

## UNIT-2

### VISUALIZATION USING R

Overview of R, Descriptive data analysis using R, Data manipulation with R Data visualization with R, R studio installation, Data manipulation with R (dplyr, data. table, reshape2package, tidyr package, Lubricate package) ,Data Visualization with R (working with Graphics,ggplot2).

-----

R programming:

- R is an open-source programming language that is widely used as a statistical software and data analysis tool. R is available across widely used platforms like Windows, Linux, and macOS.

Why Use R?

- Statistical Analysis: R is designed for analysis and It provides an extensive collection of graphical and statistical techniques, By making a preferred choice for statisticians and data analysts.
- Open Source: R is an open – source software, which means it is freely available to anyone. It can be accessible by a vibrant community of users and developers.
- Data Visualization : R boasts an array of libraries like ggplot2 that enable the creation of high-quality, customizable data visualizations.
- Data Manipulation : R offers tools that are for data manipulation and transformation.

For example: IT simplifies the process of filtering , summarizing and transforming data.

- Integration : R can be easily integrate with other programming languages and data sources. IT has connectors to various databases and can be used in conjunction with python, SQL and other tools.

- **Community and Packages:** R has vast ecosystem of packages that extend its functionality. There are packages that can help you accomplish needs of analytics.

Data types:

- **Numeric:** Numeric data types represent numeric values, including both integers and real numbers (floating-point numbers). In R, numeric values are stored as double-precision floating-point numbers by default.
- **Character:** Character data types represent text strings. They are enclosed in quotation marks (either single or double). For example, "hello" or 'world'.
- **Integer:** Integer data types specifically represent whole numbers without decimal points. They are created using the `as.integer()` function or by appending an 'L' to the numeric value. For example, 42L.
- **Logical:** Logical data types represent Boolean values, which can be either TRUE or FALSE. They are often used for logical operations and conditional expressions.
- **Complex:** Complex data types represent complex numbers with both real and imaginary parts. They are written in the form `a + bi`, where 'a' and 'b' are numeric values, and 'i' represents the imaginary unit.

Data Conversion:

- `as.numeric()`-Converts its argument to a numeric type.

```
# Convert character vector to numeric
char_vector <- c("10", "20", "30")
num_vector <- as.numeric(char_vector)
print(num_vector) # Output: 10 20 30
```

- `as.integer()`-Converts its argument to an integer type

```
# Convert numeric vector to integer
num_vector <- c(10.5, 20.7, 30.9)
int_vector <- as.integer(num_vector)
print(int_vector) # Output: 10 20 30

# Convert character vector to integer
char_vector <- c("10", "20", "30")
int_vector <- as.integer(char_vector)
print(int_vector) # Output: 10 20 30
```

- as.character()-Converts its argument to a character type.

```
# Convert numeric vector to character
num_vector <- c(10, 20, 30)
char_vector <- as.character(num_vector)
print(char_vector) # Output: "10" "20" "30"
```

- as.logical()-Converts its argument to a logical type.

```
# Convert numeric vector to logical
num_vector <- c(0, 1, 10)
logical_vector <- as.logical(num_vector)
print(logical_vector) # Output: FALSE TRUE TRUE
```

- **as.complex()**-Converts its argument to a complex type.

```
# Convert numeric vector to complex
num_vector <- c(1, 2, 3)
complex_vector <- as.complex(num_vector)
print(complex_vector) # Output: 1+0i 2+0i 3+0i

# Convert character vector to complex
char_vector <- c("1+2i", "3+4i", "5+6i")
complex_vector <- as.complex(char_vector)
print(complex_vector) # Output: 1+2i 3+4i 5+6i
```

Variable syntax:

- *Using equal to operators*  
*variable\_name = value*
- *using leftward operator*  
*variable\_name <- value*
- *using rightward operator*  
*value -> variable\_name*

**Rules to Create variable name:**

- Variable names in R must begin with a letter (uppercase or lowercase) or a dot (.).
- They cannot start with a number or any other symbol.
- After the first character, variable names can include letters, numbers, dots (.), or underscores (\_).
- They cannot include spaces or any other special characters.
- Case-sensitive: R is case-sensitive, so uppercase and lowercase letters are treated as distinct. For example, myVariable and MyVariable would be considered different variable names.
- Avoid Reserved Keywords: Avoid using reserved keywords or function names in R as variable names to prevent conflicts and confusion. For example, using if, else, for, or function as variable names is not recommended.

## Operators:

- Arithmetic operators:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division). Give the remainder of the first vector with the second
%/%	Integer Division. The result of division of first vector with second (quotient)

### Relational Operators:

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

### Logical operators:

Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
	Element-wise logical OR
	Logical OR

### Assignment operators:

Operator	Description
<code>&lt;-, &lt;&lt;-, =</code>	Left assignment
<code>-&gt;, -&gt;&gt;</code>	Right assignment

### Mixed Operators:

Operator	Description
<code>:</code>	Colon operator. It creates a sequence of numbers.
<code>%in%</code>	This operator is used to identify if an element belongs to a vector or not.
<code>%*%</code>	This operator is used for matrix multiplication. Normal <code>*</code> do elementwise multiplication.

### Taking Input from User in R Programming:

- In R, you can take input from the user using the `readline()` function for reading character input and `scan()` function for reading numeric input. Here's how you can use these functions:

```
# Prompting the user to enter a character input
user_input <- readline(prompt = "Enter your name: ")

# Displaying the input provided by the user
print(paste("Hello,", user_input))
```

```
# Prompting the user to enter a numeric input
user_number <- as.numeric(scan(""))

# Displaying the input provided by the user
print(paste("The number you entered is:", user_number))
```

## Printing Output of an R Program:

- In R, you can print output to the console using the `print()` function or simply by typing the name of the object you want to print. Here are examples of both methods:
- Using `print` function:

```
# Define a variable
x <- 10

# Print the value of the variable using the print() function
print(x)
```

## Output:

```
[1] 10
```

- Using implicitly print

```
# Define a variable
y <- "Hello, world!"

# Simply typing the name of the variable will implicitly
y
```

## Output:



```
[1] "Hello, world!"
```

## Decision making:

if Statement:

- The if statement is used to execute a block of code if a specified condition is true.

```
# Example of if statement  
x <- 10  
  
if (x > 5) {  
  print("x is greater than 5")  
}
```

- if...else Statement:
- The if...else statement is used to execute one block of code if the condition is true, and another block of code if the condition is false.

```
# Example of if...else statement
x <- 3

if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is less than or equal to 5")
}
```

else if Statement:

- The else if statement allows you to check multiple conditions and execute a corresponding block of code based on the first true condition encountered.

```
# Example of else if statement
x <- 3

if (x > 5) {
  print("x is greater than 5")
} else if (x == 5) {
  print("x is equal to 5")
} else {
  print("x is less than 5")
}
```

- Nested if...else Statements:

- You can also nest if...else statements within each other to handle more complex decision-making scenarios.

```
# Example of nested if...else statements
x <- 10
y <- 5

if (x > 5) {
  if (y > 3) {
    print("x is greater than 5 and y is greater than 3")
  } else {
    print("x is greater than 5 but y is not greater than 3")
  }
} else {
  print("x is not greater than 5")
}
```

- Switch :

In R, the switch statement provides a way to select one of several alternative blocks of code to execute based on the value of a given expression.

```
switch(expression,
       case1 = expression1,
       case2 = expression2,
       ...
       default = expression_default
)
```

Example :

```
# Example of switch statement with numeric values
number <- 3

switch(number,
  1 = print("The number is one."),
  2 = print("The number is two."),
  3 = print("The number is three."),
  4 = print("The number is four."),
  5 = print("The number is five."),
  print("The number is not in the range 1-5.")
)
```

## Loops:

- for Loop:
- The for loop iterates over a sequence of values (e.g., vector, list) and executes a block of code for each value.

```
for (value in sequence) {
  # code block to be executed
}
```

```
for (i in 1:5) {
  print(i)
}
```

- while Loop:
- The while loop repeatedly executes a block of code as long as a specified condition is true.

```
while (condition) {  
  # code block to be executed  
}
```

```
i <- 1  
while (i <= 5) {  
  print(i)  
  i <- i + 1  
}
```

- repeat Loop:
- The repeat loop executes a block of code indefinitely until a break statement is encountered.

```
repeat {  
  # code block to be executed  
  if (condition) {  
    break  
  }  
}
```

```
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break
  }
}
```

Function:

- In R, functions are blocks of code that perform a specific task and can be reused throughout a program.

### **Defining Functions:**

- You can define your own functions in R using the `function()` keyword.
- Syntax:

```
functionName <- function(arg1, arg2, ...) {
  # code block
  return(result)
}
```

- Calling Functions:
- Once defined, you can call a function by using its name followed by parentheses containing its arguments (if any).

```
square <- function(x) {  
  return(x * x)  
}
```

```
result <- square(5)  
print(result) # Output: 25
```

- **Arguments:**
- Functions can take zero or more arguments, which are placeholders for values passed to the function when it is called.
- Arguments can have default values, making them optional.

```
greet <- function(name = "World") {  
  print(paste("Hello,", name))  
}  
  
greet() # Output: Hello, World  
greet("Alice") # Output: Hello, Alice
```

- **Data structure:**  
data structures are fundamental components used to organize, store, and manipulate data.
- Some common data structures in R include:

In R, data structures are fundamental components used to organize, store, and manipulate data. Some common data structures in R include:

1. **Vectors:** A vector is a sequence of data elements of the same basic type. Vectors can be atomic types (like numeric, character, logical) or lists. They are created using the `c()` function.
2. **Lists:** Lists are collections of objects (which can be of different types) arranged in a specified order. They are created using the `list()` function.
3. **Matrices:** A matrix is a 2-dimensional array with rows and columns. Matrices can hold elements of the same atomic data type. They are created using the `matrix()` function.
4. **Arrays:** Arrays are similar to matrices, but they can have more than two dimensions. They are created using the `array()` function.
5. **Data Frames:** A data frame is a tabular structure similar to a spreadsheet or database table. It is a list of equal-length vectors, where each vector represents a column. Data frames are commonly used for handling datasets in R.
6. **Factors:** Factors are used to represent categorical data in R. They are created using the `factor()` function and are internally stored as integers with corresponding labels.

Vector:

- R Vectors are the same as the arrays in R language which are used to hold multiple data values of the same type.

1. Creating Vectors:

- You can create a vector in R using the `c()` function, which stands for "combine" or "concatenate".

```
vector_name <- c(element1, element2, ...)
```



- **Example:**

```
# Creating a numeric vector
numeric_vector <- c(1, 2, 3, 4, 5)

# Creating a character vector
character_vector <- c("apple", "banana", "orange")
```

- **Vector Types:**

- Vectors in R can contain elements of various data types, such as numeric, character, logical, complex, and raw.

```
# Numeric vector
numeric_vector <- c(1, 2, 3)

# Character vector
character_vector <- c("apple", "banana", "orange")

# Logical vector
logical_vector <- c(TRUE, FALSE, TRUE)

# Complex vector
complex_vector <- c(1 + 2i, 3 + 4i)

# Raw vector
raw_vector <- as.raw(c(0x41, 0x42, 0x43)) # Hexadecimal values
```

- **Accessing Elements:**

- You can access elements of a vector using indexing. Index values starting from 1.

```
vector_name[index]
```

```
numeric_vector <- c(1, 2, 3, 4, 5)  
print(numeric_vector[3]) # Output: 3
```

Example:

```

# Accessing vector elements using position.
t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")
u <- t[c(2,3,6)]
print(u)

# Accessing vector elements using logical indexing.
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
print(v)

# Accessing vector elements using negative indexing.
x <- t[c(-2,-5)]
print(x)

```

```

[1] "Mon" "Tue" "Fri"
[1] "Sun" "Fri"
[1] "Sun" "Tue" "Wed" "Fri" "Sat"

```

## Vector manipulation:

```

# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)

# Vector addition.
add.result <- v1+v2
print(add.result)

# Vector subtraction.
sub.result <- v1-v2
print(sub.result)

# Vector multiplication.
multi.result <- v1*v2
print(multi.result)

# Vector division.
divi.result <- v1/v2
print(divi.result)

```

## Vector Element Recycling:

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11)
# V2 becomes c(4,11,4,11,4,11)

add.result <- v1+v2
print(add.result)

sub.result <- v1-v2
print(sub.result)
```

## Output:

```
[1]  7 19  8 16  4 22
[1] -1 -3  0 -6 -4  0
```

## Vector Element Sorting:

Elements in a vector can be sorted using the `sort()` function.

```
v <- c(3,8,4,5,0,11, -9, 304)

# Sort the elements of the vector.
sort.result <- sort(v)
print(sort.result)

# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)

# Sorting character vectors.
v <- c("Red","Blue","yellow","violet")
sort.result <- sort(v)
print(sort.result)

# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
```

- Output:

```
[1] -9  0  3  4  5  8 11 304
[1] 304 11  8  5  4  3  0 -9
[1] "Blue"  "Red"   "violet" "yellow"
[1] "yellow" "violet" "Red"    "Blue"
```

Functions:

**Mathematical functions:** These functions perform mathematical operations on vectors, element-wise.

- `sum()`: Calculates the sum of all elements in a vector.

- `mean()`: Calculates the mean (average) of all elements in a vector.
- `median()`: Calculates the median of all elements in a vector.
- `sd()`: Calculates the standard deviation of all elements in a vector.
- `var()`: Calculates the variance of all elements in a vector.
- `max()`: Returns the maximum value in a vector.
- `min()`: Returns the minimum value in a vector.
- Vector manipulation functions: These functions manipulate vectors in various ways.
- `rev()`: Reverses the order of elements in a vector.
- `sort()`: Sorts the elements of a vector in ascending order.
- `order()`: Returns the indices that would sort a vector.
- `unique()`: Removes duplicate elements from a vector.
- `append()`: Appends elements to a vector.

```
# Create a numeric vector
vec <- c(10, 20, 30, 40, 50)

# Calculate the sum of all elements in the vector
sum_result <- sum(vec)
cat("Sum:", sum_result, "\n")

# Calculate the mean (average) of all elements in the vector
mean_result <- mean(vec)
cat("Mean:", mean_result, "\n")

# Calculate the median of all elements in the vector
median_result <- median(vec)
cat("Median:", median_result, "\n")
```

```
# Calculate the standard deviation of all elements in the vector
sd_result <- sd(vec)
cat("Standard Deviation:", sd_result, "\n")

# Calculate the variance of all elements in the vector
var_result <- var(vec)
cat("Variance:", var_result, "\n")

# Find the maximum value in the vector
max_value <- max(vec)
cat("Maximum Value:", max_value, "\n")

# Find the minimum value in the vector
min_value <- min(vec)
cat("Minimum Value:", min_value, "\n")
```

- **Output:**

```
Sum: 150
Mean: 30
Median: 30
Standard Deviation: 15.81139
Variance: 250
Maximum Value: 50
Minimum Value: 10
```

- **List:**  
Lists are the R objects which contain elements of different types like –

numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

- Creating a list:

```
my_list <- list(1, "a", TRUE)
```

- `length()`: Returns the number of elements in a list.

```
len <- length(my_list)
```

- `names()`: Retrieves or sets the names of the elements in a list.

```
element_names <- names(my_list)
```

### Naming List Elements:

- The list elements can be given names and they can be accessed using these names.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow =
  2),
  list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Show the list.
print(list_data)
```



Output:

```
$`1st_Quarter`  
[1] "Jan" "Feb" "Mar"
```

```
$A_Matrix  
  [,1] [,2] [,3]  
[1,]   3   5  -2  
[2,]   9   1   8
```

```
$A_Inner_list  
$A_Inner_list[[1]]  
[1] "green"
```

```
$A_Inner_list[[2]]  
[1] 12.3
```

Accessing List Elements:

- Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

```

# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow =
  2),
  list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list.
print(list_data[1])

# Access the thrid element. As it is also a list, all its elements will
  be printed.
print(list_data[3])

# Access the list element using the name of the element.
print(list_data$A_Matrix)

```

Output:

```

$`1st_Quarter`
[1] "Jan" "Feb" "Mar"

$A_Inner_list
$A_Inner_list[[1]]
[1] "green"

$A_Inner_list[[2]]
[1] 12.3

      [,1] [,2] [,3]
[1,]   3   5  -2
[2,]   9   1   8

```

## Manipulating List Elements:

- We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

### Example:

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow =
  2),
  list("green",12.3))
# Add element at the end of the list.
list_data[4] <- "New element"
print(list_data[4])
# Remove the last element.
list_data[4] <- NULL
# Print the 4th Element.
print(list_data[4])
# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

### Output:

```
[[1]]
[1] "New element"

[[1]]
NULL

[[1]]
[1] "updated element"
```

- Merging Lists: You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.  
list1 <- list(1,2,3)  
list2 <- list("Sun","Mon","Tue")  
  
# Merge the two lists.  
merged.list <- c(list1,list2)  
  
# Print the merged list.  
print(merged.list)
```

Output:

```
[[1]]  
[1] 1  
  
[[2]]  
[1] 2  
  
[[3]]  
[1] 3  
  
[[4]]  
[1] "Sun"  
  
[[5]]  
[1] "Mon"  
  
[[6]]  
[1] "Tue"
```

## Converting List to Vector:

- A list can be converted to a vector so that the elements of the vector can be used for further manipulation. To do this conversion, we use the **unlist()** function. It takes the list as input and produces a vector.
- Example:

```
# Create lists.
list1 <- list(1:5)
print(list1)

list2 <- list(10:14)
print(list2)

# Convert the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)

print(v1)
print(v2)

# Now add the vectors
result <- v1+v2
print(result)
```

Output:

```
[[1]]  
[1] 1 2 3 4 5  
  
[[1]]  
[1] 10 11 12 13 14  
  
[1] 1 2 3 4 5  
[1] 10 11 12 13 14  
[1] 11 13 15 17 19
```

Matrix:

- Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types.
- A Matrix is created using the `matrix()` function.

Syntax:

```
matrix(data, nrow, ncol, byrow, dimnames)
```

- **data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.
- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
- **dimname** is the names assigned to the rows and columns

Example:

```
# Elements are arranged sequentially by row.
M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
print(M)

# Elements are arranged sequentially by column.
N <- matrix(c(3:14), nrow = 4, byrow = FALSE)
print(N)

# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames,
  colnames))
print(P)
```

Output:

```
      [,1] [,2] [,3]
[1,]   3   4   5
[2,]   6   7   8
[3,]   9  10  11
[4,]  12  13  14
      [,1] [,2] [,3]
[1,]   3   7  11
[2,]   4   8  12
[3,]   5   9  13
[4,]   6  10  14
      col1 col2 col3
row1     3     4     5
row2     6     7     8
row3     9    10    11
row4    12    13    14
```

- **Accessing Elements of a Matrix:**

Elements of a matrix can be accessed by using the column and row index of the element.

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix.
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames,
  colnames))

# Access the element at 3rd column and 1st row.
print(P[1,3])

# Access the element at 2nd column and 4th row.
print(P[4,2])

# Access only the 2nd row.
print(P[2,])

# Access only the 3rd column.
print(P[,3])
```

**Output:**

```
[1] 5
[1] 13
col1 col2 col3
  6    7    8
row1 row2 row3 row4
```



## Matrix Computations:

### Matrix Addition & Subtraction

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)

# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)

# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

Output:

```
      [,1] [,2] [,3]
[1,]    3  -1    2
[2,]    9   4    6

      [,1] [,2] [,3]
[1,]    5   0    3
[2,]    2   9    4
Result of addition
      [,1] [,2] [,3]
[1,]    8  -1    5
[2,]   11  13   10
Result of subtraction
      [,1] [,2] [,3]
[1,]   -2  -1  -1
[2,]    7  -5    2
```

Example:

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)

# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
print(result)
```

Output:

```
      [,1] [,2] [,3]
[1,]    3  -1    2
[2,]    9   4    6
      [,1] [,2] [,3]
[1,]    5   0    3
[2,]    2   9    4
Result of multiplication
      [,1] [,2] [,3]
[1,]   15   0    6
[2,]   18  36   24
Result of division
      [,1]      [,2]      [,3]
[1,]  0.6      -Inf  0.6666667
[2,]  4.5  0.4444444  1.5000000
```

## Array:

- Arrays are the R data objects which can store data in more than two dimensions. For example – If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.
- An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array.

## Example:

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2))
print(result)
```

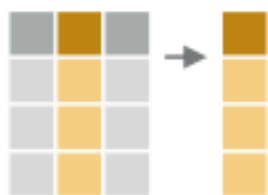
## Output:

```
, , 1
      [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

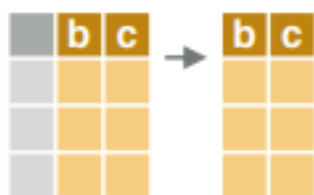
, , 2
      [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

## Data manipulation:

### EXTRACT



`dt[, c(2)]` – extract columns by number. Prefix column numbers with “-” to drop.



`dt[, .(b, c)]` – extract columns by name.

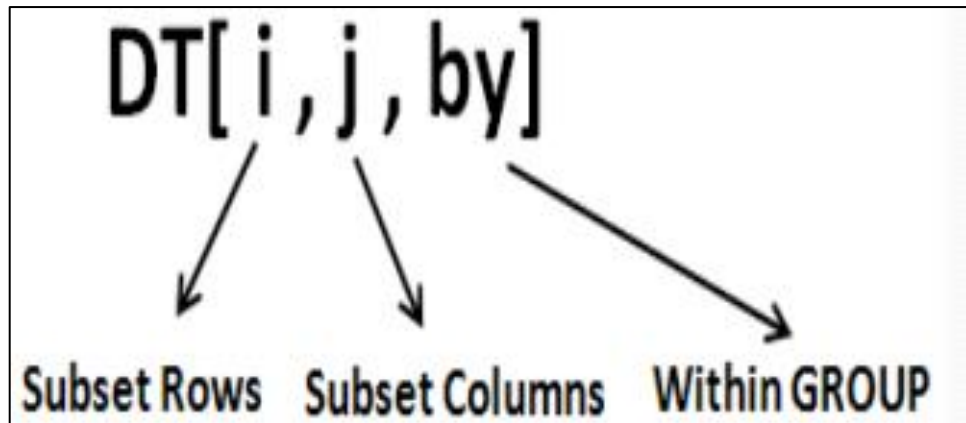
## Data manipulation in R:

### 1.Data.table:

- The data.table package in R is a powerful extension of R's data manipulation capabilities. It provides an enhanced version of the traditional data.frame object, optimized for speed and memory efficiency.

#### Reason to move from Dataframe to Data.table:

- Fast data manipulation: data.table provides fast and concise syntax for common data manipulation tasks like subsetting, grouping, joining, and aggregating large datasets.
- Efficient memory usage: It is designed to handle large datasets efficiently, using memory-mapped files and optimized algorithms. This makes it suitable for working with big data.



### How to Install and load data.table Package

```
install.packages("data.table")
```

#### **#load required library**

```
library(data.table)
```

Read Data:

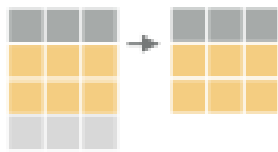
- In data.table package, `fread()` function is available to read or get data from your computer or from a web page. It is equivalent to `read.csv()` function of base R.

#### **IMPORT**

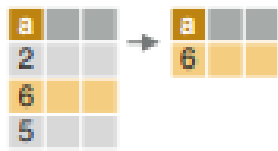
**`fread("file.csv")`** – read data from a flat file such as .csv or .tsv into R.

**`fread("file.csv", select = c("a", "b"))`** – read specified columns from a flat file into R.

Subset rows using i:



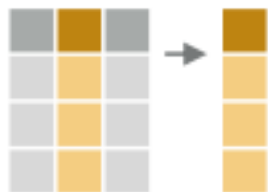
`dt[1:2, ]` – subset rows based on row numbers.



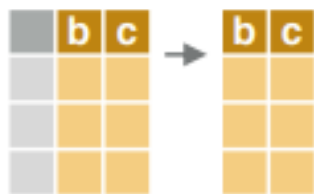
`dt[a > 5, ]` – subset rows based on values in one or more columns.

Manipulate columns with j:

### EXTRACT



`dt[, c(2)]` – extract columns by number. Prefix column numbers with “-” to drop.



`dt[, .(b, c)]` – extract columns by name.

- Accessing entire rows or columns: To access all rows of a specific column, you can use the \$ operator with the column name.

```
dt$column_name
```

- To access all rows of multiple columns, you can use a list of column names within square brackets.

```
dt[, c("column1", "column2")]
```

- **Accessing specific rows or columns:** To access specific rows based on their indices, you can use square brackets with row indices.

```
dt[c(1, 3, 5), ]
```

- To access specific rows based on conditions, you can use logical indexing within square brackets.

```
dt[condition, ]
```

- To access specific columns based on their names, you can use the `j` argument within square brackets.

```
dt[, .(column1, column2)]
```

- **Accessing rows and columns by position:**

- You can access rows and columns by their positions using integers within square brackets.

```
dt[row_index, column_index]
```

- To access a specific element at a given row and column, you can use double square brackets

```
dt[row_index, column_name]
```

Compute columns:

- In a data.table, you can compute new columns based on existing columns using the `:=` operator within square brackets. Here's how you can compute columns in a data.table:

```
# Suppose you have a data.table named dt
dt <- data.table(
  A = c(1, 2, 3),
  B = c(4, 5, 6)
)

# Compute a new column C based on existing columns A and B
dt[, C := A + B]

# Print the updated data.table
print(dt)
```

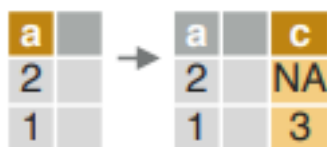


## COMPUTE COLUMNS\*



		c
		3
		3

`dt[, c := 1 + 2]` – compute a column based on an expression.



a		c
2		NA
1		3

`dt[a == 1, c := 1 + 2]` – compute a column based on an expression but only for a subset of rows.

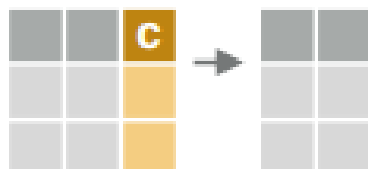


		c	d
		1	2
		1	2

`dt[, `:=` (c = 1, d = 2)]` – compute multiple columns based on separate expressions.

Delete columns:

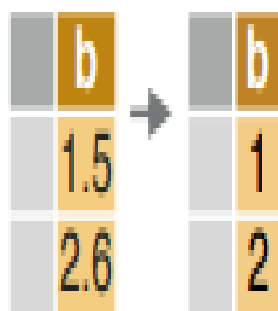
## DELETE COLUMN



		c

`dt[, c := NULL]` – delete a column.

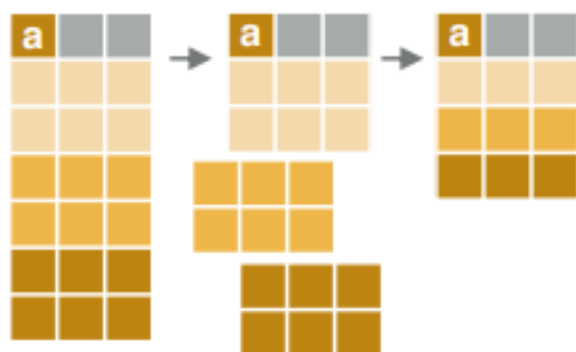
CONVERT COLUMN TYPE:



	b
	1
	2

`dt[, b := as.integer(b)]` – convert the type of a column using `as.integer()`, `as.numeric()`, `as.character()`, `as.Date()`, etc..

Group according to by:



`dt[, j, by = .(a)]` – group rows by values in specified columns.

`dt[, j, keyby = .(a)]` – group *and simultaneously sort* rows by values in specified columns.

`dt[, .(c = sum(b)), by = a]` – summarize rows within groups.

`dt[, c := sum(b), by = a]` – create a new column and compute rows within groups.

`dt[, .SD[1], by = a]` – extract first row of groups.

`dt[, .SD[.N], by = a]` – extract last row of groups.

Example:

```
result <- data[, sum_value := sum(value), by = "item no"]  
result
```

```
result <- data[, .(avg_age = mean(Age), max_value = max(value)), by = "Gender"]  
print(result)
```

Functions for data.tables:

### REORDER

a	b	
1	2	
2	2	
1	1	

 → 

a	b	
1	2	
1	1	
2	2	

**setorder**(dt, a, -b) – reorder a data.table according to specified columns. Prefix column names with “-” for descending order.

```
a=setorder(data, -Age)|
a
```

```
a=setorder(data, Age)
a
```

### UNIQUE ROWS

a	b	
1	2	
2	2	
1	2	

 → 

a	b	
1	2	
2	2	

**unique**(dt, by = c("a", "b")) – extract unique rows based on columns specified in “by”. Leave out “by” to use all columns.

```
a=unique(data, by="Age")
a
```

Output:

```
> a=unique(data,by="Age")
> a
```

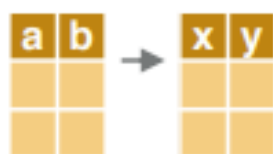
	No	Age	Gender	Order	item no	list	value	sum_value
	<char>	<int>	<char>	<int>	<int>	<int>	<int>	<int>
1:	TM195	18	Male	14	23	3	4	7
2:	TM195	19	Male	15	34	2	3	14
3:	TM195	20	Male	13	90	4	2	9
4:	TM195	21	Female	14	67	3	3	6
5:	TM195	22	Male	14	56	3	3	6
6:	TM195	23	Male	16	1	3	1	4
7:	TM195	24	Female	16	76	4	3	3

**uniqueN(dt, by = c("a", "b"))** – count the number of unique rows based on columns specified in “by”.

```
a=uniqueN(data,by="Age")
a
```

```
> a=uniqueN(data,by="Age")
> a
[1] 7
```

## RENAME COLUMNS



**setnames(dt, c("a", "b"), c("x", "y"))** – rename columns.

```
b=setnames(a, c("Age", "value"), c("x", "y"))
b
```

	No	x	Gender	Order	item no	list	y	sum_value
	<char>	<int>	<char>	<int>	<int>	<int>	<int>	<int>
1:	TM195	18	Male	14	23	3	4	7
2:	TM195	19	Male	15	34	2	3	14
3:	TM195	19	Female	14	56	4	3	6
4:	TM195	19	Male	12	87	3	3	8
5:	TM195	20	Male	13	90	4	2	9
6:	TM195	20	Female	14	45	3	3	3
7:	TM195	21	Female	14	67	3	3	6
8:	TM195	21	Male	13	89	3	3	3
9:	TM195	21	Male	15	90	5	4	9
10:	TM195	21	Female	15	34	2	3	14

Combine data.tables:

## JOIN

a	b		x	y		a	b	x
1	c	+	3	b	=	3	b	3
2	a		2	c		1	c	2
3	b		1	a		2	a	1

`dt_a[dt_b, on = .(b = y)]` – join data.tables on rows with equal values.

a	b	c		x	y	z		a	b	c	x
1	c	7	+	3	b	4	=	3	b	4	3
2	a	5		2	c	5		1	c	5	2
3	b	6		1	a	8		NA	a	8	1

`dt_a[dt_b, on = .(b = y, c > z)]` – join data.tables on rows with equal *and unequal* values.

## ROLLING JOIN

a	id	date		b	id	date	=	a	id	date	b
1	A	01-01-2010	+	1	A	01-01-2013	=	2	A	01-01-2013	1
2	A	01-01-2012		1	B	01-01-2013		2	B	01-01-2013	1
3	A	01-01-2014									
1	B	01-01-2010									
2	B	01-01-2012									

`dt_a[dt_b, on = .(id = id, date = date), roll = TRUE]` – join data.tables on matching rows in id columns but only keep the most recent preceding match with the left data.table according to date columns. “roll = -Inf” reverses direction.

## BIND

a	b		a	b	=	a	b
		+			=		

`rbind(dt_a, dt_b)` – combine rows of two data.tables.

a	b		x	y	=	a	b	x	y
		+			=				

`cbind(dt_a, dt_b)` – combine columns of two data.tables.

## data.table():

- To create a table using the `data.table()` function in R, you can specify the column names and their corresponding values as arguments to the function.

```
# Create a data.table
my_data <- data.table(
  ID = c(1, 2, 3, 4, 5),
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age = c(25, 30, 22, 35, 28)
)

# Print the data.table
print(my_data)
```

List using data table :

- `x<-list(name=c("john","rose"),age=c(34,56))` # creating list
- `X`
- Output:
- `$name`
- `[1] "john" "rose"`
- `$age`
- `[1] 34 56`

Convert list into table:

`as.data.table()` – this function is used to convert list into data table.

```
# List to data table
y<-as.data.table(x)
y
```

Output:

name age

1: john 34

2: rose 56

### **Dplyr-package:**

- The dplyr package in R is a popular package for data manipulation and transformation. It provides a set of functions that are designed to make data manipulation tasks easier and more intuitive.

key features :

- Consistent syntax: The functions in dplyr follow a consistent syntax, making it easy to learn and use. This consistency makes it easier to read and understand code written with dplyr.
- Verb-based functions: The functions in dplyr are verb-based, meaning that they are named after the actions they perform. For example, `filter()`, `select()`, `mutate()`, `arrange()`, and `summarize()` are some of the core functions in dplyr.
- Efficient backends: dplyr is designed to work efficiently with various data sources, including data frames, databases (via DBI), and big data systems (via dplyr backend packages like `dbplyr` and `sparklyr`). This makes it suitable for working with large datasets and different data storage systems.
- Piping: dplyr integrates well with the `%>%` (pipe) operator from the `magrittr` package, allowing you to chain together multiple data manipulation operations in a readable and concise manner.



Five major data manipulation commands:

- Filter -> filter the data based on the condition
- Select -> used to select a column from a data set.
- Mutate <- Used to create new variable from existing variable.
- Summarize (with group –by)-> commonly used operations like ,min, max, sum,mean
- Arrange <-used to arrange data set values on ascending and descending order.

Filter():

- The filter() function in the dplyr package is used to subset rows of a data frame or tibble based on specified conditions. It allows you to extract rows that meet certain criteria, similar to using logical indexing or conditional statements in base R.

**filter(data, condition)**

```
install.packages("dplyr")
install.packages("data.table")
library(dplyr)
library(data.table)
data=fread("C://Users/Hicet/Downloads/Cardio.csv")
data
```

```
filtered_data <- filter(data, Age > 25)
filtered_data
```

```
filtered_data <- filter(data, Age <20)
filtered_data
```

```
> filtered_data <- filter(data, Age <20)
> filtered_data
      No   Age Gender Order item no  list value
  <char> <int> <char> <int>   <int> <int> <int>
1: TM195   18  Male   14     23     3     4
2: TM195   19  Male   15     34     2     3
3: TM195   19 Female   14     56     4     3
4: TM195   19  Male   12     87     3     3
> |
```

filter 18 ,17 values in age column:

```
x=filter(data, Age %in% c("18","17"))
```

x

### Output :

No Age Gender Education item no Usage Fitnz

1: TM195 18 Male 14 23 3 4

## Select():

- the select() function in the dplyr package offers several additional operations for selecting columns beyond just specifying column names directly.
- 1. Selecting Columns by Range: You can use the : operator to select a range of columns based on their positions (indices) in the data frame.

```
#select a cloumns using range  
a= data %>% select(1:2)  
a
```

- #select one or more columns:  
data%<% select("Column-1",column-2"....)
- Selecting Columns by Pattern:
- You can use functions like starts\_with(), ends\_with(), contains(), and matches() to select columns based on patterns in their names.

```
# Select columns based on start characters  
a= data %>% select(starts_with("A"))  
a  
# Select columns based on start characters  
a= data %>% select(ends_with("R"))  
a
```

Example:

```
# Create a sample data frame
df <- data.frame(
  ID = 1:5,
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age_2020 = c(25, 30, 22, 35, 28),
  Age_2021 = c(26, 31, 23, 36, 29)
)

# Select columns containing the pattern "Age"
selected_df <- select(df, contains("Age"))

# Print the selected data frame
print(selected_df)
```

Matches():

```
# Create a sample data frame
df <- data.frame(
  ID = 1:5,
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age_2020 = c(25, 30, 22, 35, 28),
  Age_2021 = c(26, 31, 23, 36, 29)
)

# Select columns containing the pattern "Age"
selected_df <- select(df, matches("Age"))

# Print the selected data frame
print(selected_df)
```

- Output:

	Age_2020	Age_2021
1	25	26
2	30	31
3	22	23
4	35	36
5	28	29

### **Difference between the contains() and matches():**

- contains():
  - contains() selects columns that contain a specific character string anywhere within their names.
  - It performs a simple substring match, so it doesn't require regular expressions.
  - It's more straightforward to use when you want to match columns based on a specific substring.
- matches():
  - matches() selects columns based on regular expressions. It allows for more complex pattern matching.

It's useful when you need more flexible matching criteria, such as selecting columns based on patterns defined by regular expressions

- Excluding Columns:
- You can use the - operator to exclude specific columns from the selection.

```
# Select all but one column  
a=data %>% select(-"Age")  
a
```

- Renaming Columns:
- You can rename columns using the := operator within the select() function.

```
# Rename the "Name" column to "Full_Name"  
renamed_df <- select(df, Full_Name = Name)
```

- Mutate():  
The mutate() function in the dplyr package is used to add new variables (columns) to a data frame or modify existing ones.

- Syntax:

```
mutate(data, new_column = expression)
```

- Example:

```
# Sample data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age = c(25, 30, 22, 35, 28)
)
```

```
# Add a new column Age_Group based on Age
mutated_df <- mutate(df, Age_Group = ifelse(Age < 30, "Young", "Old"))

# Print the mutated data frame
print(mutated_df)
```

- Output:

	Name	Age	Age_Group
1	Alice	25	Young
2	Bob	30	Old
3	Charlie	22	Young
4	David	35	Old
5	Eve	28	Young

- Summarize():  
The summarize() function in the dplyr package is used to compute summary statistics or aggregate values within groups. It allows you to calculate summary statistics, such as mean, median, sum, etc., for each group in a data frame.

```
summarize(data, summary_variable = summary_function(column))
```

Example:

```
# Sample data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age = c(25, 30, 22, 35, 28),
  Height = c(160, 175, 168, 180, 170)
)
```

```
# Calculate mean height for each age group
summary_df <- df %>%
  group_by(Age) %>%
  summarize(mean_height = mean(Height))

# Print the summary data frame
print(summary_df)
```



Output:

```
# A tibble: 4 × 2
  Age mean_height
  <dbl>         <dbl>
1    22         168
2    25         160
3    28         170
4    30         175
```

Arrange():

- The `arrange()` function in the `dplyr` package is used to reorder rows of a data frame based on one or more variables. It allows you to sort the rows in ascending or descending order of the specified variables.

```
arrange(data, column1, column2, ...)
```

Example:

```
# Assuming 'sales_data' is your dataframe
sales_data <- data.frame(
  Date = c("2022-01-01", "2022-01-03", "2022-01-02"),
  Revenue = c(1000, 1500, 1200)
)

# Arrange the data based on the 'Date' column
arranged_data <- arrange(sales_data, Date)

# View the arranged data
print(arranged_data)
```

Output:

	Date	Revenue
1	2022-01-01	1000
2	2022-01-02	1200
3	2022-01-03	1500

- Tidyr Packages:

The tidyr package in R is designed for data manipulation tasks related to reshaping and tidying data. It provides a set of functions to organize messy datasets into a more structured format, making it easier to work with and analyze data.

- What is Tidy Data?

Tidy data adheres to a specific structure:

- Each column represents a single variable.
- Each row represents a single observation.
- Each cell contains a single value.

This consistent format makes data analysis and visualization more efficient and intuitive.

- Basic functions:

1.gather():The gather() function is used to convert data from a wide format to a long format. It takes multiple columns and gathers them into key-value pairs, where each key-value pair represents a unique observation.

- Syntax:

```
gather(data, key, value, ..., na.rm = FALSE, convert = FALSE)
```

Explanation:

- data: The data frame to be reshaped.
- key: The name of the new key column that will contain the column names that are being gathered.
- value: The name of the new value column that will contain the values associated with the key column.
- ...: Columns to gather. You can specify multiple column names separated by commas.
- na.rm: Logical value indicating whether to remove rows with NA values. Default is FALSE.
- convert: Logical value indicating whether to convert character columns to factors. Default is FALSE.

Example:

```
# Sample data frame
df <- data.frame(
  Student = c("Alice", "Bob", "Charlie"),
  Math = c(85, 90, 75),
  Science = c(70, 80, 85),
  English = c(75, 85, 80)
)

print(df)
```

Output:

	Student	Math	Science	English
1	Alice	85	70	75
2	Bob	90	80	85
3	Charlie	75	85	80

Using `gather()`:

```
library(tidyr)

# Convert from wide to long format
long_df <- gather(df, Subject, Score, -Student)

print(long_df)
```

Output:

	Student	Subject	Score
1	Alice	Math	85
2	Bob	Math	90
3	Charlie	Math	75
4	Alice	Science	70
5	Bob	Science	80
6	Charlie	Science	85
7	Alice	English	75
8	Bob	English	85
9	Charlie	English	80

**Using spread():** It spreads key-value pairs back into multiple columns, converting data from long to wide format.

```
# Convert from long to wide format
wide_df <- spread(long_df, Subject, Score)

print(wide_df)
```

Output:

	Student	English	Math	Science
1	Alice	75	85	70
2	Bob	85	90	80
3	Charlie	80	75	85

Separate():

- This function splits a single column into multiple columns based on a delimiter or a fixed number of characters.

- Syntax:

```
separate(data, col, sep = delimiter, into = new_column_names)
```

- # Sample data frame

```
data <- data.frame(
```

```
  full_name = c("Alice Smith", "Bob Jones", "Charlie Brown")
```

```
)
```

```
# Separate full name into first and last names
```

```
separated_data <- separate(data, full_name, sep = " ", into = c("first_name",  
"last_name"))
```

```
print(separated_data)
```

Output:

```
  first_name last_name  
1    Alice    Smith  
2     Bob    Jones  
3  Charlie    Brown
```

Unite():

It combines multiple columns into a single column by concatenating their values.

- Syntax:

```
unite(data, col, into, sep = "")
```

- data: The data frame you are working with.
- col: The name of the new column you want to create.
- into: A vector of character strings specifying the names of the columns you want to unite.
- sep: (Optional) The character string to use as a separator between the values from the original columns (defaults to an empty string, meaning no separator).

# Sample data frame

```
data <- data.frame(  
  first_name = c("Alice", "Bob", "Charlie"),  
  last_name = c("Smith", "Jones", "Brown")  
)
```

# Combine first and last names into a full name column

```
united_data <- unite(data, full_name, into = c("first_name", "last_name"),  
  sep = " ")  
print(united_data)
```

Output:

```
first_name full_name  
1      Alice  Alice Smith  
2        Bob  Robert Jones  
3    Charlie Charles Brown
```

Complete():

This function ensures that all combinations of specified variables are present in the data, filling in missing combinations with NA values.

```
# Original data frame
df <- data.frame(
  ID = c(1, 2, 3),
  Value = c(10, 20, 30)
)

# Reshaped data frame
completed_df <- complete(df, ID = 1:5)
```

Output:

	ID	Value
1	1	10
2	2	20
3	3	30
4	4	NA
5	5	NA



Drop\_na():

- It removes rows with missing values (NA) from a data frame.

Example:

```
# Original data frame
df <- data.frame(
  ID = c(1, 2, NA),
  Value = c(10, NA, 30)
)

# Reshaped data frame
cleaned_df <- drop_na(df)
```

Output:

	ID	Value
1	1	10

Fill():

- This function fills in missing values in a column with the most recent non-missing value above it.

```
# Original data frame
df <- data.frame(
  ID = 1:5,
  Value = c(10, NA, 20, NA, 30)
)

# Reshaped data frame
filled_df <- fill(df, Value)
```

- Output:

	ID	Value
1	1	10
2	2	10
3	3	20
4	4	20
5	5	30

## Reshape2() Packages:

- R programming language has many methods to reshape the data using reshape package. melt() and cast() are the functions that efficiently reshape the data.
- Reshaping involves reorganizing the structure of your data, typically by changing the layout of rows and columns. The package offers functionality for tasks such as converting data between wide and long formats, restructuring data frames, and aggregating data.
- reshape2 is based around two key functions:

1.melt

2.cast

- melt takes wide-format data and melts it into long-format data.
- cast takes long-format data and casts it into wide-format data.

A dataset can be written in two different formats: wide and long.

- A wide format contains values that do not repeat in the first column.
- A long format contains values that do repeat in the first column.

Wide Format			
Team	Points	Assists	Rebounds
A	88	12	22
B	91	17	28
C	99	24	30
D	94	28	31

**Long Format**

Team	Variable	Value
A	Points	88
A	Assists	12
A	Rebounds	22
B	Points	91
B	Assists	17
B	Rebounds	28
C	Points	99
C	Assists	24
C	Rebounds	30
D	Points	94
D	Assists	28
D	Rebounds	31

**Wide Format**

Team	Points	Assists	Rebounds
A	88	12	22
B	91	17	28
C	99	24	30
D	94	28	31

Each value is unique in first column

Long Format		
Team	Variable	Value
A	Points	88
A	Assists	12
A	Rebounds	22
B	Points	91
B	Assists	17
B	Rebounds	28
C	Points	99
C	Assists	24
C	Rebounds	30
D	Points	94
D	Assists	28
D	Rebounds	31

The values in the first column repeat

Reshape vs reshape2:

- Syntax: The syntax of the main functions for reshaping data differs slightly between the two packages. For example, in reshape, the main functions are melt() and cast(), while in reshape2, they are melt() and dcast(). The function names and argument conventions may vary slightly.
- Performance: reshape2 is generally considered to be more efficient and faster than reshape, especially when dealing with larger datasets. The underlying implementation of some functions in reshape2 has been optimized for better performance.

- Additional features: reshape2 introduces some additional functions and features that are not present in reshape. For example, reshape2 provides the recast() function, which offers more flexibility than cast() for reshaping data. reshape2 also includes functions like colsplit() for splitting columns and melt.data.table() for melting data frames created with the data.table package.
- Development status: As of my last update in January 2022, reshape2 was actively maintained and received updates, while reshape had not been updated for several years. This suggests that reshape2 may be more actively supported and developed.

Syntax:

```
Variable_name<- melt(data = your_data_frame, id.vars = c("id_column1",
" id_column2", ...),
                    measure.vars = c("var_column1", "var_column2", ...),
                    variable.name = "new_variable_column_name",
                    value.name = "new_value_column_name")
```

Syntax explanation:

- data: The original data frame you want to melt.
- id.vars: Columns (or variables) in the original data frame that you want to keep unchanged (i.e., not melted).
- measure.vars: Columns (or variables) in the original data frame that you want to melt into key-value pairs.
- variable.name: The name for the new column that will store the variable names.
- value.name: The name for the new column that will store the values corresponding to the variable names.

Example:

```
df <- data.frame(  
  ID = 1:3,  
  Age = c(25, 30, 35),  
  Weight = c(60, 70, 80)  
)
```

```
melted_df <- melt(data = df, id.vars = "ID",  
                  measure.vars = c("Age", "Weight"),  
                  variable.name = "Variable",  
                  value.name = "Value")
```

Output:

	ID	Variable	Value
1	1	Age	25
2	2	Age	30
3	3	Age	35
4	1	Weight	60
5	2	Weight	70
6	3	Weight	80

- Syntax:
- `casted_df <- cast(data = your_molten_data_frame, formula = formula, ..., fun.aggregate = function, ...)`

#### Syntax Expansion:

- `data`: The molten data frame (typically obtained from `melt()`) that you want to cast back into a wide format.
- `formula`: A formula specifying the variables to use for casting. The formula should be in the form `LHS ~ RHS`, where LHS represents the columns that will become the new rows in the wide format, and RHS represents the columns that will become the new columns in the wide format.
- `...:` Additional arguments passed to `aggregate()`.
- `fun.aggregate`: An optional function to use for aggregating duplicate values. The default is `length`.

```
melted_df <- data.frame(  
  ID = rep(1:3, each = 2),  
  Variable = c("Age", "Age", "Weight", "Weight", "Height", "Height"),  
  Value = c(25, 30, 60, 70, 160, 165)  
)
```

```
casted_df <- cast(data = melted_df, formula = ID ~ Variable)
```



Output:

	ID	Age	Weight	Height
1	1	25	60	160
2	2	30	70	165
3	3	35	80	NA

Example:

```
# Load the reshape2 package
library(reshape2)

# Sample data
data <- data.frame(
  Student = c("John", "John", "Jane", "Jane"),
  Exam = c("Midterm", "Final", "Midterm", "Final"),
  Score = c(85, 90, 88, 95)
)

# Display the original data
print("Original data:")
print(data)
```

```

# Reshape data from long to wide format using dcast
wide_data <- dcast(data, Student ~ Exam, value.var = "Score")

# Display the reshaped data
print("Wide format data:")
print(wide_data)

# Reshape data from wide to long format using melt
long_data <- melt(data, id.vars = "Student", measure.vars = c("Midterm", "Final"))

# Display the reshaped data
print("Long format data:")
print(long_data)

```

Output:

```

[1] "Original data:"
  Student Exam Score
1   John Midterm   85
2   John  Final   90
3   Jane Midterm   88
4   Jane  Final   95
[1] "Wide format data:"
  Student Final Midterm
1   Jane    95     88
2   John    90     85
[1] "Long format data:"
  Student variable value
1   John Midterm    85
2   John  Final    90
3   Jane Midterm    88
4   Jane  Final    95

```

## Lubridate packages:

- The lubridate package is widely used in R for handling date-time data.
- Lubridate provides tools that make it easier to parse and manipulate dates.
- This package is also part of the tidyverse core that also houses readr, dplyr, and ggplot2.

- Install package:

```
install.packages('lubridate')
```

```
library(lubridate)
```

- Creating Date/Time Data
- There are three relevant data types when we talk about date/time data:
- Date - only has the date (e.g. 2020-05-15)
- Time - only has the time (e.g. 20:45:00)
- Date time - has both the date and time (e.g. 2020-05-15 20:45:00)
- several ways for us to create date/time data:
- we can create them from raw strings, from an existing date/time data, or from a dataset
- `ymd()`: Parses dates in the "year-month-day" format.

```
ymd("2024-02-27")
```

- **`mdy()`**: Parses dates in the "month-day-year" format.

```
mdy("02-27-2024")
```

- **`dmy()`**: Parses dates in the "day-month-year" format.

```
dmy("27-02-2024")
```

- **ymd\_hms()**: Parses date-time objects in the "year-month-day hour:minute:second" format.

```
ymd_hms("2024-02-27 12:30:45")
```

- **year(), month(), day()**: Extracts year, month, and day components from date-time objects, respectively.

```
date <- ymd("2024-02-27")
year(date)
month(date)
day(date)
```

- **week()**: Extracts the week of the year from date-time objects.

```
date <- ymd("2024-02-27")
week(date)
```

- **today()**: This function returns today's date.

```
today()
# Output: [1] "2024-02-27"
```

- **now()**: This function returns the current date and time.

```
now()
# Output: [1] "2024-02-27 14:30:00 UTC"
```

- **as\_datetime()**: This function converts various objects to date-time objects.

```
as_datetime(today())
# Output: [1] "2024-02-27 UTC"
```

- **as\_date(now())**: When you apply **as\_date()** to the result of **now()**, which gives the current date and time, it extracts only the date part, discarding the time information.

```
# Get the current date and time
now_datetime <- now()

# Convert to date
as_date(now_datetime)
```

Output:

```
[1] "2024-02-27"
```

### How to Create a Range of Dates in R:

- The as.Date() method takes as input a character date object and converts it to a Date object.

- Seq() method:

Syntax:

```
seq(from, to, by, length.out)
```

Parameters:

- from: Starting element of the sequence
- to: Ending element of the sequence
- by: Difference between the elements
- length.out: Maximum length of the vectors

creating date with help of seq function:

```
date<-as.Date("2021-03-02")
```

```
len<-10
```

```
seq(date,by="day",length.out=len)
```

- Output :

```
[1] "2021-03-02" "2021-03-03" "2021-03-04" "2021-03-05"  
[5] "2021-03-06" "2021-03-07" "2021-03-08" "2021-03-09"  
[9] "2021-03-10" "2021-03-11"
```

Increment by month:

```
date<-as.Date("2021-03-02")
```

```
len<-10
```

```
seq(date,by="month",length.out=len)
```

- Output:

```
[1] "2021-03-02" "2021-04-02" "2021-05-02" "2021-06-02"  
[5] "2021-07-02" "2021-08-02" "2021-09-02" "2021-10-02"  
[9] "2021-11-02" "2021-12-02"
```

**Define start and end date increment by days :**

```
Start<-as.Date("2021-03-02")
```

```
End<-as.Date("2021-03-10")
```

```
x<-seq(Start,End,"days")
```

```
print(x)
```

- Output:

```
[1] "2021-03-02" "2021-03-03" "2021-03-04" "2021-03-05"  
[5] "2021-03-06" "2021-03-07" "2021-03-08" "2021-03-09"  
[9] "2021-03-10"
```

Creating a range of dates:

```
Start<-as.Date("2021-03-02")
```

```
End<-as.Date("2021-03-10")
```

```
x<-seq(Start,End,"days")
```

```
print(x)
```

- Output:

```
[1] "2021-03-02" "2021-03-03" "2021-03-04" "2021-03-05"
```

```
[5] "2021-03-06" "2021-03-07" "2021-03-08" "2021-03-09"
```

```
[9] "2021-03-10"
```

## **R-Graph:**

### **Data visualization working with R graphics**

#### **Pie –chart:**

In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input. The additional parameters are used to control labels, color, title etc.

#### **Syntax**

```
pie(x, labels, radius, main, col, clockwise)
```

#### **Parameter description:**

**x** is a vector containing the numeric values used in the pie chart.

**labels** is used to give description to the slices.

**radius** indicates the radius of the circle of the pie chart.(value between –1 and +1).

**main** indicates the title of the chart.

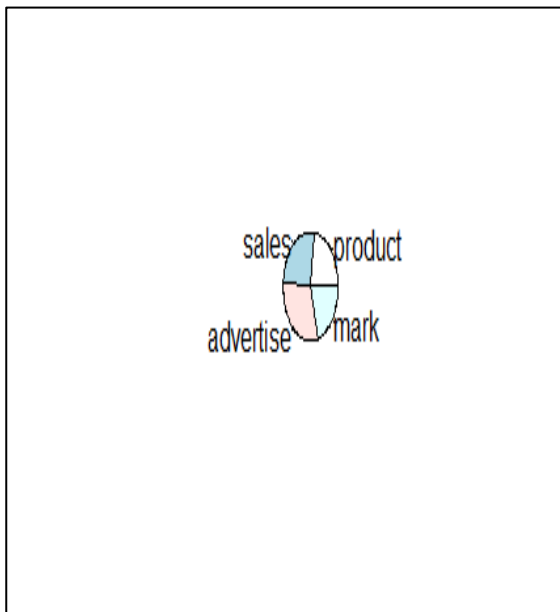
**col** indicates the color palette.

**clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

# create pie chart:

```
x<-c(60,70,80,55)
```

```
labels<-c("product","sales","advertise","mark") pie(x,labels)
```



# main fun() # of chart

```
pie(x,labels,main="Departments")
```





