

R Programming

UNIT – IV

Factors

Factors and Levels:

A **factor** is a data structure, which is used to represent categorical data. It stores a vector of values along with their unique categories (*levels*).

Examples of categorical data are:

- Gender: Male ,Female
- Music: Rock, Pop, Classic, Jazz
- Result : Pass , Fail
- Temperature: Low, Medium, High

Creating a Factor:

To create a factor in R you need to use the function called ***factor()***. The argument to this factor() is the vector.

Syntax:

```
fact<- factor(vector)
```

Example:

```
# Vector
> gender <- c("Male", "Female", "Male", "Male", "Female", "Male")
# Factor
> gen_factor <- factor(gender)
> print(gen_factor)
# output:
[1] Male   Female Male    Male   Female Male
Levels: Female Male
```

Levels in Factors:

Levels in a factor are the unique categorical values that are recognized by R. By default, R arranges them in alphabetical order, but you can change this manually.

The ***levels()*** function is used to get the levels from the factor and rename/change the levels in the factor.

Example:

```
# To get the levels
> levels(gen_factor)
[1] "Female" "Male"
```

```
# To change/rename the levels
> levels(gen_factor) <- c("F", "M")
```

```
# Updated levels
> levels(gen_factor)
[1] "F" "M"
```

```
# factor with Updated levels
> print(gen_factor)
[1] M F M M F M
Levels: F M
```

- To see the internal structure of a factor, use **str()** function.

Example:

```
> str(gen_factor)
# Output:
Factor w/ 2 levels "F", "M": 2 1 2 2 1 2
```

- To find the length of a factor, use **length()** function.

Example:

```
> print(length(gen_factor))
# Output:
[1] 6
```

The length of a factor is still determined by the length of the data instead of being based on the number of levels.

Common Functions Used with Factors:

In addition to factors, we have another function in the apply family called **tapply()**. We will explore this function along with two other functions that are often used with factors: **split()** and **by()**.

- **tapply() function:**

The tapply() function applies a function to subsets of a vector, grouped by a factor.

Syntax:

tapply(x, f, g)

where,

x as a vector,

f as a factor, and

g as a function.

Example 1:

```
> # vector
> ages <- c(21,22,24,41,36,25)
> # factor
> gen_factor <- factor(c("Male", "Female", "Male", "Male",
"Female", "Male"))
> # tapply()
> tapply(ages,gen_factor,mean)
# output:
Female    Male
  29.00   27.75
```

Example 2:

```
> # vector
> salary <- c(50000, 55000, 48000, 60000, 65000, 70000)
> # factor
> department <- factor(c("HR", "IT", "HR", "IT", "Finance",
"Finance"))
> # tapply()
> tapply(salary, department, mean)
# output:
Finance      HR      IT
  67500   49000   57500
```

- ***split()* function:**

The `split()` function divides a vector or dataset into groups based on a factor.

Syntax:

```
split(x,f)
```

where,

x as a vector or data frame,

f as a factor

Example 1:

```
> # split()
> split(ages,gen_factor)
# Output:
$Female
[1] 22 36

$Male
[1] 21 24 41 25
```

Note: Each gender gets a separate list.

Example 2:

```
> # split()
> split(salary, department)
# Output:
$Finance
[1] 65000 70000

$HR
[1] 50000 48000

$IT
[1] 55000 60000
```

Note: Each department gets a separate list.

- ***by()* function:**

The ***by()*** function is useful when working with data frames, applying functions to subsets.

Syntax:

by(x, f, g)

where,

x as data frame or matrix

f as a factor

g as a function

Example:

```
> # creating Data Frame
> Name <- c("Madhu", "Mohan", "Madhan", "Murali", "Mahesh", "Manoj")
> Salary <- c(50000, 55000, 48000, 60000, 65000, 70000)
> Department <- c("HR", "IT", "HR", "IT", "Finance", "Finance")
> emp_df <- data.frame(Name, Salary, Department)
> print(emp_df)
# Output:
  Name Salary Department
1 Madhu  50000         HR
2 Mohan  55000         IT
3 Madhan 48000         HR
4 Murali 60000         IT
5 Mahesh 65000    Finance
6 Manoj  70000    Finance
```

```

# by() function
> result <- by(emp_df$Salary, emp_df$Department, mean)
> print(result)
# Output:
emp_df$Department: Finance
[1] 67500
-----
emp_df$Department: HR
[1] 49000
-----
emp_df$Department: IT
[1] 57500

```

Function	Purpose
tapply()	Applies a function to subsets of a vector, grouped by a factor.
split()	Divides a dataset into subsets based on a factor.
by()	Applies a function to a data frame by groups.

Tables

Working with Tables:

In R, **tables** are used to summarize categorical data and count occurrences of values.

Creation of Table:

To create table in R, use **table()** function. And it has arguments as vectors.

Syntax:

```
table (data,...)
```

Example:

```

# Working with single vector
# vector
> gender <- c("Male", "Female", "Male", "Female", "Male", "Male")

# table creation
> tbl_gender <- table(gender)

> print(tbl_gender)
# Output:
gender
Female    Male
      2      4

```

Note: The function returns a frequency table for each unique value in a vector.

Example:

```
# Working with two vectors
# vector1
> gender <- c("Male", "Female", "Male", "Female", "Male", "Male")

# vector 2
> department <- c("HR", "IT", "HR", "IT", "Finance", "Finance")

# table creation
> tbl_emp <- table(department,gender)

> print(tbl_emp)
# output:
```

	gender	
department	Female	Male
Finance	0	2
HR	0	2
IT	2	0

Note: This creates a cross-tabulation of department and gender. This is also called as Two-Way Table (Contingency Table).

Matrix/Array-Like Operations on Tables:

In R, tables can be treated similarly to matrices and arrays, allowing you to perform various operations like **indexing**, **applying functions**, and **aggregation**.

- **Indexing Tables:**

You can **access elements** from tables using **row and column indices** just like a matrix.

Example:

```
> print(tbl_emp)
# output:
```

	gender	
department	Female	Male
Finance	0	2
HR	0	2
IT	2	0

```
# Accessing specific elements
> print(tbl_emp["IT","Female"])
# Output : [1] 2

> print(tbl_emp[1,1])
# Output : [1] 0

> print(tbl_emp[1,2])
# Output : [1] 2
```

- **Subsetting Tables**

Similar to matrices, you can subset tables using **row and column indices**.

Example:

```
# All genders in Finance
> sub_tab1 <- tbl_emp["Finance", ]
> print(sub_tab1)
# Output:
Female    Male
      0      2
# All departments with Male gender
> sub_tab2 <- tbl_emp[, "Male" ]
> print(sub_tab2)
# Output:
Finance      HR      IT
      2      2      0
# All genders in Finance and HR
> subset1 <- tbl_emp[1:2, ]
> print(subset1)
# Output:
      gender
department Female Male
  Finance      0      2
    HR      0      2
# All departments with Female gender
> subset2 <- tbl_emp[, 1 ]
> print(subset2)
# Output:
Finance      HR      IT
      0      0      2
```

- **Applying Functions to Tables**

Use the **apply()** function to perform **row-wise or column-wise operations**.

Example:

```
# row sums
> row_sums <- apply(tbl_emp, 1, sum)
> print(row_sums)
# Output:
Finance      HR      IT
      2      2      2
# column sums
> col_sums <- apply(tbl_emp, 2, sum)
> print(col_sums)
# Output:
Female    Male
      2      4
```

- **Aggregating Tables**

Use **addmargins()** function to **aggregate along a margin** (rows or columns).

Example:

```
# Sum across rows and columns
```

```
> addmargins(tbl_emp)
```

```
      gender
department Female Male Sum
  Finance      0     2    2
    HR      0     2    2
    IT      2     0    2
    Sum      2     4    6
```

```
# Sum across rows (margin = 1)
```

```
> addmargins(tbl_emp,1)
```

```
      gender
department Female Male
  Finance      0     2
    HR      0     2
    IT      2     0
    Sum      2     4
```

```
# Sum across columns (margin = 2)
```

```
> addmargins(tbl_emp,2)
```

```
      gender
department Female Male Sum
  Finance      0     2    2
    HR      0     2    2
    IT      2     0    2
```

- **Dimensions and Levels of a Table**

Use **dimnames()** function to extract dimensions and levels from a table.

Example:

```
# To get Dimensions and Levels
```

```
> dimnames(tbl_emp)
```

```
# Output:
```

```
$department
```

```
[1] "Finance" "HR"      "IT"
```

```
$gender
```

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

Extended Example: Extracting a Subtable

Let us continue working with our employee table example:

```
> print(tbl_emp)
# output:
      gender
department Female Male
Finance      0      2
HR           0      2
IT           2      0
```

Suppose we wish to extract a subtable with the details of “**Male**” employees in “**HR**” department.

A subtable that looks like this:

```
      gender
department Male
HR         2
```

To achieve this, we will create a function **subtable()** below performs subtable extraction. It has two arguments:

- *tbl*: The table of interest, of class "table".
- *subnames*: A list specifying the desired subtable extraction.

Example:

```
# subtable() function
subtable <- function(tbl, subnames) {
  # Get all dimension names (like row and column labels)
  dims <- dimnames(tbl)

  # Create a list to hold the parts we want to keep
  ls <- list()

  # Go through each dimension of the table
  for (dim_name in names(dims)) {
    if (dim_name %in% names(subnames)) {
      # If the user gave a filter for this dimension, use it
      ls[[dim_name]] <- subnames[[dim_name]]
    } else {
      # Otherwise, keep all values (no filter)
      ls[[dim_name]] <- dims[[dim_name]]
    }
  }

  # Extract and return the subtable using the selected parts
  return(tbl[ls[[1]], ls[[2]], drop = FALSE])
}
```

```

# Sample Run 1:

# Calling subtable() function to
# Extract rows for HR only and gender Male only
result <- subtable(tbl_emp, list(department = c("HR"),
                                gender = c("Male")))

print(result)

# Output:
      gender
department Male
      HR      2

# Sample Run 2:

# Calling subtable() function to
# Extract rows for HR & IT and gender Female only
result <- subtable(tbl_emp, list(
  department = c("HR", "IT"),
  gender = c("Female")
))

print(result)

# Output:
      gender
department Female
      HR      0
      IT      2

```

Extended Example: Finding the Largest Cells in a Table

It can be challenging to view a huge table, with many rows or columns. One approach might be to focus on the cells with the largest frequencies.

To achieve this, we will create a function **tabdom()**. It shows the most common frequencies in a table.

tabdom(tbl,k)

It has two arguments:

- *tbl*: The table of interest, of class "table".
- *k*: number of cells.

Example:

```
# finds the cells in table tbl_emp with the k highest frequencies
tabdom <- function(tbl, k) {
  # Convert the table to a data frame
  tbl_df <- as.data.frame(tbl)

  # Sort the rows by frequency (highest first)
  sorted_tbl <- tbl_df[order(tbl_df$Freq, decreasing = TRUE), ]

  # Get the top k rows
  top_k <- sorted_tbl[1:k, ]

  # Return the result
  return(top_k)
}

# Finding the top 2 Largest Cells
> result <- tabdom(tbl_emp,2)
> print(result)
# Output:
  department gender Freq
3         IT Female    2
4   Finance   Male    2

# Finding the top 3 Largest Cells
> result <- tabdom(tbl_emp,3)
> print(result)
# Output:
  department gender Freq
3         IT Female    2
4   Finance   Male    2
5         HR   Male    2
```

In the above example,

as.data.frame(tbl) : Turns the table into a data frame with row/column names and frequencies

order (... , decreasing = TRUE): Sorts the data frame by frequency from high to low

[1:k,]: Picks the top k rows from the sorted list.

Math Functions

R includes an extensive set of built-in math functions. Here is a partial list math functions:

Common Math Functions:

- **exp()**: It returns Exponential (e^x) value with base e.

Example:

```
> print(exp(1))  
[1] 2.718282
```

```
> print(exp(2))  
[1] 7.389056
```

- **log()**: It returns Natural logarithm value.

Example:

```
> print(log(2))  
[1] 0.6931472
```

```
> print(log(10))  
[1] 2.302585
```

- **log10()**: It returns Logarithm base 10 value.

Example:

```
> print(log10(100))  
[1] 2
```

```
> print(log10(10))  
[1] 1
```

- **sqrt()**: It returns Square root value.

Example:

```
> print(sqrt(16))  
[1] 4
```

```
> print(sqrt(36))  
[1] 6
```

- **abs()**: It returns Absolute value.

Example:

```
> print(sqrt(16))  
[1] 4
```

```
> print(sqrt(24))  
[1] 4.898979
```

Trigonometric Math Functions:

- **sin():** It returns Trigonometric sine value.

Example:

```
> print(sin(90))  
[1] 0.8939967  
  
> print(sin(0))  
[1] 0
```

- **cos():** It returns Trigonometric cosine value.

Example:

```
> print(cos(0))  
[1] 1  
  
> print(cos(45))  
[1] 0.525322
```

- **tan():** It returns Trigonometric Tangent value.

Example:

```
> print(tan(45))  
[1] 1.619775  
  
> print(tan(90))  
[1] -1.9952
```

Other Math Functions:

- **min()** and **max()**: It returns a Minimum value and maximum value within a vector.

Example:

```
> nums <- c(23,12,45,6,78,24,67,55)  
> print(min(nums))  
[1] 6  
  
> print(max(nums))  
[1] 78
```

- **which.min ()** and **which.max ()**: It returns the index of the minimal element and maximal element of a vector.

Example:

```
> nums <- c(23,12,45,6,78,24,67,55)  
> print(which.min(nums))  
[1] 4  
> print(which.max(nums))  
[1] 5
```

- **pmin()** and **pmax()**: It returns Pair Wise / Element-wise minima and maxima of several vectors.

Example:

```
> nums_1 <- c(23,12,45,6,78,24,67,55)
> nums_2 <- c(45,42,15,9,68,54,27,85)

> print(pmin(nums_1,nums_2))
[1] 23 12 15 6 68 24 27 55

> print(pmax(nums_1,nums_2))
[1] 45 42 45 9 78 54 67 85
```

- **sum()** and **prod()**: Sum and product of the elements of a vector.

Example:

```
> num_v <- c(1,2,3,4,5)
> print(sum(num_v))
[1] 15
> print(prod(num_v))
[1] 120
```

- **cumsum()** and **cumprod()**: Cumulative sum and product of the elements of a vector.

Example:

```
> num_v <- c(1,2,3,4,5)
> print(cumsum(num_v))
[1] 1 3 6 10 15

> print(cumprod(num_v))
[1] 1 2 6 24 120
```

- **round()**: It round number to the closest integer.

Example:

```
> print(round(3.6))
[1] 4

> print(round(3.2))
[1] 3
```

- **floor()**: It round number to the closest integer which is less than given number.

Example:

```
> print(floor(3.6))
[1] 3

> print(floor(3.2))
[1] 3
```

- **ceiling():** It round to the closest integer which is greater than number.

Example:

```
> print(ceiling(3.6))
[1] 4

> print(ceiling(3.2))
[1] 4
```

- **factorial():** It returns the Factorial of the given number.

Example:

```
> print(factorial(5))
[1] 120

> print(factorial(4))
[1] 24
```

Extended Example: Calculating a Probability

Suppose we have n independent events, and the i th event has the probability p_i of occurring. What is the probability of **exactly one** of these events occurring?

Suppose first that $n = 3$ and our events are named A, B, and C. Then we break down the computation as follows:

$$\begin{aligned}
 P(\text{exactly one event occurs}) = & \\
 & P(\text{A and not B and not C}) + \\
 & P(\text{not A and B and not C}) + \\
 & P(\text{not A and not B and C})
 \end{aligned}$$

In the above, $P(\text{A and not B and not C})$ would be $(p_A * (1 - p_B) * (1 - p_C))$, and so on.

Example:

Here is the code to compute this, with our probabilities p_i contained in the vector p .

```
#function to calculate probability of exactly one event occurs
exactlyone <- function(p) {
  n <- length(p)
  total <- 0
  # Iterate through all event probabilities
  for (i in 1:n) {
    prob_one <- p[i] * prod(1 - p[-i])
    total <- total + prob_one
  }
  return(total)
}
```

Sample Run 1:

```
# Calling Function
p <- c(0.2, 0.5, 0.3)
event_pr <- exactlyone(p)
cat("The Probability that exactly one event occurs:",event_pr)
```

Output:

The Probability that exactly one event occurs: 0.47

Sample Run 2:

```
# Calling Function
p <- c(0.3, 0.5, 0.7)
event_pr <- exactlyone(p)
cat("The Probability that exactly one event occurs:",event_pr)
```

Output:

The Probability that exactly one event occurs: 0.395

Cumulative Sums and Products:

The functions *cumsum()* and *cumprod()* return cumulative sums and products.

Example:

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

# Cumulative Sums
> cs <- cumsum(x)
> print(cs)
# Output:
[1] 1 3 6 10 15

# Cumulative Products
> cp <- cumprod(x)
> print(cp)
# Output:
[1] 1 2 6 24 120
```

In the above, the function **cumsum()** works as follows:

In a vector *x*, the sum of the first element is **1**, the sum of the first two elements is **3**, the sum of the first three elements is **6**, the sum of the first four elements is **10**, and the sum of the first five elements is **15**.

The function **cumprod()** works as follows:

In a vector *x*, the product of the first element is **1**, the product of the first two elements is **2**, the product of the first three elements is **6**, the product of the first four elements is **24**, and the product of the first five elements is **120**.

Minima and Maxima:

There is quite a difference between ***min()*** and ***pmin()***. The former ***min()*** function simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if ***pmin()*** is applied to two or more vectors, it returns a vector of the *pair-wise minima*, hence the name ***pmin()***.

The ***max()*** and ***pmax()*** functions work similarly to the ***min()*** and ***pmin()*** functions.

Example: In case of **vectors**

```
> x <- c(12,9,23)
> y <- c(4,13,44)
> print(min(x,y))
# Output:
[1] 4
> print(max(x,y))
# Output:
[1] 44
> print(pmin(x,y))
# Output:
[1] 4 9 23
> print(pmax(x,y))
# Output:
[1] 12 13 44
```

Example: In case of **Matrix**

```
> z <- matrix(c(1,5,6,2,3,2),nrow=3,ncol=2)
> print(z)
# Output:
      [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2
> print(min(z))
# Output:
[1] 1
> print(min(z[,1],z[,2]))
# Output:
[1] 1
> print(pmin(z[,1],z[,2]))
# Output:
[1] 1 3 2
> print(pmax(z[,1],z[,2]))
# Output:
[1] 2 5 6
```

In the above example, ***min()*** computed the smallest value in (1,5,6,2,3,2). But the call to ***pmin()*** computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

Calculus:

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

Example:

```
> D(expression(exp(x^2)), "x") # derivative
exp(x^2) * (2 * x)

> integrate(function(x) x^2, 0, 1)
0.3333333 with absolute error < 3.7e-15
```

You can find R packages for differential equations (*odesolve*), for interfacing R with the Yacas symbolic math system (*ryacas*), and for other calculus operations. These packages, and thousands of others, are available from the Comprehensive R Archive Network (CRAN).

Functions for Statistical Distributions:

In R, you can work with statistical distributions using a variety of built-in functions. Below are some of the key functions for common distributions:

Distribution	Density/pmf	cdf	Quantiles	Random Numbers
<i>Normal</i>	<code>dnorm()</code>	<code>pnorm()</code>	<code>qnorm()</code>	<code>rnorm()</code>
<i>Chi square</i>	<code>dchisq()</code>	<code>pchisq()</code>	<code>qchisq()</code>	<code>rchisq()</code>
<i>Binomial</i>	<code>dbinom()</code>	<code>pbinom()</code>	<code>qbinom()</code>	<code>rbinom()</code>

Normal Distributions Functions:

dnorm():

The *dnorm(x, mean, sd)* returns the height of the normal distribution curve (i.e., probability density) at point x.

Example: Normal Distribution with mean = 100 and sd = 15

```
> dnorm(100, mean = 100, sd = 15)
# Output:
[1] 0.02659615
```

It checks the density at the mean, where the bell curve is tallest.

pnorm():

The *pnorm(x, mean, sd)* gives the cumulative probability that a normally distributed random variable is less than or equal to x.

Example: Normal Distribution with mean = 100 and sd = 15

```
> pnorm(110, mean = 100, sd = 15)
# Output:
[1] 0.7475075
```

About 74.75% of values are below 110.

qnorm():

The *qnorm(p, mean, sd)* returns the quantile (or cut-off value) for a given cumulative probability p.

Example: Normal Distribution with mean = 100 and sd = 15

```
> qnorm(0.90, mean = 100, sd = 15)
# Output:
[1] 119.2233
```

90% of the students scored below 119.23

rnorm():

The *rnorm(n, mean, sd)* generates n random values from a normal distribution with the given mean and standard deviation.

Example: Normal Distribution with mean = 100 and sd = 15

```
> rnorm(5, mean = 100, sd = 15)
# Output:
[1] 108.17404 79.47250 107.41886 77.67176 120.89173
```

These could represent randomly generated student test scores.

Chi square Distributions Functions:

dchisq():

The *dchisq(x, df)* returns the density (height) of the Chi-square distribution at a value x, with df = degrees of freedom.

Example: Density at x = 2 with df = 3

```
> dchisq(2, df = 3)
# Output:
[1] 0.2075537
```

This means the value of the Chi-square PDF at x = 2 is about 0.2076 when degrees of freedom = 3.

pchisq():

The *pchisq(q, df)* gives the cumulative probability for the Chi-square distribution.

Example: Probability $X \leq 4$ with df = 3

```
> pchisq(4, df = 3)
# Output:
[1] 0.7385359
```

There is a 73.85% chance that a Chi-square random variable with df = 3 is less than or equal to 4.

qchisq():

The *qchisq(p, df)* returns the **Chi-square value x** such that:

$$P(X \leq x) = p,$$

where X follows a **Chi-square distribution** with df degrees of freedom.

Example: 95th percentile (critical value) for $df = 4$

```
> qchisq(0.95, df = 4)
# Output:
[1] 9.487729
```

The value **9.49** is the **cut-off** where **95% of values lie below it**. (i.e., $P(X \leq 9.49) = 0.95$ for $df = 4$)

rchisq():

The *rchisq(n, df)* generates n random numbers from a Chi-square distribution with df degrees of freedom.

Example: Generate 5 random Chi-square values with $df = 4$

```
> rchisq(5, df = 4)
# Output:
[1] 0.7686753 9.2089134 0.1694429 2.9136730 1.2949039
```

These values are **random samples** from a Chi-square distribution.

Binomial Distributions Functions:

dbinom():

The *dbinom(x, size, prob)* returns the probability of getting exactly x successes in $size$ trials, where each trial has a success probability of $prob$.

Example: $P(X = 6)$ for 10 fair coin flips

```
> dbinom(6, size = 10, prob = 0.5)
# Output:
[1] 0.2050781
```

So there's about a 20.5% chance of getting exactly 6 heads.

pbinom():

The *pbinom(q, size, prob)* gives you the **probability of getting $\leq q$ successes** in $size$ trials, where the probability of success in each trial is $prob$.

Example: $P(X \leq 6)$ in 10 coin flips ($p = 0.5$)

```
> pbinom(6, size = 10, prob = 0.5)
# Output:
[1] 0.828125
```

There is about an 82.81% chance of getting 6 or fewer heads in 10 coin tosses.

qbinom():

The `qbinom(p, size, prob)` returns the smallest number of successes x such that the cumulative probability $P(X \leq x)$ is at least p when performing `size` trials with success probability `prob`.

Example: 90th percentile for a binomial process.

```
> qbinom(0.90, size = 15, prob = 0.3)
# Output:
[1] 7
```

This gives the minimum number of defective items for which the cumulative probability reaches 90%.

rbinom():

The `rbinom(n, size, prob)` returns a vector of random numbers, where each number represents the **count of successes** in `size` trials.

Example:

```
> results <- rbinom(n = 5, size = 10, prob = 0.5)
> print(results)
[1] 4 3 6 4 5
```

Each value in `results` indicates the number of heads obtained in 10 coin flips in that experiment. For instance, if one of the outcomes is 6, it means in that experiment 6 heads came up out of 10 flips.