Unit-4: Data Filtering and cleaning

4.1 Subsetting and filtering data

Creating a subset from a larger data frame is called “filtering”.

Filtering data (i.e., subsetting data) is an important data-management process, as it allows us to:

- Select or remove a subset of cases from a data frame based on their scores on one or more variables;
- Select or remove a subset of variables from a data frame.

Both subset() and filter() functions are used for selecting specific rows of a data frame based on specified conditions.

Working with Columns (subsetting / Slicing columns)

Subsetting / Slicing in R Using [ ] Operator

Using the ‘[ ]’ operator, elements of vectors and observations from data frames can be accessed. To neglect some indexes, ‘-‘ is used to access all other indexes of vector or data frame.

Example 1:
In this example, let us create a vector and perform subsetting using the [ ] operator.

```
# Create vector
x <- 1:15
# Print vector
cat("Original vector: ", x, "\n")
# Subsetting vector
cat("First 5 values of vector: ", x[1:5], "\n")
cat("Without values present at index 1, 2 and 3: ",x[-c(1, 2, 3)], "\n")
```

Output:
Original vector: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
First 5 values of vector: 1 2 3 4 5
Without values present at index 1, 2 and 3: 4 5 6 7 8 9 10 11 12 13 14 15

Example 2

let us use mtcars data frame present in R base package for subsetting.

- mpg: Miles/(US) gallon
- cyl: Number of cylinders
- disp: Displacement (cu.in.)
- hp: Gross horsepower
- drat: Rear axle ratio
- wt: Weight (1000 lbs)
- qsec: 1/4 mile time

- vs: Engine (0 = V-shaped, 1 = straight)
- am: Transmission (0 = automatic, 1 = manual)
- gear: Number of forward gears
- carb: Number of carburetors

```
# Dataset
cat("Original dataset: \n")
print(mtcars)
# Subsetting data frame
cat("HP values of all cars:\n")
print(mtcars['hp'])
# First 10 cars
cat("Without mpg and cyl column:\n")
print(mtcars[1:10, -c(1, 2)])
```

Code	Result
DF[1:3]	All Rows 3 Columns
DF[1:3,]	3 Rows All Columns
DF[,1:3]	All Rows 3 Columns
DF[1:3,1:3]	3 Rows 3 Columns
DF[1,1:3]	1st Row 3 Columns
DF[1:3,1]	3 Rows 1st Columns
DF[1,1]	1st row 1st Column

Output:

Original Dataset:

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
2	Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
6	Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
7	Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4

HP values of all cars:

	hp
Mazda RX4	110
Mazda RX4 Wag	110
Datsun 710	93
Hornet 4 Drive	110
Hornet Sportabout	175
Valiant	105
Duster 360	245
Merc 240D	62
Merc 230	95

Merc 280 123

Without mpg and cyl column:

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

Subsetting / Slicing in R Using [[ ]] Operator

[[ ]] operator is used for subsetting of list-objects. This operator is the same as [ ] operator but the only difference is that [[ ]] selects only one element whereas [ ] operator can select more than 1 element in a single command.

Example 1: In this example, let us create a list and select the elements using [[]] operator.

```
# Create list
ls <- list(a = 1, b = 2, c = 10, d = 20)
# Print list
cat("Original List: \n")
print(ls)
# Select first element of list
cat("First element of list: ", ls[[1]], "\n")
```

Output:

```
Original List:
$a
[1] 1
$b
[1] 2
$c
[1] 10
$d
[1] 20
First element of list:  1
```

Example

let us create a list and recursively select elements using c() function.

```
# Create list
z <- list(a = list(x = 1, y = "GFG"), b = 1:10)
# Print list
cat("Original list:\n")
```

```
print(z)
# Print GFG using c() function
cat("Using c() function:\n")
print(z[[c(1, 2)]])
# Print GFG using only [[]] operator
cat("Using [[]] operator:\n")
print(z[[1]][[2]])
```

Output

```
Original list:
$a
$a$x
[1] 1
$a$y
[1] "GFG"
$b
 [1]  1  2  3  4  5  6  7  8  9 10
Using c() function:
[1] "GFG"
Using [[]] operator:
[1] "GFG"
```

Subsetting / Slicing in R Using $ Operator

$ operator can be used for lists and data frames in R. Unlike [ ] operator, it selects only a single observation at a time. It can be used to access an element in named list or a column in data frame. $ operator is only applicable for recursive objects or list-like objects.

Example 1: In this example, let us create a named list and access the elements using $ operator

```
# Create list
ls <- list(a = 1, b = 2, c = "Hello", d = "GFG")
# Print list
cat("Original list:\n")
print(ls)
# Print "GFG" using $ operator
cat("Using $ operator:\n")
print(ls$d)
```

Output:

```
Original list:
$a
[1] 1
$b
[1] 2
$c
[1] "Hello"
$d
[1] "GFG"
Using $ operator:
[1] "GFG"
```

Example

let us use the mtcarsdataframe and select a particular column using $ operator.

```
# Access hp column
```

```
cat("Using $ operator:\n")
print(mtcars$hp)
```

Output

```
Using $ operator:
[1] 110 110  93 110 175 105 245  62  95 123
```

Working with Rows (Filtering data / rows)

The filter() function is part of the dplyr package and is also used for subsetting / filtering data frames based on specified conditions.

Syntax: filter(.data, ..., .preserve = FALSE)

Where:

.data: The data frame to be filtered.

….: The conditions for selecting rows.

.preserve: If TRUE, the original row order will be preserved.

Comparison Operator	Rows Returned from Original Data Frame
var==value	All rows where var IS equal to value
var!=value	All rows where var is NOT equal to value
var %in% c(value1,value2)	All rows where var IS IN vector of values
var>value	All rows where var is greater than value
var>=value	All rows where var is greater than or equal to value
var<value	All rows where var is less than value
var<=value	All rows where var is less than or equal to value
condition1,condition2	All rows where BOTH conditions are true
condition1 | condition2	All rows where ONE or BOTH conditions are true

What is piping?

The pipe operator, represented by %>%, is a powerful tool in the R programming language that allows for seamless data manipulation and transformation. It simplifies the code by enabling the user to chain several operations together in a concise and readable manner. The pipe operator works by taking the output of one operation and using it as the input for the next operation, allowing for a streamlined workflow. This is especially useful when working with large datasets or performing complex calculations, as it eliminates the need for intermediate variables and reduces the likelihood of errors. The pipe operator enhances code readability and improves code efficiency, making it an essential tool for any R programmer. With its ability to organize code logically and neatly, the pipe operator is a valuable addition to the arsenal of techniques used in data analysis, modeling, and visualization in R.

Before using the filter() function, you must install and load the dplyr package:

```
# Install dplyr package if not already installed

if (!requireNamespace("dplyr", quietly = TRUE)) {

install.packages("dplyr")

}
```

```
 # Load dplyr package

library(dplyr)
```

Example

```
# Load the mtcars dataset

data(mtcars)

 # Filter the data to include only cars with 6 cylinders and horsepower greater than 110

selected_data<- mtcars %>% filter(cyl == 6 & hp > 110)
```

Note:

- Tibbles are the core data structure of the tidyverse and are used to facilitate the display and analysis of information in a tidy format. Tibbles is a new form of Data Frames where data frames are the most common data structures used to store data sets in R.
- Reading with builtin read.csv() function will output data frames, while reading with read_csv() in "readr" package inside tidyverse will output tibbles.

Example

```
# Access readr package

library(readr)

# Read data and name data frame (tibble) objects

mergeddf<- read_csv("PersPerfData.csv")

print(mergeddf)
```

	id	lastname	firstname	startdate	gender	perf_q1	perf_q2	perf_q3	perf_q4
	<dbl>	<chr>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	153	Sanchez	Alejandro	1/1/2016	male	3.9	4.8	4.9	5
2	154	McDonald	Ronald	1/9/2016	male	NA	NA	NA	NA
3	155	Smith	John	1/9/2016	male	NA	NA	NA	NA
4	165	Doe	Jane	1/4/2016	female	NA	NA	NA	NA
5	125	Franklin	Benjamin	1/5/2016	male	2.1	1.9	2.1	2.3
6	111	Newton	Isaac	1/9/2016	male	3.3	3.3	3.4	3.3
7	198	Morales	Linda	1/7/2016	female	4.9	4.5	4.4	4.8
8	201	Providence	Cindy	1/9/2016	female	1.2	1.1	1	1
9	282	Legend	John	1/9/2016	male	2.2	2.3	2.4	2.5

With Pipes

```
# Filter in by gender with pipe
filterdf<- mergeddf %>% filter(gender=="female")
# Print filtered data frame
filterdf

# Filter out by gender with pipe
filterdf<- mergeddf %>% filter(gender!="female")
filterdf

# Filter by perf_q2 with pipe
filterdf<- mergeddf %>% filter(perf_q2>4.0)
filterdf
```

```
# Filter by gender or perf_q2 with pipe
filterdf<- mergeddf %>% filter(gender=="female" | perf_q2>4.0)
filterdf

# Filter by gender and perf_q2 with pipe
filterdf<- mergeddf %>% filter(gender=="female" & perf_q2>4.0)
filterdf

# Filter by two values of firstname with pipe
filterdf<- mergeddf %>% filter(firstname=="John" | firstname=="Jane")
filterdf

# Filter by two ranges of values of perf_q1 with pipe
filterdf<- mergeddf %>% filter(perf_q1<=2.5 | perf_q1>=4.0)
filterdf
```

The filter function can also be used to remove multiple specific cases (such as from a unique identifier variable), which might be useful when you've identified outliers that need to be removed.

```
# Filter out id of 198 and 201 with pipe
filterdf<- mergeddf %>% filter(!id %in% c(198,201))
filterdf

# Filter in id of 198 and 201 with pipe
filterdf<- mergeddf %>% filter(id %in% c(198,201))
filterdf

# Filter out id of 198 with pipe
filterdf<- mergeddf %>% filter(id!=198)
filterdf
```

When working with variables of type Date, things can get a bit trickier. When we applied the str function from base R, we found that the startdate variable was read in and joined as a character variable as opposed to a date variable. As such, we need to convert the startdate variable using the as.Date function from base R. First, type the name of the data frame object (mergeddf), followed by the $ operator and the name of whatever you want to call the new variable (startdate2); remember, the $ operator tells R that a variable belongs to (or will belong to) a particular data frame. Second, type the <- operator. Third, type the name of the as.Date function. Fourth, in the function parentheses, as the first argument, enter the as.character function with the name of the data frame object (mergeddf), followed by the $ operator and the name the original variable (startdate) as the sole argument. Fifth, as the second argument in the as.Date function, type format="%m/%d/%Y" to indicate the format for the data variable; note that the capital Y in %Y implies a 4-digit year, whereas a lower case would imply a 2-digit year.

```
# Convert character startdate variable to the Date type startdate2 variable

mergeddf$startdate2 <- as.Date(as.character(mergeddf$startdate),
format="%m/%d/%Y")
```

To verify that the new startdate2 variable is of type date, use the str function from base R, and enter the name of the data frame object (mergeddf) as the sole argument. As you will see, the new startdate2 variable is now of type Date.

```
# Verify that the startdate2 variable is now a variable of type Date
str(mergeddf)
# Filter by startdate2 with pipe
```

```
filterdf<- mergeddf %>% filter(startdate2 >as.Date("2016-01-07"))
filterdf

library(stringr)
filterdf<- mergeddf %>% filter(str_detect(firstname,'Jo'))
filterdf

filterdf<- mergeddf %>% filter(is.na(firstname))
filterdf

filterdf<- mergeddf %>% filter(!is.na(firstname))
filterdf
```

complete.cases() helper

Similar to is.na(), we can check for the presence of NA values across all columns of a dataframe using complete.cases(). This function is not part of the tidyverse package, so it requires a period . within the brackets, to indicate that we want to search across the entire dataframe.

To filter for only the rows with no missing values:

```
filterdf<- mergeddf %>% filter(complete.cases(.))

filterdf

filterdf<- mergeddf %>% filter(!complete.cases(.))

filterdf
```

Without Pipes

```
# Filter in by gender without pipe
filterdf<- filter(mergeddf, gender=="female")
filterdf

# Filter in by gender without pipe
filterdf<- filter(mergeddf, gender!="female")
filterdf

# Filter by perf_q2 without pipe
filterdf<- filter(mergeddf, perf_q2>4.0)
filterdf

# Filter by gender or perf_q2 without pipe
filterdf<- filter(mergeddf, gender=="female" | perf_q2>4.0)
filterdf

# Filter by gender and perf_q2 without pipe
filterdf<- filter(mergeddf, gender=="female" & perf_q2>4.0)
filterdf

# Filter by two values of firstname without pipe
filterdf<- filter(mergeddf, firstname=="John" | firstname=="Jane")
filterdf

# Filter by two ranges of values of perf_q1 without pipe
filterdf<- filter(mergeddf, perf_q1<=2.5 | perf_q1>=4.0)
filterdf

# Filter out id of 198 and 201 without pipe
filterdf<- filter(mergeddf, !id %in% c(198,201))
filterdf
```

```
# Filter in id of 198 and 201 without pipe
filterdf<- filter(mergeddf, id %in% c(198,201))
filterdf

# Filter in id of 198 without pipe
filterdf<- filter(mergeddf, id!=198)

# Filter in id of 201 without pipe
filterdf<- filter(filterdf, id!=201)
filterdf


# Convert character startdate variable to the date type startdate2
variable
mergeddf$startdate2   <-   as.Date(as.character(mergeddf$startdate),
format="%m/%d/%Y")

# Verify that the startdate2 variable is now a variable of type date
str(mergeddf)

# Filter by startdate2 without pipe
filterdf<- filter(mergeddf, startdate2 >as.Date("2016-01-07"))
filterdf
```

Example

Dataframe in use:

```
# load the package
library(dplyr)
# create the dataframe with three columns
# id , department and salary with 8 rows
data=data.frame(id=c(7058,7059,7060,7089,7072,7078,7093,7034),
           department=c('IT','sales','finance','IT','finance',
                      'sales','HR','HR'),
           salary=c(34500.00,560890.78,67000.78,25000.00,
                      78900.00,25000.00,45000.00,90000))
#display actual dataframe
print(data)
print("==========================")
#filter dataframe with department is sales
print(filter(data,department=="sales"))

# filter dataframe with department is sales and salary is greater
than 27000
print(filter(data,department=="sales" & salary >27000))

# filter dataframe with department is IT or salary is greater than
27000
print(filter(data,department=="IT" |  salary >27000))

# filter dataframe with department sales and salary greater than
27000 or salaryless than 5000
print(filter(data,department=="sales"    &    salary    >27000    |
salary<5000))

# display top 3 values with slice_head (similar to head() of base R)
data %>% slice_head(n=3)
```

```
# display top 5 values with slice_head
data %>% slice_head(n=5)

  # display top 1 value  withslice_head
data %>% slice_head(n=1)

# display last 3 values with slice_tail (similar to tail() of base
R)
data %>% slice_tail(n=3)

# display last 5 values with slice_tail
data %>% slice_tail(n=5)

# display last 1 value  withslice_tail
data %>% slice_tail(n=1)

# display last 3 values with top_n
data %>% top_n(n=3)  # it will select the column / variable on which
it will order

# display last 5 values with top_n
data %>% top_n(n=5, Salary)

# display last 1 value  withtop_n
top_n(data, n=1, id)
top_n(Data_Frame1,n=-2,Salary) # last 2

# display last 3 values with slice_sample (it will randomly select
the rows)
data %>% slice_sample(n=3)

# display last 5 values with slice_sample
data %>% slice_sample(n=5)

# display last 1 value  withslice_sample
data %>% slice_sample(n=1)

# return top 3 maximum rows based on salary    column in the
dataframe
print(data %>% slice_max(salary, n = 3))

# return top 5 maximum rows based on department column in the
dataframe
print(data %>% slice_max(department, n = 5))

# return top 3 minimum rows based on salary column in the dataframe
print(data %>% slice_min(salary, n = 3))

# return top 5 minimum rows based on department column in the
dataframe
print(data %>% slice_min(department, n = 5))

# return 25 % of rows
print(sample_frac(data,0.25))
```

```
# return 50 % of rows
print(sample_frac(data,0.50)

# return 100 % of rows
print(sample_frac(data,1))
```

Remove Duplicate rows in R using Dplyr

distinct()

This function is used to remove the duplicate rows in the dataframe and get the unique data

Syntax:

```
distinct(dataframe)
```

We can also remove duplicate rows based on the multiple columns/variables in the dataframe

Syntax:

```
distinct(dataframe,column1,column2,.,column n)
```

Example 1: R program to remove duplicate rows from the dataframe

```
# load the package
library(dplyr)
# createdataframe with three columns named id,name and address
data1=data.frame(id=c(1,2,3,4,5,6,7,1,4,2),
          name=c('sravan','ojaswi','bobby',
'gnanesh','rohith','pinkey',
               'dhanush','sravan','gnanesh', 'ojaswi'),
          address=c('hyd','hyd','ponnur','tenali',
          'vijayawada','vijayawada','guntur',
'hyd','tenali','hyd'))
data1
# remove duplicate rows
print(distinct(data1))
```

```
   id    name    address
1   1  sravan        hyd
2   2  ojaswi        hyd
3   3   bobby     ponnur
4   4 gnanesh     tenali
5   5  rohith vijayawada
6   6  pinkey vijayawada
7   7 dhanush     guntur
8   1  sravan        hyd
9   4 gnanesh     tenali
10  2  ojaswi        hyd
```

```
> distinct(data1)
  id    name    address
1  1  sravan        hyd
2  2  ojaswi        hyd
3  3   bobby     ponnur
4  4 gnanesh     tenali
5  5  rohith vijayawada
6  6  pinkey vijayawada
7  7 dhanush     guntur
```

Example 2: Remove duplicate rows based on single column

```
# remove duplicate rows based on name column

print(distinct(data1,name))
```

```
> distinct(data1,name)
     name
1  sravan
2  ojaswi
3   bobby
4 gnanesh
5  rohith
6  pinkey
7 dhanush
```

Example 3: Remove duplicate rows based on multiple columns

```
# remove duplicate rows based on name and address columns

print(distinct(data1,address,name))
```

```
> distinct(data1,address,name)
     name    address
1  sravan        hyd
2  ojaswi        hyd
3   bobby     ponnur
4 gnanesh     tenali
5  rohith vijayawada
6  pinkey vijayawada
7 dhanush     guntur
```

using duplicated() function

duplicated() function will return the duplicated rows and !duplicated() function will return the unique rows.

Syntax: `dataframe[!duplicated(dataframe$column_name), ]`

Here, dataframe is the input dataframe and column_name is the column in dataframe, based on that column the duplicate data is removed.

Example: R program to remove duplicate data based on particular column

```
# load the package
library(dplyr)
# create dataframe with three columns named id,name and address
data1=data.frame(id=c(1,2,3,4,5,6,7,1,4,2),
            name=c('sravan','ojaswi','bobby', 'gnanesh','rohith','pinkey',
                   'dhanush','sravan','gnanesh', 'ojaswi'),
            address=c('hyd','hyd','ponnur','tenali',
            'vijayawada','vijayawada','guntur', 'hyd','tenali','hyd'))
data1
```

```
   id    name    address
1   1  sravan        hyd
2   2  ojaswi        hyd
3   3   bobby     ponnur
4   4 gnanesh     tenali
5   5  rohith vijayawada
6   6  pinkey vijayawada
7   7 dhanush     guntur
8   1  sravan        hyd
9   4 gnanesh     tenali
10  2  ojaswi        hyd
```

```
# remove duplicate rows using duplicated() function based on name column
print(data1[!duplicated(data1$name), ] )
print("======================")
# remove duplicate rows using duplicated() function based on id column
print(data1[!duplicated(data1$id), ] )
print("======================")
# remove duplicate rows using duplicated() function based on address column
print(data1[!duplicated(data1$address), ] )
print("======================")
```

```
  id    name    address
1  1  sravan        hyd
2  2  ojaswi        hyd
3  3   bobby     ponnur
4  4 gnanesh     tenali
5  5  rohith vijayawada
6  6  pinkey vijayawada
7  7 dhanush     guntur
[1] "======================"
  id    name    address
1  1  sravan        hyd
2  2  ojaswi        hyd
3  3   bobby     ponnur
4  4 gnanesh     tenali
5  5  rohith vijayawada
6  6  pinkey vijayawada
7  7 dhanush     guntur
[1] "======================"
  id    name    address
1  1  sravan        hyd
3  3   bobby     ponnur
4  4 gnanesh     tenali
5  5  rohith vijayawada
7  7 dhanush     guntur
[1] "======================"
```

Using unique() function

unique() function is used to remove duplicate rows by returning the unique data

Syntax:`unique(dataframe)`

To get unique data from column pass the name of the column along with the name of the dataframe,

Syntax:`unique(dataframe$column_name)`

Where, dataframe is the input dataframe and column_name is the column in the dataframe.

Example 1: R program to remove duplicates using unique() function

```
# load the package
```

```
library(dplyr)
# create dataframe with three columns named id, name and address
data1=data.frame(id=c(1,2,3,4,5,6,7,1,4,2),
            name=c('sravan','ojaswi','bobby', 'gnanesh','rohith','pinkey',
                   'dhanush','sravan','gnanesh', 'ojaswi'),
            address=c('hyd','hyd','ponnur','tenali',
            'vijayawada','vijayawada','guntur', 'hyd','tenali','hyd'))
data1
# get unique data from the dataframe
print(unique(data1))
```

```
   id    name    address
1   1  sravan        hyd
2   2  ojaswi        hyd
3   3   bobby     ponnur
4   4 gnanesh     tenali
5   5  rohith vijayawada
6   6  pinkey vijayawada
7   7 dhanush     guntur
8   1  sravan        hyd
9   4 gnanesh     tenali
10  2  ojaswi        hyd
```

```
  id    name    address
1  1  sravan        hyd
2  2  ojaswi        hyd
3  3   bobby     ponnur
4  4 gnanesh     tenali
5  5  rohith vijayawada
6  6  pinkey vijayawada
7  7 dhanush     guntur
```

Example 2: R program to remove duplicate in particular column

```
# get unique data from the dataframe in id column

print(unique(data1$id))

# get unique data from the dataframe in name column

print(unique(data1$name))

# get unique data from the dataframe in address column

print(unique(data1$address))
```

```
[1] 1 2 3 4 5 6 7
[1] "sravan"  "ojaswi"  "bobby"   "gnanesh" "rohith"  "pinkey"  "dhanush"
[1] "hyd"       "ponnur"    "tenali"    "vijayawada" "guntur"
```

Working with rows and columns both (Subsetting)

subset() function in R programming is used to create a subset of vectors, matrices, or data frames based on the conditions provided in the parameters.

Syntax: `subset(x, subset, select, drop = FALSE)`

Where:

x: The data frame to be subsetted.

subset: The conditions for selecting rows.

select: The columns to be selected.

drop: If TRUE, the result will be coerced to the lowest possible dimension.

Example

First, let's load the mtcars dataset:

```
data(mtcars)
```

Then, let's subset the data to include only cars with 6 cylinders and horsepower greater than 110:

```
selected_data<- subset(mtcars, cyl == 6 & hp > 110)
```

The resulting 'selected_data' data frame will contain only the rows from the 'mtcars' dataset where the 'cyl' column value is 6 and the 'hp' column value is greater than 110.

Example: In this example, let us use mtcars data frame present in R base package and select gear where hp< 65.

```
selected_data<- subset(mtcars, hp>123, select = c(gear))

print(selected_data)
```

Output

```
gear

3

3
```

Example: In this example, let us use mtcars data frame present in R base package and selects the car with 5 gears and hp > 200.

```
mtc<- subset(mtcars, gear == 5 & hp > 200, select = c(gear, hp))
print(mtc)
```

Example : Basic example of R – subset() Function

```
# Creating a Data Frame
df<-data.frame(col1 = 0:2, col2 = 3:5, col3 = 6:8)
print ("Original Data Frame")
print (df)
```

Output:

```
[1] "Original Data Frame"
  col1 col2 col3
1   0   3   6
2   1   4   7
3   2   5   8
```

```
# Creating a Subset
df1<-subset(df, select = col2)
print("Modified Data Frame")
print(df1)
```

Output:

```
[1] "Modified Data Frame"
  col2
1   3
2   4
3   5
```

Here, in the above code, the original data frame remains intact while another subset of data frame is created which holds a selected row from the original data frame.

Example: Create Subsets of Data frame in R Language

```
df<-subset(df, select = -c(col2, col3))
print("Modified Data Frame")
print(df)
```

Output:

```
[1] "Modified Data Frame"
 col1
1   0
2   1
3   2
```

Here, in the above code, the rows are permanently deleted from the original data frame.

Example: Logical AND and OR using subset

```
# Create a data frame
df<- data.frame(
ID = 1:5,
Name = c("Nishant", "Vipul", "Jayesh", "Abhishek", "Shivang"),
Age = c(25, 30, 22, 35, 28)
)
Df
```

Output

	ID	Name	Age
1	1	Nishant	25
2	2	Vipul	30
3	3	Jayesh	22
4	4	Abhishek	35
5	5	Shivang	28

```
# Subset based on age greater than 25 and ID less than 4
subset_df<- subset(df, subset = Age > 25 & ID < 4)
print(subset_df)
```

Output:

```
     ID   Name           Age
2    2    Vipul          30
```

```
# Subset based on age greater than 30 or ID equal to 2
subset_df2 <- subset(df, subset = Age > 30 | ID == 2)
print(subset_df2)
```

Output

```
     ID   Name       Age
2    2    Vipul      30
4    4    Abhishek   35
```

Example: Subsetting with Missing Values

```
# Create a data frame
df<- data.frame(
ID = 1:5,
Name = c("Nishant", "Vipul", NA, "Abhishek", NA),
Age = c(25, 30, NA, 35, NA)
)
```

```
df
```

Output

```
   ID      Name       Age
1  1       Nishant    25
2  2       Vipul      30
3  3       <NA>  NA
4  4       Abhishek   35
5  5       <NA>       NA
```

```
subset_df<- subset(df, subset = !is.na(Age))
print(subset_df)
```

Output:

```
  ID      Name       Age
1  1      Nishant    25
2  2      Vipul      30
4  4      Abhishek   35
```

In summary, both the subset() and filter() functions are used for subsetting data frames based on specified conditions. The main differences between them are:

- Package: subset() is part of base R, while filter() is part of the dplyr package.
- Syntax: The syntax for the filter() function is more concise and is often used in combination with other dplyr functions through the use of the pipe operator (%>%).
- Row order preservation: The filter() function provides an option to preserve the original row order through the .preserve argument, while subset() does not have a similar option.

Overall, while the subset() function is more accessible since it is part of base R, the filter() function is more flexible and can be easily combined with other dplyr functions, making it the preferred choice for many R users.

4.2 Removing, Adding, & Changing Variable Names

After reading data into R as a data frame object, you may encounter situations in which it makes sense to remove the variable names (and not the data associated with the variable names), to add or replace variable names, or to just rename (change) certain variables. For example, perhaps the variable names from the original data file don't adhere to your preferred naming conventions, and thus you wish to change the variable names. As another example, sometimes the variable names are divorced from the associated data, and thus as an initial data management step, we need to add the variable names to the associated data in R.

Function	Package
`names`	base R
`c`	base R
`head`	base R
`rename`	`dplyr`

Example

```
install.packages("readr")
```

```
library(readr)
```

```
personaldata<- read_csv("PersData.csv")
```

```
# Print the names of the variables in the data frame (tibble) object

names(personaldata)
```

Output

```
[1] "id"        "lastname"  "firstname" "startdate" "gender"

print(personaldata)
```

Output

	id	lastname	firstname	startdate	gender
	<dbl>	<chr>	<chr>	<chr>	<chr>
1	153	Sanchez	Alejandro	1/1/2016	male
2	154	McDonald	Ronald	1/9/2016	male
3	155	Smith	John	1/9/2016	male
4	165	Doe	Jane	1/4/2016	female
5	125	Franklin	Benjamin	1/5/2016	male
6	111	Newton	Isaac	1/9/2016	male
7	198	Morales	Linda	1/7/2016	female
8	201	Providence	Cindy	1/9/2016	female
9	282	Legend	John	1/9/2016	male

Remove Variable Names from a Data Frame Object

Note:

- Tibbles are the core data structure of the tidyverse and are used to facilitate the display and analysis of information in a tidy format. Tibbles is a new form of Data Frames where data frames are the most common data structures used to store data sets in R.
- Reading with builtin read.csv() function will output data frames, while reading with read_csv() in “readr” package inside tidyverse will output tibbles.

To remove variable names, just apply the names function with the data frame name as the argument, and then use either the <- operator with NULL to remove the variable names.

Example

```
# Remove variable names

names(personaldata) <- NULL

# Print just the first 6 rows of the data frame object in Console

head(personaldata)
```

Output

```
Error in names[old] <- names(x)[j[old]] : replacement has length zero
```

If you get the above error message, then you need need to convert your object from a tibble to a data frame object prior to removing the variable names. When we use the read_csv function from the readr package to read in data, we technically read in the data as a tibble as opposed to a standard data frame object.

To convert the object to a data frame object, we can use the as.data.frame object from base R as follows and then re-try the previous step.

Example

```
library(readr)
personaldata<- read_csv("PersData.csv")
# Print the names of the variables in the data frame (tibble) object
names(personaldata)
```
Output
```
[1] "id"        "lastname"  "firstname" "startdate" "gender"
print(personaldata)
```
Output

	id	lastname	firstname	startdate	gender
	<dbl>	<chr>	<chr>	<chr>	<chr>
1	153	Sanchez	Alejandro	1/1/2016	male
2	154	McDonald	Ronald	1/9/2016	male
3	155	Smith	John	1/9/2016	male
4	165	Doe	Jane	1/4/2016	female
5	125	Franklin	Benjamin	1/5/2016	male
6	111	Newton	Isaac	1/9/2016	male
7	198	Morales	Linda	1/7/2016	female
8	201	Providence	Cindy	1/9/2016	female
9	282	Legend	John	1/9/2016	male

```
# If error message appears, convert object to data frame

personaldata<- as.data.frame(personaldata)

# Remove variable names

names(personaldata) <- NULL

# Print just the first 6 rows of the data frame object in Console

head(personaldata)
```

1	153	Sanchez	Alejandro	1/1/2016	male
2	154	McDonald	Ronald	1/9/2016	male
3	155	Smith	John	1/9/2016	male
4	165	Doe	Jane	1/4/2016	female
5	125	Franklin	Benjamin	1/5/2016	male
6	111	Newton	Isaac	1/9/2016	male

Add Variable Names to a Data Frame Object

In other instances, you might find yourself with a dataset that lacks variable names (or has variable names that need to be replaced), which means that you will need to add those variable names to the data frame.

To add variable names, we can use the names function from base R, and enter the name of the data frame as the argument. Using the <- operator, we can specify the variable names using the c (combine) function that contains a vector of variable names in quotation marks (" ") as the arguments. Remember to type a comma (,) between the function arguments, as commas are used to separate arguments from one another when there are more than one. Please note that the it's important that the vector of variable names contains the same number of names as the data frame object has columns.

Example

```
# Add (or replace) variable names to data frame object

names(personaldata) <- c("id", "lastname", "firstname", "startdate",
"gender")

# Print just the first 6 rows of data in Console

head(personaldata)
```

```
   id lastname firstname startdate gender
1 153  Sanchez Alejandro  1/1/2016   male
2 154 McDonald    Ronald  1/9/2016   male
3 155    Smith      John  1/9/2016   male
4 165      Doe      Jane  1/4/2016 female
5 125 Franklin  Benjamin  1/5/2016   male
6 111   Newton     Isaac  1/9/2016   male
```

Change Specific Variable Names in a Data Frame Object

We'll begin by specifying the name of our data frame object personaldata, followed by the <- operator so that we can overwrite the existing personaldata frame object with one that contains the renamed variables. Next, type the name of the rename function. As the first argument in the function, type the name of the data frame object (personaldata). As the second argument, let's change the lastname variable to Last_Name by typing the name of our new variable followed by = and, in quotation marks (" "), the name of the original variable (Last_Name="lastname"). As the third argument, let's apply the same process as the second argument and change the firstname variable to First_Name by typing the name of our new variable followed by = and, in quotation marks (" "), the name of the original variable (First_Name="firstname").

Example

```
# Add (or replace) variable names to data frame object

personaldata<- rename(personaldata,

Last_Name="lastname",

First_Name="firstname")
```

Using the head function from base R, let's verify that we renamed the two variables successfully.

```
# View just the first 6 rows of data in Console

head(personaldata)
```

```
   id Last_Name First_Name startdate gender
1 153   Sanchez  Alejandro  1/1/2016   male
2 154  McDonald     Ronald  1/9/2016   male
3 155     Smith       John  1/9/2016   male
4 165       Doe       Jane  1/4/2016 female
5 125  Franklin   Benjamin  1/5/2016   male
6 111    Newton      Isaac  1/9/2016   male
```

Example

```
# Change startdate to Start_Date

names(personaldata)[names(personaldata) == "startdate"] <- "Start_Date"

# View the new names

names(personaldata)
```

Output:

```
 [1] "id"         "Last_Name"    "First_Name"   "Start_Date"     "gender"
```

To rename all the variables, assign a vector of names (which means that for variable names you do not wish to rename, you must specify the existing name).

Example

```
personaldata<- personaldata %>% rename(id = ID)
```

Output:

```
 [1] "ID"  "Last_Name"    "First_Name"   "Start_Date"     "gender"
```

rename() overwrites an existing variable. If you used mutate() instead of rename() above, then you would create a new, additional, variable that is a copy of the old one and the old one would remain, as well.

change/rename specific attributes within a data frame column

```
cols <- c("green", "green", "red", "blue", "black", "blue")
num<- c(1, 1, 2, 3, 4, 3)
df<- data.frame(cols, num)
df$cols<- as.character(df$cols)
>df
```

Output

```
   cols num
1 green   1
2 green   1
3   red   2
4  blue   3
5 black   4
6  blue   3
```

Let's say there was a mistake in my data and that all of my "green" attributes need to be "purple."

```
df[df$cols == "green","cols" ] <- "purple"
>df
```

Output

```
     cols      num
1    purple    1
2    purple    1
3    red       2
4    blue      3
5    black     4
6    blue      3
```

4.3 Data Cleaning and transformation

For data analyses to be valid, accurate and transparent, the data themselves need to be correct.

Data cleaning is the process that removes data that does not belong in your dataset. Data transformation is the process of converting data from one format or structure into another.

Data Cleaning

Data cleaning is the process of fixing or removing incorrect, corrupted, incorrectly formatted, duplicate, or incomplete data within a dataset. When combining multiple data sources, there are many opportunities for data to be duplicated or mislabelled. If data is incorrect, outcomes and algorithms are

unreliable, even though they may look correct. There is no one absolute way to prescribe the exact steps in the data cleaning process because the processes will vary from dataset to dataset. But it is crucial to establish a template for your data cleaning process so you know you are doing it the right way every time.

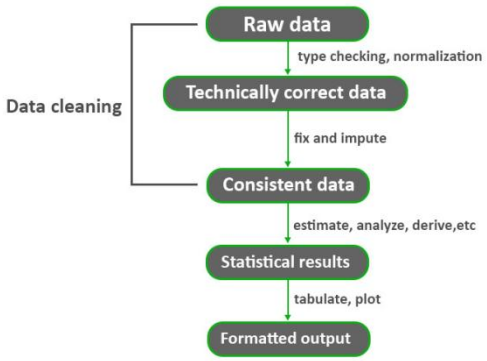
Purpose of Data Cleaning

The following are the various purposes of data cleaning in R:

- Eliminate Errors
- Eliminate Redundancy
- Increase Data Reliability
- Delivery Accuracy
- Ensure Consistency
- Assure Completeness
- Standardize your approach

Overview of a typical data analysis chain

Each rectangle in the figure represents data in a certain state while each arrow represents the activities needed to get from one state to the other. The first state (Raw data) is the data as it comes in. Raw data may lack headers, contain wrong data types, wrong category labels, unknown or unexpected character encoding, and so on. Once this pre-processing has taken place, data can be deemed Technically correct Data. That is, in this state data can be read into an R dataframe, with correct names, types, and labels, without further trouble. However, this does not mean that the values are error-free or complete. Consistent data is the stage where data is ready for statistical inference. It is the data that most statistical theories use as a starting point.



How to clean data in R ?

Here, this involves various steps, as from the initial raw data have to move toward the consistent and highly efficient data which is ready to be implemented as per the requirements and produces highly precise and accurate statistical results. The steps vary from data to data in this case the user should be aware of the data he/she is using for the results. As there are many characteristics and common symptoms of messy data which totally depend on the data used by the user for analysis.

Characteristics of clean data include data are:

- Free of duplicate rows/values
- Error-free (misspellings free)
- Relevant (special characters free)
- The appropriate data type for analysis
- Free of outliers (or only contain outliers that have been identified/understood)
- Follows a “tidy data” structure

Common symptoms of messy data:

- Special characters (e.g. commas in numeric values)
- Numeric values stored as text/character data types
- Duplicate rows
- Misspellings
- Inaccuracies
- White space
- Missing data
- Zeros instead of null values vary.

More specifically, data cleaning often entails (but is not limited to):

- Identifying and correcting data-entry errors and inconsistent coding;
- Evaluating missing data and determining how to handle them;
- Flagging and correcting out-of-bounds scores for variables;
- Addressing open-ended and/or "other" responses from employee surveys;
- Flagging and potentially removing untrustworthy variables.

With categorical (i.e., nominal, ordinal) variables containing text values, sometimes different spellings or formatting (e.g., uppercase, lowercase) are used (mistakenly) to represent the same category. Such issues broadly fall under data-entry errors and inconsistent coding reason for data cleaning. While the human eye can usually discern what the intended category is, many software programs and programming languages will be unable to automatically or correctly determine which text values represent which category. For example, in the table below, the facility variable is categorical and contains text values meant to represent different facilities at this hypothetical organization. Human eyes can quickly pick out that there are two facility locations represented in this table: Beaverton and Portland. With that said, for the R programming language, without direction, the different spellings and different cases (i.e., lowercase vs. uppercase "B") for the Beaverton facility (i.e., Beaverton, beaverton, beverton) will be treated as unique categories (i.e., facilities in this context). Often this is the result of data-entry errors and/or the lack of data validation. To clean the Facility variable, we could convert all instances of "beaverton" and "beverton" to "Beaverton".

EmployeeID	Facility	StartDate
EP1619	Beaverton	05/15/2010
EP1845	beaverton	01/31/2008
EP4321	Beaverton	02/05/2017
EP0214	Portland	09/19/2018
EP9746		03/23/2010
EP1475	beverton	06/01/2011
EP9952		05/15/2010

Data-entry errors and inconsistent coding: In this table, different spelling and letter cases (e.g., uppercase vs. lowercase) appear for what is supposed to be the same facility location: Beaverton.

Missing data (i.e., missing scores) for certain observations (i.e., cases) should also be addressed during data cleaning. For the Facility variable in the table shown below, note how facility location data are missing for the employees with IDs EP9746 and EP9952. In this example, we could likely find other employee records or contact the employees (or their supervisors) in question to verify the facility location where these two employees work. Provided we find the facility locations for these two employees, we could then replace the missing values with the correct facility location information. In other instances, it may prove to be more difficult to replace missing data, such as when organization administers an anonymous employee survey and certain respondents have missing responses to certain questions or items. In such instances, we may decide to tolerate a small

percentage of missing data (e.g., < 10%) when we go to analyze the data; however, if the percentage of missing data is sufficiently large and if we plan to analyze the data to make inferences about underlying population from which the sample data were attained, we may begin to think more seriously about whether the data are missing completely at random, missing at random, or missing at not at random.

EmployeeID	Facility	StartDate
EP1619	Beaverton	05/15/2010
EP1845	beaverton	01/31/2008
EP4321	Beaverton	02/05/2017
EP0214	Portland	09/19/2018
EP9746		03/23/2010
EP1475	beverton	06/01/2011
EP9952		05/15/2010

Missing data: In this table, data are missing for the employees with IDs EP9746 and EP9952.

In some instances, we might encounter out-of-bounds scores, which refer to values that simply are too high or low, or that are just too unrealistic or implausible to be correct. In the table below, the Base Salary variable includes salaries that are all below 75,000 – with the notable exception of salary associated with employee ID EP0214. Let's imagine that this table is only supposed to include data for a specific job category; knowing that, a base salary of 789,120,000 seems excessively high in a global sense and extraordinarily high in a local sense. It could be that someone entered the base salary data incorrectly for this employee, perhaps by adding four extra zeroes at the end of the actual base salary amount. In this case, we would try to find verify the correct base salary for this individual and then make the correction to the data.

EmployeeID	Base Salary	Tenure (Years)
EP1619	$54,310	2.5
EP1845	$29,911	0.5
EP4321	$31,471	3.5
EP0214	$789,120,000	0.5
EP9746	$74,031	-9999
EP1475	$42,985	1.0
EP9952	$55,541	1.0

Out-of-bounds scores: In this table, the base salary for the individual with employee ID EP0214 seems extraordinarily high and is almost certainly a data entry error.

When an open-ended response or "other" response field is provided (as opposed to a close-ended response field with predetermined response options), the individual who enters the data can type in whatever they would like (in most instances) provided that they limit their response to the allotted space. This challenge crops up frequently in employee surveys, such as when employees may select an "other" response for one survey question that then branches them to a follow-up question that is open-ended. In the table below, survey respondents' close-ended response options for the Number of Direct Reports variable include an "Other" option; respondents who responded with "Other" had an opportunity to indicate their number of direct reports using an open-ended survey question associated with the Number of Direct Reports (Other) variable. When cleaning such data, we often must determine what to do with the affect variables on a case-by-case basis. For example, the individual

who responded to the survey associated with a unique identifier of 2 responded with "Not a supervisor". If this survey was intended to acquire data from supervisors only, then we might decide to remove the row of data associated with that individual's response, as they likely do not fit our definition of the target population.

Survey Number	Number of Direct Reports	Number of Direct Reports (Other)
1	3	
2	Other	Not a supervisor
3	6	
4	5	
5	Other	Dotted-line supervisor of 3 employees
6	11	
7	5	

Open-ended and/or "other" responses: In this table, survey respondents' close-ended response options for the Number of Direct Reports variable include an "Other" option; for respondents who responded with "Other", they then had an opportunity to indicate their number of direct reports using an open-ended survey question associated with the Number of Direct Reports (Other) variable.

Finally, sometimes scores for a variable (or even missing scores for a variable) seem "off," incorrect, or implausible – or in other words, untrustworthy. For example, in the table below, a variable called Training Post Test is meant to include the scores on a post-training assessment; yet, we can see that the individual with employee ID EP1475 has a score of 99 even though the adjacent variable indicates that the individual did not complete the training. Now, it's entirely possible that this person (and others) were part of a control group (i.e., comparison group) intended to be used as part of a training evaluation design, but that then begs the question why only one individual in this table has a training post-test score. At first glance, data associated with the Training Post Test variable seem untrustworthy, and they very well may be in reality. As a next step, we would want to do some sleuthing to figure out what errors or issues may be at work in these data, and whether we should remove the potentially untrustworthy variable in question.

EmployeeID	Completed Training	Training Post Test
EP1619	No	
EP1845	No	
EP4321	No	
EP0214	No	
EP9746	No	
EP1475	No	99
EP9952	No	

Untrustworthy variables: In this table, data are missing for all but employee ID EP1475 on the Training Post Test variable, and furthermore, the Completed Training variable indicates that none of the employees who appear in the table received training.

Function	Package
View	base R
count	dplyr
str	base R
mutate	dplyr
replace	base R
match	base R
ifelse	base R
is.na	base R
toupper	base R
tolower	base R
names	base R
function	base R
clean_names	janitor

Example

```
install.packages("readr")
library(readr)
df<- read_csv("DataCleaningExample.csv")
names(df)
```

Output

```
[1] "EmpID" "Facility" "JobLevel" "StartDate" "Org_Tenure_Yrs"
"OnboardingCompleted"
```

```
df
```

Output

EmpID	Facility	JobLevel	StartDate	Org_Tenure_Yrs	OnboardingCompleted
EP1201	Beaverton	1	2010-05-05	8.6	No
EP1202	beaverton	1	2008-01-31	10.9	No
EP1203	Beaverton	-9999	2017-02-05	1.9	No
EP1204	Portland	5	2018-09-19	0.3	No
EP1205		2	2018-09-19	0.3	No
EP1206	Beaverton	1	2010-03-23	8.8	No
EP1207	beverton	1	2011-06-01	7.6	No
EP1208	Portland	4	2010-05-15	8.6	No
EP1209	Portland	3	2011-04-29	7.7	No
EP1210	Beaverton	11	2012-07-11	6.5	No

Note in the data frame that the EmpID field/variable is a unique identifier variable, which means that each case/observation has a unique value on this field/variable.

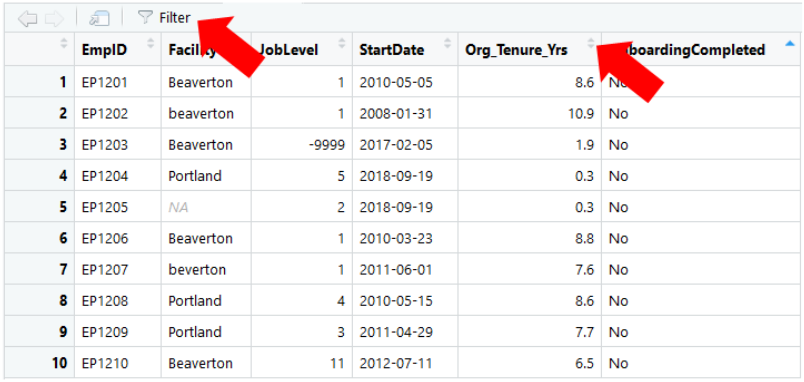
Review Data

There are different tools and techniques we can use to review the cleanliness or integrity of the available data. Often, it's a good idea to have the "ocular test" on the data which simply means to scan the raw data using your eyes. The ocular test can give you an idea of the types of data cleaning issues you'll need to address. The View function from base R is a great tool for this, as it allows you to look at the data frame object in a viewer tab, and using the arrows at the top of each column, you can sort each field (i.e., variable) manually.

```
# View data frame object using View function

View(df)
```

Output



	EmpID	Facility	JobLevel	StartDate	Org_Tenure_Yrs	boardingCompleted
1	EP1201	Beaverton	1	2010-05-05	8.6	No
2	EP1202	beaverton	1	2008-01-31	10.9	No
3	EP1203	Beaverton	-9999	2017-02-05	1.9	No
4	EP1204	Portland	5	2018-09-19	0.3	No
5	EP1205	NA	2	2018-09-19	0.3	No
6	EP1206	Beaverton	1	2010-03-23	8.8	No
7	EP1207	beverton	1	2011-06-01	7.6	No
8	EP1208	Portland	4	2010-05-15	8.6	No
9	EP1209	Portland	3	2011-04-29	7.7	No
10	EP1210	Beaverton	11	2012-07-11	6.5	No

Reviewing data using the View function from base R.

In addition, applying the count function from the dplyr package can provide us with an understanding of the values (or lack thereof) associated with each variable in your data frame object. The count function groups and tells the number of observations (i.e., frequencies) by value/level within a variable. Using this function we can (hopefully) identify any values that might be considered "out of bounds" or erroneous.

We can achieve the same output with and without the use of the pipe (%>%) operator because the dplyr package is built upon the magrittr package.

Using the pipe (%>%) operator, first, type the name of the data frame object to which a variable belongs (e.g., df). Second, type the %>% operator. Third, type the name of the count function, and within the parentheses, enter the exact name of the variable (e.g., Facility) as the sole argument.

```
Library(dplyr)

# Apply count function to Facility variable using pipe

df %>% count(Facility)

Or

# Apply count function to Facility variable without using pipe

count(df, Facility)
```

Output

```
## # A tibble: 5 × 2
##   Facility      n
##   <chr>     <int>
## 1 Beaverton     4
## 2 Portland      3
## 3 beaverton     1
## 4 beverton      1
## 5 <NA>          1
```

Note that the output yields a table object (or more specifically a "tibble", which is used in the tidyverse collection of functions). There are two columns: (a) the variable name, below which appears all values/levels of that variable, and (b) the n column, which displays the number of cases/observations associated with each value/level of the variable in question. As you can see, there appear to be some errors. Perhaps the database used to gather these data lacked data validation rules for certain variables, which allowed users to enter different spellings or formatting of the same value/level of the variables. For example, the Beaverton facility is spelled/formatted in three ways: Beaverton, beaverton, and beverton. Because R is case sensitive, Beaverton and beaverton are treated as to distinct levels/values of the Facility variable. Further, beverton is a misspelling of Beaverton and lacks the capital B. Finally, note that there is one NA value, which indicates that someone forgot to enter the name of the facility where that employee works. Clearly, we need to correct these errors and clean up the data residing within this variable.

Now let's apply the count function to the JobLevel variable. All we need to do is replace Facility with JobLevel in the syntax/code we wrote previously.

Example

```
# Apply count function to JobLevel variable using pipe

df %>% count(JobLevel)
```

Output

```
## # A tibble: 7 × 2
##   JobLevel     n
##      <dbl> <int>
## 1    -9999     1
## 2        1     4
## 3        2     1
## 4        3     1
## 5        4     1
## 6        5     1
## 7       11     1
```

For the sake of this example, let's assume that the company has only five job levels (1 = lowest, 5 = highest). In the output, it is apparent that there are two out-of-bounds values: -9999 and 11. Rather than leave cell blank when entering data, some people prefer to use extreme negative values, such as -9999, to flag missing data, and let's assume that is the case in this example. A potential issue with this approach to code missing data is that R assumes -9999 is a real value, and in R, NA is often used to indicate missing data to avoid this issue. Regarding the 11 value, it is likely that someone made an error when entering the data value in the database by typing 1 twice by mistake; let's assume that we verified that this was the case and that 11 should be replaced with 1.

Lastly, let's apply the count function to the OnboardingCompleted variable.

```
# Apply count function to OnboardingCompleted variable using pipe

df %>% count(OnboardingCompleted)
```

Output

```
## # A tibble: 1 × 2
##   OnboardingCompleted     n
##   <chr>               <int>
## 1 No                     10
```

In this example, employees either completed (Yes) or did not complete (No) the onboarding for new employees. As you can see in the output, there are no missing data (i.e., all 10 cases have a value); however, note that every single case/observation (i.e., employee) has the value No when in actuality (let's assume) every single case/observation should have the value Yes because we learned that every single employee in this data frame completed the onboarding program.

Clean Data

As is often the case in R (and in life), there are multiple approaches to cleaning data. Let's see two of them: (a) replacing a value for specific cases/observations with the correct value and (b) replacing a specific value for all cases that have the same value on that variable.

Replace a Specific Value for a Specific Case

To do so, we'll apply three functions: mutate from the dplyr package and replace and match from base R. Using the pipe (%>%) operator, we can replace a specific value for a specific case by:

1. Type the name of a new data frame object newdf1.

2. Type the <- operator to the right of the new data frame object so that the results of the subsequent operations can be assigned to the new object.

3. Type the name of the original data frame object (df), followed by the pipe (%>%) operator.

4. Type the name of the **mutate** function which creates new columns that are functions of existing variables. It can also modify and delete columns. Within the mutate function parentheses, provide the name of an existing or new variable; in this case, we will overwrite the existing Facility variable by typing the same variable name. Next, enter the = operator to the right of the mutate function to specify how values for the Facility variable will be determined.

5. Type the name of the **replace** function, which is a function that allows one to replace values in specific location within a vector or data frame. Within the replace function parentheses, as the first argument, type the name of the variable for which you wish to replace specific values (Facility). As the second argument, type the name of the **match** function, which is a function that allows us to find matched value(s) within a variable (i.e., vector); s the first argument within the match function, enter the value you wish to find a match for ("EP1202"), and as the second argument, enter the name of the variable (EmpID) to which that value belongs. Essentially, this match function and its two arguments will instruct R to identify the case in which EmpID (variable name) is equal to "EP1202". Because the variable EmpID is of type character (chr), the value identified from that variable should be placed in quotation marks (" "); if the variable were of type numeric, you would not include the quotation marks (" "), and as shown below, the str (structure) function can be used to identify the variable type for the variables within a data frame object (e.g., df); as the final (third) argument within the replace function, enter the new value (e.g., "Beaverton") that you would like to replace the existing value with for the particular case you identified.

Example

```
# Identify variable type

str(df)
```

Output

```
## spc_tbl_ [10 × 6] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ EmpID             : chr [1:10] "EP1201" "EP1202" "EP1203" "EP1204" ...
##  $ Facility          : chr [1:10] "Beaverton" "beaverton" "Beaverton" "Portland" ...
##  $ JobLevel          : num [1:10] 1 1 -9999 5 2 ...
##  $ StartDate         : Date[1:10], format: "2010-05-05" "2008-01-31" "2017-02-05" "2018-09-19"
##  $ Org_Tenure_Yrs    : num [1:10] 8.6 10.9 1.9 0.3 0.3 8.8 7.6 8.6 7.7 6.5
##  $ OnboardingCompleted: chr [1:10] "No" "No" "No" "No" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   EmpID = col_character(),
##   ..   Facility = col_character(),
##   ..   JobLevel = col_double(),
##   ..   StartDate = col_date(format = ""),
##   ..   Org_Tenure_Yrs = col_double(),
##   ..   OnboardingCompleted = col_character()
##   .. )
##  - attr(*, "problems")=<externalptr>
```



```
# For EmpID equal to EP1202, replace "beaverton" with "Beaverton" using pipe

newdf1 <- df %>%

  mutate(Facility = replace(Facility, match("EP1202", EmpID), "Beaverton"))

# Print new data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>             <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05           8.6 No
##  2 EP1202 Beaverton        1 2008-01-31          10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05           1.9 No
##  4 EP1204 Portland         5 2018-09-19           0.3 No
##  5 EP1205 <NA>             2 2018-09-19           0.3 No
##  6 EP1206 Beaverton        1 2010-03-23           8.8 No
##  7 EP1207 beverton         1 2011-06-01           7.6 No
##  8 EP1208 Portland         4 2010-05-15           8.6 No
##  9 EP1209 Portland         3 2011-04-29           7.7 No
## 10 EP1210 Beaverton       11 2012-07-11           6.5 No
```

In the new data frame object (newdf1), you should now see that “beaverton” has been replaced with “Beaverton” for the case in which EmpID is equal to “EP1202”.

To apply the mutate function without a pipe (%>%) operator, we simply remove the pipe (%>%) operator and enter the name of our original data frame object (df) as the first argument of the mutate function.

```
# For EmpID equal to EP1202, replace "beaverton" with "Beaverton" without using pipe

newdf1 <- mutate(df,

                 Facility = replace(Facility,
match("EP1202",EmpID),"Beaverton"))

# Print new data frame object
```

```
print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 beverton         1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

Now let's take the newdf1 data frame object we just created.

```
# For EmpID equal to EP1207, replace "beverton" with "Beaverton" using pipe
newdf1 <- newdf1 %>%
  mutate(Facility = replace(Facility,match("EP1207", EmpID), "Beaverton"))
# Print new data frame object
print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

Let's assume that after looking through other organizational records we found that the employee with EmpID equal to "EP1205" works at the "Portland" facility. Accordingly, we want to replace the NA value for the Facility variable for that employee with "Portland". When adapting the previous script/code, all we need to do is replace "EP1207" with "EP1205" and "Beaverton" with "Portland".

```
# For EmpID equal to EP1205, replace NA with "Portland" using pipe
newdf1 <- newdf1 %>%
  mutate(Facility = replace(Facility, match("EP1205", EmpID), "Portland"))
```

```
# Print new data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 Portland         2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

Now that the Facility variable has been cleaned, let's move on to the JobLevel variable. In this, we determined that there were two problematic values for two cases: -9999 (associated with EmpID equal to "1203") and 11 (associated with EmpID equal to "1210"). We decided that -9999 should be replace with NA because it represents a missing value, and 11 should be replaced with 1 because it represents a data entry error. Check out the script/code below to see how these replacements were handled. Also, note that because the JobLevel variable is of type numeric, we don't put the replacement values of NA and 1 in direct quotations.

```
# For EmpID equal to EP1203, replace -9999 with NA using pipe

newdf1 <- newdf1 %>%

  mutate(JobLevel = replace(JobLevel,match("EP1203", EmpID), NA))

# Print new data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 Portland         2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

```
# For EmpID equal to EP1210, replace 11 with 1 using pipe

newdf1 <- newdf1 %>%

  mutate(JobLevel = replace(JobLevel,match("EP1210", EmpID), 1))
```

```
# Print new data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 Portland         2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton        1 2012-07-11            6.5 No
```

Finally, let's assume that for the OnboardingCompleted variable, two of the cases have values that should in fact be NA (missing). Specifically, because the hypothetical onboarding program takes 1.0 year to complete, the two employees (cases) with Org_Tenure_Yrs (organizational tenure in years) equal to 0.3 have not worked in the company long enough to have completed the 1-year-long onboarding program. Consequently, we decide that it is not appropriate to put "No" or "Yes" for these two employees, which means treating those values as missing (NA) would be most appropriate. The two employees in question have EmpID variable values of "EP1204" and "EP1205". Note that we can enter NA as the final argument within the replace function parentheses, just as we would with an actual value. Check out the script/code below to see how we replace 0.3 with NA on the OnboardingCompleted variable for the cases with the EmpID unique identifier variable equal to "EP1204" and "EP1205".

```
# For EmpID equal to EP1204, replace 0.3 with NA using pipe

newdf1 <- newdf1 %>%

  mutate(OnboardingCompleted  =  replace(OnboardingCompleted,match("EP1204",
EmpID), NA))

# For EmpID equal to EP1205, replace 0.3 with NA using pipe

newdf1 <- newdf1 %>%

  mutate(OnboardingCompleted  =  replace(OnboardingCompleted,match("EP1205",
EmpID),NA))

# Print new data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 <NA>
##  5 EP1205 Portland         2 2018-09-19            0.3 <NA>
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton        1 2012-07-11            6.5 No
```

Replace a Specific Value for All Cases with a Particular Value

If you need to systematically clean multiple values for a given variable, there is another approach that is more efficient. Note that this approach is only appropriate if you have identified that the values in question all need to be changed to the same value. For the sake of this example, let's pretend that all of the "No" values for the OnboardingCompleted variable were entered incorrectly. Instead of "No", the value should be "Yes" for each of these cases.

First, we will learn how to replace these values using the mutate function from the dplyr package and the ifelse function from base R.

Second, we will learn how to do the same thing without the use of a specific function.

Let's start with the first approach, which involves the mutate and ifelse functions. Using the pipe (%>%) operator, we can do the following:

1. Type the name of a new data frame object called newdf1.

2. Type the <- operator to the right of the new data frame object so that the results of the subsequent operations can be assigned to the new object.

3. Type the name of the original data frame object (newdf1), followed by the pipe (%>%) operator.

4. Type the name of the mutate function. Within the mutate function parentheses, provide the name of an existing or a new variable; in this case, we will overwrite the existing OnboardingCompleted variable by typing the same name as the existing variable. To the right of the mutate function, type the = operator to specify how values for the OnboardingCompleted variable will be determined.

5. Type the name of the ifelse function, which is a function that can be used for conditional value (element) selection (identification) and replacement. As the first argument, type a conditional statement for the variable whose values you wish to replace; because we wish to replace all "No" values with "Yes" for the OnboardingCompleted variable, we'll specify the conditional statement as OnboardingCompleted=="No". As the second argument in the ifelse function, type the replacement value when the aforementioned conditional statement is true for a given value of the OnboardingCompleted variable, which is "Yes" in this example. Because the OnboardingCompleted variable is of type character, this value needs to be in quotation marks (" "). As the final argument function, instruct the ifelse function what the replacement value should be when the aforementioned conditional statement is false, which in this; in this example, we will replace all other values with the existing values of the OnboardingCompleted variable, and thus we'll enter that variable name as the third argument.

Example

```
# For OnboardingCompleted variable, replace "No" values with "Yes" using pipe

newdf1 <- newdf1 %>%

mutate(OnboardingCompleted = ifelse(OnboardingCompleted=="No","Yes",

OnboardingCompleted))

# Print new data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>             <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05           8.6 Yes
##  2 EP1202 Beaverton        1 2008-01-31          10.9 Yes
##  3 EP1203 Beaverton       NA 2017-02-05           1.9 Yes
##  4 EP1204 Portland         5 2018-09-19           0.3 <NA>
##  5 EP1205 Portland         2 2018-09-19           0.3 <NA>
##  6 EP1206 Beaverton        1 2010-03-23           8.8 Yes
##  7 EP1207 Beaverton        1 2011-06-01           7.6 Yes
##  8 EP1208 Portland         4 2010-05-15           8.6 Yes
##  9 EP1209 Portland         3 2011-04-29           7.7 Yes
## 10 EP1210 Beaverton        1 2012-07-11           6.5 Yes
```

Now let's pretend that we wish to convert the "Yes" values for the OnboardingCompleted variable with the numeric value of 1. The format of the code remains virtually the same, except that we change the conditional statement to OnboardingCompleted=="Yes" as the first argument in the ifelse function, and in the second argument, we change the replacement value to 1.

```
# For OnboardingCompleted variable, replace "Yes" values with 1 using pipe
newdf1 <- newdf1 %>%
mutate(OnboardingCompleted = ifelse(OnboardingCompleted=="Yes", 1,
OnboardingCompleted))
# Print new data frame object
print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>             <dbl>              <dbl>
##  1 EP1201 Beaverton        1 2010-05-05           8.6                  1
##  2 EP1202 Beaverton        1 2008-01-31          10.9                  1
##  3 EP1203 Beaverton       NA 2017-02-05           1.9                  1
##  4 EP1204 Portland         5 2018-09-19           0.3                 NA
##  5 EP1205 Portland         2 2018-09-19           0.3                 NA
##  6 EP1206 Beaverton        1 2010-03-23           8.8                  1
##  7 EP1207 Beaverton        1 2011-06-01           7.6                  1
##  8 EP1208 Portland         4 2010-05-15           8.6                  1
##  9 EP1209 Portland         3 2011-04-29           7.7                  1
## 10 EP1210 Beaverton        1 2012-07-11           6.5                  1
```

As an alternative approach, we can write the following code. First, specify the name of the data frame object (newdf1), followed by the $ operator and the name of the variable in question (OnboardingCompleted). Second, type brackets ([ ]). Third, within the brackets, enter a conditional statement; for the sake of example, let's say that we want to identify all instances in which the OnboardingCompleted variable is equal to 1 (newdf1$OnboardingCompleted==1). Fourth, type the <- operator, followed by the value you wish to use to replace the existing values for which the conditional statement you previously specified is true; in this example, we enter 2. Because the two values are numeric, we do not use quotation marks (" ").

```
# For OnboardingCompleted variable, replace 1 values with 2

newdf1$OnboardingCompleted[newdf1$OnboardingCompleted==1] <- 2

# Print data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl>               <dbl>
##  1 EP1201 Beaverton        1 2010-05-05            8.6                   2
##  2 EP1202 Beaverton        1 2008-01-31           10.9                   2
##  3 EP1203 Beaverton       NA 2017-02-05            1.9                   2
##  4 EP1204 Portland         5 2018-09-19            0.3                  NA
##  5 EP1205 Portland         2 2018-09-19            0.3                  NA
##  6 EP1206 Beaverton        1 2010-03-23            8.8                   2
##  7 EP1207 Beaverton        1 2011-06-01            7.6                   2
##  8 EP1208 Portland         4 2010-05-15            8.6                   2
##  9 EP1209 Portland         3 2011-04-29            7.7                   2
## 10 EP1210 Beaverton        1 2012-07-11            6.5                   2
```

If we wish to change numeric values to character values, be sure to put the character values in quotation marks (" "). In the following example, all instances in which the OnboardingCompleted variable is equal to 2 are changed to “Yes”.

```
# For OnboardingCompleted variable, replace 2 values with "Yes"

newdf1$OnboardingCompleted[newdf1$OnboardingCompleted==2] <- "Yes"

# Print data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 Yes
##  2 EP1202 Beaverton        1 2008-01-31           10.9 Yes
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 Yes
##  4 EP1204 Portland         5 2018-09-19            0.3 <NA>
##  5 EP1205 Portland         2 2018-09-19            0.3 <NA>
##  6 EP1206 Beaverton        1 2010-03-23            8.8 Yes
##  7 EP1207 Beaverton        1 2011-06-01            7.6 Yes
##  8 EP1208 Portland         4 2010-05-15            8.6 Yes
##  9 EP1209 Portland         3 2011-04-29            7.7 Yes
## 10 EP1210 Beaverton        1 2012-07-11            6.5 Yes
```

let’s change all of the “Yes” values for the OnboardingCompleted variable back to “No” using the code below.

```
# For OnboardingCompleted variable, replace "Yes" values with "No"

newdf1$OnboardingCompleted[newdf1$OnboardingCompleted=="Yes"] <- "No"

# Print data frame object

print(newdf1)
```

Output

```
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 <NA>
##  5 EP1205 Portland         2 2018-09-19            0.3 <NA>
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton        1 2012-07-11            6.5 No
```

Finally, imagine we wish to replace the NA values in the OnboardingCompleted variable with the value 2 using this approach. To do so, instead of making our own conditional statement, within the brackets ([ ]), apply the is.na function from base R, and type the name of the data frame, followed by the $ operator and the name of the variable containing the NAs. To the right of the bracket, use the <- operator followed by the value with which you wish to replace the NAs.

```
# For OnboardingCompleted variable, replace NA values with 2

newdf1$OnboardingCompleted[is.na(newdf1$OnboardingCompleted)] <- 2

# Print data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 2
##  5 EP1205 Portland         2 2018-09-19            0.3 2
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton        1 2012-07-11            6.5 No
```

Rename Variables

In some cases you may wish to rename an existing variable. One of the simplest ways to do this is to create a new variable and then delete the old variable. Let's rename the Org_Tenure_Yrs variable as simply Tenure.

First, specify the name of the data frame object (newdf1), followed by the $ operator and the new name of the variable (Tenure). Second, type the <- operator. Third, specify the name of the data frame object (newdf1), followed by the $ operator and the name of the old variable (Org_Tenure_Yrs).

Example

```
# Create new variable called Tenure based on Org_Tenure_Yrs variable

newdf1$Tenure <- newdf1$Org_Tenure_Yrs

# Print data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 7
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted Tenure
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>                <dbl>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No                     8.6
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No                    10.9
##  3 EP1203 Beaverton       NA 2017-02-05            1.9 No                     1.9
##  4 EP1204 Portland         5 2018-09-19            0.3 2                      0.3
##  5 EP1205 Portland         2 2018-09-19            0.3 2                      0.3
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No                     8.8
##  7 EP1207 Beaverton        1 2011-06-01            7.6 No                     7.6
##  8 EP1208 Portland         4 2010-05-15            8.6 No                     8.6
##  9 EP1209 Portland         3 2011-04-29            7.7 No                     7.7
## 10 EP1210 Beaverton        1 2012-07-11            6.5 No                     6.5
```

Note that there is now a new variable called Tenure and that the old variable called Org_Tenure_Yrs remains. To remove the old variable called Org_Tenure_Yrs, first, specify the name of the data frame object (newdf1), followed by the $ operator and the name of the old variable (Org_Tenure_Yrs). Second, type the <- operator. Third, enter NULL.

```
# Remove Org_Tenure_Yrs variable from data frame

newdf1$Org_Tenure_Yrs <- NULL

# Print data frame object

print(newdf1)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  OnboardingCompleted Tenure
##    <chr>  <chr>        <dbl> <date>     <chr>                <dbl>
##  1 EP1201 Beaverton        1 2010-05-05 No                     8.6
##  2 EP1202 Beaverton        1 2008-01-31 No                    10.9
##  3 EP1203 Beaverton       NA 2017-02-05 No                     1.9
##  4 EP1204 Portland         5 2018-09-19 2                      0.3
##  5 EP1205 Portland         2 2018-09-19 2                      0.3
##  6 EP1206 Beaverton        1 2010-03-23 No                     8.8
##  7 EP1207 Beaverton        1 2011-06-01 No                     7.6
##  8 EP1208 Portland         4 2010-05-15 No                     8.6
##  9 EP1209 Portland         3 2011-04-29 No                     7.7
## 10 EP1210 Beaverton        1 2012-07-11 No                     6.5
```

As you can see, the Org_Tenure_Yrs variable is now gone.

Other Approaches to Cleaning Data

Changing the Case of Variable Names

In some instances, we may wish to systematically change the case of all variable or column names. There are a few functions that can be quite handy in this regard. Let's revert back the name of the data frame object we initially read in and names: df.

```
# Access readr package

library(readr)
```

```
# Read in set of data as data frame

df<- read_csv("DataCleaningExample.csv")
```

To change the case of variable or column names such that they are all lower case, we can use the tolower function from base R. Let's change the variable names to be all lower case. We will need to use the names function from base R as well to signal that we are referencing the column names as opposed to the values for each variable. First, type the name of the names function with the name of the new data frame object you are creating and naming as the sole argument; here, we will put in the same name of the existing data frame to overwrite it. Second, enter the <- operator to create and name the new data frame object. Third, enter the tolower function, and as the sole argument, include the names function again with the name of the focal data frame object as its sole argument.

```
# Change case of variable names to all lower case
names(df) <- tolower(names(df))
# Print data frame object
print(df)
```

Output

```
## # A tibble: 10 × 6
##    empid  facility  joblevel startdate  org_tenure_yrs onboardingcompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 beverton         1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

We can use the toupper function to change the variable names to all upper case.

```
# Change case of variable names to all upper case
names(df) <- toupper(names(df))
# Print data frame object
print(df)
```

Output

```
## # A tibble: 10 × 6
##    EMPID  FACILITY  JOBLEVEL STARTDATE  ORG_TENURE_YRS ONBOARDINGCOMPLETED
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 beverton         1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

If you wanted to capitalize just the first letter in each variable name, we can create our function to do so. Let's call this function firstletterupper() and use the function called function from base R to program our own function. In this function, we will define what the function will do. Essentially, we are going to create two strings of text and then concatenate them using the paste function. As the first argument in the paste function and to select a string of text, we will pull just the first letter of each variable name using the substring function; the second and third numeric arguments in this function indicate the range of letters that we will pull out, and because we put 1 and 1, we are saying retain just the first letter in each name. We then enter this substring function as an argument in the toupper function so that we will just capitalize the first letter of each variable. We will then enter the second argument in the paste function, which is another substring function; this time, we will indicate that we simply wish to retain the second letter in each name followed by any remaining letters, and we do so by entering the numeral 2 as the second argument. We then enter this substring function as an argument in the tolower() function to make all but the first letter in each variable name lower case. As the final argument in the paste function, we enter sep="" to signify that we do not want any space between these letters when they are concatenated; note that there is no space within the quotation marks. We can now run this script to program our new function called firstletterupper().

```
# Create function to change just first letter of variable name to upper case

firstletterupper<- function(x) {paste(toupper(substring(x, 1, 1)),

tolower(substring(x, 2)),

sep="")}
```

With our new DIY function called firstletterupper we can apply it to the variable names using the same approach we did above with the toupper and tolower functions.

```
# Change just first letter of variable name to upper case

names(df) <- firstletterupper(names(df))

# Print data frame object

print(df)
```

Output

```
## # A tibble: 10 × 6
##    Empid  Facility  Joblevel Startdate  Org_tenure_yrs Onboardingcompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 beverton         1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

Alternatively, we could use the clean_names function from the janitor package. Be sure to install and access the package if you haven't already.

```
# Install janitor package if you haven't already

install.packages("janitor")

# Access janitor package

library(janitor)
```

When coupled with the case=upper_camel argument, the clean_names function will capitalize the first letter in variable names, and if there is an underscore within the variable name (_), the function will capitalize the letter that comes immediately after the underscore.

```
# Change just first letter of variable name to upper case

df<- clean_names(df, case="upper_camel")

# Print data frame object

print(df)
```

Output

```
## # A tibble: 10 × 6
##    Empid  Facility  Joblevel Startdate  OrgTenureYrs Onboardingcompleted
##    <chr>  <chr>        <dbl> <date>           <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05         8.6 No
##  2 EP1202 beaverton        1 2008-01-31        10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05         1.9 No
##  4 EP1204 Portland         5 2018-09-19         0.3 No
##  5 EP1205 <NA>             2 2018-09-19         0.3 No
##  6 EP1206 Beaverton        1 2010-03-23         8.8 No
##  7 EP1207 beverton         1 2011-06-01         7.6 No
##  8 EP1208 Portland         4 2010-05-15         8.6 No
##  9 EP1209 Portland         3 2011-04-29         7.7 No
## 10 EP1210 Beaverton       11 2012-07-11         6.5 No
```

There are many other case= arguments that you could use for the clean_name functions to different variations of capitalization. Just access the help menu for the function as shown below.

 # Access help information for clean_names function

```
?clean_names()
```

Changing the Case of Character Variable Values

In other instances, we may wish to systematically change the case of values for a character variable.

```
# Access readr package

library(readr)

# Read in set of data as data frame

df<- read_csv("DataCleaningExample.csv")

# Change case of Facility variable's values to all lower case

df$Facility<- tolower(df$Facility)
```

```
# Print data frame object

print(df)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 beaverton        1 2010-05-05            8.6 No
##  2 EP1202 beaverton        1 2008-01-31           10.9 No
##  3 EP1203 beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 beaverton        1 2010-03-23            8.8 No
##  7 EP1207 beverton         1 2011-06-01            7.6 No
##  8 EP1208 portland         4 2010-05-15            8.6 No
##  9 EP1209 portland         3 2011-04-29            7.7 No
## 10 EP1210 beaverton       11 2012-07-11            6.5 No
```

We can similarly use the toupper function to change a character variable's values to all upper case.

```
# Change case of Facility variable's values to all upper case

df$Facility<- toupper(df$Facility)

# Print data frame object

print(df)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 BEAVERTON        1 2010-05-05            8.6 No
##  2 EP1202 BEAVERTON        1 2008-01-31           10.9 No
##  3 EP1203 BEAVERTON    -9999 2017-02-05            1.9 No
##  4 EP1204 PORTLAND         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 BEAVERTON        1 2010-03-23            8.8 No
##  7 EP1207 BEVERTON         1 2011-06-01            7.6 No
##  8 EP1208 PORTLAND         4 2010-05-15            8.6 No
##  9 EP1209 PORTLAND         3 2011-04-29            7.7 No
## 10 EP1210 BEAVERTON       11 2012-07-11            6.5 No
```

We can now run this script to program our new function called firstletterupper. Note that this is the same function we created above for changing the case of variable names, but we will take it to another level in this instance. Specifically, we are now ready to insert the paste function we just specified into the ifelse function from base R. Specifically, in case there are any missing values (NAs), we want to exclude those when running this paste function. As the first argument in the ifelse function enter !is.na(x) to indicate we want to select those values for that are not NA. As the second argument, we enter what we wrote for the paste function. As the third argument, we address the "else" part of the ifelse function; meaning, we need to indicate how values that are NA should be treated. By entering the name of the object x as this final argument, we are indicating that the original value (which in this case is actually missing as NA) will be retained.

```
# Create function to change just first letter of character variable's values to upper case

firstletterupper<- function(x) {ifelse(!is.na(x),

paste(toupper(substring(x, 1, 1)),

tolower(substring(x, 2)),

sep=""), x)}
```

With our new DIY function called firstletterupper we can apply it to the character variable we wish to change (Facility).

```
# Change just first letter of character variable's values to upper case

df$Facility<- firstletterupper(df$Facility)

# Print data frame object

print(df)
```

Output

```
## # A tibble: 10 × 6
##    EmpID  Facility  JobLevel StartDate  Org_Tenure_Yrs OnboardingCompleted
##    <chr>  <chr>        <dbl> <date>              <dbl> <chr>
##  1 EP1201 Beaverton        1 2010-05-05            8.6 No
##  2 EP1202 Beaverton        1 2008-01-31           10.9 No
##  3 EP1203 Beaverton    -9999 2017-02-05            1.9 No
##  4 EP1204 Portland         5 2018-09-19            0.3 No
##  5 EP1205 <NA>             2 2018-09-19            0.3 No
##  6 EP1206 Beaverton        1 2010-03-23            8.8 No
##  7 EP1207 Beverton         1 2011-06-01            7.6 No
##  8 EP1208 Portland         4 2010-05-15            8.6 No
##  9 EP1209 Portland         3 2011-04-29            7.7 No
## 10 EP1210 Beaverton       11 2012-07-11            6.5 No
```

Data Transformation in R

Data transformation is the process of converting, cleansing, and structuring data into a usable format that can be analyzed to support decision making processes.Transformation processes can also be referred to as data wrangling, or data munging, transforming and mapping data from one "raw" data form into another format for warehousing and analyzing.

The data transformation in R is mostly handled by the external packages tidyverse and dplyr . These packages provide many methods to carry out the data simulations. There are a large number of ways to simulate data transformation in R. These methods are widely available using these packages, which can be downloaded and installed using the following command :

```
install.packages("tidyverse")
```

Method 1: Using Arrange() method

For data transformation in R, we will use the arrange() method, to create an order for the sequence of the observations given. It takes a single column or a set of columns as the input to the method and creates an order for these.

The arrange() method in the tidyverse package inputs a list of column names to rearrange them in a specified order. By default, the arrange() method arranges the data in ascending order. It has the following syntax :

```
Syntax: arrange(col-name)
```

Parameter:

col-name – Name of the column.

The data frame can be supplied with a pipe operator followed by the application of arrange() method to reflect the changes.

Example

```
# Importing tidyvverse
library(tidyverse)
# Creating a data frame
data_frame = data.frame(col1 = c(2,4,1,7,5,3,5,8),
      col2 = letters[1:8],
      l3 = c(0,1,1,1,0,0,0,0))
# Assigning row names
rownames(data_frame) <- c("r1","r2","r3","r4","r5","r6","r7","r8")
print("Data Frame")
print(data_frame)
# Arranging a single column in ascending order
arr_data_frame<- data_frame %>% arrange(col1)
print("Arranged Data Frame")
print(arr_data_frame)
```

Output:

```
           col1       col2       col3
r1         2          a          0
r2         4          b          1
r3         1          c          1
r4         7          d          1
r5         5          e          0
r6         3          f          0
r7         5          g          0
r8         8          h          0
```

[1] "Arranged Data Frame"

```
           col1       col2       col3
r3         1          c          1
r1         2          a          0
r6         3          f          0
r2         4          b          1
r5         5          e          0
r7         5          g          0
r4         7          d          1
r8         8          h          0
```

The column values can also be arranged in descending order by specifying the order explicitly using the following syntax :

Syntax: arrange(desc(col-name))

Example

```
# Arranging column in descending order
arr_data_frame<- data_frame %>%arrange(desc(col1))
print("Arranged Data Frame")
print(arr_data_frame)
```

Output:

```
[1] "Arranged Data Frame"

     col1 col2 col3
r8    8    h    0
r4    7    d    1
r5    5    e    0
r7    5    g    0
r2    4    b    1
r6    3    f    0
r1    2    a    0
r3    1    c    1
```

Method 2: Using select() method

Data transformation in R of the data frame can also be fetched using the select() method in tidyverse package. The columns are fetched in the order of their specification in the argument list of the select() method call. This method results in a subset of the data frame as the output. The following syntax is followed :

```
Syntax: select(list-of-col-names)
```

Parameter:

```
list-of-col-names - List of column names separated by comma.
```

Example

```
# Importing tidyverse
library(tidyverse)
# Creating a data frame
data_frame = data.frame(col1 = c(2,4,1,7,5,3,5,8),
     col2 = letters[1:8],
     col3 = c(0,1,1,1,0,0,0,0),
     col4 = c(9:16))
print("Data Frame")
# Printing data_frame
print(data_frame)
# Selecting a range of columns in df
arr_data_frame<- data_frame %>%    select(col2,col4)
print("Selecting col2 and col4 in Data Frame")
print(arr_data_frame)
```

Output:

```
     col1 col2 col3 col4
1    2    a    0    9
2    4    b    1   10
3    1    c    1   11
4    7    d    1   12
5    5    e    0   13
6    3    f    0   14
7    5    g    0   15
8    8    h    0   16
[1] "Selecting col2 and col4 in Data Frame"
     col2 col4
1    a    9
2    b   10
3    c   11
4    d   12
5    e   13
6    f   14
7    g   15
8    h   16
```

The col2 and col4 of the data frame are selected and displayed as the output.

A consecutive range of columns can also be fetched from the data frame by specifying the colon operator. For instance, the following code snippet indicates that a range of columns can be extracted using the command col2:col4, which fetches all the columns in order beginning from col2 of the data frame.

Since there is a colon operator between col2 and col4, all the columns in the data frame are selected beginning from col2 and ending at col4 in order. Therefore, col2, col3, and col4 are returned as the output data frame.

```
Syntax: select(begin-col : end-col)
```

Example

```
# Selecting a range of columns in df
arr_data_frame<- data_frame %>% select(col2:col4)
print("Selecting col2 to col4 in Data Frame")
print(arr_data_frame)
```

Output:

```
 [1] "Selecting col2 to col4 in Data Frame"
  col2 col3 col4
1    a    0    9
2    b    1   10
3    c    1   11
4    d    1   12
5    e    0   13
6    f    0   14
7    g    0   15
8    h    0   16
```

Method 3: Using filter() method

The filter() method in the tidyverse package is used to apply a range of constraints and conditions to the column values of the data frame in data transformation in R. It filters the data and results in the smaller output returned by the column values satisfying the specified condition. The conditions are specified using the logical operators, and values are validated then. A data frame can be supplied with the pipe operator and then using the filter condition.

```
Syntax: filter(cond1, cond2)
```

Parameter:

cond1, cond2 – Condition to be applied on data.

Example

```
# Importing tidyverse
library(tidyverse)
# Creating a data frame
data_frame = data.frame(col1 = c(2,4,1,7,5,3,5,8),
     col2 = letters[1:8],
     col3                                                    =
c("this","that","there","here","there","this","that","here"),
     col4 = c(9:16))
print("Data Frame")
# Printing data frame
print(data_frame)
# Selecting values where col1 value is greater than 4
arr_data_frame<-       data_frame       %>%filter(col3       ==
c("there","this"))
print("Selecting col1>4 ")
# Printing data frame after
# applying filter
print(arr_data_frame)
```

Output:

```
  col1 col2  col3 col4
1    2    a  this    9
2    4    b  that   10
3    1    c there   11
4    7    d  here   12
5    5    e there   13
6    3    f  this   14
7    5    g  that   15
8    8    h  here   16
[1] "Selecting col3 value is either there or this"
  col1 col2  col3 col4
1    1    c there   11
2    5    e there   13
3    3    f  this   14
```

Multiple conditions can also be checked in the filter method and combined using the comma using filter() method. For instance, the below code checks for the col3 value equal to “there” and col1 value equivalent to 5, respectively. The output data frame contains the rows of the

original data frame where the col1 value is equivalent to 5 and the col3 value is equivalent to "there." The rows are fetched in the order in which they occur in the original data frame.

Example

```
# Selecting values where col3 value is there and col1 is 5
arr_data_frame<- data_frame %>%filter(col3=="there",col1==5)
print("Selecting col3 value is there and col1 is 5")
print(arr_data_frame)
```

Output:

```
[1] "Selecting col3 value is there and col1 is 5"
  col1 col2  col3 col4
1    5    e there   13
```

Method 4: Using spread() method

For the data transformation in the R, spread() method is used to spread any key-value pair in multiple columns in the data frame. It is used to increase the readability of the data specified in the data frame. The data is rearranged according to the list of columns in the spread() method. All the data in col2 is repeated until the values in col3 are exhausted. The entire data frame is returned as the output. It has the following syntax :

```
Syntax: spread(col-name)
```

Parameter:

col-name – Name of one or more columns according to which data is to be structured.

The following code arranges the data such that the values in col2 are assigned as column headings and their corresponding values as cell values of the data frame :

Example

```
# Importing tidyr
library(tidyr)
# Creating a data frame
data_frame = data.frame(col1 =
c("A","A","A","A","A","A","B","B","B","B","B","B"),
     col2 = c("Eng","Phy","Chem","MAQ","Bio","SST",
          "Eng","Phy","Chem","MAQ","Bio","SST"),
     col3 = c(34,56,46,23,72,67,89,43,88,45,78,99)
)
print("Data Frame")
print(data_frame)
# Selecting values by col2 and col3
arr_data_frame<- data_frame %>%spread(col2,col3)
print("Spread using col2 and col3")
print(arr_data_frame)
```

Output:

```
   col1 col2 col3
1     A  Eng   34
2     A  Phy   56
3     A Chem   46
4     A  MAQ   23
```

```
5     A  Bio   72
6     A  SST   67
7     B  Eng   89
8     B  Phy   43
9     B Chem   88
10    B  MAQ   45
11    B  Bio   78
12    B  SST   99
[1] "Spread using col2 and col3"
  col1 Bio Chem Eng MAQ Phy SST
1    A  72   46  34  23  56  67
2    B  78   88  89  45  43  99
```

The following code arranges the data such that the values in col2 are assigned as row headings and their corresponding values as cell values of the data frame :

```
# Selecting values by col1 and col3
arr_data_frame<- data_frame %>%    spread(col1,col3)
print("Spread using col1 and col3")
print(arr_data_frame)
```

Output:

```
[1] "Spread using col1 and col3"
  col2  A  B
1  Bio 72 78
2 Chem 46 88
3  Eng 34 89
4  MAQ 23 45
5  Phy 56 43
6  SST 67 99
```

Method 5: Using mutate() method

For the data transformation in R, mutate() method is used to create and modify new variables in the specified data frame. A new column name can be assigned to the data frame and evaluated to an expression where constants or column values can be used. The output data frame has the new columns created. The method has the following syntax :

Syntax: mutate (new-col-name = expr)

Parameters:

new-col-name – Name of column to be created.

expr – Expression which is applied on new column.

Multiple columns can also be added to the data frame and separated using the comma operator. The following code snippet illustrates the addition of two new columns, col5, which is the summation of col1 and col4 values, and col6, which is constant 1 added to the col3 values. For example, in first row col1 = 2 and col4 = 9 , therefore col5 = 2 + 9 = 11. And col6 is the addition of col3 value and 1.

```
# Importing tidyverse
library(tidyverse)
# Creating a data frame
data_frame = data.frame(col1 = c(2,4,1,7,5,3,5,8),
```

```
     col2 = letters[1:8],
     col3 = c(0,1,1,1,0,0,0,0),
     col4 = c(9:16))
print("Data Frame")
print(data_frame)
# Added new columns
data_frame_mutate<- data_frame %>% mutate(col5 = col1 + col4
,col6 = col3+1)
print("Mutated Data Frame")
print(data_frame_mutate)
```

Output:

```
 col1 col2 col3 col4
1    2    a    0    9
2    4    b    1   10
3    1    c    1   11
4    7    d    1   12
5    5    e    0   13
6    3    f    0   14
7    5    g    0   15
8    8    h    0   16
[1] "Mutated Data Frame"
  col1 col2 col3 col4 col5 col6
1    2    a    0    9   11    1
2    4    b    1   10   14    2
3    1    c    1   11   12    2
4    7    d    1   12   19    2
5    5    e    0   13   18    1
6    3    f    0   14   17    1
7    5    g    0   15   20    1
8    8    h    0   16   24    1
```

Method 6: Using group_by() and summarise() method

For the data transformation in R,group_by() and summarise() methods are used collectively to group by variables of the data frame and reduce multiple values down to a single value. It is used to make the data more readable. The column name can be specified in R's group_by() method. The data can be arranged in groups and then further summarised using the base aggregate methods in this package.

```
Syntax: group_by(col-name)
Syntax: group_by(col,..) %>% summarise(action)
```

The data in the data frame are grouped according to the col3 value. The count column indicates the number of records in each group; for instance, there are five rows with col3 = 0. The mean is then calculated for all the elements in a. particular group.

Example

```
# Importing dplyr
library(dplyr)
# Creating a data frame
data_frame = data.frame(col1 = c(2,4,1,7,5,3,5,8),
     col2 = letters[1:8],
```

```
     col3 = c(0,1,1,1,0,0,0,0),
     col4 = c(9:16))
print("Data Frame")
print(data_frame)
# Mutate data using group_by()and summarise()
data_frame_mutate<- data_frame %>% group_by(col3) %>%
                    summarise(count    =    n(),mean_col1    =
              mean(col1)
)
print("Mutated Data Frame")
print(data_frame_mutate)
```

Output:

```
 col1 col2 col3 col4
1    2    a    0    9
2    4    b    1   10
3    1    c    1   11
4    7    d    1   12
5    5    e    0   13
6    3    f    0   14
7    5    g    0   15
8    8    h    0   16
[1] "Mutated Data Frame"
# A tibble: 2 x 3
   col3 count mean_col1
<dbl><int><dbl>
1     0     5       4.6
2     1     3       4
```

Method 7: Using the gather() method

In R many data columns indicate the values that should be stored in a single column of the data frame and are unnecessarily bifurcated. This can be done using the gather() method. It collects the key-value pairs and rearranges them in new columns. The column values are specified as arguments of the gather() method. For instance, the following code snippet illustrates the combination of col2 to col4 of the data frame under argument 2 of the gather() method. The particular column from which the value was chosen is assigned the tag under the column name “Subject” given by argument 1 of the called method.

```
Syntax: gather(data, key, value)
```

The col2, col3, and col4 values in the data frame are clubbed under the column “Subject” in the output data frame. The marks of each student in each subject are then assigned in the col3 value.

Example

```
# Importing dplyr
Library(tidyr)
library(dplyr)
# Creating a data frame
data_frame            =            data.frame(col1            =
c("Jack","Jill","Yash","Mallika",
```

```
                "Muskan","Keshav","Meenu","Sanjay"),
     Maths = c(26,47,14,73,65,83,95,48),
     Physics = c(24,53,45,88,68,35,78,24),
     Chemistry = c(67,23,79,67,33,66,25,78)
     )
print("Data Frame")
print(data_frame)
data_frame_mutate<-                                    data_frame
%>%gather("Subject","Marks",2:4)
print("Mutated Data Frame")
# Printing after rearrange data
print(data_frame_mutate)
```

Output:

```
     col1 Maths Physics Chemistry
1    Jack    26      24        67
2    Jill    47      53        23
3    Yash    14      45        79
4 Mallika    73      88        67
5  Muskan    65      68        33
6  Keshav    83      35        66
7   Meenu    95      78        25
8  Sanjay    48      24        78
[1] "Mutated Data Frame"
      col1   Subject Marks
1     Jack     Maths    26
2     Jill     Maths    47
3     Yash     Maths    14
4  Mallika     Maths    73
5   Muskan     Maths    65
6   Keshav     Maths    83
7    Meenu     Maths    95
8   Sanjay     Maths    48
9     Jack   Physics    24
10    Jill   Physics    53
11    Yash   Physics    45
12 Mallika   Physics    88
13  Muskan   Physics    68
14  Keshav   Physics    35
15   Meenu   Physics    78
16  Sanjay   Physics    24
17    Jack Chemistry    67
18    Jill Chemistry    23
19    Yash Chemistry    79
20 Mallika Chemistry    67
21  Muskan Chemistry    33
22  Keshav Chemistry    66
23   Meenu Chemistry    25
24  Sanjay Chemistry    78
```

Method 8: Using recode() method (Recoding variables – Topic 4.5 of your syllabus)

It recode values in a column i.e. the values of the variables

Data Formatting

In R, data formatting typically involves preparing and structuring your data in a way that is suitable for analysis or visualization. The exact steps for data formatting may vary depending on your specific dataset and the analysis you want to perform.

Formatting data in R is an essential part of data preprocessing and analysis. Depending on our specific needs, we want to manipulate data types, change the structure of our data frame, or format variables. Here are some common tasks related to data formatting in R:

Import Data

Use functions like read.csv(), read.table(), read.xlsx() from packages like readr, readxl, or others to import our data into R. Ensure that our data is in a supported file format such as CSV, Excel, or a database.

Checking Data Structure

Use str() to check the structure of your data frame. This function displays the data type and structure of each column in your data frame.

```
str(your_data_frame)
```

Changing Data Types (Type Conversion - Topic 4.5 of your syllabus)

By using pre-defined functions of R we can convert columns to appropriate data types.

Handling Missing Values (Identifying and handling Missing values - Topic 4.4 of your syllabus)

By using pre-defined functions of R we can identify and handle the missing values in the columns i.e. values of the variables / attributes

Renaming Columns

Rename columns using the colnames() function or by directly assigning new column names to the names() attribute of your data frame.

Formatting Dates and Times

Use functions like as.Date()

Aggregating Data

Use functions like aggregate() or dplyr functions like group_by() and summarize() to aggregate data based on specific variables. We can summarize data using aggregation functions like mean(), sum(), etc.

```
# Calculate the mean mpg for each number of cylinders
mean_marks_by_subjects<- aggregate(data_frame_mutate$Marks, by
          = list(data_frame_mutate$Subject), FUN = mean)
colnames(mean_marks_by_subjects) <- c("Subjects", "MeanMarks")
mean_marks_by_subjects
```

Output:

```
     Subjects MeanMarks
1 Chemistry    54.750
2     Maths    56.375
3   Physics    51.875
```

Selecting Columns

We can select specific columns of the dataset using the $ operator or the subset() function.

Filtering Rows

We can filter rows based on certain conditions using the subset() or [ ] indexing.

Sorting Data

You can sort the data by one or more columns using the order() function.

```
# Sort the data by marks in descending order (- minus sign)
sorted_data<-data_frame_mutate[order(-
                                data_frame_mutate$Marks), ]
 head(sorted_data)
```

Output

```
     col1   Subject Marks
7    Meenu     Maths    95
12 Mallika   Physics    88
6   Keshav     Maths    83
19    Yash Chemistry    79
15   Meenu   Physics    78
24  Sanjay Chemistry    78
```

Creating New Variables

We can create new variables based on existing ones.

```
# Create a new variable "MarksCategory" based on "Marks"
data_frame_mutate$MarksCategory<-
          ifelse(data_frame_mutate$Marks>60, "High", "Low")
data_frame_mutate$MarksCategory
```

Output:

```
[1] "Low"   "Low"   "Low"   "High" "High" "High" "High" "Low"
"Low"  "Low"  "Low"  "High" "High" "Low"  "High" "Low"  "High"
"Low"  "High"
[20] "High" "Low"  "High" "Low"  "High"
```

4.4 Identifying and Handling Missing Values

Use functions like is.na(), complete.cases(), or na.omit() to identify and handle missing values (NA) appropriately. You can choose to remove rows with missing values or impute missing values with mean, median, or other strategies.

```
# Remove rows with missing values
your_data_frame<-
your_data_frame[complete.cases(your_data_frame), ]
# Impute missing values with mean
your_data_frame$numeric_column[is.na(your_data_frame$numeric_c
olumn)] <- mean(your_data_frame$numeric_column, na.rm = TRUE)
```

In data science, one of the common tasks is dealing with missing data. As the name indicates, Missing values are those elements that are not known. NA or NaN are reserved words that indicate a missing value in R Programming language.

If we have missing data in your dataset, there are several ways to handle it in R programming. One way is to simply remove any rows or columns that contain missing data. Another way to handle missing data is to impute the missing values using a statistical method. This means replacing the missing values with estimates based on the other values in the dataset. For example, we can replace missing values with the mean or median value of the variable in which the missing values are found.

Dealing Missing Values in R

Missing Values in R, are handled with the use of some pre-defined functions:

is.na() Function for Finding Missing values:

A logical vector is returned by this function that indicates all the NA values present. It returns a Boolean value. If NA is present in a vector it returns TRUE else FALSE.

Example

```
x<- c(NA, 3, 4, NA, NA, NA)
```

is.na(x)

Output:

```
[1]  TRUE FALSE FALSE  TRUE  TRUETRUE
```

Properties of Missing Values:

- For testing objects that are NA use is.na()
- For testing objects that are NaN use is.nan()
- There are classes under which NA comes. Hence integer class has integer type NA, the character class has character type NA, etc.
- A NaN value is counted in NA but the reverse is not valid.

The creation of a vector with one or multiple NAs is also possible.

```
x<- c(NA, 3, 4, NA, NA, NA)
x
```

Output:

```
[1] NA  3  4 NA NANA
```

Removing NA or NaN values

There are two ways to remove missing values:

Extracting values except for NA or NaN values:

Example 1:

```
x <- c(1, 2, NA, 3, NA, 4)
d <- is.na(x)
x[! d]
```

Output:

```
[1] 1 2 3 4
```

Example 2:

```
x <- c(1, 2, 0 / 0, 3, NA, 4, 0 / 0)
x
x[! is.na(x)]
```

Output:

[1] 1 2 NaN3 NA 4 NaN
[1] 1 2 3 4

A function called **complete.cases()** can also be used. This function also works on data frames.

Missing Value Filter Functions

The modeling functions in R language acknowledge a na.action argument which provides instructions to the function regarding its response if NA comes in its way.

And hence this way the function calls one of the missing value filter functions. Missing Value Filter Functions alter the data set and in the new data set the value of NAs has been changed. The default Missing Value Filter Function is na.omit. It omits every row containing even one NA. Some other Missing Value Filter Functions are:

na.omit– omits every row containing even one NA

na.fail– halts and does not proceed if NA is encountered

na.exclude– excludes every row containing even one NA but keeps a record of their original position

na.pass– it just ignores NA and passes through it

Example

```
# Creating a data frame
```

```
df<- data.frame (c1 = 1:8,
     c2 = factor (c("B", "A", "B", "C","A", "C", "B", "A")))
df
```
Output
```
c1 c2
1  1  B
2  2  A
3  3  B
4  4  C
5  5  A
6  6  C
7  7  B
8  8  A
```

```
# Filling some NA in data frame
df[4, 1] <- df[6, 2] <- NA
df
```
Output
```
c1   c2
1  1    B
2  2    A
3  3    B
4 NA    C
5  5    A
6  6<NA>
7  7    B
8  8    A
```

```
na.omit(df)
```
Output
```
  c1 c2
1  1  B
2  2  A
3  3  B
5  5  A
7  7  B
8  8  A
```
```
# Printing all the levels(NA is not considered one)
levels(df$c2)
```
Output
```
[1] "A" "B" "C"
```

```
# fails if NA is encountered
na.fail (df)
```
Output
```
Error in na.fail.default(df) : missing values in object
```
```
# excludes every row containing even one NA

na.exclude (df)
```
Output:
```
c1 c2
1  1  B
```

```
2  2  A
3  3  B
5  5  A
7  7  B
8  8  A
```

Find and Remove NA or NaN values from a dataset

In R we can remove and find missing values from the entire dataset. there are some main functions we can use and perform the tasks.

First, we will create one data frame and then we will find and remove all the missing values which are present in the data.

Example

```
# Create a data frame with 5 rows and 3 columns
data <- data.frame(A = c(1, 2, NA, 4, 5),
     B = c(NA, 2, 3, NA, 5),
     C = c(1, 2, 3, NA, NA)
)
# View the resulting data frame
data
```

Output:

```
A  B  C
1  1 NA  1
2  2  2  2
3 NA  3  3
4  4 NA NA
5  5  5 NA
```

Find all the missing values in the data

```
# Finding missing values in data.
sum(is.na(data)) # finds the count of missing values
```

Output:

```
[1] 5
```

Find all the missing values in the columns

```
# Finding missing values column wise
colSums(is.na(data))
```

Output:

```
A B C
1 2 2
```

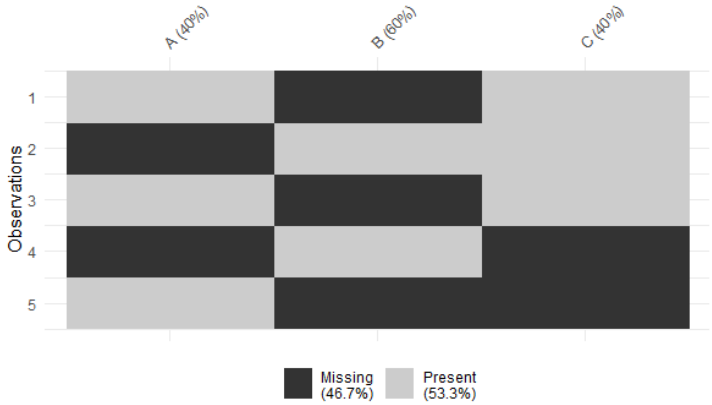
Sometimes values are stored as 99 that you can convert into NA using the following command.

```
df$v1[df$v1==99] <- NA
```

Visualization of missing values of a dataset

```
# Install and load the 'visdat' package
install.packages("visdat")
library(visdat)
# Create a data frame with missing values
data <- data.frame(A = c(1, NA, 3, NA, 5),
          B = c(NA, 2, NA, 4, NA),
          C = c(1, 2, 3, NA, NA)
)
# Plot the missing value diagram
vis_miss(data)
```

Output



Remove missing values from dataframe

```
# Remove missing values using na.omit function.
data<- na.omit(data)
data
```

Output:

```
[1] A B C
<0 rows> (or 0-length row.names)
```

identify NAs in Vector

```
myVector<- c(NA, "TP", 4, 6.7, 'c', NA, 12)
which(is.na(myVector)) # finds the location of missing values
Output
[1] 1 6
```

Selecting values that are not NA or NAN

In order to select only those values which are not missing, firstly we are required to produce a logical vector having corresponding values as True for NA or NAN value and False for other values in the given vector.

Example

```
myVector1 <- c(200, 112, NA, NA, NA, 49, NA, 190)
```

```
logicalVector1 <- is.na(myVector1)
newVector1 = myVector1[! logicalVector1]
print(newVector1)
```
Output
```
[1] 200 112  49 190

myVector2 <- c(100, 121, 0 / 0, 123, 0 / 0, 49, 0 / 0, 290)
logicalVector2 <- is.nan(myVector2)
newVector2 = myVector2[! logicalVector2]
print(newVector2)
```
Output
```
[1] 100 121 123  49 290
```

Filling Missing Values with Mean or Median

In this section, we will see how we can fill or populate missing values in a dataset using mean and median. We will use the apply method to get the mean and median of missing columns.

Step 1 − The very first step is to get the list of columns that contain at least one missing value (NA) value.

Example

```
# Create a data frame
dataframe<- data.frame(
Name = c("Bhuwanesh", "Anil", "Jai", "Naveen"),
   Physics = c(98, 87, 91, 94),
   Chemistry = c(NA, 84, 93, 87),
   Mathematics = c(91, 86, NA, NA) )
#Print dataframe
print(dataframe)
```
Output
```
       Name   Physics Chemistry Mathematics
1 Bhuwanesh        98        NA          91
2      Anil        87        84          86
3       Jai        91        93          NA
4    Naveen        94        87          NA
```

Let's print the column names having at least one NA value.

apply() - returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

```
Syntax: apply(X, MARGIN, FUN, ...)
```

Arguments

X - an array, including a matrix.

MARGIN - a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.

FUN - the function to be applied. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.

... - optional arguments to FUN.

Example

```
listMissingColumns<- colnames(dataframe)[ apply(dataframe, 2,
anyNA)]
print(listMissingColumns)
```

Output

```
[1] "Chemistry"   "Mathematics"
In our dataframe, we have two columns with NA values.
```

Step 2 − Now we are required to compute the mean and median of the corresponding columns. Since we need to omit NA values in the missing columns, therefore, we can pass "na.rm = True" argument to the apply() function.

```
meanMissing<- apply(dataframe[,colnames(dataframe) %in%
listMissingColumns], 2, mean, na.rm =  TRUE)
print(meanMissing)
```

Output

```
Chemistry Mathematics
   88.0          88.5
The mean of Column Chemistry is 88.0 and that of Mathematics is 88.5.
```

Now let's find the median of the columns −

```
medianMissing<- apply(dataframe[,colnames(dataframe) %in%
listMissingColumns], 2, median, na.rm =  TRUE)
print(medianMissing)
```

Output

```
Chemistry Mathematics
87.0        88.5
```

The median of Column Chemistry is 87.0 and that of Mathematics is 88.5.

Step 3 − Now our mean and median values of corresponding columns are ready. In this step, we will replace NA values with mean and median using mutate() function which is defined under "dplyr" package.

Example

```
# Importing library
library(dplyr)
newDataFrameMean<- dataframe %>% mutate(
          Chemistry = ifelse(is.na(Chemistry), meanMissing[1],
          Chemistry),
       Mathematics         =         ifelse(is.na(Mathematics),
       meanMissing[2],
       Mathematics))
newDataFrameMean
```

Output

```
       Name    Physics Chemistry Mathematics
```

```
1 Bhuwanesh          98         88         91.0
2       Anil         87         84         86.0
3        Jai         91         93         88.5
4     Naveen         94         87         88.5
```
Notice the missing values are filled with the mean of the corresponding column.

Example

Now let's fill the NA values with the median of the corresponding column.

```
# Importing library
library(dplyr)
newDataFrameMedian<- dataframe %>% mutate(
   Chemistry = ifelse(is.na(Chemistry), medianMissing[1],
Chemistry),
   Mathematics =  ifelse(is.na(Mathematics), medianMissing[2],
Mathematics))
print(newDataFrameMedian)
```
Output
```
     Name  Physics Chemistry Mathematics
1 Bhuwanesh      98        87         91.0
2       Anil      87        84         86.0
3        Jai      91        93         88.5
4     Naveen      94        87         88.5
```
The missing values are filled with the median of the corresponding column.

Deleting rows containing missing values, lead to a reduction in sample size and avoid some good representative values also.

Delete Missing Data leads to the following major issues

- Data loss
- Bias data

Missing data

Now we need to calculate what **percentage of data** is missing from each variable.

```
p <- function(x) {
     sum(is.na(x))/length(x)*100
     }
apply(dataframe, 2, p) #consider original dataframe from above
example
```
Output
```
Name     Physics   Chemistry Mathematics
0           0          25          50
```

Example

```
#create data frame
df<- data.frame(
team=c('A', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'),
```

```
                  points=c(4, 4, NA, 8, 6, 12, 14, 86, 13, 8),
                  rebounds=c(9, 9, 7, 6, 8, NA, 9, 14, 12, 11),
                  assists=c(2, 2, NA, 7, 6, 6, 9, 10, NA, 14))
df
```

Output

```
   team points rebounds assists
1     A      4        9       2
2     A      4        9       2
3     B     NA        7      NA
4     C      8        6       7
5     D      6        8       6
6     E     12       NA       6
7     F     14        9       9
8     G     86       14      10
9     H     13       12      NA
10    I      8       11      14
```

Remove Rows with Missing Values

```
library(dplyr)
new_df<- df %>% na.omit()
new_df
```

Output

```
   team points rebounds assists
1     A      4        9       2
2     A      4        9       2
4     C      8        6       7
5     D      6        8       6
7     F     14        9       9
8     G     86       14      10
10    I      8       11      14
```

Replace Missing Values with Another Value

The **na.rm** option tells R to remove missing values when it calculates the mean/median.

In R, the median of a vector is calculated using the **median**() function.

replace_na: Replace missing values. If data is a vector, a single value used for replacement.

The **across**() function is designed to apply a calculation or function on one or several columns at once.

Where() select the variable

We can use the following to replace any missing values with the median value of each column:

```
library(dplyr)

library(tidyr)
```

#replace missing values in each numeric column with median value of column

```
new_df<-df %>% mutate(across(where(is.numeric),~replace_na(.,median(.,na.rm=TRUE))))

new_df
```

Output

```
   team points rebounds assists
1     A      4        9     2.0
2     A      4        9     2.0
3     B      8        7     6.5
4     C      8        6     7.0
5     D      6        8     6.0
6     E     12        9     6.0
7     F     14        9     9.0
8     G     86       14    10.0
9     H     13       12     6.5
10    I      8       11    14.0
```

Note that you could also replace median in the formula with mean to instead replace missing values with the mean value of each column.

Note: We also had to load the tidyr package in this example because the drop_na() function comes from this package.

Remove Duplicate Rows

```
library(dplyr)
#remove duplicate rows
new_df<- df %>% distinct(.keep_all=TRUE) # keep all variables
in data
new_df
```

Output

```
  team points rebounds assists
1    A      4        9       2
2    B     NA        7      NA
3    C      8        6       7
4    D      6        8       6
5    E     12       NA       6
6    F     14        9       9
7    G     86       14      10
8    H     13       12      NA
9    I      8       11      14
```

Special Cases

There are two special cases where NA is denoted or presented differently:

- Factor Vectors– is the symbol displayed in factor vectors for missing values.
- NaN – This is a special case of NA only. It is displayed when an arithmetic operation yields a result that is not a number. For example, dividing zero by zero produces NaN.