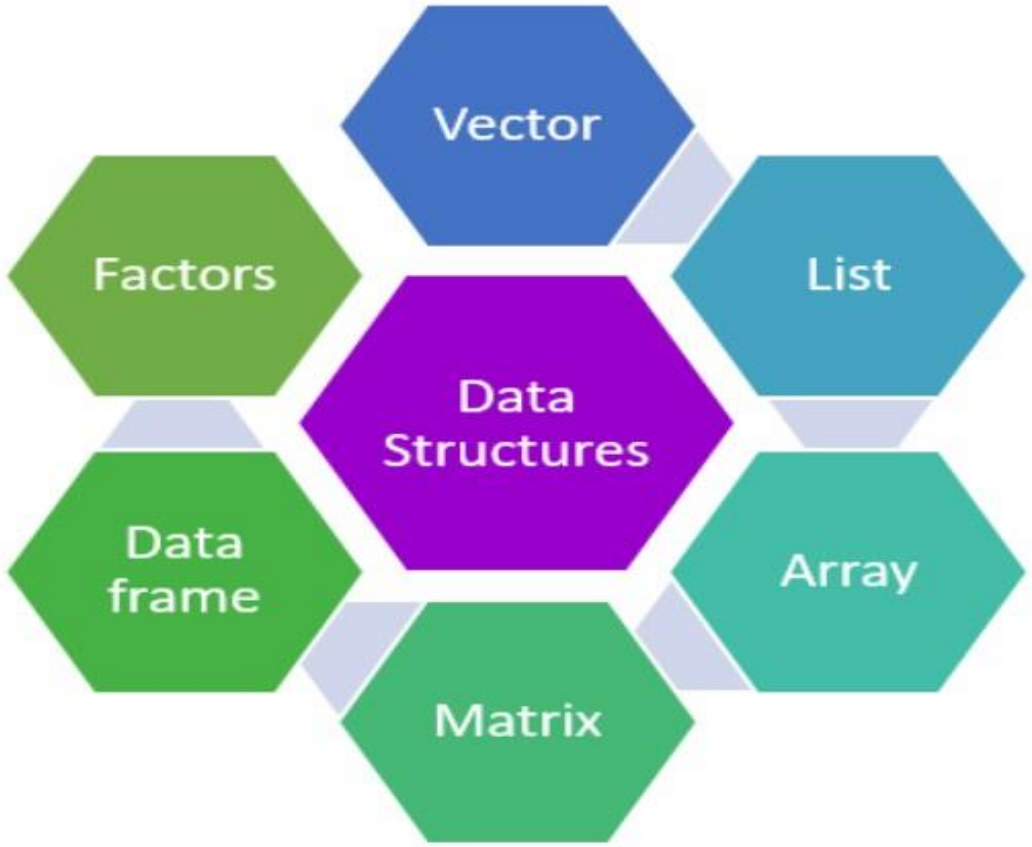
Data Structures in R

- Like all other programming languages, **R** also has its own data structure which is a fundamental concept that every learner, developer, and researcher should know.
- A data structure is a collection of data that are composed of similar or different basic data types.
- The following are the different types of Data Structures in R:
    1. Vector
    2. List
    3. Arrays
    4. Matrix
    5. Data frame
    6. Strings
    7. Factors

Data Structures



What is a Vector

- When you start to write your own functions into R, you need to learn about vectors. If you've learned R in a more traditional way, you're probably already familiar with vectors, as most R resources start with vectors.
- Vectors are particularly important as most of the functions you will write will work with vectors.
- The fundamental data type in R is the *vector*. A vector is a collection of elements, all of the same type.

  Eg. c(1, 3, 2, 1, 5) is a vector consisting of the numbers 1, 3, 2, 1, 5.

  c("R", "Excel", "SAS", "Excel") is a vector of the character elements.

What is a Vector

- In R, a sequence of elements that share the same data type is known as vector.
- The elements which are contained in the vector are known as **components** of the vector.
- How to create a vector in R?
  1. **Using the colon(:) operator:** We can create a vector with the help of the colon operator. There is the following syntax to use colon operator:

     Syntax: z<-**x : y** / z<- **-x:-y** /z<- **x:-y** / z<- **x: y**

     Example: a<**-4:-10**

```
          a
Output: [1]  4  3  2  1  0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
```

2. **Using the seq() function:** We can create a vector with the help of the **seq()** function. A sequence function creates a sequence of elements as a vector. The **seq()** function is used in two ways,

   - by setting step size with **by** parameter or
   - specifying the length of the vector with the '**length.out**' feature.

   Example: seq_vec<-**seq**(1,4,by=0.5)

   seq_vec

   class(seq_vec)

   seq_vec<-**seq**(1,4,length.out=6)

   seq_vec

   class(seq_vec)

3. Using the c() function: We can create a vector with the help of the **c()** function. This type of creating vectors with the help of **c()** is called as Atomic vectors.

Example: c(1,5,6,0,2)

c(“StudID”, “Roll No”, “Section”, “Branch”)

c(“R”, 1, TRUE)

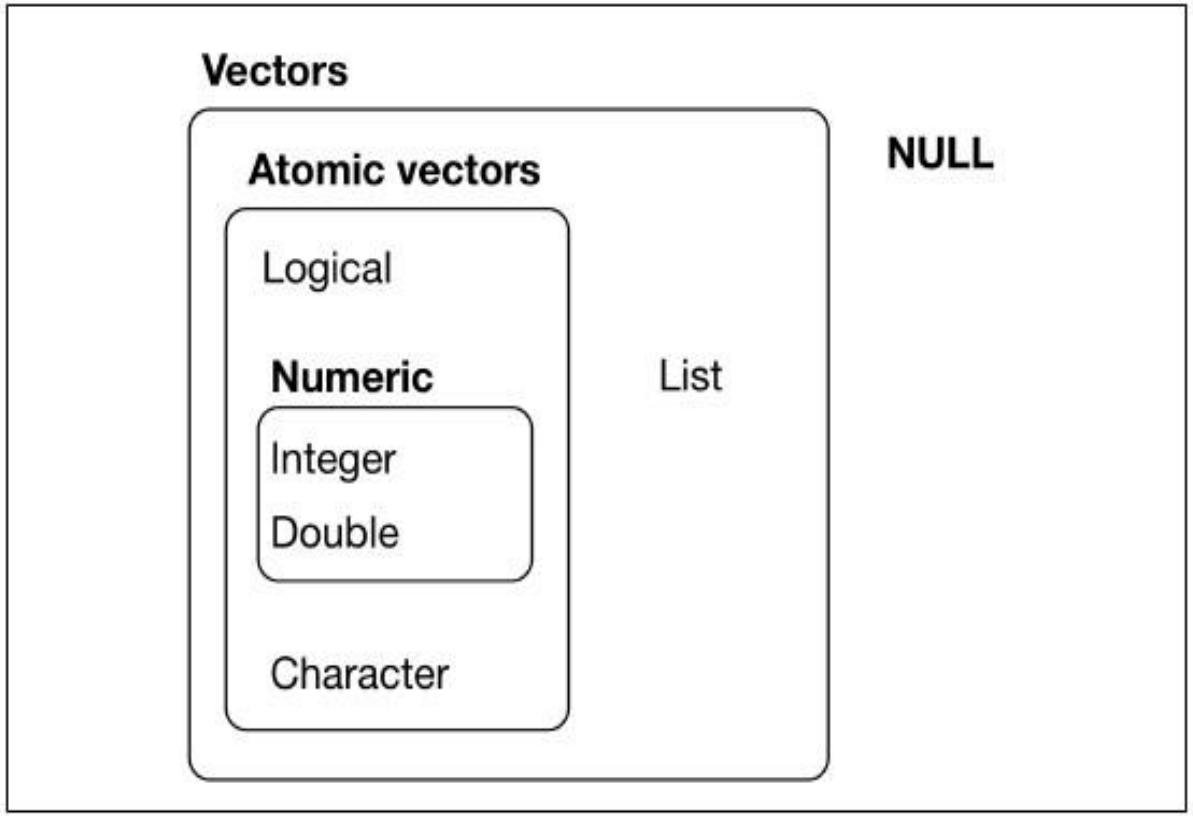
Vector Basics

- There are two types of vectors:

1. ***Atomic*** vectors, of which there are six types: *logical*, *integer*, *double*, *character*, *complex*, and *raw*. Integer and double vectors are collectively known as *numeric* vectors.
2. ***Lists***, which are sometimes called **recursive** vectors because lists can contain other lists.

S.No	Atomic Vectors	Lists
1.	Homogeneous	Heterogeneous
2.	All the elements are of the same type	The elements are of different data types.
3.	Atomic vectors are not recursive	Lists are recursive
4.	It is a one-dimensional object	It is a Multi-dimensional object

Hierarchy of R's vector types



Properties of Vectors

1. Every vector has two key properties functions:
   - Its ***type***, which you can determine with typeof():

     Input: typeof(**letters**)

     Output: *[1] "character"*

     Input: typeof(1L:10L)

     Output: *[1] "integer"*
   - Its ***length***, which you can determine with length():

     Input: x <- list("a", "b", 1:10)

     length(x)

     Output: *[1] 3*

- Vectors can also contain arbitrary additional metadata in the form of attributes.
- These attributes are used to create ***augmented vectors***, which build on additional behavior. There are four important types of augmented vectors:
  - ✓Factors are built on top of integer vectors.
  - ✓Dates and date-times are built on top of numeric vectors.
  - ✓Data frames and tibbles are built on top of lists.

Important Types of Atomic Vector

- As we have already discussed that we have four most important types of atomic vectors: logical, (integer, double) as numeric and character.
- Raw and complex are rarely used during data analysis.

1. **Logical Vector:** Logical vectors are the simplest type of atomic vector because they can take only three possible values: FALSE, TRUE, and NA. You can also create them c():

   Eg. **Input:** c(**TRUE**, **TRUE**, **FALSE**, **NA**)

   Output:[1] TRUE TRUE FALSE NA

2. **Numeric Vector:** Integer and double vectors are known collectively as numeric vectors. In R, numbers are doubles by default. To make an integer, place a **L** after the number:

   Eg. **Input:** typeof(1)

   Output:*[1] "double"*

   Input: typeof(1L)

   Output: *[1] "integer“*

There are two important differences that you should be aware of integer and double:

1. Doubles represent floating-point numbers that cannot always be precisely represented with a fixed amount of memory.

   Example: what is the square of the square root of two?

```
x <- sqrt(2) ^ 3
x
[1] 2
x - 2
[1] 4.44e-16
```

Instead of comparing floating-point numbers using ==, you should use **dplyr::near()**, which allows for some numerical tolerance.

2. Integers have one special value, NA, while doubles have four, NA, NaN, Inf, and -Inf. All three special values can arise during division:

   Example: c(-1, 0, 1) / 0

   [1] -Inf NaN Inf

Avoid using == to check for these other special values. Instead use the helper functions **is.finite()**, **is.infinite()**, and **is.nan()**:

	0	Inf	NA	NaN
is.finite()	x			
is.infinite()		x		
is.na()			x	x
is.nan()				x

- # Create a vector with some NaN values
- x <- c(1, NaN, 12, NaN, 50, 30)

- # Identify positions with NaN values
- which(is.nan(x)) # Output: 2 4

- # Count total NaN values
- sum(is.nan(x)) # Output: 2

- # Create a new vector with NaN values removed
- x_new <- x[!is.nan(x)]
- # View the new vector: 1 12 50 30

- # Replace NaN values with zeros
- x[is.nan(x)] <- 0
- # View the updated vector: 1 0 12 0 50 30

3. **Character Vector:** Character vectors are the most complex type of atomic vector because each element of a character vector is a string.

- A string can contain an arbitrary amount of data.
- R uses a **global string pool** which is one of the important feature for the string implementation.
- This mean that each unique string is only stored in memory once, and every use of the string points to that representation.
- This reduces the amount of memory needed by duplicated strings. To make use of it in R we use **pryr::object_size().**

Example: x <- "This is a reasonably long string."

pryr::object_size(x)

152 B

install.packages("pryr")

```
y <- rep(x, 1000)
pryr::object_size(y)
8.14 kB
```

- A pointer is 8 bytes, so 1000 pointers to a 136 B string is 8 * 1000 + 136 = 8.14 kB.

4. **Missing Values:** Each type of atomic vector has its own missing value:

Example:

```
NA # logical
> [1] NA
NA_integer_ # integer
> [1] NA
NA_real_ # double
> [1] NA
NA_character_ # character
> [1] NA
```

- Normally you don’t need to know about these different types of NA representation.
- You can always use NA and it will be converted to the correct type using implicit coercion.
- There are some functions that are strict about their inputs, so it’s useful to have this knowledge, so you can be specific when needed.

Using Atomic Vectors

- How to convert from one type to another, and when that happens automatically.
- How to tell if an object is a specific type of vector.
- What happens when you work with vectors of different lengths.
- How to name the elements of a vector.
- How to pull out elements of interest.

Coercion

- Coercion includes type conversions – which means the change of one type of data into another type of data. There are two ways to convert, or coerce, one type of vector to another:
  - Explicit Coercion
  - Implicit Coercion

1. **Explicit coercion** we can change one data type to another data type by calling a function like as.logical(), as.integer(), as.double(), or as.character(), etc.

S.No	Function	Description
1.	as.logical()	Converts the value to logical type. • If 0 is present then it is converted to FALSE • Any other value is converted to TRUE
2.	as.integer()	Converts the object to integer type
3.	as.double()	Converts the object to double precision type
4.	as.character(x, encoding = NULL)	Converts a numeric object to a string data type or a character object.
5.	as.list()	It accepts only dictionary type or vector as input arguments in the parameter

Example: # Creating a vector

```
x<-c(0, 1, 0, 3)
class(x) # Checking its class
as.numeric(x) # Converting it to integer type
as.double(x) # Converting it to double type
as.logical(x) # Converting it to logical type
as.list(x) # Converting it to a list
as.complex(x) # Converting it to complex numbers
```

2. **Implicit coercion** we can change one data type to another data type by itself without using any functions. In this case TRUE is converted to 1 and FALSE is converted to 0.

Example: x<- 1:3

x

typeof(x)

x[2] <- “hello”

x

typeof(x)

Example: convert logical data into numeric data

Y<-c(TRUE, FALSE, 6)

Y

Difference between conversion, coercion and cast:

- **Type Conversion** means the change of one type of data into another type of data. It signifies both coercion and casting.
- **Coercion** includes type conversions- Explicit coercion & Implicit coercion
- **Casting** means whatever is converted implicitly is referred to as coercion and if converted explicitly then it is known as casting.

Example: x<- c(“a”, “b”, “c”)

y<- as.numeric(x)

y

- We are converting character data type to numeric data type. It will show NA . It will show missing in out object.
- It will not change character data to numeric because it includes values “a” which cannot be changed to numeric data.
- An atomic vector cannot have a mix of different types because the type is a property of the complete vector, not the individual elements.
- If you need to **mix multiple types** in the same vector, you should **use a list**, which you’ll learn about shortly.

Naming Vectors

- All types of vectors can be named. You can name them during creation with c():

Example: c(x = 1, y = 2, z = 4)

> *x y z*

> *1 2 4*

Or

With **purrr::set_names():**

Example: setNames(1:3, c("a", "b", "c"))

> *a b c*

> *1 2 3*

- Named vectors are most useful for subsetting.

Adding and Deleting Vector Elements

- The size of a vector is determined at its creation, so if you wish to add or delete elements, you'll need to reassign the vector.

Example: Let's add an element to the middle of a four-element vector:

```
x <- c(88,5,12,13)
x <- c(x[1:3],168,x[4]) # insert 168 before the 13
x
```

This creates a *new* five-element vector,

- **append():** in R is used for merging vectors or adding more elements to a vector.

  Syntax: append(x, value, index(optional))

  Example: x <- c(10:15)
  x

  Output: [1] 10 11 12 13 14 15

  Input: y <- append(x, 1, 1)
  print(y)

  Output: [1] 10 1 11 12 13 14 15

  Example: x <- rep(1:5)
  x

  Output: [1] 1 2 3 4 5
  y <- append(x, 15)
  print(y)

  Output: [1] 1 2 3 4 5 15

Obtaining the Length of a Vector

- You can obtain the length of a vector in R programming, call **length()** function and pass the vector to it.

  Syntax: length()

  Example: x<- c(5,9,3,4,1,0)

  vectorlength <- length(x)

  cat("Length of vector x is:", vectorlength)

  Output: Length of vector x is: 6

Declarations

- Instead, you must create y first, for instance this way:

  Example: y<- vector(length=3)

  y[1]<-6

  y[2]<-4

  y[3]<- 5

  y

  Output: [1] 6 4 5

- This approach is all right because on the right-hand side we are creating a new vector, to which we then bind y.

Recycling

- When applying an operation to two vectors that require them to be the **same length**, R automatically ***recycles*** or **repeats**, the shorter one, until it is long enough to match the longer one.

  Example1 : c(1,2,4) + c(6,0,9,20,22)

  Which shows you a warning message longer object length and does not execute.

  Example2 : c(1,2,4) + c(6,0,9,20,22,33)

  Example3: c(1,2,4,1,2) + c(6,0,9,20,22)

Common Vector Operations

- As R is a functional language we are having two common vector operations:

1. Arithmetic operations and logical operations
2. Vector Indexing
3. Generating Useful Vectors with the : Operator / Range Indexes
4. Generating Vector Sequences with seq()
5. Repeating Vector Constants with rep()

1. **Arithmetic operations and logical operations**

Example: 2+3

"+"(2,3)

scalars are actually **one-element** vectors. So, we can add vectors, and the + operation will be applied **element-wise**.

```
x <- c(1,2,4)
x + c(5,0,-1)
x * c(5,0,-1) # element by element
x / c(5,4,-1)
x <- c(10, 15, 20)
result <- x %% c(5, 4, -1)
result
# Output:
# [1] 0 3 0
```

2. **Vector Indexing:** We can access the elements of a vector with the help of vector indexing. Indexing denotes the position where the value in a vector is stored.

   Syntax: vector1[vector2]

   Example: y <- c(1.2,3.9,0.4,0.12)

   ```
   y[c(1,3)] # extract elements 1 and 3 of y
   y[2:3]
   v <- 3:4
   y[v]
   ```

✓**Duplicates** - An index vector allows duplicate values which means we can access one element twice in one operation.

Example: x <- c(4,2,17,5)

```
y <- x[c(1,1,3)]
y
```

Eliminating the elements in a vector

- This can be done by two methods:
  - ✓Negative subscripts and
  - ✓Length() function
- ✓**Negative subscripts -** to exclude the given elements in our output.

  Example: z <- c(5,12,13)

  z[-1] # exclude element 1

  z[-1:-2] # exclude elements 1 through 2
- ✓**length() function** – Most of the time we use this function in R. This function will specify the length of the vector and also helps in removing the elements in our output.

  Example: z <- c(5,12,13)

  z[1:(length(z)-1)] / z[-length(z)]

Using all() and any()

- **any()** function: Takes a vector and a logical condition as input arguments. It checks the vector against the condition and creates a **logical vector**. It then returns **TRUE**, if **any one** of the elements in the logical vector is **TRUE**.

  Syntax: any(vector logical condition value)

  Example: x <- 1:10

  ```
  any(x > 8)
  [1] TRUE
  any(x > 88)
  [1] FALSE
  ```

❖**all()** function: Takes a vector and a logical condition as input arguments. It checks the vector against the condition and creates a logical vector. It then returns **TRUE** if **all** the elements in the logical vector are **TRUE**, and **FALSE** if **all elements** are not **TRUE**.

- **Syntax:** all(vector logical condition value)
- **Example:** x <- 1:10

```
all(x > 88)
[1] FALSE
all(x > 0)
[1] TRUE
```

Vectorized Operations

- Many operations in R are ***vectorized***, - that operations occur in parallel in certain R objects.
- This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages. This can really simplify our code.

  Example: adding two vectors.

  1. *Vector In, Vector Out*
  2. *Vector In, Matrix Out*

Example: u <- c(5,2,8)

v <- c(1,3,9)

u > v

- Here, the > function was applied to u[1] and v[1], resulting in TRUE, then to u[2] and v[2], resulting in FALSE, and so on.

Example2: u<-c(5,2,8)

```
w <- function(u)
    return(u+1)
w(u)
```

- The transcendental functions—square roots, logs, trig functions, and so on—are vectorized.
- In R we have many built-in functions, and remember that scalars are single-element vectors.

Example: sqrt(1:9)

```
y <- c(1.2,3.9,0.4)
z <- round(y)
z
y <- c(12,5,13)
'+'(y,4)
```

NA and NULL Values

- R is a scripting language that actually has two such values: NA and NULL.
- In statistical datasets, we often encounter missing data, which we represent in R with the value NA.
- NULL, on the other hand, represents that the value simply doesn't exist.

Using NA

- R has many statistical functions, we can instruct the function to skip over any missing values or NAs.

Example: x <- c(88,NA,12,168,13)

```
x
mean(x)
```

mean(x) refused to calculate, as one value in x was NA.

```
mean(x,na.rm=T)
x <- c(88,NULL,12,168,13)
mean(x)
```

- By setting the optional argument **na.rm** (*NA remove*) to **true (T)**, we calculated the mean of the remaining elements.
- There are multiple NA values, one for each mode:

  Example:

```
x <- c(5,NA,12)
mode(x[1])
mode(x[2])
y <- c("abc","def",NA)
mode(y[2])
mode(y[3])
```

Using NULL

- NULL is the absence of anything. It is not exactly missingness, it is nothingness.
- An important difference between NA and NULL is that NULL is atomical and cannot exist within a vector. If used inside a vector, it simply disappears.

Example: we build up a vector of even numbers:

```
z <- NULL
for (i in 1:10)
    if (i %%2 == 0)
      z <- c(z,i)
z
```

- If we were to use NA instead of NULL in the preceding example, we would pick up an unwanted NA:

```
z <- NA
for (i in 1:10)
  if (i %%2 == 0)
    z <- c(z,i)
z
```

- NULL values really are counted as nonexistent. NULL is a special R object with no mode.

Subsetting a R Objects

- In R we have operators that can be used to extract subsets of R objects. There are three operators:
  - The [] operator always returns an object of the same class as the original. It can be used to select multiple elements of an object.
  - The [[]] operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object of a list or data frame.
  - The $ operator is used to extract elements of a list or data frame. Its semantics are similar to that of the [[ ]] operator.

Subsetting a Vector

- Vectors are basic objects in R and they can be subsetted using the [] operator. The [] operator always returns an object of the same class as the original. It can be used to select multiple elements of an object.

- A numeric vector containing only integers. The integers must either be all positive, all negative, or zero. Subsetting with positive integers keeps the elements at those positions:

  Example: x <- c("one", "two", "three", "four", "five")

  x[c(3, 2, 5)]

  Output: [1] "three" "two" "five"

- **Repeating a position:**

  Example: x <- c("one", "two", "three", "four", "five")

  x[c(1, 1, 5, 5, 5, 2)]

  Output:

  Example: x<- c(“a”, ”b”, “c”, “c”, “d”, “a”)

  x[1]

  x[2]

  x[1:4]

  u <- x > “a”

  u

  x[u] / x[x>”a”]

- Negative values drop the elements at the specified positions:

  Input: x <- c("one", "two", "three", "four", "five")

  x[c(-1, -3, -5)]

  Output:

- It’s an error to mix positive and negative values:

  Example: x[c(1, -1)]

  Error in x[c(1, -1)]: only 0's may be mixed with negative subscripts

- The error message mentions subsetting with zero, which returns no values:

  Example: x[0]

  character(0)

- Subsetting with a logical vector keeps all values corresponding to a TRUE value.

  Example: x <- c(10, 3, **NA**, 5, 8, 1, **NA**)

  x[!is.na(x)] *# All non-missing values of x*

  [1] 10 3 5 8 1

  x[x %% 2 == 0] *# All odd (or missing!) values of x*

  Output: [1] 10 NA 8 NA

- If you have a named vector, you can subset it with a character vector:

  Example: x <- c(abc = 1, def = 2, xyz = 5)

  x[c("xyz", "def")]

  Output:

- The simplest type of subsetting is []. This is not useful for subsetting vectors, but it is useful when subsetting matrices which is of a high-dimensional structures because it lets you select all the rows or all the columns.

Out-of-order Indexes

- In R, the index vector can be out-of-order.
- Below is an example in which a vector slice with the order of first and second values reversed.

Example: q<-**c**("shubham","arpita","nishka","gunjan","vaishali","sumit")

b<-**q**[2:5]

q[c(2,1,3,4,5,6)]

lapply ()

- We have the **lapply()** function under R. The **lapply()** function takes a vector as an argument and then applies a specific function on each element of that vector, **list** or data frame.
- Finally, this function returns a **list** as an output once the function is applied to each element.

  Example: names <- c("JOHN","RICK","RAHUL","ABDUL")

  lapply(names,tolower) /names_low<- lapply(names,tolower)

- Even if you use **lapply()** on a vector, the final output will be a list.
- The **lapply()** function was specifically designed for working on with lists. The **“l”** under the function name stands for the “lists”.
- It allows us to work on with other data structures as well such as vectors, and data frames.

sapply()

- In **lapply()**, we were restricted to the final output it was always a list.

- The **sapply()** function under R feels that the final output will also look good in vector format, it will convert it into a vector rather than generating it as a list.
- The **sapply()** is very similar to the **lappy()** function. The **“s”** under the function name stands for the “simplify”.

  Example: names <- c("JOHN","RICK","RAHUL","ABDUL")

  sapply(names,tolower) /names_low<- sapply(names,tolower)

Basic operations

- head(variable_name) #shows the first 5 rows
  Example: head(iris)
- tail(variable_name) #shows the last 5 rows
  Example: tail(iris)
- str(variable_name) #shows the variable names and types
  Example: str(iris)
- names(variable_name) #shows the variable names
  Example: names(iris)
  [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
- ls() #shows a list of objects that are available

Basic operations

- mean(x) #computes the mean of the variable x
- median(x) #computes the median of the variable x
- sd(x) #computes the standard deviation of the variable x
- IQR(x) Inter Quantile Range#computer the IQR of the variable x
- summary(x) #computes the 5-number summary and the mean of the variable x