

- `unique(), duplicated()`
- `list.files()`
  - Pattern matching using regex
  - All files starting from “s”: `"^s"`
  - All files starting with “b” or “d”: `"^(b|d)“`
  - All CSV files: `".*.csv"`
- `order()`
  - Sort data/dataframes
  - It gives the sequence of ordered indexes NOT the ordered numbers
  - Can do 2-way and 3-way sorts
- `union(), intersect()`
- `cumsum(), cumprod()`
  - Can you write your own version of `cumprod()` using only `cumsum()` ?

Loop Functions

Loop Functions

- Writing loops in a single command. Can come very handy and compact. The function name ends with “apply”.
  - `lapply(), apply(), sapply(), tapply(), mapply()`

Loop Functions

- Writing loops in a single command. Can come very handy and compact. The function name ends with “apply”.
    - `lapply(), apply(), sapply(), tapply(), mapply()`
- These functions work on a list of inputs, not just one input!
    - A data.frame is also a list.
    - Loop functions (esp. `lapply`) will be heavily used later!

Loop Functions

- Writing loops in a single command. Can come very handy and compact. The function name ends with “apply”.
    - `lapply(), apply(), sapply(), tapply(), mapply()`
- These functions work on a list of inputs, not just one input!
    - A data.frame is also a list.
    - Loop functions (esp. `lapply`) will be heavily used later!
- E.g.
    - `X = list(a = 1:10, b = rnorm(100, 0, 1), c = runif(1e3, 9, 91));`
    - `lappy(X, mean); # returns a list`
    - `sapply(X, mean); # returns a vector`

Loop Functions

- Writing loops in a single command. Can come very handy and compact. The function name ends with "apply".
  - `lapply(), apply(), sapply(), tapply(), mapply()`
- These functions work on a list of inputs, not just one input!
  - A data.frame is also a list.
  - Loop functions (esp. `lapply`) will be heavily used later!
- E.g.
  - `X = list(a = 1:10, b = rnorm(100, 0, 1), c = runif(1e3, 9, 91));`
  - `lappy(X, mean); # returns a list`
  - `sapply(X, mean); # returns a vector`
- Let's say we want to find `cor(X,X^i)` for `i in 1:10` w/o writing a loop?
  - `X = rnorm(1000, 0, 1);`
  - `sapply(1:10, function(i) cor(X,X^i));`
    - Here we have used anonymous function, i.e. a function w/o a name.

- `apply()` is mostly used for applying functions on rows or cols of a matrix
  - Like taking means by rows
  - `M = matrix(1:50, nrow = 10, ncol = 5); # data filled by column (default)`
  - `apply(M, 1, mean); # mean of each row (1st dimension)`
  - `apply(M, 2, mean); # mean of each col (2nd dimension)`
  - You can do the above using a loop also, but `apply()` is more compact.
  - The faster version of above are also available:
    - `rowSums(x)` is equivalent to `apply(x, 1, sum)`
    - `colMeans(x)` is equivalent to `apply(x, 2, mean)`
  - `apply()` works with multi-dimensional (> 2) arrays as well

- `apply()` is mostly used for applying functions on rows or cols of a matrix
  - Like taking means by rows
  - `M = matrix(1:50, nrow = 10, ncol = 5); # data filled by column (default)`
  - `apply(M, 1, mean); # mean of each row (1st dimension)`
  - `apply(M, 2, mean); # mean of each col (2nd dimension)`
  - You can do the above using a loop also, but `apply()` is more compact.
  - The faster version of above are also available:
    - `rowSums(x)` is equivalent to `apply(x, 1, sum)`
    - `colMeans(x)` is equivalent to `apply(x, 2, mean)`
  - `apply()` works with multi-dimensional (> 2) arrays as well

- `mapply()` is a multi-variate version of lapply/sapply:
  - Let's say `set.seed(1); u = rnorm(1000, 0, 1); v = rnorm(1000, 0, 1);`
    `X = u + v; Y = u - v;`
  - Suppose we need to find $cor(X^p, Y^q)$ for different values of p and q
    - We can't do this with `lapply()` since it only accepts one argument for looping
  - `mapply(p = 1:5, q = 5:1, function(p,q) cor(X^p, Y^q))`