Introduction to R

Hari Patchigolla

7/17/2023

R is a high-level programming language and open-source software environment extensively utilized in bioinformatics for a wide array of applications. Its popularity in this field stems from its robust data manipulation capabilities and an extensive library of statistical and bioinformatics packages. Researchers in bioinformatics use R to analyze biological data, perform genomic sequencing, identify gene expressions, and conduct statistical analyses on large datasets. Moreover, R's powerful visualization tools enable the creation of compelling graphs and charts to visualize genomic patterns and analyze biological networks. As a result, R plays a vital role in bioinformatics, empowering scientists to unravel complex biological insights and facilitate advancements in genomics, proteomics, and other critical areas of life sciences.

In this section we are going to dive deeper into R. We will cover various topic to help you get familiar with its core concepts:

1) Data Types in R
2) Objects in R
3) Data Management
4) Basic Statistical Functions
5) Visualizations
6) Conditional Statements, Loops, and Functions in R
7) Working with Packages & Libraries within R

Data Types in R

Like Python, R allows you to store and create variables. Thus, variables can be of different types. The syntax for declaring a variable is show below:

```r
x <- 3 # arrow sign
y = 7 # equals sign
```

Notice how in your environment on RStudio (top right) you will see key value pairs of the variable name and its corresponding values. This is extremely useful as it allows you to consistently keep track of your variables and can help to debug.

There are a few basic data types in R: `numeric`, `character`, `logical` , `integer`, `complex`:

-`numeric`: these type of variables can represent integers and decimal numbers - `character`: Represents text or strings of characters. Strings are enclosed in quotes, either single (') or double ("). - `logical`: Represents Boolean values, which can be either `TRUE` or `FALSE` - `Integer`: Represents whole numbers. Integers are a subset of numeric values but have a distinct data type in R. - `complex`: Represents complex numbers with both real and imaginary parts. Complex numbers are written in the form `a + bi`, where `a` represents the real part and `b` represents the imaginary part.

Run the following snippet to get a better understanding of these types of data types. Note that the `class()` function gives you more details on the data type of a variable if you ever need to know more. (We Will cover more about functions later on)

```
x <- 13
class(x) # this will output numeric but it is also an integer!
```

```
[1] "numeric"
```

```
x <- 45.3
class(x)
```

```
[1] "numeric"
```

```
x <- TRUE
class(x)
```

```
[1] "logical"
```

```
x <- "Hello World"
class(x)
```

```
[1] "character"
```

```
x <- 98i
class(x)
```

```
[1] "complex"
```

Quick did you notice that the value of `x` in your environment changed several times? This is important because you can very easily override a variable in R, *so be careful!!*

Arithmetic

These built-in data types provide the core methods of computation. `R` allows various forms of arithmetic such as addition, division, modulo, etc.

```
x <- 10
y <- 3

x + y
```

```
[1] 13
```

```
x - y
```

```
[1] 7
```

```
x * y
```

```
[1] 30
```

```
x / y
```

```
[1] 3.333333
```

```
x^y
```

```
[1] 1000
```

```
y %% x
```

```
[1] 3
```

Hopefully this was relatively simple to follow as the syntax is actually quite similar to Python!

Relational Operators

The similarity between Python and R syntax does not stop there though. The syntax for relational operation is also the same! Relational operators define some kind of a relation between variable like greater than, equal to, etc.

```
x <- 50
y <- 35

x == y
```

```
[1] FALSE
```

```
x != y
```

```
[1] TRUE
```

```
x < y
```

```
[1] FALSE
```

```
x <= y
```

```
[1] FALSE
```

```
y > x
```

```
[1] FALSE
```

```
y >= x
```

```
[1] FALSE
```

Notice how the outputs of these operators are either `TRUE` or `FALSE`. This will come in handy when we talk about control flow in `R` later on in this module, so keep this in mind!

Objects in R

In R, an object is a fundamental concept that represents data or information stored in memory. It serves as a storage box for holding values (like numbers, text, vectors, matrices, lists, and more complex data structures). Each object in R has a unique name assigned to it, allowing for easy reference and manipulation. Objects in R are not just simple variables but are entities with specific attributes and behaviors. They can be created, modified, and interacted with during the execution of an R program. Objects in R enable users to store, organize, and process data efficiently, supporting a wide range of operations and analyses.

So technically simple data types like integers are indeed objects. But bjects can get a lot more complicated. Lets learn of a few:

Vectors

In R, vectors are one-dimensional objects that can store multiple values of the same data type. They are a fundamental data structure and can be created using the `c()` function or generated by other functions. Here's an explanation along with some code examples:

```
numbers <- c(1, 2, 3, 4, 5)
```

Here we create a vector called `numbers`, notice how it is also stored in your `R` environment.

There are other ways to initialize vectors as well:

```
years <- seq(2000, 2023)
years
```

```
 [1] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014
[16] 2015 2016 2017 2018 2019 2020 2021 2022 2023
```

```
(x <- rep(FALSE, 10)) # wrapping in an extra set of parentheses will
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# automatically print out the result

(y <- 1:10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Since vectors are a sequence of numbers you and easily access elements using bracket notation `[]`.

```
years[4]
```

```
[1] 2003
```

```
years[c(5, 7, 9)]
```

```
[1] 2004 2006 2008
```

```r
years[5:9]
```

```
[1] 2004 2005 2006 2007 2008
```

```r
years[seq(3, 5)]
```

```
[1] 2002 2003 2004
```

You can pass in specific index values to extract values from, you can also pass in another vector too!

NOTE: Unlike Python R is not zero index, so the first element of a vector is at index position 1 (not 0)

```r
years[1]
```

```
[1] 2000
```

Matrix

In R, a matrix is a two-dimensional data structure that represents a rectangular grid of elements. It consists of rows and columns, where each element holds a value of the same data type. Matrices are useful for organizing and manipulating tabular data, as well as for performing matrix algebra and mathematical operations.

Unlike vectors, which are one-dimensional, matrices provide a structured way to store data in a grid-like format. The rows of a matrix represent observations or cases, while the columns represent variables or attributes. Each element within the matrix can be accessed using its row and column index.

Matrices in R can contain numeric values, characters, logical values, or other data types. They are created using the `matrix()` function, specifying the data elements, the number of rows (`nrow`), and the number of columns (`ncol`). Once created, elements within a matrix can be accessed, modified, and manipulated using indexing operations.

The matrix function has the following parameters: `(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)`

- `data`: a vector of element for the matrix
- `norw` and `ncol`: the row x col dimensions of the matrix
- `dimnames`: optional names for the rows and cols of your matrix
- `byrow`: when the matrix is filled with `data`, `FALSE` will have data populated by columns and `TRUE` will have the data populated by column

```r
(y <- matrix(50:61, nrow=4, ncol=3))
```

```
     [,1] [,2] [,3]
[1,]   50   54   58
[2,]   51   55   59
[3,]   52   56   60
[4,]   53   57   61
```

```r
(y <- matrix(50:61, nrow=4, ncol=3, byrow=TRUE))
```

```
     [,1] [,2] [,3]
[1,]   50   51   52
[2,]   53   54   55
[3,]   56   57   58
[4,]   59   60   61
```

Below is how you can label the rows and columns

```r
(y <- matrix(50:61,
             nrow=4,
             ncol=3,
             byrow=TRUE,
             dimnames=list(c("Row 1", "Row 2", "Row 3", "Row 4"),
                           c("Col 1", "Col 2", "Col 3"))))
```

```
      Col 1 Col 2 Col 3
Row 1    50    51    52
Row 2    53    54    55
Row 3    56    57    58
Row 4    59    60    61
```

```r
# sometime when you have a lot of parameters it is best to split
# into different lines
```

In case there is ever a function that you do not remember the parameters for you can:

1) Type in the function name to the R console (bottom left)
2) Search it up in the "Help" tab, bottom right

You can also use the `help()` function:

```r
help(matrix)
```

```
starting httpd help server ... done
```

Below we will go over how to index a matrix using the `[row, col]` syntax:

```r
(y[1,]) # this will get the first row in the y matrix
```

```
Col 1 Col 2 Col 3
   50    51    52
```

```r
(y[3,]) # this gets the third row
```

```
Col 1 Col 2 Col 3
   56    57    58
```

You can also use the labels of the rows and columns:

```
y["Row 1",]
```

```
Col 1 Col 2 Col 3
   50    51    52
```

```
y[,"Col 2"]
```

```
Row 1 Row 2 Row 3 Row 4
   51    54    57    60
```

```
y["Row 2", "Col 3"] # you can get a specific element in the matrix
```

```
[1] 55
```

Arthmetic and Relation with Vectors and Matricies

Now that we covered the basics of vectors and matrices lets apply similar concepts that we previously learned to them.

```
x <- c(1,2,3,4,5)

y <- c(6,7,8,9,10)

x + y
```

```
[1]  7  9 11 13 15
```

```
x - y
```

```
[1] -5 -5 -5 -5 -5
```

```
x * y
```

```
[1]  6 14 24 36 50
```

```
x / y
```

```
[1] 0.1666667 0.2857143 0.3750000 0.4444444 0.5000000
```

Notice how for vectors though the syntax doesn't change too much the operation is applied to each element of the vector element wise

```
(x <- matrix(1:12, nrow=4, ncol=3))
```

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

```
(y <- matrix(13:24, nrow=4, ncol=3))
```

```
     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

```
x + y
```

```
     [,1] [,2] [,3]
[1,]   14   22   30
[2,]   16   24   32
[3,]   18   26   34
[4,]   20   28   36
```

```
cat("\n")
```

```
x - y
```

```
     [,1] [,2] [,3]
[1,]  -12  -12  -12
[2,]  -12  -12  -12
[3,]  -12  -12  -12
[4,]  -12  -12  -12
```

```
cat("\n")
```

```
x * y
```

```
     [,1] [,2] [,3]
[1,]   13   85  189
[2,]   28  108  220
[3,]   45  133  253
[4,]   64  160  288
```

```
cat("\n")
```

```
x / y
```

```
           [,1]      [,2]      [,3]
[1,] 0.07692308 0.2941176 0.4285714
[2,] 0.14285714 0.3333333 0.4545455
[3,] 0.20000000 0.3684211 0.4782609
[4,] 0.25000000 0.4000000 0.5000000
```

Notice how even for matrices it is element-wise operations. However matrix multiplication does not actually work like that:

```r
x <- matrix(1:12, nrow=4, ncol=3)
y <- matrix(13:24, nrow=3, ncol=4)
x %*% y
```

```
     [,1] [,2] [,3] [,4]
[1,]  218  263  308  353
[2,]  260  314  368  422
[3,]  302  365  428  491
[4,]  344  416  488  560
```

The `%*%` allows us to perform matrix (you can also use the `crossprod()` function)

As for relational operators the same concepts of element wise apply:

```r
x <- c(2,1,4,3,5)
y <- c(3,1,7,3,8)

x == y
```

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

```r
x < y
```

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

```r
3 %in% x
```

```
[1] TRUE
```

The relational operators apply element wise for vectors. Note the use for the `%in%` operate to test is an element exists in a vector

```r
x <- matrix(1:12, nrow=4, ncol=3)
y <- matrix(13:24, nrow=4, ncol=3)


x == y
```

```
      [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
```

```r
x < y
```

```
     [,1] [,2] [,3]
[1,] TRUE TRUE TRUE
[2,] TRUE TRUE TRUE
[3,] TRUE TRUE TRUE
[4,] TRUE TRUE TRUE
```

```r
3 %in% x
```

```
[1] TRUE
```

The same goes for matrices!!

Data Frames

A dataframe is more general than a matrix in that different columns can contain different modes of data (numeric, character, and so on).

They are similar to Excel sheet or database tables, consisting of rows and columns where each column can contain different types of data. Data frames are particularly useful for working with real-world datasets.

Data frames are the most common data structure you'll deal with in R, so lets get comfortable with them!!

A data frame is created with the `data.frame()` function:

```r
# This Creates a dataframe with column names
df <- data.frame(
  Name = c("John", "Joe", "Mary"),
  Age = c(25, 30, 35),
  City = c("New York", "San Francisco", "Boston")
)
```

In your R environment you would see the `df` variable with a blue drop down arrow (clicking on it will show the column names and values of your data frame), and double clicking will open up a new window that renders the dataset.

Indexing a dataframe is similar to that of a matrix:

```r
df[, "Name"] # by specifying nothing for the rows you are asking for all rows
```

```
[1] "John" "Joe"  "Mary"
```

```r
df[1:2,] # by specifying nothing for the columns you are asking for all columns
```

```
  Name Age          City
1 John  25      New York
2  Joe  30 San Francisco
```

You can also use the `$` to get a specific column (this is more commonly used)

```r
df$Age
```

```
[1] 25 30 35
```

Dataframes offer several methods for manipulating and transforming data. You can add or remove columns, compute new variables based on existing ones, filter rows based on specific conditions, and more.

```
df$Salary <- c(50000, 60000, 70000) # Adding a new column

df <- subset(df, select = -City) # Removing a column

df <- subset(df, Age > 25) # Filtering rows based on a condition
```

Notice the changes in `df` in your environment as you run the above cell.

The `subset()` function, n R, is used to extract a subset of a dataframe or a vector based on specified conditions. It allows you to filter rows based on logical expressions or certain criteria. It takes in the dataframe and the logical expression you want to subset by.

Dataframes can also be useful for when you read in a data table:

```
df <- read.table("TCGA_GBM_LGG_Mutations_all.csv", header=TRUE, sep=",")
head(df)
```

```
  ï..Grade  Project      Case_ID Gender   Age_at_diagnosis
1      LGG TCGA-LGG TCGA-DU-8164   Male 51 years 108 days
2      LGG TCGA-LGG TCGA-QH-A6CY   Male 38 years 261 days
3      LGG TCGA-LGG TCGA-HW-A5KM   Male  35 years 62 days
4      LGG TCGA-LGG TCGA-E1-A7YE Female 32 years 283 days
5      LGG TCGA-LGG TCGA-S9-A6WG   Male 31 years 187 days
6      LGG TCGA-LGG TCGA-DB-A4X9 Female  33 years 78 days
        Primary_Diagnosis  Race    IDH1        TP53        ATRX        PTEN
1  Oligodendroglioma, NOS white MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2            Mixed glioma white MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3        Astrocytoma, NOS white MUTATED     MUTATED     MUTATED NOT_MUTATED
4 Astrocytoma, anaplastic white MUTATED     MUTATED     MUTATED NOT_MUTATED
5 Astrocytoma, anaplastic white MUTATED     MUTATED     MUTATED NOT_MUTATED
6            Mixed glioma white MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
         EGFR         CIC       MUC16      PIK3CA         NF1      PIK3R1
1 NOT_MUTATED NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED     MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
        FUBP1          RB1      NOTCH1        BCOR       CSMD3     SMARCA4
1     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
       GRIN2A        IDH2        FAT4      PDGFRA
1 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
```

This dataset, which can be found here, is from the UCIrvine Machine learning repository. Here is the description of the dataset: `Gliomas are the most common primary tumors of the brain.`

```
They can be graded as LGG (Lower-Grade Glioma) or GBM (Glioblastoma Multiforme) depending
on the histological/imaging criteria. Clinical and molecular/mutation factors are also
very crucial for the grading process. Molecular tests are expensive to help accurately
diagnose glioma patients. In this dataset, the most frequently mutated 20 genes and 3
clinical features are considered from TCGA-LGG and TCGA-GBM brain glioma projects. The
prediction task is to determine whether a patient is LGG or GBM with a given clinical and
molecular/mutation features. The main objective is to find the optimal subset of mutation
genes and clinical features for the glioma grading process to improve performance and
reduce costs.
```

Here we use the `read.table()` function to read a `.csv` file into R as a dataframe. Now all the methods we just talked about are applicable here.

Factors

Factors are used in R to represent categorical or nominal data. They are especially useful when working with variables that have a limited set of distinct values or categories. Factors allow for efficient storage and manipulation of categorical data, and they play a crucial role in statistical analysis and data visualization.

You can create a factor in R using the `factor()` function. The `factor()` function takes a vector of categorical data as input and automatically identifies unique levels or categories.

```r
gender <- c("Male", "Female", "Male", "Male", "Female")
(factor_gender <- factor(gender))
```

```
[1] Male   Female Male   Male   Female
Levels: Female Male
```

In this example, we create a factor called factor_gender from the vector gender, which represents the gender of individuals. The `factor()` function automatically identifies the unique levels, "Male" and "Female," in the gender vector and creates a factor object accordingly.

Factors have several properties and functions that make them particularly useful for handling categorical data.

```r
levels(factor_gender) # This gives you ordering/levels in the factor
```

```
[1] "Female" "Male"
```

```r
table(factor_gender) # Counts the frequency of each category
```

```
factor_gender
Female   Male
     2      3
```

```r
levels(factor_gender) <- c("M", "F", "M", "M", "F") # This renames the factor's
                                                    # levels
```

In this example, we demonstrate some operations on factors. The `levels()` function retrieves the distinct levels or categories of the `factor_gender` factor. The `table()` function provides a frequency count of each level in the factor. Additionally, you can rename factor levels by assigning a new set of level names using the `levels()` function.

Factors can be ordered to represent ordinal variables with a natural ordering of levels. The `ordered()` function is used to create an ordered factor.

```r
rating <- c("Good", "Poor", "Excellent", "Fair", "Good")
(ordered_rating <- ordered(rating, levels = c("Poor", "Fair", "Good",
                                              "Excellent")))
```

```
[1] Good      Poor      Excellent Fair      Good     
Levels: Poor < Fair < Good < Excellent
```

Essentially by ordinal we mean that they are categories with order within them.

Lists

Lists are versatile data structures in R that can hold elements of different data types, including vectors, matrices, data frames, or even other lists. Lists provide a flexible way to organize and store heterogeneous data, making them suitable for various complex data structures and analyses.

You can create a list in R using the list() function. Each element of the list can be of any data type, and they are combined into a single list object.

```r
person <- list(
  name = "John Doe",
  age = 30,
  is_student = TRUE
)
```

We create a list called person using the `list()` function. The list contains three elements: `name`, `age`, and `is_student`. Each element is assigned a value, representing the name of a person, their age, and whether they are a student.

Elements within a list can be accessed using the `$` operator or the bracket (`[ ]`) notation.

```r
(person_name <- person$name)
```

```
[1] "John Doe"
```

```r
(person_age <- person[["age"]])
```

```
[1] 30
```

Lists offer various methods for manipulating and modifying their elements. You can add new elements, remove existing elements, or modify the values of specific elements within a list.

```r
person$city <- "New York" # Adds a new element

person$is_student <- NULL # Removes an element

person$name <- "Jane Smith" # Modifies an element
```

Data Management

As `R` is primarily used for work with data, is this section lets learn of different methods to manipulate data.

We will continue using the Glioma Grading Dataset from above

```
head(df)
```

```
  ï..Grade  Project      Case_ID Gender  Age_at_diagnosis
1      LGG TCGA-LGG TCGA-DU-8164   Male 51 years 108 days
2      LGG TCGA-LGG TCGA-QH-A6CY   Male 38 years 261 days
3      LGG TCGA-LGG TCGA-HW-A5KM   Male  35 years 62 days
4      LGG TCGA-LGG TCGA-E1-A7YE Female 32 years 283 days
5      LGG TCGA-LGG TCGA-S9-A6WG   Male 31 years 187 days
6      LGG TCGA-LGG TCGA-DB-A4X9 Female  33 years 78 days
        Primary_Diagnosis  Race    IDH1        TP53        ATRX        PTEN
1  Oligodendroglioma, NOS white MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2            Mixed glioma white MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3        Astrocytoma, NOS white MUTATED     MUTATED     MUTATED NOT_MUTATED
4 Astrocytoma, anaplastic white MUTATED     MUTATED     MUTATED NOT_MUTATED
5 Astrocytoma, anaplastic white MUTATED     MUTATED     MUTATED NOT_MUTATED
6            Mixed glioma white MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
         EGFR         CIC       MUC16      PIK3CA         NF1      PIK3R1
1 NOT_MUTATED NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED     MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
        FUBP1         RB1      NOTCH1        BCOR       CSMD3     SMARCA4
1     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
       GRIN2A        IDH2        FAT4      PDGFRA
1 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
```

Lets restructure the columns `Age_at_diagnosis` to have a column called `AgeInYears` where we convert everything in terms of years.

```
typeof(df$Age_at_diagnosis)
```

```
[1] "character"
```

Clearly it doesn't make sense for this column to be of a character type, but we need it as a numeric type.

```
years <- as.numeric(gsub(" years.*", "", df$Age_at_diagnosis))
```

```
Warning: NAs introduced by coercion
```

```
days <- as.numeric(gsub(".*years | days", "", df$Age_at_diagnosis))
```

```
Warning: NAs introduced by coercion
```

```
age_in_years <- years + days / 365
df$AgeInYears <- age_in_years
head(df)
```

```
  ï..Grade  Project      Case_ID Gender  Age_at_diagnosis
1      LGG TCGA-LGG TCGA-DU-8164   Male 51 years 108 days
2      LGG TCGA-LGG TCGA-QH-A6CY   Male 38 years 261 days
3      LGG TCGA-LGG TCGA-HW-A5KM   Male  35 years 62 days
4      LGG TCGA-LGG TCGA-E1-A7YE Female 32 years 283 days
5      LGG TCGA-LGG TCGA-S9-A6WG   Male 31 years 187 days
6      LGG TCGA-LGG TCGA-DB-A4X9 Female  33 years 78 days
        Primary_Diagnosis  Race    IDH1        TP53        ATRX        PTEN
1  Oligodendroglioma, NOS white MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2            Mixed glioma white MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3        Astrocytoma, NOS white MUTATED     MUTATED     MUTATED NOT_MUTATED
4 Astrocytoma, anaplastic white MUTATED     MUTATED     MUTATED NOT_MUTATED
5 Astrocytoma, anaplastic white MUTATED     MUTATED     MUTATED NOT_MUTATED
6            Mixed glioma white MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
         EGFR         CIC       MUC16      PIK3CA         NF1      PIK3R1
1 NOT_MUTATED NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED     MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
        FUBP1         RB1      NOTCH1        BCOR       CSMD3     SMARCA4
1     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
2 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
4 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
       GRIN2A        IDH2        FAT4      PDGFRA AgeInYears
1 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED   51.29589
2 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED   38.71507
3 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED   35.16986
4 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED   32.77534
5 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED   31.51233
6 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED   33.21370
```

The above few lines of code will create a column called `AgeInYears`. Lets break it down:

In R, the `gsub()` function is used for pattern matching and replacement in strings. It stands for "global substitution" and is a part of the base R package. The function has the following structure: `gsub(pattern, replacement, vector)`: - `pattern`: The pattern or regular expression you want to match in the string. - `replacement`: The replacement string that will replace the matched pattern. - `vector`: The input vector or string where the pattern matching and replacement will be performed.

In the above code we use the `gsub()` function to remove the substring " years" followed by any characters (`.*`) from the `Age_at_diagnosis` column in the dataframe. We do a similar process for the days.

Then it is nothing but a simple calculation to generate the age in years and then add it back to the df!

```
typeof(df$AgeInYears)
```

```
[1] "double"
```

Notice how the type of the columns is now a numeric type!

Now lets use a boolean mask to create a categorical variable called `Age Range`. But first, what is a boolean mask: A boolean mask on data refers to a logical representation or a logical condition applied to the elements of a dataset. It is typically represented as a boolean (`TRUE/FALSE`) value for each element in the dataset, indicating whether a specific condition is met or not. A boolean mask can be used to filter or select specific elements or rows based on the condition it represents. By applying the boolean mask to the dataset, only the elements or rows that satisfy the condition will be included, while the others will be excluded. This allows for effective data manipulation and analysis by focusing on the desired subset of the data.

For example:

```
df$AgeInYears > 60
```

```
  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE    NA FALSE FALSE
 [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
 [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
 [37] FALSE FALSE FALSE FALSE FALSE    NA FALSE FALSE FALSE  TRUE FALSE FALSE
 [49]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [61]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE
 [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
 [85] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
 [97] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[109] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
[121] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
[133]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[145] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[157] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
[169]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[181] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[193] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[217] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[229] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
[241] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[253] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[265]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[277] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[289] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[301] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[313]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[325] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[337] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[349]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
[361] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[373] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[385] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[397] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

```
[409] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
[421] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[433] FALSE FALSE FALSE FALSE FALSE    NA FALSE FALSE FALSE FALSE FALSE FALSE
[445]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[457] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[469] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[481] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
[493] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[505]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
[517]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
[529] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
[541]  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
[553]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE
[565]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
[577]    NA FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
[589]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
[601]  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
[613]  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
[625]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
[637]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
[649] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
[661]  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE    NA
[673] FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[685]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
[697] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE    NA  TRUE
[709] FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[721]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
[733] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
[745] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[757] FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
[769] FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
[781]  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
[793]  TRUE FALSE    NA  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
[805] FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[817] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
[829] FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
[841] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
[853] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

This logical expression is evaluated on all of the values in the `AgeInYears` columns and returns TRUE or FALSE depending on if the condition is met.

If you look closely at this output you will realize that there are `NA` values. `NA` means that the value at that specific data cell is missing. Why is this the case?

```
sum(is.na(df$AgeInYears))
```

```
[1] 7
```

When running the above line you can see that there are actually 7 missing values in the `AgeInYears` columns. Lets take a look at these rows.

```
df[is.na(df$AgeInYears),] # boolean masking to get the NA values
```

```
    ï..Grade  Project      Case_ID Gender Age_at_diagnosis
10       LGG TCGA-LGG TCGA-DU-A76K   Male         87 years
42       LGG TCGA-LGG TCGA-R8-A6YH     --               --
438      LGG TCGA-LGG TCGA-W9-A837   Male               --
577      GBM TCGA-GBM TCGA-06-0744   Male         67 years
672      GBM TCGA-GBM TCGA-28-2501     --               --
707      GBM TCGA-GBM TCGA-28-2510     --               --
795      GBM TCGA-GBM TCGA-16-1048     --               --
         Primary_Diagnosis  Race        IDH1        TP53        ATRX
10  Oligodendroglioma, NOS white NOT_MUTATED NOT_MUTATED NOT_MUTATED
42                      --    --     MUTATED     MUTATED     MUTATED
438 Oligodendroglioma, NOS white     MUTATED NOT_MUTATED NOT_MUTATED
577           Glioblastoma white NOT_MUTATED     MUTATED NOT_MUTATED
672                     --    -- NOT_MUTATED NOT_MUTATED NOT_MUTATED
707                     --    -- NOT_MUTATED NOT_MUTATED NOT_MUTATED
795                     --    -- NOT_MUTATED NOT_MUTATED NOT_MUTATED
           PTEN        EGFR         CIC       MUC16      PIK3CA         NF1
10  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
42  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
438 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
577 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
672 NOT_MUTATED NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED
707     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
795 NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
         PIK3R1       FUBP1         RB1      NOTCH1        BCOR       CSMD3
10  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
42  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
438 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
577 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
672 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
707 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
795 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
        SMARCA4      GRIN2A        IDH2        FAT4      PDGFRA AgeInYears
10  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
42  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
438 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
577 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
672 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
707 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
795 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED         NA
```

You can see that in rows 10, 42, 438, 577, 672, 707, and 795 the format of the `Age_at_diagnosis` is weird. It takes on values like "–" and are not specified to be any number of days old as well. To handle for these cases we can rewrite the code above as follows:

```
years <- as.numeric(gsub(" years.*", "", df$Age_at_diagnosis))
```

```
Warning: NAs introduced by coercion
```

```
days <- as.numeric(gsub(".*years | days", "", df$Age_at_diagnosis))
```

```
Warning: NAs introduced by coercion
```

```
days[is.na(days)] <- 0  # Assign 0 for missing 'days' values
years[is.na(years)] <- 0 # Assign 0 for missing 'years' values
age_in_years <- years + days / 365
df$AgeInYears <- age_in_years
```

Now if we re-run the same command we can see that there are no missing values in the `AgeInYears` column

```
sum(is.na(df$AgeInYears))
```

```
[1] 0
```

Lets go back to creating the `AgeRange` columns.

```
df$AgeRange[df$AgeInYears < 40] <- "Young"
df$AgeRange[df$AgeInYears >= 40 & df$AgeInYears < 65] <- "Old"
df$AgeRange[df$AgeInYears >= 65] <- "Elder"
```

The expressions within the brackets create a boolean mask to which we can subset the df and replace with a specific value!

```
(df$AgeRange[seq(1,10)])
```

```
 [1] "Old"   "Young" "Young" "Young" "Young" "Young" "Young" "Old"   "Young"
[10] "Elder"
```

Now, lets look at the columns of the `df`

```
names(df) # use the names() function to return the column names
```

```
 [1] "ï..Grade"          "Project"           "Case_ID"
 [4] "Gender"            "Age_at_diagnosis"  "Primary_Diagnosis"
 [7] "Race"              "IDH1"              "TP53"
[10] "ATRX"              "PTEN"              "EGFR"
[13] "CIC"               "MUC16"             "PIK3CA"
[16] "NF1"               "PIK3R1"            "FUBP1"
[19] "RB1"               "NOTCH1"            "BCOR"
[22] "CSMD3"             "SMARCA4"           "GRIN2A"
[25] "IDH2"              "FAT4"              "PDGFRA"
[28] "AgeInYears"        "AgeRange"
```

Notice that one of the columns is “ï..Grade”, I don’t know where this random character came from but it shouldn’t be too hard of a fix:

```
names(df)[1] <- "Grade"
```

If you look back at the column names you can see that the changes have been made:

```
names(df)
```

```
 [1] "Grade"            "Project"          "Case_ID"         
 [4] "Gender"           "Age_at_diagnosis" "Primary_Diagnosis"
 [7] "Race"             "IDH1"             "TP53"            
[10] "ATRX"             "PTEN"             "EGFR"            
[13] "CIC"              "MUC16"            "PIK3CA"          
[16] "NF1"              "PIK3R1"           "FUBP1"           
[19] "RB1"              "NOTCH1"           "BCOR"            
[22] "CSMD3"            "SMARCA4"          "GRIN2A"          
[25] "IDH2"             "FAT4"             "PDGFRA"          
[28] "AgeInYears"       "AgeRange"        
```

Unfortunately, there are a few other columns in the `df` that are "improperly" formatted. For example:

```
unique(df$Race)
```

```
[1] "white"                            "asian"                           
[3] "black or african american"        "--"                              
[5] "not reported"                     "american indian or alaska native"
```

If we look at the unique values of the `Race` columns, which we can do with the `unique()` function, you will once again see the "–" values.

So lets make so that anytime we see a value of "–" we replace it with `NA` so that we don't get confused.

```
df <- replace(df, df == "--", NA)
```

In the above line we use the `replace()` function that takes in a dataframe, a boolean mask, and the value to replace with.

```
unique(df$Race)
```

```
[1] "white"                            "asian"                           
[3] "black or african american"        NA                                
[5] "not reported"                     "american indian or alaska native"
```

Now we see that there is an `NA` value instead of "–" for the unique values in the `Race` columns

```
df[!complete.cases(df), ]
```

```
    Grade  Project      Case_ID Gender Age_at_diagnosis      Primary_Diagnosis
42    LGG TCGA-LGG TCGA-R8-A6YH   <NA>             <NA>                   <NA>
438   LGG TCGA-LGG TCGA-W9-A837   Male             <NA> Oligodendroglioma, NOS
672   GBM TCGA-GBM TCGA-28-2501   <NA>             <NA>                   <NA>
707   GBM TCGA-GBM TCGA-28-2510   <NA>             <NA>                   <NA>
```

```
795   GBM TCGA-GBM TCGA-16-1048   <NA>             <NA>                   <NA>
     Race        IDH1        TP53        ATRX        PTEN        EGFR
42   <NA>     MUTATED     MUTATED     MUTATED NOT_MUTATED NOT_MUTATED
438 white     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
672  <NA> NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
707  <NA> NOT_MUTATED NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED
795  <NA> NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED     MUTATED
            CIC       MUC16      PIK3CA         NF1      PIK3R1       FUBP1
42  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
438 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
672 NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
707 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
795 NOT_MUTATED NOT_MUTATED     MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
            RB1      NOTCH1        BCOR       CSMD3     SMARCA4      GRIN2A
42  NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
438 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
672 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
707 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
795 NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED NOT_MUTATED
           IDH2        FAT4      PDGFRA AgeInYears AgeRange
42  NOT_MUTATED NOT_MUTATED NOT_MUTATED          0    Young
438 NOT_MUTATED NOT_MUTATED NOT_MUTATED          0    Young
672 NOT_MUTATED NOT_MUTATED NOT_MUTATED          0    Young
707 NOT_MUTATED NOT_MUTATED NOT_MUTATED          0    Young
795 NOT_MUTATED NOT_MUTATED NOT_MUTATED          0    Young
```

Here we use the `complete.cases()` function, which returns a logical vector indicating which rows are complete (no NAs) or incomplete (contain at least one NA). By negating the result with `!`, we get a boolean mask for rows with at least one NA value.

The output includes all the rows with at least one missing values, thankfully there aren't that many.

When cleaning and manipulating a dataset, the way you deal with missing values can be very important. This area of missing value imputation has various strategies from imputing with the mean or median, to using machine learning models to estimate and replace missing values. This is however outside the scope of this tutorial, but here is a good link to learn more!

For now we will just go ahead and delete all these row with a missing value using the `na.omit()` function:

```r
df <- na.omit(df)
```

Lets look at the data types of all the columns int eh `df`

```r
str(df)
```

```
'data.frame':   857 obs. of  29 variables:
 $ Grade            : chr  "LGG" "LGG" "LGG" "LGG" ...
 $ Project          : chr  "TCGA-LGG" "TCGA-LGG" "TCGA-LGG" "TCGA-LGG" ...
 $ Case_ID          : chr  "TCGA-DU-8164" "TCGA-QH-A6CY" "TCGA-HW-A5KM" "TCGA-E1-A7YE" ...
 $ Gender           : chr  "Male" "Male" "Male" "Female" ...
 $ Age_at_diagnosis : chr  "51 years 108 days" "38 years 261 days" "35 years 62 days" "32 years 283 day
 $ Primary_Diagnosis: chr  "Oligodendroglioma, NOS" "Mixed glioma" "Astrocytoma, NOS" "Astrocytoma, ana
 $ Race             : chr  "white" "white" "white" "white" ...
 $ IDH1             : chr  "MUTATED" "MUTATED" "MUTATED" "MUTATED" ...
 $ TP53             : chr  "NOT_MUTATED" "NOT_MUTATED" "MUTATED" "MUTATED" ...
 $ ATRX             : chr  "NOT_MUTATED" "NOT_MUTATED" "MUTATED" "MUTATED" ...
 $ PTEN             : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ EGFR             : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ CIC              : chr  "NOT_MUTATED" "MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ MUC16            : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "MUTATED" ...
 $ PIK3CA           : chr  "MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ NF1              : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ PIK3R1           : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "MUTATED" ...
 $ FUBP1            : chr  "MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ RB1              : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ NOTCH1           : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ BCOR             : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ CSMD3            : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ SMARCA4          : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ GRIN2A           : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ IDH2             : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ FAT4             : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "MUTATED" ...
 $ PDGFRA           : chr  "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" "NOT_MUTATED" ...
 $ AgeInYears       : num  51.3 38.7 35.2 32.8 31.5 ...
 $ AgeRange         : chr  "Old" "Young" "Young" "Young" ...
 - attr(*, "na.action")= 'omit' Named int [1:5] 42 438 672 707 795
  ..- attr(*, "names")= chr [1:5] "42" "438" "672" "707" ...
```

clearly, the columns labelled as “chr” should be factors, so lets convert them!

```r
categorical_cols <- names(df)[names(df) != "AgeInYears"]
df[categorical_cols] <- lapply(df[categorical_cols], factor)
str(df)
```

```
'data.frame':   857 obs. of  29 variables:
 $ Grade            : Factor w/ 2 levels "GBM","LGG": 2 2 2 2 2 2 2 2 2 2 ...
 $ Project          : Factor w/ 2 levels "TCGA-GBM","TCGA-LGG": 2 2 2 2 2 2 2 2 2 2 ...
 $ Case_ID          : Factor w/ 857 levels "TCGA-02-0003",..: 478 730 690 516 772 386 705 564 657 495 .
 $ Gender           : Factor w/ 2 levels "Female","Male": 2 2 2 1 2 1 1 1 1 2 ...
 $ Age_at_diagnosis : Factor w/ 837 levels "14 years 154 days",..: 410 223 174 126 104 148 177 315 142 8
 $ Primary_Diagnosis: Factor w/ 6 levels "Astrocytoma, anaplastic",..: 6 4 2 1 1 4 5 4 6 6 ...
 $ Race             : Factor w/ 5 levels "american indian or alaska native",..: 5 5 5 5 5 5 5 5 5 5 ...
 $ IDH1             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 1 1 1 1 1 1 1 1 1 2 ...
 $ TP53             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 1 1 1 2 1 1 1 2 ...
 $ ATRX             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 1 1 1 1 2 1 1 2 ...
 $ PTEN             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ EGFR             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ CIC              : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 1 2 2 2 2 2 2 2 2 ...
 $ MUC16            : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 1 2 2 2 2 2 2 ...
 $ PIK3CA           : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 1 2 2 2 2 2 2 2 2 2 ...
 $ NF1              : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ PIK3R1           : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 1 2 2 2 2 2 2 ...
 $ FUBP1            : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 1 2 2 2 2 2 2 2 2 2 ...
 $ RB1              : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ NOTCH1           : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ BCOR             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ CSMD3            : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ SMARCA4          : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ GRIN2A           : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ IDH2             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ FAT4             : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 1 2 2 2 2 2 2 ...
 $ PDGFRA           : Factor w/ 2 levels "MUTATED","NOT_MUTATED": 2 2 2 2 2 2 2 2 2 2 ...
 $ AgeInYears       : num  51.3 38.7 35.2 32.8 31.5 ...
 $ AgeRange         : Factor w/ 3 levels "Elder","Old",..: 2 3 3 3 3 3 3 2 3 1 ...
 - attr(*, "na.action")= 'omit' Named int [1:5] 42 438 672 707 795
  ..- attr(*, "names")= chr [1:5] "42" "438" "672" "707" ...
```

Here we use boolean masking, yet again, to get all the columns, but the `AgeInYear` one, and then use the `lapply()` function to apply the `factor()` function to all of the categorical columns.

Basic Statistical Functions

In this section we will cover some basic functions you can use to learn more about your data.

The `mean()` and `median()` functions are used to calculate the mean and median of a vector:

```
(mean(df$AgeInYears))
```

```
[1] 50.92447
```

```
(median(df$AgeInYears))
```

```
[1] 51.55068
```

The mean and median are pretty close to each other indicating that there are not that many outliers in the dataset.

We can also use the `sd()` for standard deviation or `var()` for variance to understand the spread of our dataset.

```
(sd(df$AgeInYears))
```

```
[1] 15.7328
```

```
(var(df$AgeInYears))
```

```
[1] 247.5209
```

Here are a few more functions to let you learn more about your data:

```
(sum(df$AgeInYears)) # sum of all the values in the column
```

```
[1] 43642.27
```

```
(min(df$AgeInYears)) # minimum of all the values in the column
```

```
[1] 14.42192
```

```r
(max(df$AgeInYears)) # maximum of all the values in the column
```

```
[1] 89.28767
```

```r
(range(df$AgeInYears)) # the minimum and the maximum of the column
```

```
[1] 14.42192 89.28767
```

The `quantile()` function is also pretty useful. For a probability (p) 0 to 1, `quantile(x,probs=p)` returns a number Q such that 100p% of the sample are less than Q.

```r
(quantile(df$AgeInYears, probs = 0.3))
```

```
     30%
40.11123
```

30% of the samples in the dataset are less the 40.11 years old.

To get these results of all the columns we can use the `summary()` function. This function provides a summary of the distribution of each variable in a dataframe or a numeric vector. It includes the minimum, 1st quartile, median, mean, 3rd quartile, and maximum values. And for categorical (factor) variables it will return the counts for all of the categories.

```r
summary(df)
```

```
 Grade         Project             Case_ID        Gender
 GBM:360   TCGA-GBM:360   TCGA-02-0003:  1   Female:359
 LGG:497   TCGA-LGG:497   TCGA-02-0033:  1   Male  :498
                          TCGA-02-0047:  1
                          TCGA-02-0055:  1
                          TCGA-02-2466:  1
                          TCGA-02-2470:  1
                          (Other)     :851
         Age_at_diagnosis                     Primary_Diagnosis
 64 years 298 days:  3     Astrocytoma, anaplastic      :129
 30 years 32 days :  2     Astrocytoma, NOS             : 58
 31 years 187 days:  2     Glioblastoma                 :360
 35 years 68 days :  2     Mixed glioma                 :128
 38 years 13 days :  2     Oligodendroglioma, anaplastic: 75
 38 years 203 days:  2     Oligodendroglioma, NOS       :107
 (Other)          :844
                              Race            IDH1             TP53
 american indian or alaska native:  1   MUTATED    :412   MUTATED    :353
 asian                           : 14   NOT_MUTATED:445   NOT_MUTATED:504
 black or african american       : 59
 not reported                    : 18
 white                           :765


         ATRX              PTEN              EGFR              CIC
 MUTATED    :219   MUTATED    :143   MUTATED    :113   MUTATED    :114
 NOT_MUTATED:638   NOT_MUTATED:714   NOT_MUTATED:744   NOT_MUTATED:743




         MUC16             PIK3CA              NF1            PIK3R1
 MUTATED    : 99   MUTATED    : 76   MUTATED    : 69   MUTATED    : 57
 NOT_MUTATED:758   NOT_MUTATED:781   NOT_MUTATED:788   NOT_MUTATED:800




         FUBP1               RB1            NOTCH1             BCOR
 MUTATED    : 47   MUTATED    : 41   MUTATED    : 38   MUTATED    : 29
 NOT_MUTATED:810   NOT_MUTATED:816   NOT_MUTATED:819   NOT_MUTATED:828




         CSMD3            SMARCA4           GRIN2A             IDH2
 MUTATED    : 28   MUTATED    : 28   MUTATED    : 27   MUTATED    : 23
 NOT_MUTATED:829   NOT_MUTATED:829   NOT_MUTATED:830   NOT_MUTATED:834




          FAT4             PDGFRA        AgeInYears      AgeRange
 MUTATED    : 23   MUTATED    : 22   Min.   :14.42   Elder:177
 NOT_MUTATED:834   NOT_MUTATED:835   1st Qu.:38.02   Old  :424
                                     Median :51.55   Young:256
                                     Mean   :50.92
                                     3rd Qu.:62.77
                                     Max.   :89.29
```

Visualizations

R's graphing capabilities enable users to transform data into informative and visually appealing representations, such as bar charts, pie charts, histograms, and more. In this section we will learn some basic charting abilities within R

Bar Plot

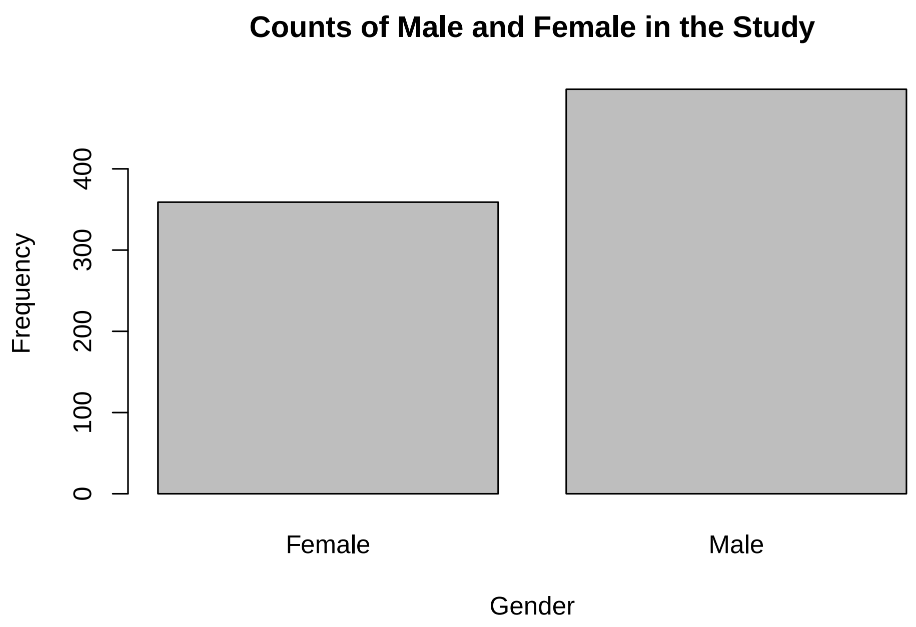
1) Bar plots show the frequency of a categorical variable
2) You should only use this if the number of categories are low

We use the `barplot(vector)` function to generate a bar plot where `vector` is the vector of frequencies.

```r
(counts <- table(df$Gender))
```

```
Female   Male 
   359    498 
```

```r
barplot(counts,
        main="Counts of Male and Female in the Study", #title or the chart
        xlab="Gender", # x-axis label
        ylab="Frequency") # y-axis label
```



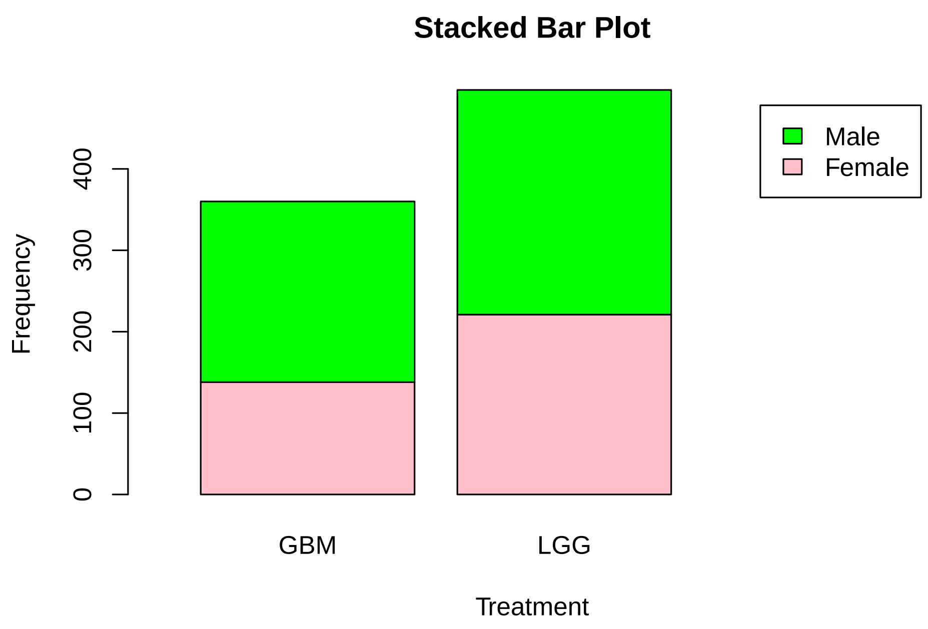
This chart shows that there are more Male participants that female participants in the study

We can create a little more for a fancier bar plot called a stacked bar plot:

```r
(counts <- table(df$Gender, df$Grade))
```

```
        
         GBM LGG
  Female 138 221
  Male   222 276
```

```r
barplot(counts,
        xlim=c(0,3.5),
        main="Stacked Bar Plot",
        xlab="Treatment", ylab="Frequency",
        col=c("pink","green"),
        legend=rownames(counts))
```



This stacked bar plot shows us which proportion of the Glioma Grading types are Male and Female

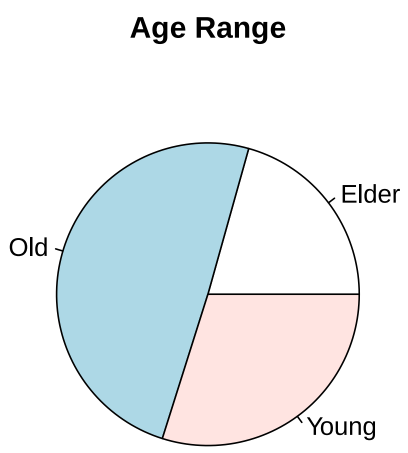
Pie Charts

1) These kind of charts are used to show proportions of different categories out of 100%.

We can use the `pie(vector)` function to generate a pie chart in which `vector` is the sequence of variables.

```
freq <- table(df$AgeRange)

pie(freq, main="Age Range")
```



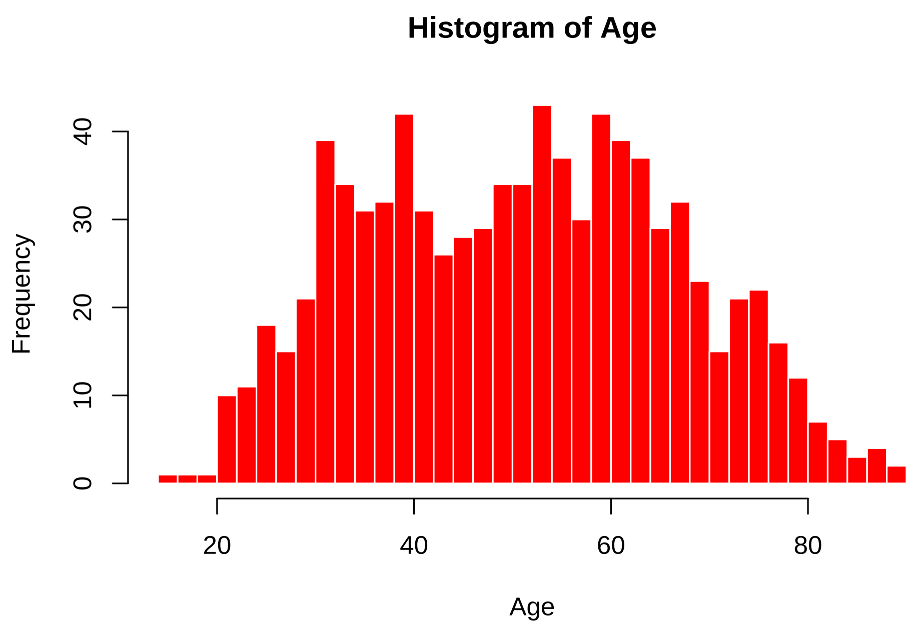
Clearly the majority of the participants in the dataset are "Old".

Histograms

1) A histogram displays the distribution of a continuous variable by dividing the range of values into a number of bins (groups of data points) on the x-axis and displaying the frequency of each bin on the y-axis.

We use the `hist(vector)` function to generate histograms which take as an input a `vector` of numbers.

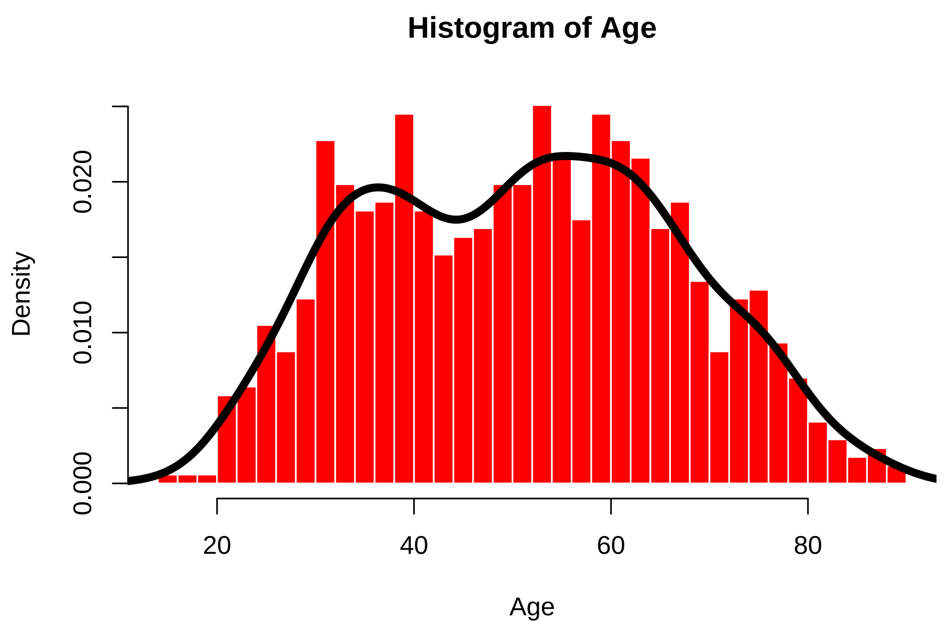
```
hist(df$AgeInYears,
     breaks=50,
     main="Histogram of Age",
     col="red",
     border="white",
     xlab = "Age")
```



While histograms show the distribution of a column, to see a better representation of the distribution we can use the kernel density plot, via the `density()` function.

A kernel density plot works by estimating the density of data points at different values along the variable's range.

```
hist(df$AgeInYears,
     breaks=50,
     main="Histogram of Age",
     col="red",
     border="white",
     xlab = "Age",
     freq=FALSE)
lines(density(df$AgeInYears), col="black", lwd=5)
```



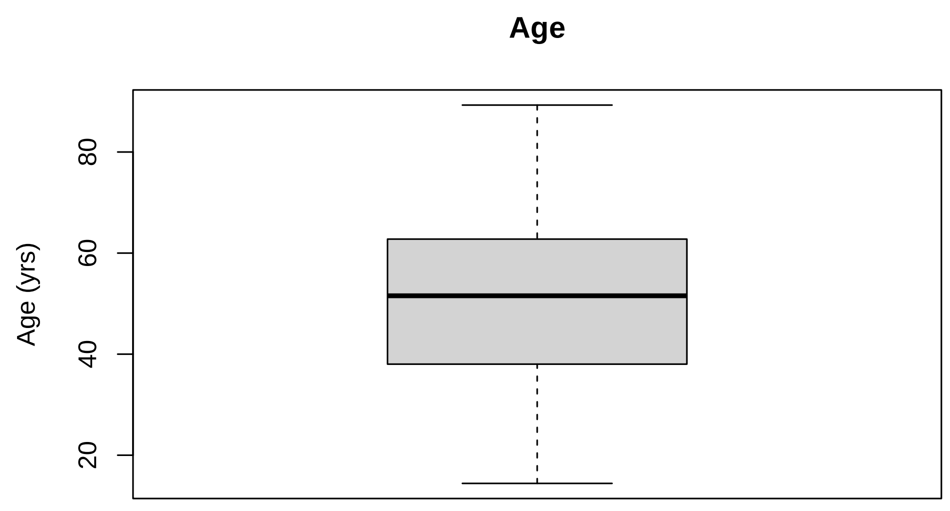
The data is bi-modally distributed.

Box & Whisker plots

1) These plot can also describe the distribution of a continuous variable (column) by giving a visual of the 5 number summary (minimum, lower quartile (25th percentile), median (50th percentile), upper quartile (75th percentile), and maximum).

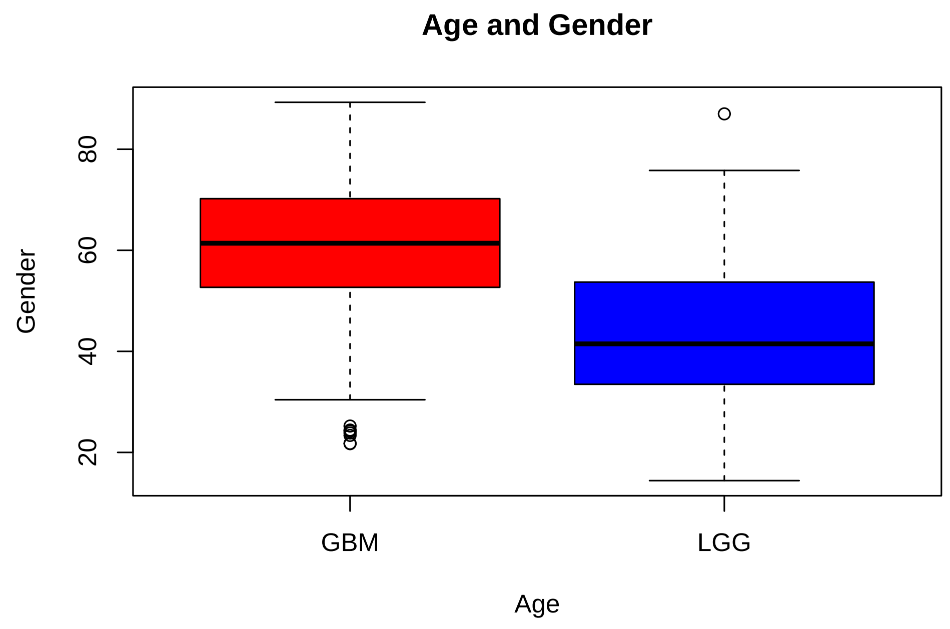
We use the `boxplot(vector)` to display a box and whisker plot:

```
boxplot(df$AgeInYears,
        main="Age",
        ylab="Age (yrs)")
```



You can also create a box plot based on a group:

```
boxplot(AgeInYears~Grade,
        data=df,
        main="Age and Gender",
        xlab="Age",
        ylab="Gender",
        col = c("red", "blue"))
```



The ages of the participants with with GBM seem to be older than those with LGG.

AgeInYears ~ Gender is a formula notation used to specify a relationship or model between the dependent variable AgeInYears and the independent variable Gender in a statistical analysis or data modeling context.

The tilde (~) operator in the formula notation separates the dependent variable from the independent variable(s). In this case, AgeInYears is the dependent variable, and Gender is the independent variable.

When using this formula notation, it implies that you want to study or model the relationship between AgeInYears and Gender. It suggests that you are interested in understanding how the variable Gender might influence or be associated with the variable AgeInYears.

Conditional Statements, Loops, and Functions in R

Conditional Statements

Conditional statements allow you to execute different blocks of code based on certain conditions. In R, common conditional statements include if, if-else, if-else if-else, and the switch statement. These statements evaluate conditions and perform specific actions accordingly.

```r
x <- 10
if (x > 0) {
  print("x is positive")
}
```

```
[1] "x is positive"
```

```r
x <- -5
if (x > 0) {
  print("x is positive")
} else {
  print("x is non-positive")
}
```

```
[1] "x is non-positive"
```

```r
x <- 0
if (x > 0) {
  print("x is positive")
} else if (x < 0) {
  print("x is negative")
} else {
  print("x is zero")
}
```

```
[1] "x is zero"
```

```r
day <- 2
weekday <- switch(day,
                  "Monday",
                  "Tuesday",
                  "Wednesday",
                  "Thursday",
                  "Friday",
                  "Saturday",
                  "Sunday")
print(weekday)
```

```
[1] "Tuesday"
```

The if, if-else, and if-else if-else statements evaluate conditions and execute the corresponding code blocks based on the results, similar to the previous explanation.

The switch statement evaluates an expression (day in the example) and matches it to different cases. It executes the code block corresponding to the matched case. In this example, if day is 2, it assigns "Tuesday" to the weekday variable.

NOTE: We are using the `print()` function to also display the output.

Loops

Loops are used to repeatedly execute a block of code. R supports for loops and while loops, providing flexibility for different iteration scenarios.

```r
for (i in 1:5) {
  print(paste("For loop Iteration", i))
}
```

```
[1] "For loop Iteration 1"
[1] "For loop Iteration 2"
[1] "For loop Iteration 3"
[1] "For loop Iteration 4"
[1] "For loop Iteration 5"
```

```r
i <- 1
while (i <= 5) {
  print(paste("While loop Iteration", i))
  i <- i + 1
}
```

```
[1] "While loop Iteration 1"
[1] "While loop Iteration 2"
[1] "While loop Iteration 3"
[1] "While loop Iteration 4"
[1] "While loop Iteration 5"
```

The for loop iterates over a sequence (1:5 in the example), executing the code block for each iteration. It prints "For loop Iteration 1" through "For loop Iteration 5".

The while loop continues executing the code block as long as the condition remains TRUE. It increments the i variable within the loop, and the loop terminates when i becomes greater than 5.

Functions

Functions in R allow you to encapsulate reusable blocks of code, making your programs more modular and organized. You can define your own functions or use built-in functions from packages.

```r
calculate_sum <- function(a, b) {
  sum <- a + b
  return(sum)
}

result <- calculate_sum(3, 5)
print(result)
```

```
[1] 8
```

Functions can be used to perform specific tasks, enhance code reusability, and promote efficient programming practices.

Working with Packages & Libraries within R

R's package ecosystem is a vast collection of pre-built tools, functions, and libraries contributed by the R community. These packages extend the functionality of the base R system and offer specialized tools for various tasks in data analysis, visualization, machine learning, and more. The package ecosystem in R is one of its key strengths, providing users with a wide range of options to efficiently perform data analysis tasks.

For this section we will be using the famous `seqinr` package. The `seqinr` library is a comprehensive R package specifically designed for the analysis and manipulation of biological sequence data, including DNA, RNA, and protein sequences. It offers a wide range of functionalities that allows bioinformaticians to perform various sequence-related tasks, such as computing sequence statistics (e.g., length, GC content), performing pairwise sequence alignments, and searching for sequence motifs

To make use of specific packages, you need to install them first. The `install.packages()` function in R allows you to download and install packages from the comprehensive CRAN (Comprehensive R Archive Network) repository or other repositories. For example:

```r
# install.packages("seqinr"), run this to install the package
```

Once a package is installed, you can load it into your R session using the library() or require() functions. This makes the package's functions, data sets, and other resources available for use in your code. For example:

```r
library(seqinr)
```

```
Warning: package 'seqinr' was built under R version 4.1.3
```

The `seqinr` comes with built-in datasets that allow us to practice without the need for external data. Let's explore one of these datasets, the `fasta` dataset, which contains DNA sequences for the OPA protein-coding genes:

Lets load in this dataset:

```r
data(fasta)
DNA_SEQ <- fasta[3]
first_seq <- DNA_SEQ$seq[[1]][1]
second_seq <- DNA_SEQ$seq[[2]][1]
```

Here we use the `translate()` function to translate the sequence to its amino acid sequence.

```r
protein_sequence <- translate(unlist(strsplit(first_seq, "")))
protein_sequence
```

```
  [1] "M" "M" "S" "A" "E" "P" "P" "S" "S" "Q" "P" "Y" "I" "S" "D" "V" "L" "R"
 [19] "R" "Y" "Q" "L" "E" "R" "F" "Q" "C" "A" "F" "A" "S" "S" "M" "T" "I" "K"
 [37] "D" "L" "L" "A" "L" "Q" "P" "E" "D" "F" "N" "R" "Y" "G" "V" "V" "E" "A"
 [55] "M" "D" "I" "L" "R" "L" "R" "D" "A" "I" "E" "Y" "I" "K" "A" "N" "P" "L"
 [73] "P" "A" "S" "R" "S" "G" "S" "D" "V" "L" "D" "N" "D" "G" "D" "G" "D" "G"
 [91] "D" "D" "S" "T" "P" "E" "G" "K" "E" "G" "C" "S" "T" "E" "R" "R" "R" "Q"
[109] "Y" "T" "A" "R" "G" "T" "T" "V" "L" "C" "R" "S" "T" "D" "T" "A" "E" "E"
[127] "V" "K" "R" "K" "S" "R" "I" "L" "V" "A" "I" "R" "K" "R" "P" "L" "S" "A"
[145] "G" "E" "Q" "T" "N" "G" "F" "T" "D" "I" "M" "D" "A" "D" "N" "S" "G" "E"
[163] "I" "V" "L" "K" "E" "P" "K" "V" "K" "V" "D" "L" "R" "K" "Y" "T" "H" "V"
[181] "H" "R" "F" "F" "F" "D" "E" "V" "F" "D" "E" "A" "C" "D" "N" "V" "D" "V"
[199] "Y" "N" "R" "A" "A" "R" "A" "L" "I" "D" "T" "V" "F" "D" "G" "G" "C" "A"
[217] "T" "C" "F" "A" "Y" "G" "Q" "T" "G" "S" "G" "K" "T" "H" "T" "M" "L" "G"
[235] "K" "G" "P" "E" "P" "G" "L" "Y" "A" "L" "A" "A" "K" "D" "M" "F" "D" "R"
[253] "L" "T" "S" "D" "T" "R" "I" "V" "V" "S" "F" "Y" "E" "I" "Y" "S" "G" "K"
[271] "L" "F" "D" "L" "L" "N" "G" "R" "R" "P" "L" "R" "A" "L" "E" "D" "D" "K"
[289] "G" "R" "V" "N" "I" "R" "G" "L" "T" "E" "H" "C" "S" "T" "S" "V" "E" "D"
[307] "L" "M" "T" "I" "I" "D" "Q" "G" "S" "G" "V" "R" "S" "C" "G" "S" "T" "G"
[325] "A" "N" "D" "T" "S" "S" "R" "S" "H" "A" "I" "L" "E" "I" "K" "L" "K" "A"
[343] "K" "R" "T" "S" "K" "Q" "S" "G" "K" "F" "T" "F" "I" "D" "L" "A" "G" "S"
[361] "E" "R" "G" "A" "D" "T" "V" "D" "C" "A" "R" "Q" "T" "R" "L" "E" "G" "A"
[379] "E" "I" "N" "K" "S" "L" "L" "A" "L" "K" "E" "C" "I" "R" "F" "L" "D" "Q"
[397] "N" "R" "K" "H" "V" "P" "F" "R" "G" "S" "K" "L" "T" "E" "V" "L" "R" "D"
[415] "S" "F" "I" "G" "N" "C" "R" "T" "V" "M" "I" "G" "A" "V" "S" "P" "S" "N"
[433] "N" "N" "A" "E" "H" "T" "L" "N" "T" "L" "R" "Y" "A" "D" "R" "V" "K" "E"
[451] "L" "K" "R" "N" "A" "T" "E" "R" "R" "T" "V" "C" "M" "P" "D" "D" "Q" "E"
[469] "E" "A" "F" "F" "D" "T" "T" "E" "S" "R" "P" "P" "S" "R" "R" "T" "T" "T"
[487] "R" "L" "S" "T" "A" "A" "P" "L" "F" "S" "G" "S" "S" "T" "A" "A" "P" "A"
[505] "L" "R" "S" "T" "L" "L" "S" "S" "R" "S" "V" "N" "T" "L" "S" "P" "S" "S"
[523] "Q" "A" "K" "S" "T" "L" "V" "T" "P" "K" "P" "P" "S" "R" "D" "R" "T" "P"
[541] "D" "M" "V" "C" "T" "K" "R" "P" "R" "D" "S" "D" "R" "S" "G" "E" "D" "E"
[559] "V" "V" "A" "R" "P" "S" "G" "R" "P" "S" "F" "K" "R" "F" "E" "S" "G" "A"
[577] "E" "L" "V" "A" "A" "Q" "R" "S" "R" "V" "I" "D" "Q" "Y" "N" "A" "Y" "L"
[595] "E" "T" "D" "M" "N" "C" "I" "K" "E" "E" "Y" "Q" "V" "K" "Y" "D" "A" "E"
[613] "Q" "M" "N" "A" "N" "T" "R" "S" "F" "V" "E" "R" "A" "R" "L" "L" "V" "S"
[631] "E" "K" "R" "R" "A" "M" "E" "S" "F" "L" "T" "Q" "L" "E" "E" "L" "D" "K"
[649] "I" "A" "Q" "Q" "V" "A" "D" "I" "T" "A" "F" "Q" "Q" "H" "L" "P" "P" "T"
```