Basic R

Md Rasheduzzaman

2025-05-09

Data types, variables, vectors, data frame, functions

Table of contents

1 L2: Data Representation **3**
1.1 Using R as a Calculator . . . 3
1.2 Variables . . . 4
1.2.1 Integer and Modulus division again . . . 5
1.3 Rounding . . . 6
1.4 Logical Operations . . . 7
1.5 Help and Documentation . . . 8
1.6 Working with Vectors . . . 9
1.6.1 Vector Operations . . . 10
1.7 Data Frame . . . 11
1.7.1 Gene Expression Table . . . 11
1.8 Homeworks . . . 12
1.8.1 Deadline . . . 13

2 L3: Data Transformation **13**
2.1 Getting Started . . . 13
2.1.1 Installation of R Markdown . . . 13
2.1.2 Basic Setup for Today's Session . . . 14
2.1.3 Building on Last HW: . . . 14
2.1.4 Preamble on random variables (RV): . . . 16
2.1.5 Basic Stuffs: Atomic Vector . . . 17
2.1.6 Basic Stuffs: Matrices . . . 18
2.1.7 Basic Stuffs: List . . . 31
2.2 Homeworks: Matrix and List Operations . . . 38
2.2.1 Protein Quantification in Biological Samples . . . 38
2.2.2 Tasks . . . 39

2.2.3 Interpretation Questions . . . 39
2.2.4 Gene-to-Protein Translation . . . 39
2.2.5 Tasks . . . 40
2.2.6 Visualization Tasks . . . 40
2.2.7 Interpretation Questions . . . 41
2.2.8 Animal Breeding – Economic Ranking of Bulls by Traits . . . 41
2.2.9 Tasks . . . 42
2.2.10 Visualization Tasks . . . 42
2.2.11 Interpretation Questions . . . 42
2.2.12 Plant Breeding – Trait Contributions from Parental Lines . . . 43
2.2.13 Tasks . . . 43
2.2.14 Visualization Tasks . . . 44
2.2.15 Interpretation Questions . . . 44
2.2.16 Managing Matrices and Weight Vectors Using Lists in R . . . 45
2.2.17 Step 1: Create a master list . . . 45
2.2.18 Tasks . . . 45
2.2.19 Visualization Tasks . . . 46
2.2.20 Interpretation Questions . . . 46
2.3 Factor Variables . . . 46
2.3.1 Creating Factors . . . 47
2.3.2 Factor Operations . . . 48
2.4 Subsetting Data . . . 51
2.4.1 Vectors . . . 51
2.4.2 Data Frames . . . 52
2.4.3 Row Names in Data Frames . . . 55
2.5 Homework: Factors, Subsetting, and Biological Insight . . . 57
2.6 Handling Missing/Wrong Values . . . 59
2.6.1 Identifying Issues . . . 59
2.6.2 Fixing Data . . . 60
2.7 Data Transformation . . . 61
2.7.1 Introduction to Outliers . . . 61
2.7.2 Identifying Outliers . . . 62
2.7.3 Transforming Vectors . . . 62
2.7.4 Logical Expressions . . . 65
2.7.5 Logical Operators . . . 66
2.7.6 Logical Functions . . . 67
2.8 Conditionals . . . 68
2.8.1 `if-else` statement . . . 68
2.8.2 `elseif` statement for vectors . . . 68
2.8.3 `for` loop . . . 71
2.8.4 `while` loop . . . 72
2.8.5 `next` and `break` . . . 73

2.9 Writing Functions in R 73
2.9.1 Flag gene expression 73
2.9.2 Function with multiple arguments 74
2.9.3 Return a list 74
2.9.4 Practical Session 75
2.10 Summary of the Lesson 75
2.11 Homework 76

1 L2: Data Representation

1.1 Using R as a Calculator

Let's do some basic calculation.

```
5+3
```

```
[1] 8
```

```
3+2
```

```
[1] 5
```

```
3-2
```

```
[1] 1
```

```
3*2
```

```
[1] 6
```

```
3/2 #normal division
```

```
[1] 1.5
```

```
7 %/% 2 #integer division, only the quotient
```

```
[1] 3
```

```r
5 %% 3 #modulus division, the remainder
```

```
[1] 2
```

```r
(10-5)*(2+4) #use of parentheses
```

```
[1] 30
```

```r
10-5*2+4 #Noticed BODMAS?
```

```
[1] 4
```

```r
(10-5)*(2+4) #Noticed BODMAS
```

```
[1] 30
```

```r
7/(1+3); 7/1+3 #multi-line codes, separated with semi-colon
```

```
[1] 1.75
```

```
[1] 10
```

```r
1+2; log(1); 1/10 #more multi-line codes
```

```
[1] 3
```

```
[1] 0
```

```
[1] 0.1
```

1.2 Variables

Variables are variable. We have freedom to name them as we wish. But make any variable name meaningful and identifiable.

```
a <- 5 #assign value 5 to a
b = 10
a
```

```
[1] 5
```

```
b
```

```
[1] 10
```

```
a <- a + 10
b = b + 15
a
```

```
[1] 15
```

```
a^2 #a squared
```

```
[1] 225
```

```
a**2 #a squared again, in a different way.
```

```
[1] 225
```

```
a^3 #a qubed
```

```
[1] 3375
```

> **Note**
>
> `<-` and `=` are used to assign values. It is not mathematical equality. `b <- b + 15` might make better sense than `b = b + 15`.

1.2.1 Integer and Modulus division again

Do some more practice.

```
7/3
```

```
[1] 2.333333
```

```
7%/%3
```

```
[1] 2
```

```
7%%3
```

```
[1] 1
```

1.3 Rounding

Some important functions we apply on numerical values

```
x <- 9/4
floor(x)
```

```
[1] 2
```

```
ceiling(x)
```

```
[1] 3
```

```
round(x)
```

```
[1] 2
```

```
round(x, 2) #round till 2 decimal points
```

```
[1] 2.25
```

1.4 Logical Operations

Get to know TRUE/FALSE in R.

```
a = 5
b = 7
c = 10
d = 3
a == b #is a equal to b? Ans: No/FALSE
```

```
[1] FALSE
```

```
a != b #is a not equal to b? Ans: Yes/TRUE
```

```
[1] TRUE
```

```
a > b #is a greater than b? Ans: FALSE
```

```
[1] FALSE
```

```
a < b #is a less than b? Ans: TRUE
```

```
[1] TRUE
```

```
a >= b #is a greater than or equal to b? Ans: FALSE
```

```
[1] FALSE
```

```
a <= b #is a less than or equal to b? Ans: TRUE
```

```
[1] TRUE
```

```
a < b | d > b #is a less than b OR d greater than b?
```

```
[1] TRUE
```

```
#It's answer will be TRUE OR FALSE --> So, TRUE
a < b & c > d #is a less than b AND a greater than b? It's answer will be TRUE AND TRUE --
```

```
[1] TRUE
```

```
a < b & d > c #is a less than b AND a greater than b? It's answer will be TRUE AND FALSE -
```

```
[1] FALSE
```

1.5 Help and Documentation

But how to know more about a function? The package/library developer have written helpful documentation for us.

```
?log
example(log)
```

```
log> log(exp(3))
[1] 3

log> log10(1e7) # = 7
[1] 7

log> x <- 10^-(1+2*1:9)

log> cbind(deparse.level=2, # to get nice column names
log+       x, log(1+x), log1p(x), exp(x)-1, expm1(x))
          x   log(1 + x)     log1p(x)   exp(x) - 1     expm1(x)
 [1,] 1e-03 9.995003e-04 9.995003e-04 1.000500e-03 1.000500e-03
 [2,] 1e-05 9.999950e-06 9.999950e-06 1.000005e-05 1.000005e-05
 [3,] 1e-07 1.000000e-07 1.000000e-07 1.000000e-07 1.000000e-07
 [4,] 1e-09 1.000000e-09 1.000000e-09 1.000000e-09 1.000000e-09
 [5,] 1e-11 1.000000e-11 1.000000e-11 1.000000e-11 1.000000e-11
 [6,] 1e-13 9.992007e-14 1.000000e-13 9.992007e-14 1.000000e-13
 [7,] 1e-15 1.110223e-15 1.000000e-15 1.110223e-15 1.000000e-15
 [8,] 1e-17 0.000000e+00 1.000000e-17 0.000000e+00 1.000000e-17
 [9,] 1e-19 0.000000e+00 1.000000e-19 0.000000e+00 1.000000e-19
```

```
?log()
```

1.6 Working with Vectors

What is a vector? See the example and think.

```
x <- c(1, 2, 3, 4, 5) #c means concatenate
z <- 1:5 #consecutively, from 1 through 5. A short-hand notation using :
y <- c(3, 6, 9, 12, 15, 20)
length(x)
```

```
[1] 5
```

```
mode(x)
```

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

```
is(x)
```

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

```
x[1] #first entry in vector y
```

```
[1] 1
```

```
x[2:5] #2nd to 5th entries in vector y
```

```
[1] 2 3 4 5
```

```
DNA <- c("A", "T", "G", "C") #character vector. Notice the quotation marks.
dec <- c(10.0, 20.5, 30, 60, 80.9, 90, 100.7, 50, 40, 45, 48, 56, 55) #vector of floats. A
dec[c(1:3, 7:length(dec))] #1st to 3rd and then 7th till the end of vector `dec`. Output a
```

```
 [1]  10.0  20.5  30.0 100.7  50.0  40.0  45.0  48.0  56.0  55.0
```

1.6.1 Vector Operations

Notice the element-wise or index-wise mathematical operations (`+`, `/`, `log2()`, `round()`, etc.). Noticed?

```
x <- 1:10
y <- 2:11
#x and y are of same length
x + y
```

```
 [1]  3  5  7  9 11 13 15 17 19 21
```

```
y / x
```

```
 [1] 2.000000 1.500000 1.333333 1.250000 1.200000 1.166667 1.142857 1.125000
 [9] 1.111111 1.100000
```

```
log2(x)
```

```
 [1] 0.000000 1.000000 1.584963 2.000000 2.321928 2.584963 2.807355 3.000000
 [9] 3.169925 3.321928
```

```
round(log2(x), 1) #log2 of all the values of `x`, 1 digit after decimal to round.
```

```
 [1] 0.0 1.0 1.6 2.0 2.3 2.6 2.8 3.0 3.2 3.3
```

```
round(log2(x), 3) #same logic
```

```
 [1] 0.000 1.000 1.585 2.000 2.322 2.585 2.807 3.000 3.170 3.322
```

> **Note**
>
> Nested functions work inside out. Think again about `round(log2(x), 1)` and you will see it. At first, it is making `log2` of vector `x` and then it is rounding the log2 values to one digit after decimal. Got it?

1.7 Data Frame

Now, it's time to use vectors to make data sets…..

```
names <- c("Mina", "Raju", "Mithu", "Lali")
gender <- c("Female", "Male", "Female", "Female")
age <- c(15, 12, 2, 3)
is_human <- c(TRUE, TRUE, FALSE, FALSE)
cartoon <- data.frame(names, gender, age, is_human)
write.table(cartoon, "cartoon.csv", sep = ",", col.names = TRUE)
df <- read.table("cartoon.csv", header = TRUE, sep = ",")
dim(df) #`dim` means dimension. so, rows * columns
```

```
[1] 4 4
```

```
str(df) #structure of `df`
```

```
'data.frame':   4 obs. of  4 variables:
 $ names   : chr  "Mina" "Raju" "Mithu" "Lali"
 $ gender  : chr  "Female" "Male" "Female" "Female"
 $ age     : int  15 12 2 3
 $ is_human: logi  TRUE TRUE FALSE FALSE
```

We made the vectors first, and the used them to make the `cartton` data frame or table. We learned how to export the data frame using write.table function. Also, we learned to import or read back the table using `read.table` function. What are the `sep`, `col.names`, `header` arguments there? Why do we need them? Think. Try thinking of different properties of a data set.

1.7.1 Gene Expression Table

```r
gene_expr <- data.frame(
  genes = c("TP53", "BRCA1", "MYC", "EGFR", "GAPDH", "CDC2"),
  sample1 = c(8.2, 6.1, 9.5, 7.0, 10.0, 12),
  Sample2 = c(5.9, 3.9, 7.2, 4.8, 7.9, 9),
  Sample3 = c(8.25, 6.15, 9.6, 7.1, 10.1, 11.9),
  pathways = c("Apoptosis", "DNA Repair", "Cell Cycle", "Signaling", "Housekeeping", "Cell
)
write.table(gene_expr, "gene_expr.csv", sep = ",", col.names = TRUE)
gene_set <- read.table("gene_expr.csv", header = TRUE, sep = ",")
```

Note

Here, we directly used the vectors as different columns while making the data frame. Did you notice that? Also, the syntax is different here. We can't assign the vectors with the assignment operator (means we can't use `<-` sign. We have to use the `=` sign). Try using the `<-` sign. Did you notice the column names?

1.8 Homeworks

1. Compute the difference between this year (2025) and the year you started at the university and divide this by the difference between this year and the year you were born. Multiply this with 100 to get the percentage of your life you have spent at the university.
2. Make different kinds of variables and vectors with the data types we learned together.
3. What are the properties of a data frame?

Hint: Open an excel/csv/txt file you have and try to "generalize".

4. Can you make logical questions on the 2 small data sets we used? Try. It will help you understanding the logical operations we tried on variables. Now we are going to apply them on vectors (columns) on the data sets. For example, in the `cartoon` data set, we can ask/try to subset the data set filtering for females only, or for both females and age greater than 2 years.
5. If you are writing or practicing coding in R, write comment for each line on what it is doing. It will help to chunk it better into your brain.
6. Push the script and/or your answers to the questions (with your solutions) to one of your GitHub repo (and send me the repo link).

1.8.1 Deadline

Friday, 10pm BD Time.

2 L3: Data Transformation

Firstly, how did you solve the problems?

Give me your personal Mindmap. Please, send it in the chat!

2.1 Getting Started

2.1.1 Installation of R Markdown

We will use `rmarkdown` to have the flexibility of writing codes like the one you are reading now. If you haven't installed the `rmarkdown` package yet, you can do so with:

```
# Install rmarkdown package
#install.packages("rmarkdown")
library(rmarkdown)
# Other useful packages we might use
#install.packages("dplyr")    # Data manipulation
library(dplyr)
#install.packages("readr")    # Reading CSV files
library(readr)
```

Remove the hash sign before the `install.packages("rmarkdown")`, `install.packages("dplyr")`, `install.packages("readr")` if the library loading fails. That means the package is not there to be loaded. We need to download/install first.

> **Note**
>
> Do you remember this book by Hadley Wickham?. Try to follow it to get the hold on the basic R syntax and lexicon.

2.1.2 Basic Setup for Today's Session

```
# Clear environment
rm(list = ls())

# Check working directory
getwd()

# Set working directory if needed
# setwd("path/to/your/directory")  # Uncomment and modify as needed
```

2.1.3 Building on Last HW:

```
cartoon <- data.frame(
  names = c("Mina", "Raju", "Mithu", "Lali"),
  gender = c("Female", "Male", "Female", "Female"),
  age = c(15, 12, 2, 3),
  is_human = c(TRUE, TRUE, FALSE, FALSE)
)
cartoon
```

```
  names gender age is_human
1  Mina Female  15     TRUE
2  Raju   Male  12     TRUE
3 Mithu Female   2    FALSE
4  Lali Female   3    FALSE
```

```
dim(cartoon)
```

```
[1] 4 4
```

```
str(cartoon)
```

```
'data.frame':   4 obs. of  4 variables:
 $ names   : chr  "Mina" "Raju" "Mithu" "Lali"
 $ gender  : chr  "Female" "Male" "Female" "Female"
 $ age     : num  15 12 2 3
 $ is_human: logi  TRUE TRUE FALSE FALSE
```

```r
length(cartoon$names)
```

```
[1] 4
```

```r
##subseting
cartoon[1:2, 2:3] #row 1-2, column 2-3
```

```
  gender age
1 Female  15
2   Male  12
```

```r
cartoon[c(1, 3), c(1:3)] #row 1-3, column 1-3
```

```
  names gender age
1  Mina Female  15
3 Mithu Female   2
```

```r
#condition for selecting only male characters
male_df <- cartoon[cartoon$gender == "Male", ]
male_df
```

```
  names gender age is_human
2  Raju   Male  12     TRUE
```

```r
#condition for selecting female characters with age more than 2 years
female_age <- cartoon[cartoon$gender == "Female" & cartoon$age > 2, ]
female_age
```

```
  names gender age is_human
1  Mina Female  15     TRUE
4  Lali Female   3    FALSE
```

```r
sum(female_age$age) #sum of age of female_age dataset
```

```
[1] 18
```

```r
sd(cartoon$age) #standard deviation of age of main cartoon dataset
```

```
[1] 6.480741
```

```
mean(cartoon$age) #mean of age of main cartoon dataset
```

```
[1] 8
```

Check your colleague's repo for the Q3.

Logical Operators

Operator	Meaning	Example
`==`	Equal to	`x == 5`
`!=`	Not equal	`x != 5`
`<`	Less than	`x < 5`
`>`	Greater than	`x > 5`
`<=`	Less or equal	`x <= 5`
`>=`	Greater or equal	`x >= 5`
`!`	Not	`!(x < 5)`
`|`	OR	`x < 5 | x > 10`
`&`	AND	`x > 5 & x < 10`

2.1.4 Preamble on random variables (RV):

RV is so fundamental of an idea to interpret and do better in any kind of data analyses. But what is it? Let's imagine this scenario first. You got 30 mice to do an experiment to check anti-diabetic effect of a plant extract. You randomly assigned them into 3 groups. `control`, `treat1` (meaning insulin receivers), and `treat2` (meaning your plant extract receivers). Then you kept testing and measuring. You have mean glucose level of every mouse and show whether the mean value of `treat1` is equal to `treat2` or not. So, are you done? Not really. Be fastidious about the mice. What if you got some other 30 mice? Are they the same? Will their mean glucose level be the same? No, right. We would end up with different mean value. We call this type of quantities RV. Mean, Standard deviation, median, variance, etc. all are RVs. Do you see the logic? That's why we put this constraint and look for p-value, confidence interval (or CI), etc. by (null) hypothesis testing and sample distribution analyses. We will get into these stuffs later. But let's check what I meant. Also ponder about `sample` vs `population`.

Let's download the data first.

```
# Download small example dataset
download.file("https://raw.githubusercontent.com/genomicsclass/dagdata/master/inst/extdata
```

```
              destfile = "mice.csv")

# Load data
mice <- read.csv("mice.csv")
```

Let's check now.

```
control <- sample(mice$Bodyweight,12)
mean(control)
```

```
[1] 23.91167
```

```
control1 <- sample(mice$Bodyweight,12)
mean(control1)
```

```
[1] 23.48917
```

```
control2 <- sample(mice$Bodyweight,12)
mean(control2)
```

```
[1] 24.2175
```

Do you see the difference in the mean value now?

2.1.5 Basic Stuffs: Atomic Vector

```
atomic_vec <- c(Human=0.5, Mouse=0.33)
```

It is fast, but has limited access methods.

How to access elements here?

```
atomic_vec["Human"]
```

```
Human
  0.5
```

```
atomic_vec["Mouse"]
```

```
Mouse
 0.33
```

2.1.6 Basic Stuffs: Matrices

Matrices are essential for biologists working with expression data, distance matrices, and other numerical data.

```
# Create a gene expression matrix: rows=genes, columns=samples
expr_matrix <- matrix(
  c(12.3, 8.7, 15.2, 6.8,
    9.5, 11.2, 13.7, 7.4,
    5.6, 6.8, 7.9, 6.5),
  nrow = 3, ncol = 4, byrow = TRUE
)

# Add dimension names
rownames(expr_matrix) <- c("BRCA1", "TP53", "GAPDH")
colnames(expr_matrix) <- c("Control_1", "Control_2", "Treatment_1", "Treatment_2")
expr_matrix
```

```
      Control_1 Control_2 Treatment_1 Treatment_2
BRCA1      12.3       8.7        15.2         6.8
TP53        9.5      11.2        13.7         7.4
GAPDH       5.6       6.8         7.9         6.5
```

```
# Matrix dimensions
dim(expr_matrix)       # Returns rows and columns
```

```
[1] 3 4
```

```
nrow(expr_matrix)      # Number of rows
```

```
[1] 3
```

```
ncol(expr_matrix)      # Number of columns
```

```
[1] 4
```

```r
# Matrix subsetting
expr_matrix[2, ]       # One gene, all samples
```

```
  Control_1   Control_2 Treatment_1 Treatment_2
        9.5        11.2        13.7         7.4
```

```r
expr_matrix[, 3:4]     # All genes, treatment samples only
```

```
      Treatment_1 Treatment_2
BRCA1        15.2         6.8
TP53         13.7         7.4
GAPDH         7.9         6.5
```

```r
expr_matrix["TP53", c("Control_1", "Treatment_1")]  # Specific gene and samples
```

```
  Control_1 Treatment_1
        9.5        13.7
```

```r
# Matrix calculations (useful for bioinformatics)
# Mean expression per gene
gene_means <- rowMeans(expr_matrix)
gene_means
```

```
BRCA1  TP53 GAPDH
10.75 10.45  6.70
```

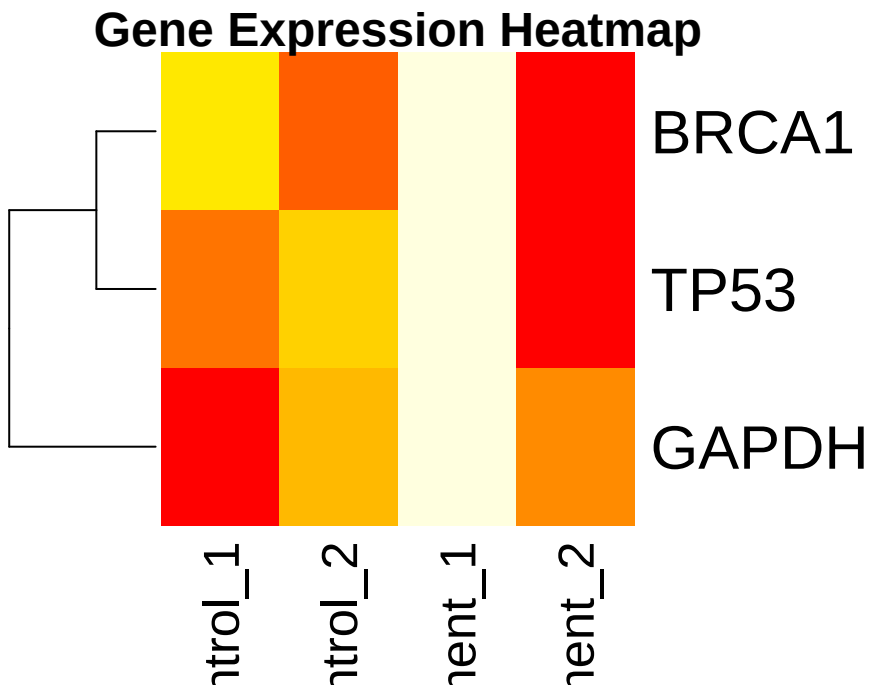
```r
# Mean expression per sample
sample_means <- colMeans(expr_matrix)
sample_means
```

```
  Control_1   Control_2 Treatment_1 Treatment_2
   9.133333    8.900000   12.266667    6.900000
```

```r
# Calculate fold change (Treatment vs Control)
control_means <- rowMeans(expr_matrix[, 1:2])
treatment_means <- rowMeans(expr_matrix[, 3:4])
fold_change <- treatment_means / control_means
fold_change
```

```
   BRCA1     TP53    GAPDH
1.047619 1.019324 1.161290
```

```r
# Matrix visualization
# Heatmap of expression data
heatmap(expr_matrix,
        Colv = NA,          # Don't cluster columns
        scale = "row",      # Scale by row (gene)
        col = heat.colors(16),
        main = "Gene Expression Heatmap")
```



2.1.6.1 More Matrix Practice:

```r
#Create a simple Gene Expression matrix (RNA-seq style)

Gene_Expression <- matrix(c(
  5.2, 3.1, 8.5,   # Sample 1
  6.0, 2.8, 7.9    # Sample 2
), nrow = 2, byrow = TRUE)
```

```r
rownames(Gene_Expression) <- c("Sample_1", "Sample_2")
colnames(Gene_Expression) <- c("GeneA", "GeneB", "GeneC")

print("Gene Expression Matrix:")
```

```
[1] "Gene Expression Matrix:"
```

```r
print(Gene_Expression)
```

```
         GeneA GeneB GeneC
Sample_1   5.2   3.1   8.5
Sample_2   6.0   2.8   7.9
```

```r
#1. Transpose: Genes become rows, Samples become columns

Gene_Expression_T <- t(Gene_Expression)
print("Transpose of Gene Expression Matrix:")
```

```
[1] "Transpose of Gene Expression Matrix:"
```

```r
print(Gene_Expression_T)
```

```
      Sample_1 Sample_2
GeneA      5.2      6.0
GeneB      3.1      2.8
GeneC      8.5      7.9
```

```r
#2. Matrix multiplication
# Suppose each gene has an associated "gene weight" (e.g., biological importance)

Gene_Weights <- matrix(c(0.8, 1.2, 1.0), nrow = 3, byrow = TRUE)
rownames(Gene_Weights) <- c("GeneA", "GeneB", "GeneC")
colnames(Gene_Weights) <- c("Weight")

Total_Weighted_Expression <- Gene_Expression %*% Gene_Weights
print("Total Weighted Expression per Sample:")
```

```
[1] "Total Weighted Expression per Sample:"
```

```
print(Total_Weighted_Expression)
```

```
         Weight
Sample_1  16.38
Sample_2  16.06
```

```
# 3. Matrix addition
# Hypothetically increase expression by 1 TPM everywhere (technical adjustment)

Adjusted_Expression <- Gene_Expression + 1
print("Expression Matrix after adding 1 TPM:")
```

```
[1] "Expression Matrix after adding 1 TPM:"
```

```
print(Adjusted_Expression)
```

```
         GeneA GeneB GeneC
Sample_1   6.2   4.1   9.5
Sample_2   7.0   3.8   8.9
```

```
# 4. Identity matrix
I <- diag(3)
rownames(I) <- c("GeneA", "GeneB", "GeneC")
colnames(I) <- c("GeneA", "GeneB", "GeneC")

print("Identity Matrix (for genes):")
```

```
[1] "Identity Matrix (for genes):"
```

```
print(I)
```

```
      GeneA GeneB GeneC
GeneA     1     0     0
GeneB     0     1     0
GeneC     0     0     1
```

```r
# Multiplying Gene Expression by Identity
Identity_Check <- Gene_Expression %*% I
print("Gene Expression multiplied by Identity Matrix:")
```

```
[1] "Gene Expression multiplied by Identity Matrix:"
```

```r
print(Identity_Check)
```

```
         GeneA GeneB GeneC
Sample_1   5.2   3.1   8.5
Sample_2   6.0   2.8   7.9
```

```r
# 5. Scalar multiplication
# Suppose you want to simulate doubling expression values

Doubled_Expression <- 2 * Gene_Expression
print("Doubled Gene Expression:")
```

```
[1] "Doubled Gene Expression:"
```

```r
print(Doubled_Expression)
```

```
         GeneA GeneB GeneC
Sample_1  10.4   6.2  17.0
Sample_2  12.0   5.6  15.8
```

```r
# 6. Summations

# Total expression per sample
Total_Expression_Per_Sample <- rowSums(Gene_Expression)
print("Total Expression per Sample:")
```

```
[1] "Total Expression per Sample:"
```

```r
print(Total_Expression_Per_Sample)
```

```
Sample_1 Sample_2
    16.8     16.7
```

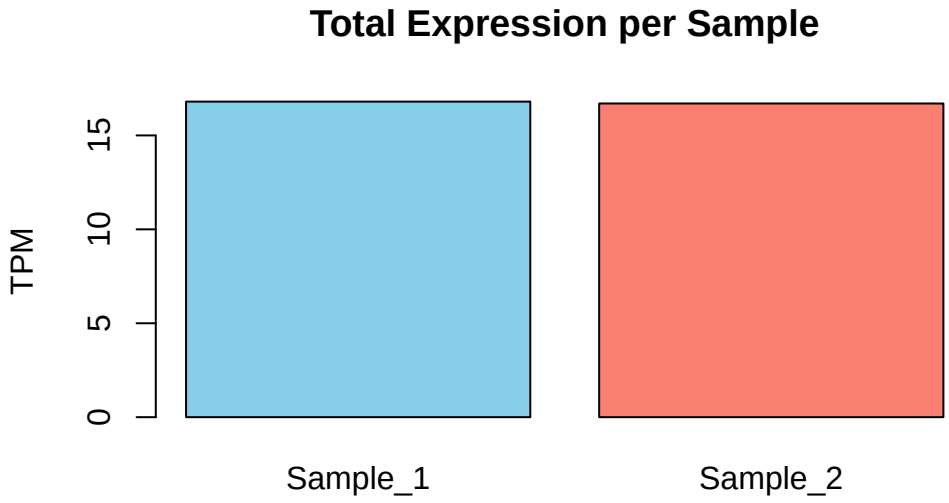
```
# Total expression per gene
Total_Expression_Per_Gene <- colSums(Gene_Expression)
print("Total Expression per Gene:")
```

```
[1] "Total Expression per Gene:"
```

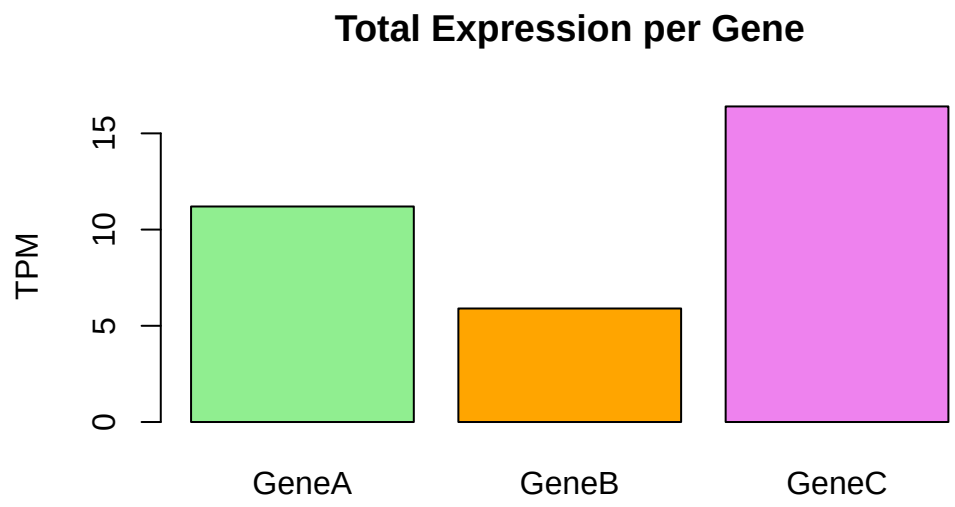
```
print(Total_Expression_Per_Gene)
```

```
GeneA GeneB GeneC
 11.2   5.9  16.4
```

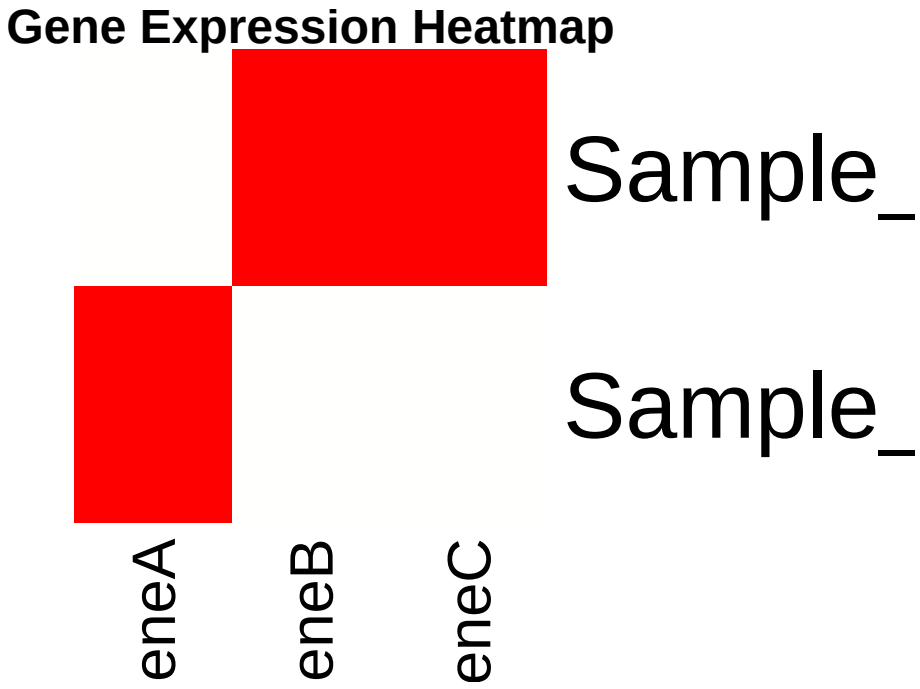
```
# 7. Simple plots

# Barplot: Total expression per sample
barplot(Total_Expression_Per_Sample, main="Total Expression per Sample", ylab="TPM", col=c
```




```
# Barplot: Total expression per gene
barplot(Total_Expression_Per_Gene, main="Total Expression per Gene", ylab="TPM", col=c("li
```




```
# Heatmap: Expression matrix
heatmap(Gene_Expression, Rowv=NA, Colv=NA, col=heat.colors(256), scale="column", main="Gen
```



Another Example: You have counts of cells in different organs for two animal species.

You also have a matrix with average cell sizes (micrometer, μm^2) for each organ.

You can then multiply count $\times$ size to get total cell area for each species in each organ.

```r
# Create a matrix: Cell counts
Cell_Counts <- matrix(c(500, 600, 300, 400, 700, 800), nrow = 2, byrow = TRUE)
rownames(Cell_Counts) <- c("Mouse", "Rat")
colnames(Cell_Counts) <- c("Heart", "Liver", "Brain")

print("Cell Counts Matrix:")
```

```
[1] "Cell Counts Matrix:"
```

```r
print(Cell_Counts)
```

```
      Heart Liver Brain
Mouse   500   600   300
Rat     400   700   800
```

```r
# Create a matrix: Average cell size in µm²
Cell_Size <- matrix(c(50, 200, 150), nrow = 3, byrow = TRUE)
rownames(Cell_Size) <- c("Heart", "Liver", "Brain")
colnames(Cell_Size) <- c("Avg_Cell_Size")

print("Cell Size Matrix (µm²):")
```

```
[1] "Cell Size Matrix (µm²):"
```

```r
print(Cell_Size)
```

```
      Avg_Cell_Size
Heart            50
Liver           200
Brain           150
```

```r
# 1. Transpose of Cell Counts
Cell_Counts_T <- t(Cell_Counts)
print("Transpose of Cell Counts:")
```

```
[1] "Transpose of Cell Counts:"
```

```r
print(Cell_Counts_T)
```

```
      Mouse Rat
Heart   500 400
Liver   600 700
Brain   300 800
```

```r
# 2. Matrix multiplication: Total cell area
# (2x3) %*% (3x1) => (2x1)
Total_Cell_Area <- Cell_Counts %*% Cell_Size
colnames(Total_Cell_Area) <- "Cell_area"
print("Total Cell Area (Counts × Size) (µm²):")
```

```
[1] "Total Cell Area (Counts × Size) (µm²):"
```

```r
print(Total_Cell_Area)
```

```
      Cell_area
Mouse    190000
Rat      280000
```

```r
# 3. Matrix addition: Add 10 cells artificially to all counts (for example)
Added_Cells <- Cell_Counts + 10
print("Cell Counts after adding 10 artificial cells:")
```

```
[1] "Cell Counts after adding 10 artificial cells:"
```

```r
print(Added_Cells)
```

```
      Heart Liver Brain
Mouse   510   610   310
Rat     410   710   810
```

```r
# 4. Identity matrix
I <- diag(3)
rownames(I) <- c("Heart", "Liver", "Brain")
colnames(I) <- c("Heart", "Liver", "Brain")
```

```r
print("Identity Matrix:")
```

```
[1] "Identity Matrix:"
```

```r
print(I)
```

```
      Heart Liver Brain
Heart     1     0     0
Liver     0     1     0
Brain     0     0     1
```

```r
# 5. Multiplying Cell Counts by Identity Matrix (no real change but shows dimension rules)
Check_Identity <- Cell_Counts %*% I
print("Cell Counts multiplied by Identity Matrix:")
```

```
[1] "Cell Counts multiplied by Identity Matrix:"
```

```r
print(Check_Identity)
```

```
      Heart Liver Brain
Mouse   500   600   300
Rat     400   700   800
```

```r
# 6. Scalar multiplication: double the counts (hypothetical growth)
Double_Cell_Counts <- 2 * Cell_Counts
print("Doubled Cell Counts:")
```

```
[1] "Doubled Cell Counts:"
```

```r
print(Double_Cell_Counts)
```

```
      Heart Liver Brain
Mouse  1000  1200   600
Rat     800  1400  1600
```

```r
# Total number of cells per animal (row sums)
Total_Cells_Per_Species <- rowSums(Cell_Counts)
print("Total number of cells per species:")
```

```
[1] "Total number of cells per species:"
```

```r
print(Total_Cells_Per_Species)
```

```
Mouse   Rat
 1400  1900
```

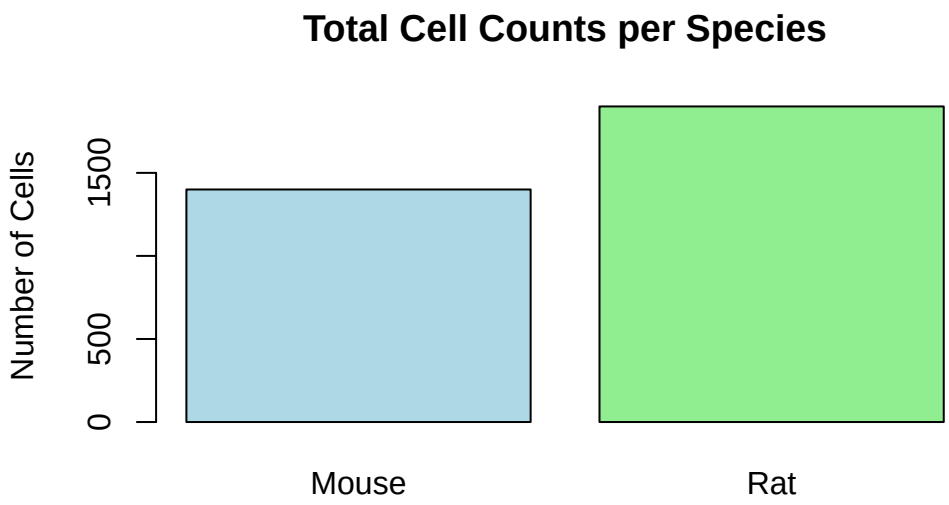
```r
# Total number of cells per organ (column sums)
Total_Cells_Per_Organ <- colSums(Cell_Counts)
print("Total number of cells per organ:")
```

```
[1] "Total number of cells per organ:"
```

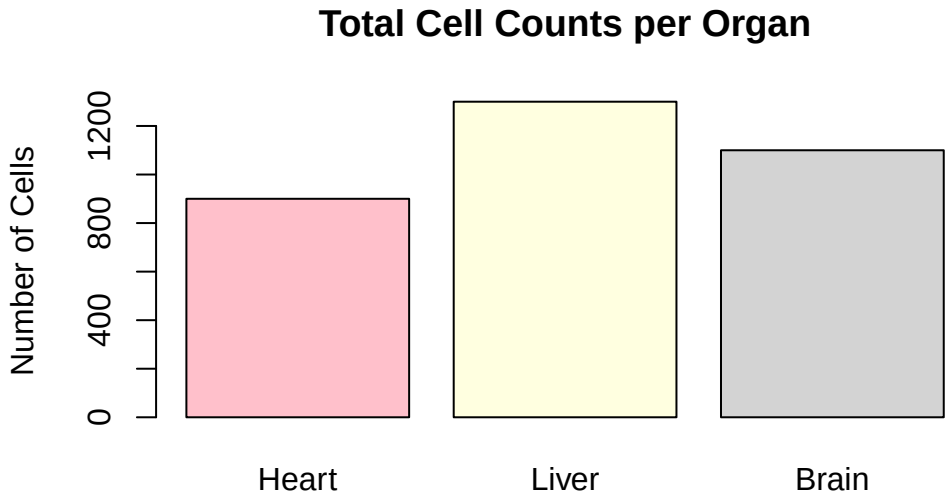
```r
print(Total_Cells_Per_Organ)
```

```
Heart Liver Brain
  900  1300  1100
```

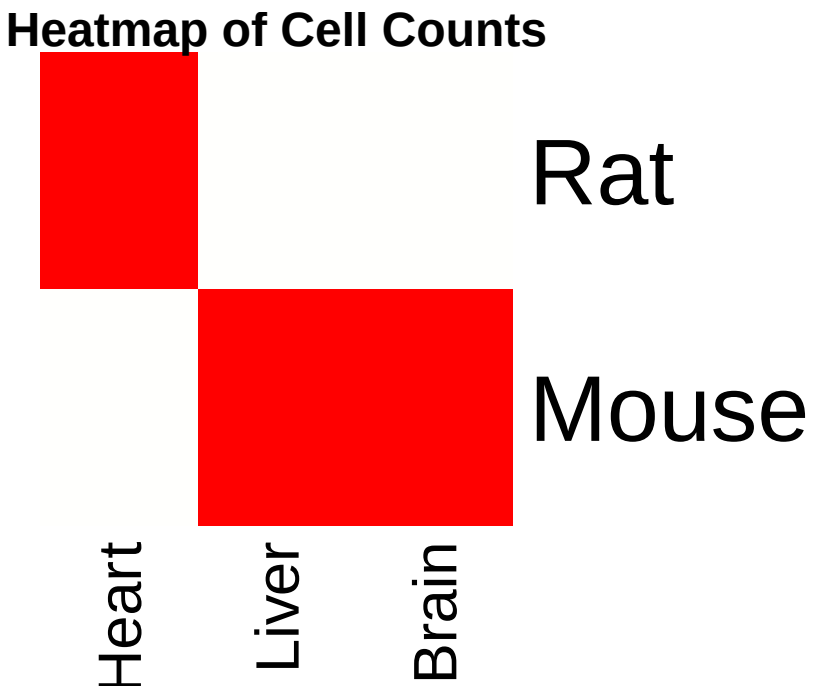
```r
# --- Simple plots ---

# Bar plot of total cells per species
barplot(Total_Cells_Per_Species, main="Total Cell Counts per Species", ylab="Number of Cel
```




```
# Bar plot of total cells per organ
barplot(Total_Cells_Per_Organ, main="Total Cell Counts per Organ", ylab="Number of Cells",
```




```
# Heatmap of the original Cell Counts matrix
heatmap(Cell_Counts, Rowv=NA, Colv=NA, col=heat.colors(256), scale="column", main="Heatmap
```



Operation	Explanation	R Function/Example
Matrix Creation	Create gene expression matrix	`matrix()`
Transpose	Flip genes and samples	`t(Gene_Expression)`
Matrix Multiplication	Calculate weighted sums	`Gene_Expression %*% Gene_Weights`
Matrix Addition	Adjust counts	`Gene_Expression + 1`
Identity Matrix	Special neutral matrix	`diag(3)`
Scalar Multiplication	Simulate overall increase	`2 * Gene_Expression`
Row/Column Summation	Total per sample/gene	`rowSums()`, `colSums()`
Plotting	Visualize expression patterns	`barplot()`, `heatmap()`

2.1.7 Basic Stuffs: List

Lists are the most flexible data structure in R - they can hold any combination of data types, including other lists! This makes them essential for biological data analysis where we often deal with mixed data types.

```r
# A list storing different types of genomic data
genomics_data <- list(
  gene_names = c("TP53", "BRCA1", "MYC"),                # Character vector
  expression = matrix(c(1.2, 3.4, 5.6, 7.8, 9.1, 2.3), nrow=3),    # Numeric matrix
  is_cancer_gene = c(TRUE, TRUE, FALSE),                  # Logical vector
  metadata = list(                                        # Nested list!
    lab = "CRG",
    date = "2023-05-01"
  )
)
```

How to Access Elements of a List?

```r
# Method 1: Double brackets [[ ]] for single element
genomics_data[[1]]  # Returns gene_names vector
```

```
[1] "TP53"  "BRCA1" "MYC"
```

```r
# Method 2: $ operator with names (when elements are named)
genomics_data$expression  # Returns the matrix
```

```
     [,1] [,2]
[1,]  1.2  7.8
[2,]  3.4  9.1
[3,]  5.6  2.3
```

```r
# Method 3: Single bracket [ ] returns a sublist
genomics_data[1:2]  # Returns list with first two elements
```

```
$gene_names
[1] "TP53"  "BRCA1" "MYC"

$expression
     [,1] [,2]
[1,]  1.2  7.8
[2,]  3.4  9.1
[3,]  5.6  2.3
```

Key Difference from Vectors:

```
# Compare to your prop.table() example:
atomic_vec["Human"]    # Returns named numeric (vector)
```

```
Human
  0.5
```

```
atomic_vec["Mouse"]
```

```
Mouse
 0.33
```

```
genomics_data[1] # Returns list containing the vector
```

```
$gene_names
[1] "TP53"  "BRCA1" "MYC"
```

Why Biologists Need Lists?
`lm()`, `prcomp()` functions, RNAseq analysis packages produces list. So, we need to learn how to handle lists.

See these examples:

A. Storing BLAST results

```
blast_hits <- list(
  query_id = "GeneX",
  hit_ids = c("NP_123", "NP_456"),
  e_values = c(1e-50, 3e-12),
  alignment = matrix(c("ATG...", "CTA..."), ncol=1))
```

B. Handling Mixed Data

```
patient_data <- list(
  id = "P1001",
  tests = data.frame(
    test = c("WBC", "RBC"),
    value = c(4.5, 5.1)
  ),
  has_mutation = TRUE
)
```

Common List Operations

```
# Add new element
genomics_data$sequencer <- "Illumina"

# Remove element
genomics_data$is_cancer_gene <- NULL

# Check structure (critical for complex lists)
str(genomics_data)
```

```
List of 4
 $ gene_names: chr [1:3] "TP53" "BRCA1" "MYC"
 $ expression: num [1:3, 1:2] 1.2 3.4 5.6 7.8 9.1 2.3
 $ metadata  :List of 2
  ..$ lab : chr "CRG"
  ..$ date: chr "2023-05-01"
 $ sequencer : chr "Illumina"
```

By the way, how would you add more patients?

```
# Add new patient
patient_data$P1002 <- list(
  id = "P1002",
  tests = data.frame(
    test = c("WBC", "RBC", "Platelets"),
    value = c(6.2, 4.8, 150)
  ),
  has_mutation = FALSE
)
# Access specific patient
patient_data$P1001$test
```

```
NULL
```

For Batch Processing:

```
patients <- list(
  list(
    id = "P1001",
    tests = data.frame(test = c("WBC", "RBC"), value = c(4.5, 5.1)),
```

```
    has_mutation = TRUE
  ),
  list(
    id = "P1002",
    tests = data.frame(test = c("WBC", "RBC", "Platelets"), value = c(6.2, 4.8, 150)),
    has_mutation = FALSE
  )
)

# Access 2nd patient's WBC value
patients[[2]]$tests$value[patients[[2]]$tests$test == "WBC"]
```

```
[1] 6.2
```

Converting Between Structures

```
# List → Vector
unlist(genomics_data[1:3])
```

```
  gene_names1   gene_names2   gene_names3   expression1   expression2
       "TP53"        "BRCA1"         "MYC"         "1.2"         "3.4"
  expression3   expression4   expression5   expression6  metadata.lab
       "5.6"         "7.8"         "9.1"         "2.3"         "CRG"
metadata.date
 "2023-05-01"
```

Visualization

```
# Base R plot from list data
barplot(unlist(genomics_data[2]),
        names.arg = genomics_data[[1]])
```

This code won't work if you run. `unlist(genomics_data[2]` creates a vector of length 6 from our 3*2 matrix but `genomics_data[[1]]` has 3 things inside the `gene_names` vector. Debug like this:

```
dim(genomics_data$expression)  # e.g., 2 rows x 2 cols
```

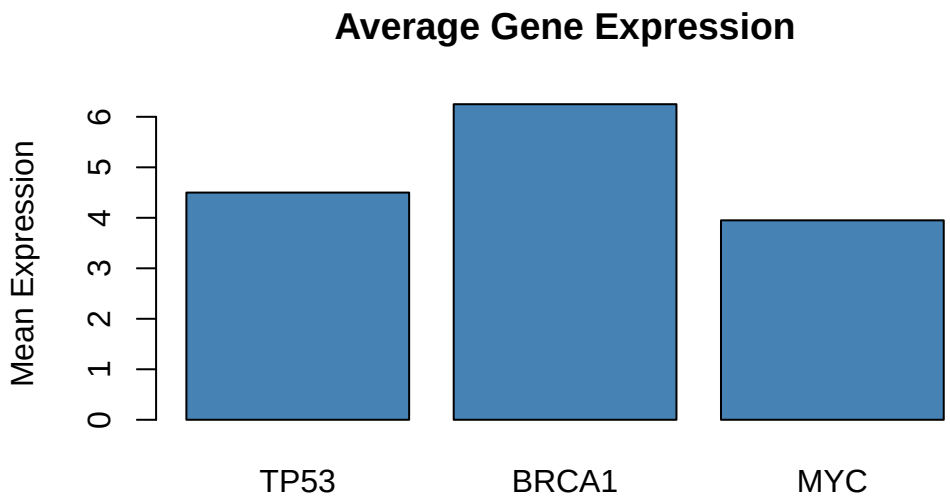
```
[1] 3 2
```

```r
length(genomics_data$gene_names) # e.g., 3 genes
```

```
[1] 3
```

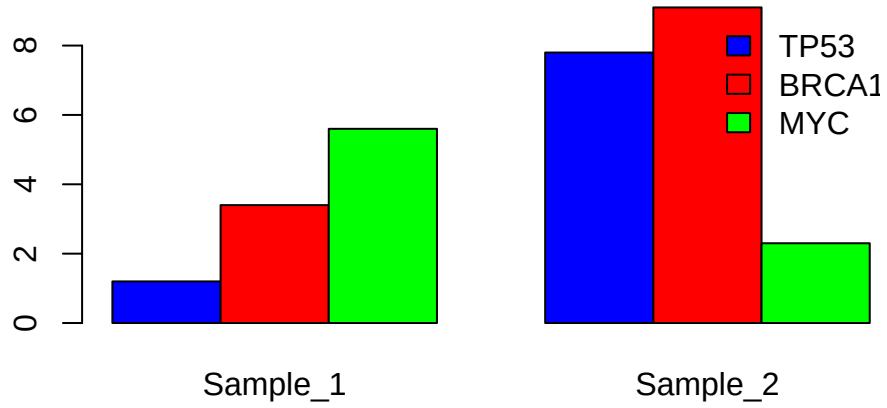
A. Gene-Centric (Mean Expression)

```r
barplot(rowMeans(genomics_data$expression),
        names.arg = genomics_data$gene_names,
        col = "steelblue",
        ylab = "Mean Expression",
        main = "Average Gene Expression")
```



B. Sample-Centric (All Measurements)

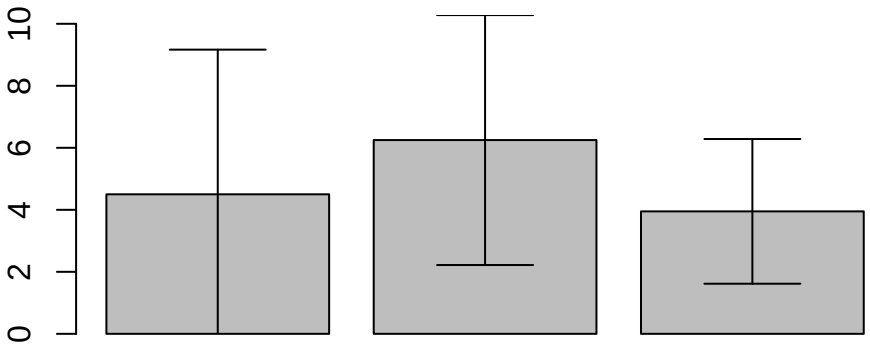
```r
barplot(genomics_data$expression,
        beside = TRUE,
        names.arg = paste0("Sample_", 1:ncol(genomics_data$expression)),
        legend.text = genomics_data$gene_names,
        args.legend = list(x = "topright", bty = "n"),
        col = c("blue", "red", "green"),
        main = "Expression Across Samples")
```



Note

This matches real-world scenarios:
RNA-seq: Rows=genes, cols=samples
rowMeans() = average expression per gene
beside=TRUE => compare samples within genes
Proteomics: Rows=proteins, cols=replicates
Same principles apply

```
# Calculate stats
gene_means <- rowMeans(genomics_data$expression)
gene_sds <- apply(genomics_data$expression, 1, sd)

# Plot with error bars
bp <- barplot(gene_means, ylim = c(0, max(gene_means + gene_sds)))
arrows(bp, gene_means - gene_sds, bp, gene_means + gene_sds,
       angle = 90, code = 3)
```



Task: Create a list containing:

i) A character vector of 3 gene names

ii) A numeric matrix of expression values

iii) A logical vector indicating pathway membership

iv) A nested list with lab metadata

2.2 Homeworks: Matrix and List Operations

2.2.1 Protein Quantification in Biological Samples

You are given the following protein concentration matrix:

$$\text{ProteinMatrix} = \begin{bmatrix} 5 & 3 & 2 \\ 7 & 6 & 4 \end{bmatrix}$$

- Rows represent samples:
  - Sample1
  - Sample2
- Columns represent proteins:
  - ProteinX
  - ProteinY
  - ProteinZ

You are also given a weight (importance) matrix for the proteins:

$$\text{WeightVector} = \begin{bmatrix} 0.5 \\ 1.0 \\ 1.5 \end{bmatrix}$$

2.2.2 Tasks

1. Make the matrices (with exact names) and multiply the ProteinMatrix by the WeightVector.

That is:

$$\text{ProteinMatrix} \times \text{WeightVector}$$

2. Transpose the ProteinMatrix and show what it looks like.
3. Create the Identity matrix of compatible size and show what happens when you multiply:

$$\text{ProteinMatrix} \times I$$

4.Do the calculations (rowSums, colSums, etc.) and visualization (barplot, heatmap) as shown in the class.

2.2.3 Interpretation Questions

- What does multiplying the protein levels by the weight vector mean biologically?
- What does the result tell you about total protein burden (or total protein impact) for each sample?
- What do the identity matrix represent in the context of protein interactions or measurement biases?
- If you changed the weight of ProteinZ to 3.0, how would the result change?

2.2.4 Gene-to-Protein Translation

You are given the following matrix representing normalized gene expression levels (e.g., TPM):

$$\text{GeneExpression} = \begin{bmatrix} 10 & 8 & 5 \\ 15 & 12 & 10 \end{bmatrix}$$

- Rows = Samples:
  - Sample1
  - Sample2
- Columns = Genes:
  - GeneA
  - GeneB
  - GeneC

Each gene translates into proteins with a certain efficiency. The efficiency of translation from each gene to its corresponding protein is given by the following diagonal matrix:

$$\text{TranslationMatrix} = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 1.2 & 0 \\ 0 & 0 & 1.8 \end{bmatrix}$$

This means:

- GeneA → ProteinA with 1.5× efficiency
- GeneB → ProteinB with 1.2× efficiency
- GeneC → ProteinC with 1.8× efficiency

2.2.5 Tasks

1. Make the matrices and multiply GeneExpression × TranslationMatrix to compute the resulting ProteinMatrix.

- Show the result step-by-step.

2. Transpose the GeneExpression matrix. What does this new matrix represent?
3. Create the Identity matrix I and multiply it with the TranslationMatrix. What happens?
4. Create a new matrix containing only the expression of GeneA and GeneB across both samples. Call this submatrix A.

- Compute the inverse A^{1} using `solve()` function.
- Then verify:
  $A \times A^{1} = I$

2.2.6 Visualization Tasks

5. Plot a MARplot-style scatter plot:

- x-axis: Gene expression values (GeneExpression matrix, flattened)
- y-axis: Corresponding Protein values (ProteinMatrix, flattened)
- Label each point as "Sample-Gene"

6. Generate a heatmap of the ProteinMatrix using R's `heatmap()` function.

- Add meaningful row and column labels.
- Enable clustering by rows and columns.

2.2.7 Interpretation Questions

- What does matrix multiplication represent biologically in this case?
- Why does the diagonal TranslationMatrix make sense biologically?
- What does it mean if Sample2 has higher protein levels even with similar gene expression?
- How does the MARplot help interpret translation efficiency?
- How does clustering in the heatmap reveal relationships between samples and proteins?

2.2.8 Animal Breeding – Economic Ranking of Bulls by Traits

You are evaluating two bulls for use in a dairy breeding program. Their Estimated Breeding Values (EBVs) are:

$$\text{BullEBVs} = \begin{bmatrix} 400 & 1.2 & 0.8 \\ 500 & 1.5 & 0.6 \end{bmatrix}$$

- Rows:
  - Bull1
  - Bull2
- Columns:
  - Trait1 = Milk yield (liters/year)
  - Trait2 = Growth rate (kg/day)
  - Trait3 = Fertility (calving interval adjustment)

You assign economic weights to each trait:

$$\text{EconomicWeights} = \begin{bmatrix} 0.002 \\ 50 \\ 100 \end{bmatrix}$$

2.2.9 Tasks

1. Compute:

$$\text{TotalValue} = \text{BullEBVs} \times \text{EconomicWeights}$$

- What are the resulting values?
- Which bull is more valuable economically?

2. Interpret what multiplying by the economic weights means biologically.

3. Create the 3×3 identity matrix I and multiply it with BullEBVs.

- What does it return?
- What does the identity matrix mean in this case?

4. Subset the BullEBVs matrix to remove Trait1 (milk yield) and recalculate TotalValue.

- How does this change the ranking?

2.2.10 Visualization Tasks

5. Create a bar plot comparing TotalValue for Bull1 and Bull2.

6. Create a heatmap of the EBVs.

- Label rows and columns.
- Enable clustering.

2.2.11 Interpretation Questions

- How do economic weights affect trait importance?
- Why might you ignore milk yield in some breeding programs?
- What is the value of heatmaps in visualizing multivariate trait data?
- Can this method be extended to more bulls and more traits?

2.2.12 Plant Breeding – Trait Contributions from Parental Lines

You are breeding a new rice variety from three parental lines. The key traits are:

- T1 = Drought resistance
- T2 = Yield
- T3 = Maturation time

The following trait values (normalized 1–10) have been measured:

$$\text{ParentTraits} = \begin{bmatrix} 7 & 5 & 3 \\ 6 & 8 & 4 \\ 5 & 6 & 6 \end{bmatrix}$$

- Rows:
  - P1 (Parent 1)
  - P2 (Parent 2)
  - P3 (Parent 3)
- Columns:
  - T1 = Drought resistance
  - T2 = Yield
  - T3 = Maturation time

You design a hybrid with contributions from each parent as follows:

$$\text{HybridWeights} = \begin{bmatrix} 0.5 \\ 0.3 \\ 0.2 \end{bmatrix}$$

2.2.13 Tasks

1. Compute the HybridTrait vector by:

$$\text{HybridTraits} = \text{HybridWeights}^T \times \text{ParentTraits}$$

Show the steps and result.

2. Explain what it means biologically when one parent contributes more to a particular trait.

3. Create an identity matrix I and multiply it with ParentTraits.

- What do you observe?
- What does I × ParentTraits represent?

4. Subset the ParentTraits matrix to include only T1 and T2. Recalculate the hybrid traits.

- Discuss how removing a trait affects your outcome.

2.2.14 Visualization Tasks

5. Generate a heatmap of the ParentTraits matrix.

- Label the rows with parent names and columns with trait names.
- Enable row/column clustering.

6. Create a bar plot showing the HybridTraits (T1, T2, T3).

- Color code each bar by trait.
- What trait contributes most?

2.2.15 Interpretation Questions

- How does the weighting of parents affect the hybrid's performance?
- What does the identity matrix represent here?
- If you used equal weights (for each), how would the hybrid traits change?
- What real-world limitations does this simplified model ignore?

2.2.16 Managing Matrices and Weight Vectors Using Lists in R

Now that you have completed four biological matrix problems — Protein concentration, gene-to-protein mapping, bull breeding value ranking, and plant trait combinations — it's time to organize your data and weights using R's list structure.

In this task, you will:

- Group each example's matrix and its corresponding weight vector inside a named list.
- Combine these named lists into a larger list called bioList.
- Use list indexing to repeat your earlier calculations and visualizations.
- Reflect on the benefits and challenges of using structured data objects.

2.2.17 Step 1: Create a master list

You should now build a named list called bioList containing the following four elements:

- ProteinConc = list(matrix = ProteinMatrix, weights = WeightVector)
- ProteinMap = list(matrix = ProteinMapping, weights = TranslationWeights)
- Plant = list(matrix = ParentTraits, weights = HybridWeights)
- Animal = list(matrix = BullEBVs, weights = EconomicWeights)

Hint: Each inner list should contain both:

- matrix = the main data matrix
- weights = the vector used for multiplication

No R code is required here (You have them from previous part, use inside same rmd/notebook file) — just structure your data like this in your workspace.

2.2.18 Tasks

1. List the full names of each component in bioList. What are the names of the top-level and nested components?
2. Access each matrix and its corresponding weights using list indexing.

- How would you extract only the matrix of the Plant entry?
- How would you extract the weights for the Protein concentration entry?

3. Use the correct matrix and weights to perform:

- ProteinConc: Weighted gene expression score
- ProteinMap: Contribution of transcripts to each protein
- Plant: Hybrid trait values

- Animal: Bull total economic value

4. Subset one matrix in each sublist (e.g., drop a trait or feature) and repeat the weighted calculation.

- What changes in the results?
- Which traits/genes have the strongest influence?

2.2.19 Visualization Tasks

5. Generate one heatmap for any matrix stored in bioList.

- Choose one (e.g., ProteinMap or Plant)
- Apply clustering to rows and/or columns
- Label appropriately

6. Generate two bar plots:

- One showing the result of weighted trait aggregation for the Plant hybrid
- One showing the total breeding values for each bull

2.2.20 Interpretation Questions

- How does structuring your data using a list help with clarity and reproducibility?
- What risks or challenges might occur when accessing elements from nested lists?
- Could this structure be scaled for real datasets with many samples or traits?
- How would you loop over all elements in bioList to apply the same function?
- How can this list structure be useful for building automated bioinformatics pipelines?

Your Rmarkdown file(s) should include:

- All matrix calculations (tasks). Also, name the rows and columns of each matrix accordingly.
- All interpretation answers
- All plots (output from embedded code)
- And your commentary blocks for each code chunk

knit your rmd (or Notebook) file as html/pdf file and push both the rmd (or Notebook) and html/pdf files

2.3 Factor Variables

Important for categorical data

2.3.1 Creating Factors

Factors are used to represent categorical data in R. They are particularly important for biological data like genotypes, phenotypes, and experimental conditions.

```
# Simple factor: DNA sample origins
origins <- c("Human", "Mouse", "Human", "Zebrafish", "Mouse", "Human")
origins_factor <- factor(origins, levels = c("Human", "Zebrafish", "Mouse"))
origins_factor
```

```
[1] Human     Mouse     Human     Zebrafish Mouse     Human
Levels: Human Zebrafish Mouse
```

```
# Check levels (categories)
levels(origins_factor)
```

```
[1] "Human"     "Zebrafish" "Mouse"
```

```
# Create a factor with predefined levels
treatment_groups <- factor(c("Control", "Low_dose", "High_dose", "Control", "Low_dose"),
                           levels = c("Control", "Low_dose", "High_dose"))
treatment_groups
```

```
[1] Control   Low_dose  High_dose Control   Low_dose
Levels: Control Low_dose High_dose
```

```
# Ordered factors (important for severity, stages, etc.)
disease_severity <- factor(c("Mild", "Severe", "Moderate", "Mild", "Critical"),
                           levels = c("Mild", "Moderate", "Severe", "Critical"),
                           ordered = TRUE)
disease_severity
```

```
[1] Mild     Severe   Moderate Mild     Critical
Levels: Mild < Moderate < Severe < Critical
```

```
# Compare with ordered factors
disease_severity[1] > disease_severity[2]  # Is Mild less severe than Severe?
```

```
[1] FALSE
```

2.3.2 Factor Operations

```
# Count frequencies
out <- table(origins_factor)
out
```

```
origins_factor
    Human Zebrafish     Mouse
        3         1         2
```

```
out["Human"]
```

```
Human
    3
```

```
# Calculate proportions
prop.table(out)
```

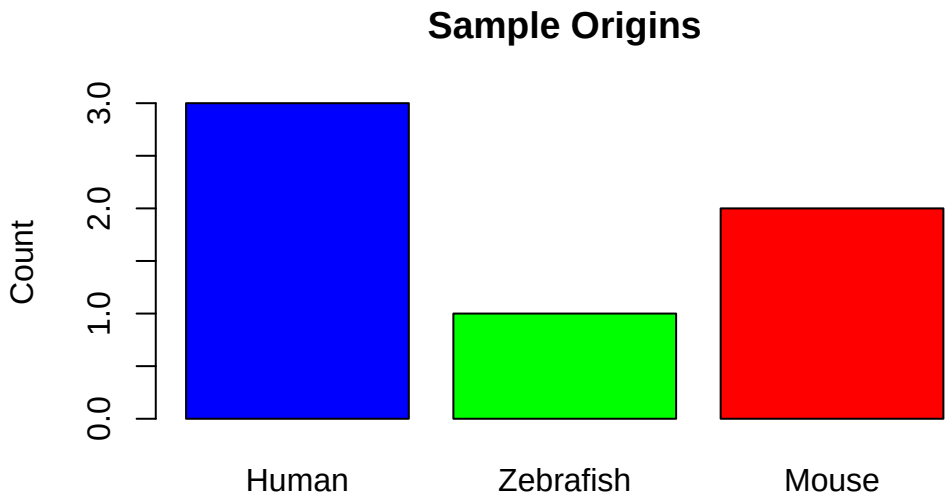
```
origins_factor
    Human Zebrafish     Mouse
0.5000000 0.1666667 0.3333333
```

```
# Change reference level (important for statistical models)
origins_factor_relevel <- relevel(origins_factor, ref = "Mouse")
origins_factor_relevel
```

```
[1] Human     Mouse     Human     Zebrafish Mouse     Human
Levels: Mouse Human Zebrafish
```

```
# Convert to character
origins_char <- as.character(origins_factor)

# Plot factors - Basic barplot
barplot(table(origins_factor),
        col = c("blue", "green", "red"),
```

```
        main = "Sample Origins",
        ylab = "Count")
```



> **Note**
>
> Factor `level`-ing and `relevel`-ing are different. `relevel` redefines what the reference should be. For example, in an experiment, you have `control`, `treatment1`, `treatment2` groups. Your reference might be `control`. So, all of your comparisons/statistics are on the basis of `control`. But you might change the reference (by `relevel` to `treatment1` and all of your comparison will be on the basis of `treatment1` group. Got it?

More advanced plot with factors:

```
gene_expr <- c(5.2, 7.8, 4.5, 12.3, 8.1, 3.7)
names(gene_expr) <- as.character(origins)

# Boxplot by factor
boxplot(gene_expr ~ origins,
        col = "lightblue",
        main = "Gene Expression by Sample Origin",
        xlab = "Origin",
        ylab = "Expression Level")
```

> **Note**
>
> Did you notice how factor `level`-ing changes the appearance of the categories in the plots? See the `barplot` and the `boxplot` again. Where are Zebrafish and Mouse now in the plots? Why are their positions on the x-axis changed?

> **Note**
>
> Keep noticing the output formats. Sometimes the output is just a number, sometimes a vector or table or list, etc. Check `prop.table(table(origins_factor))`. How is it?

> **Got it?**
>
> `prop <- prop.table(table(origins_factor))` – is a named numeric vector (atomic vector). `prop$Human` or similar won't work. Check this way: `prop prop["Human"]; prop["Mouse"]; prop["Zebrafish"]`
> Or make it a data frame (df) first, then try to use normal way of handling df.

Accessing the Output:

```
prop <- prop.table(table(origins_factor))
prop #What do you see? A data frame? No difference?
```

```
origins_factor
    Human Zebrafish     Mouse
0.5000000 0.1666667 0.3333333
```

```
prop["Human"]; prop["Mouse"]; prop["Zebrafish"]
```

```
Human
  0.5
```

```
    Mouse
0.3333333
```

```
Zebrafish
0.1666667
```

2.4 Subsetting Data

2.4.1 Vectors

```
# Create a vector
expression_data <- c(3.2, 4.5, 2.1, 6.7, 5.9, 3.3, 7.8, 2.9)
names(expression_data) <- paste0("Sample_", 1:8)
expression_data
```

```
Sample_1 Sample_2 Sample_3 Sample_4 Sample_5 Sample_6 Sample_7 Sample_8
     3.2      4.5      2.1      6.7      5.9      3.3      7.8      2.9
```

```
# Subset by position
expression_data[3]             # Single element
```

```
Sample_3
     2.1
```

```
expression_data[c(1, 3, 5)]    # Multiple elements
```

```
Sample_1 Sample_3 Sample_5
     3.2      2.1      5.9
```

```r
expression_data[2:5]           # Range
```

```
Sample_2 Sample_3 Sample_4 Sample_5
     4.5      2.1      6.7      5.9
```

```r
# Subset by name
expression_data["Sample_6"]
```

```
Sample_6
     3.3
```

```r
expression_data[c("Sample_1", "Sample_8")]
```

```
Sample_1 Sample_8
     3.2      2.9
```

```r
# Subset by condition
expression_data[expression_data > 5]              # Values > 5
```

```
Sample_4 Sample_5 Sample_7
     6.7      5.9      7.8
```

```r
expression_data[expression_data >= 3 & expression_data <= 6]  # Values between 3 and 6
```

```
Sample_1 Sample_2 Sample_5 Sample_6
     3.2      4.5      5.9      3.3
```

2.4.2 Data Frames

```r
# Create a data frame
gene_df <- data.frame(
  gene_id = c("BRCA1", "TP53", "MYC", "EGFR", "GAPDH"),
  expression = c(8.2, 6.1, 9.5, 7.0, 10.0),
  mutation = factor(c("Yes", "No", "Yes", "No", "No")),
  pathway = c("DNA Repair", "Apoptosis", "Cell Cycle", "Signaling", "Metabolism")
)
```

```r
gene_df
```

```
  gene_id expression mutation    pathway
1   BRCA1        8.2      Yes DNA Repair
2    TP53        6.1       No  Apoptosis
3     MYC        9.5      Yes Cell Cycle
4    EGFR        7.0       No  Signaling
5   GAPDH       10.0       No Metabolism
```

```r
# Subsetting by row index
gene_df[1:3, ]         # First three rows, all columns
```

```
  gene_id expression mutation    pathway
1   BRCA1        8.2      Yes DNA Repair
2    TP53        6.1       No  Apoptosis
3     MYC        9.5      Yes Cell Cycle
```

```r
# Subsetting by column index
gene_df[, 1:2]     # All rows, first two columns
```

```
  gene_id expression
1   BRCA1        8.2
2    TP53        6.1
3     MYC        9.5
4    EGFR        7.0
5   GAPDH       10.0
```

```r
# Subsetting by column name
gene_df[, c("gene_id", "mutation")]
```

```
  gene_id mutation
1   BRCA1      Yes
2    TP53       No
3     MYC      Yes
4    EGFR       No
5   GAPDH       No
```

```r
# Using the $ operator
gene_df$expression
```

```
[1]  8.2  6.1  9.5  7.0 10.0
```

```r
gene_df$mutation
```

```
[1] Yes No  Yes No  No
Levels: No Yes
```

```r
# Subsetting by condition
gene_df[gene_df$expression > 8, ]
```

```
  gene_id expression mutation    pathway
1   BRCA1        8.2      Yes DNA Repair
3     MYC        9.5      Yes Cell Cycle
5   GAPDH       10.0       No Metabolism
```

```r
gene_df[gene_df$mutation == "Yes", ]
```

```
  gene_id expression mutation    pathway
1   BRCA1        8.2      Yes DNA Repair
3     MYC        9.5      Yes Cell Cycle
```

```r
# Multiple conditions
gene_df[gene_df$expression > 7 & gene_df$mutation == "No", ]
```

```
  gene_id expression mutation    pathway
5   GAPDH         10       No Metabolism
```

Logical Operators

Operator	Meaning	Example
`==`	Equal to	`x == 5`
`!=`	Not equal	`x != 5`
`<`	Less than	`x < 5`
`>`	Greater than	`x > 5`
`<=`	Less or equal	`x <= 5`

Operator	Meaning	Example
`>=`	Greater or equal	`x >= 5`
`!`	Not	`!(x < 5)`
`|`	OR	`x < 5 | x > 10`
`&`	AND	`x > 5 & x < 10`

2.4.3 Row Names in Data Frames

Row names are particularly important in bioinformatics where genes, proteins, or samples are often used as identifiers.

```
# Setting row names for gene_df
rownames(gene_df) <- gene_df$gene_id
gene_df
```

```
      gene_id expression mutation    pathway
BRCA1   BRCA1        8.2      Yes DNA Repair
TP53     TP53        6.1       No  Apoptosis
MYC       MYC        9.5      Yes Cell Cycle
EGFR     EGFR        7.0       No  Signaling
GAPDH   GAPDH       10.0       No Metabolism
```

We can now drop the gene_id column, if required.

```
gene_df_clean <- gene_df[, -1]  # Remove the first column
gene_df_clean
```

```
      expression mutation    pathway
BRCA1        8.2      Yes DNA Repair
TP53         6.1       No  Apoptosis
MYC          9.5      Yes Cell Cycle
EGFR         7.0       No  Signaling
GAPDH       10.0       No Metabolism
```

```
# Access rows by name
gene_df_clean["TP53", ]
```

```
     expression mutation   pathway
TP53        6.1       No Apoptosis
```

```r
# Check if row names are unique
any(duplicated(rownames(gene_df_clean)))
```

```
[1] FALSE
```

```r
# Handle potential duplicated row names
# NOTE: R doesn't allow duplicate row names by default
dup_genes <- data.frame(
  expression = c(5.2, 6.3, 5.2, 8.1),
  mutation = c("Yes", "No", "Yes", "No")
)

# This would cause an error:
#rownames(dup_genes) <- c("BRCA1", "BRCA1", "TP53", "EGFR")

# Instead, we can preemptively make them unique:
proposed_names <- c("BRCA1", "BRCA1", "TP53", "EGFR")
unique_names <- make.unique(proposed_names)
unique_names  # Show the generated unique names
```

```
[1] "BRCA1"   "BRCA1.1" "TP53"    "EGFR"
```

```r
# Now we can safely assign them
rownames(dup_genes) <- unique_names
dup_genes
```

```
        expression mutation
BRCA1          5.2      Yes
BRCA1.1        6.3       No
TP53           5.2      Yes
EGFR           8.1       No
```

Note

Why is unique name important for us? Imagine this meaningful biological scenario: one gene might transcribed into many transcript isoforms and hence many protein isoforms. From RNAseq data, we might get alignment count for each gene. But then we can separate the count for each transcript. One gene has one name or ID, but the transcripts are many for the same gene! So, we can denote, for example, 21 isoform of geneA

like genA.1, geneA.2, geneA.3,……., geneA.21. See this link for MBP gene. How many transcript isoforms does it have?

2.5 Homework: Factors, Subsetting, and Biological Insight

1. **(Factor vs Character)** Explain the difference between a character vector and a factor in R. Why would `mutation_status` be a factor and not just a character vector?
2. **(Factor Level Order)** You observed the following bacterial species in gut microbiome samples:

```
species <- c("Lactobacillus", "Bacteroides", "Escherichia", "Bacteroides", "Lactobacillus"
species_factor <- factor(species, levels = c("Bacteroides", "Escherichia", "Lactobacillus"
```

What will `levels(species_factor)` return? Why?

3. Given the factor:

```
disease_severity <- factor(c("Mild", "Severe", "Moderate"), levels = c("Mild", "Moderate",
```

What will be the result of `disease_severity[1] < disease_severity[2]` and why?

4. You computed:

```
prop <- prop.table(table(species_factor))
```

How do you extract the proportion of "Escherichia" samples from prop? Is `prop$Escherichia` valid?

5. Interpret what this query returns:

```
gene_df[gene_df$expression > 7 & gene_df$mutation == "No", ]
```

What type of genes does it select?

6. You have:

```
samples <- c("WT", "KO", "WT", "KO", "WT")
expression <- c(5.2, 8.1, 4.3, 9.0, 5.7)
```

Make a dataframe using these 2 vectors first. Then,

(a) Create a factor `group_factor` for the samples.

(b) Use `tapply()` to calculate mean expression per group.

> **Note**
>
> Use `?tapply()` to see how to use it.
> **Hint:** You need to provide things for `X`, `INDEX`, `FUN`. You have `X`, `INDEX` in this small dataframe. The `FUN` should be applied thinking of what you are trying to do. You are trying to get the mean or average, right?

(c) Plot a `barplot` of average expression for each group.

7. Use the `gene_df` example. Subset the data to find genes with:

- expression > 8
- pathway is either "Cell Cycle" or "Signaling"

8. Create an ordered factor for the disease stages: `c("Stage I", "Stage III", "Stage II", "Stage IV", "Stage I")`. Then plot the number of patients per stage using `barplot()`. Confirm that `"Stage III" > "Stage I"` is logical in your factor.

9. Suppose `gene_data` has a column type with values "Oncogene", "Tumor Suppressor", and "Housekeeping".

- Subset all "Oncogene" rows where expression > 8.
- Change the reference level of the factor type to "Housekeeping"

10. Simulate expression data for 3 tissues: We are going to use `rnorm()` function to generate random values from a normal distribution for this purpose. The example values inside the `rnorm()` function means we want:

- 30 values in total,
- average or mean value = 8,
- standard deviation of expression is 2.

You can play with the numbers to make your own values.

`rep()` function is to replicate things (many times). In this example, we have `rep(c("brain", "liver", "kidney"), each = 10)`. We will be having 10x "brains", followed by 10x "liver", followed by 10x "kidney". So, if you have changed your values inside the `rnorm()` function, make this value meaningful for you. Now we have 3 things, each=10. So, 3*10=30 is matching with the total value inside `rnorm()` function. Got it?

```
set.seed(42) #just for reproducibility. Not completely needed
gene_expr <- rnorm(30, mean = 8, sd = 2)
tissue <- rep(c("brain", "liver", "kidney"), each = 10)
```

```
tissue_factor <- factor(tissue, levels = c("liver", "brain", "kidney"))
```

- Make a boxplot showing expression per tissue.
- Which tissue shows the most variable gene expression? (Use `tapply()` + `sd()`)

> **Note**
>
> **Hint:** Variability is expressed in another way (squaring) of measuring standard deviation (`sd`). Now, do you see how to use `sd` inside `tapply()` function?

Use these questions as a self-check – reflect on why each step works before moving on to the next level (question).

Push your `.Rmd` file and share by Friday 10PM BD Time.

2.6 Handling Missing/Wrong Values

2.6.1 Identifying Issues

```
# Create data with missing values
clinical_data <- data.frame(
  patient_id = 1:5,
  age = c(25, 99, 30, -5, 40),     # -5 is wrong, 99 is suspect
  bp = c(120, NA, 115, 125, 118),  # NA is missing
  weight = c(65, 70, NA, 68, -1)   # -1 is wrong
)
clinical_data
```

```
  patient_id age  bp weight
1          1  25 120     65
2          2  99  NA     70
3          3  30 115     NA
4          4  -5 125     68
5          5  40 118     -1
```

```
# Check for missing values
is.na(clinical_data)
```

```
     patient_id   age    bp weight
[1,]      FALSE FALSE FALSE  FALSE
[2,]      FALSE FALSE  TRUE  FALSE
[3,]      FALSE FALSE FALSE   TRUE
[4,]      FALSE FALSE FALSE  FALSE
[5,]      FALSE FALSE FALSE  FALSE
```

```
colSums(is.na(clinical_data))  # Count NAs by column
```

```
patient_id        age         bp     weight
         0          0          1          1
```

```
# Check for impossible values
clinical_data$age < 0
```

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

```
clinical_data$weight < 0
```

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

```
# Find indices of problematic values
which(clinical_data$age < 0 | clinical_data$age > 90)
```

```
[1] 2 4
```

2.6.2 Fixing Data

```
# Replace impossible values with NA
clinical_data$age[clinical_data$age < 0 | clinical_data$age > 90] <- NA
clinical_data$weight[clinical_data$weight < 0] <- NA
clinical_data
```

```
  patient_id age  bp weight
1          1  25 120     65
2          2  NA  NA     70
```

```
3          3  30 115     NA
4          4  NA 125     68
5          5  40 118     NA
```

```
# Replace NAs with mean (common in biological data)
clinical_data$bp[is.na(clinical_data$bp)] <- mean(clinical_data$bp, na.rm = TRUE)
clinical_data$weight[is.na(clinical_data$weight)] <- mean(clinical_data$weight, na.rm = TR
clinical_data
```

```
  patient_id age    bp   weight
1          1  25 120.0 65.00000
2          2  NA 119.5 70.00000
3          3  30 115.0 67.66667
4          4  NA 125.0 68.00000
5          5  40 118.0 67.66667
```

```
# Replace NAs with median (better for skewed data)
clinical_data$age[is.na(clinical_data$age)] <- median(clinical_data$age, na.rm = TRUE)
clinical_data
```

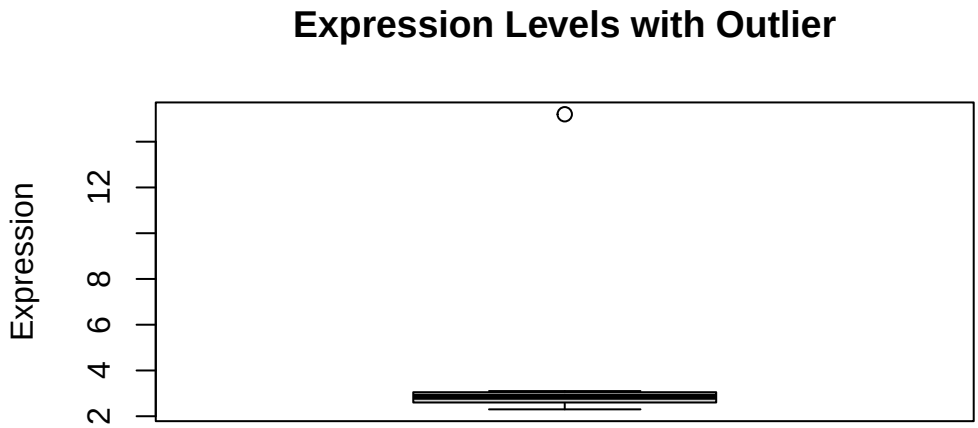
```
  patient_id age    bp   weight
1          1  25 120.0 65.00000
2          2  30 119.5 70.00000
3          3  30 115.0 67.66667
4          4  30 125.0 68.00000
5          5  40 118.0 67.66667
```

2.7 Data Transformation

2.7.1 Introduction to Outliers

Outliers can significantly affect statistical analyses, especially in biological data where sample variation can be high.

```
# Create data with outliers
expression_levels <- c(2.3, 2.7, 3.1, 2.9, 2.5, 3.0, 15.2, 2.8)
boxplot(expression_levels,
        main = "Expression Levels with Outlier",
        ylab = "Expression")
```



2.7.2 Identifying Outliers

```
# Statistical approach: Values beyond 1.5*IQR
data_summary <- summary(expression_levels)
data_summary
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.300   2.650   2.850   4.312   3.025  15.200
```

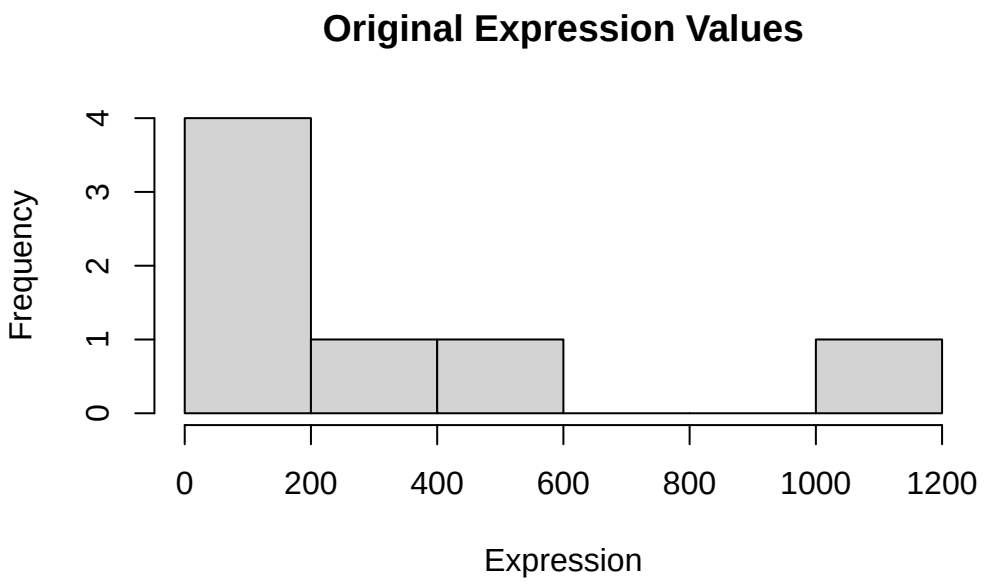
```
IQR_value <- IQR(expression_levels)
upper_bound <- data_summary["3rd Qu."] + 1.5 * IQR_value
lower_bound <- data_summary["1st Qu."] - 1.5 * IQR_value

# Find outliers
outliers <- expression_levels[expression_levels > upper_bound |
                              expression_levels < lower_bound]
outliers
```

```
[1] 15.2
```

2.7.3 Transforming Vectors

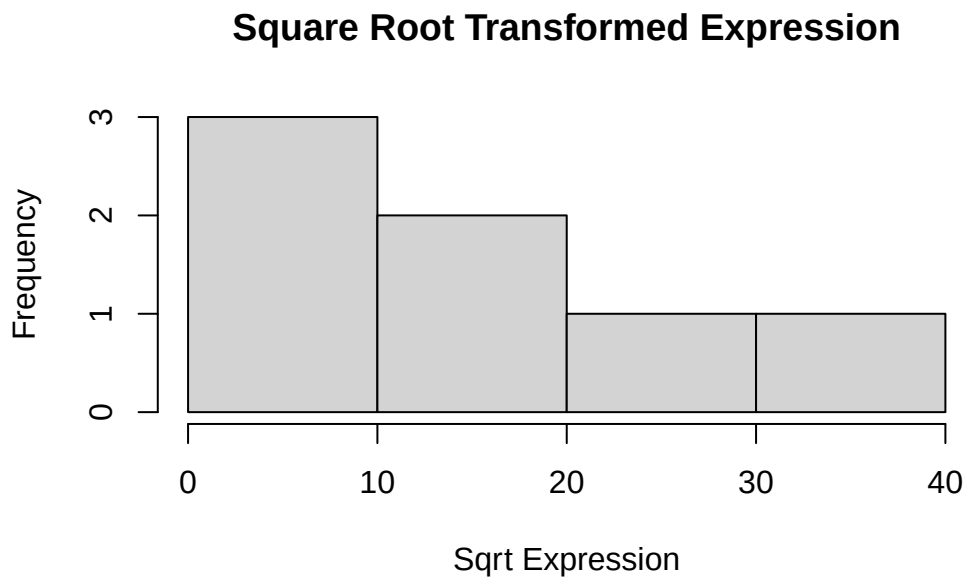
Mathematical transformations can normalize data, reduce outlier effects, and make data more suitable for statistical analyses.

```r
# Original data
gene_exp <- c(15, 42, 87, 115, 320, 560, 1120)
hist(gene_exp, main = "Original Expression Values", xlab = "Expression")
```

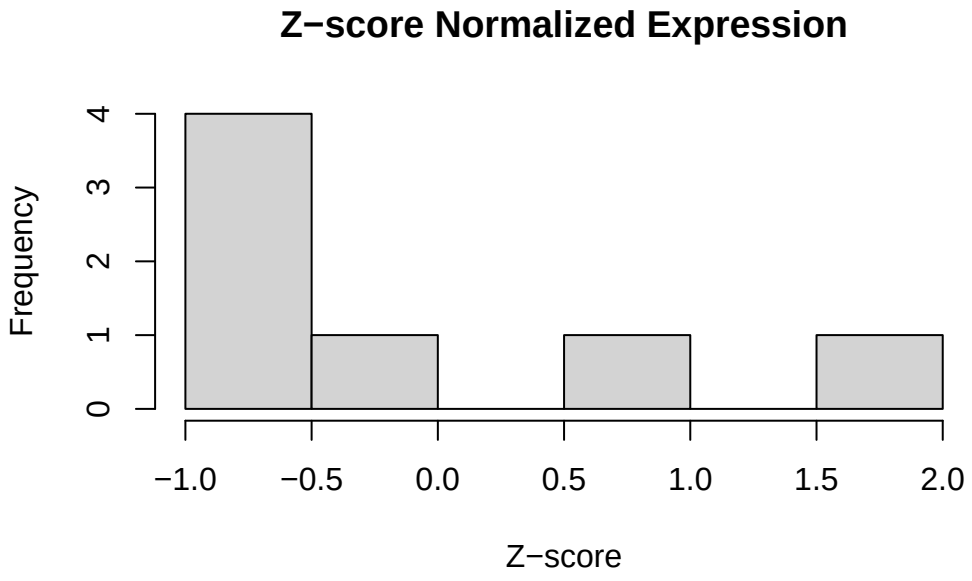



```r
# Log transformation (common in gene expression analysis)
log_exp <- log2(gene_exp)
hist(log_exp, main = "Log2 Transformed Expression", xlab = "Log2 Expression")
```

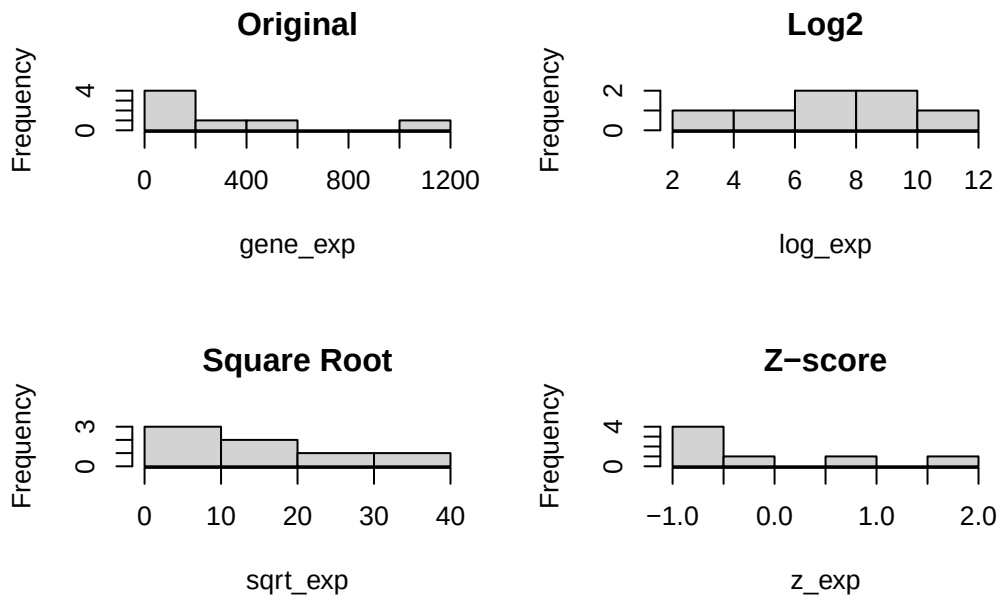
```r
# Square root transformation (less aggressive than log)
sqrt_exp <- sqrt(gene_exp)
hist(sqrt_exp, main = "Square Root Transformed Expression", xlab = "Sqrt Expression")
```




```r
# Z-score normalization (standardization)
z_exp <- scale(gene_exp)
hist(z_exp, main = "Z-score Normalized Expression", xlab = "Z-score")
```



```r
# Compare transformations
par(mfrow = c(2, 2))
hist(gene_exp, main = "Original")
hist(log_exp, main = "Log2")
hist(sqrt_exp, main = "Square Root")
hist(z_exp, main = "Z-score")
```




```r
par(mfrow = c(1, 1))  # Reset plotting layout
```

2.7.4 Logical Expressions

```r
# Create gene expression vector
exp_data <- c(5.2, 3.8, 7.1, 2.9, 6.5, 8.0, 4.3)
names(exp_data) <- paste0("Gene_", 1:7)

# Basic comparisons
exp_data > 5    # Which genes have expression > 5?
```

```
Gene_1 Gene_2 Gene_3 Gene_4 Gene_5 Gene_6 Gene_7
  TRUE  FALSE   TRUE  FALSE   TRUE   TRUE  FALSE
```

```
exp_data <= 4   # Which genes have expression <= 4?
```

```
Gene_1 Gene_2 Gene_3 Gene_4 Gene_5 Gene_6 Gene_7
 FALSE   TRUE  FALSE   TRUE  FALSE  FALSE  FALSE
```

```
# Store results in logical vector
high_exp <- exp_data > 6
high_exp
```

```
Gene_1 Gene_2 Gene_3 Gene_4 Gene_5 Gene_6 Gene_7
 FALSE  FALSE   TRUE  FALSE   TRUE   TRUE  FALSE
```

```
# Use logical vectors for subsetting
exp_data[high_exp]  # Get high expression values
```

```
Gene_3 Gene_5 Gene_6
   7.1    6.5    8.0
```

2.7.5 Logical Operators

```
# Combining conditions with AND (&)
exp_data > 4 & exp_data < 7  # Expression between 4 and 7
```

```
Gene_1 Gene_2 Gene_3 Gene_4 Gene_5 Gene_6 Gene_7
  TRUE  FALSE  FALSE  FALSE   TRUE  FALSE   TRUE
```

```
# Combining conditions with OR (|)
exp_data < 4 | exp_data > 7  # Expression less than 4 OR greater than 7
```

```
Gene_1 Gene_2 Gene_3 Gene_4 Gene_5 Gene_6 Gene_7
 FALSE   TRUE   TRUE   TRUE  FALSE   TRUE  FALSE
```

```
# Using NOT (!)
!high_exp  # Not high expression
```

```
Gene_1 Gene_2 Gene_3 Gene_4 Gene_5 Gene_6 Gene_7
  TRUE   TRUE  FALSE   TRUE  FALSE  FALSE   TRUE
```

```
# Subsetting with combined conditions
exp_data[exp_data > 4 & exp_data < 7]  # Get values between 4 and 7
```

```
Gene_1 Gene_5 Gene_7
   5.2    6.5    4.3
```

2.7.6 Logical Functions

```
# all() - Are all values TRUE?
all(exp_data > 0)  # Are all expressions positive?
```

```
[1] TRUE
```

```
# any() - Is at least one value TRUE?
any(exp_data > 7)  # Is any expression greater than 7?
```

```
[1] TRUE
```

```
# which() - Get indices of TRUE values
which(exp_data > 6)  # Which elements have expressions > 6?
```

```
Gene_3 Gene_5 Gene_6
     3      5      6
```

```
# %in% operator - Test for membership
test_genes <- c("Gene_1", "Gene_5", "Gene_9")
names(exp_data) %in% test_genes  # Which names match test_genes?
```

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

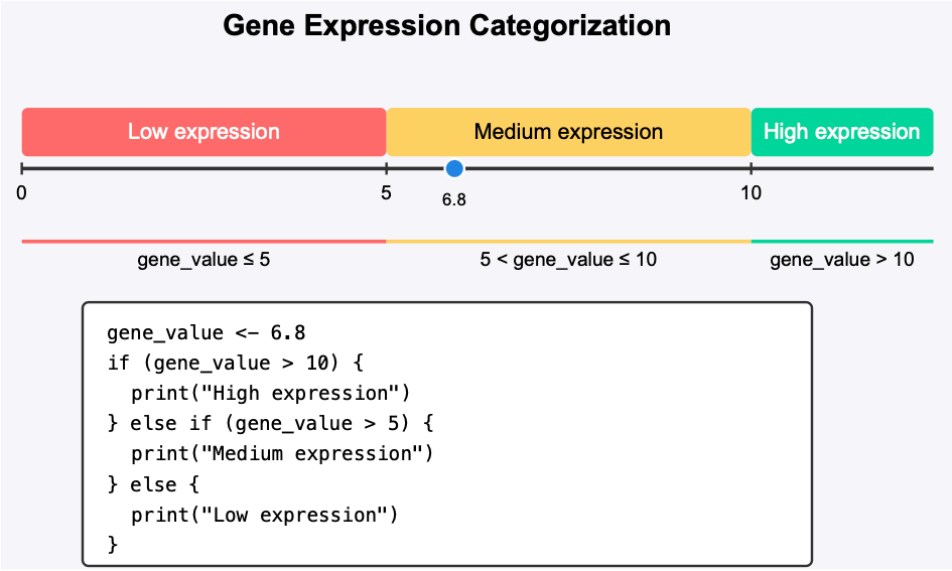
2.8 Conditionals

2.8.1 `if-else` statement

```
# if-else statement
gene_value <- 6.8

if(gene_value > 10) {
  print("High expression")
} else if(gene_value > 5) {
  print("Medium expression")
} else {
  print("Low expression")
}
```

```
[1] "Medium expression"
```

Visualize it using the image below.

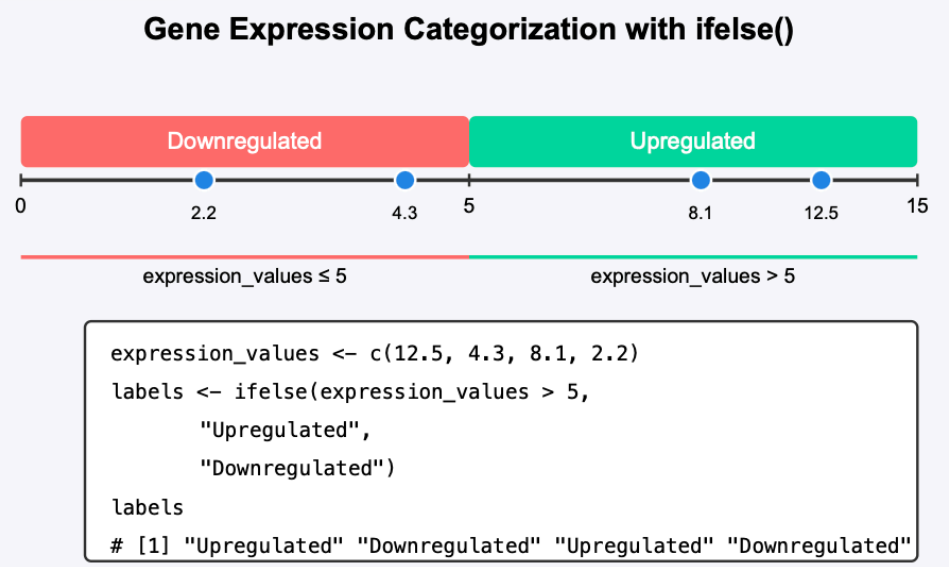


2.8.2 `elseif` statement for vectors

`elseif` is binary in nature. So, we can categorize only 2 things using `elseif`. See this example:

```r
expression_values <- c(12.5, 4.3, 8.1, 2.2)
labels <- ifelse(expression_values > 5, "Upregulated", "Downregulated")
labels
```

```
[1] "Upregulated"   "Downregulated" "Upregulated"   "Downregulated"
```

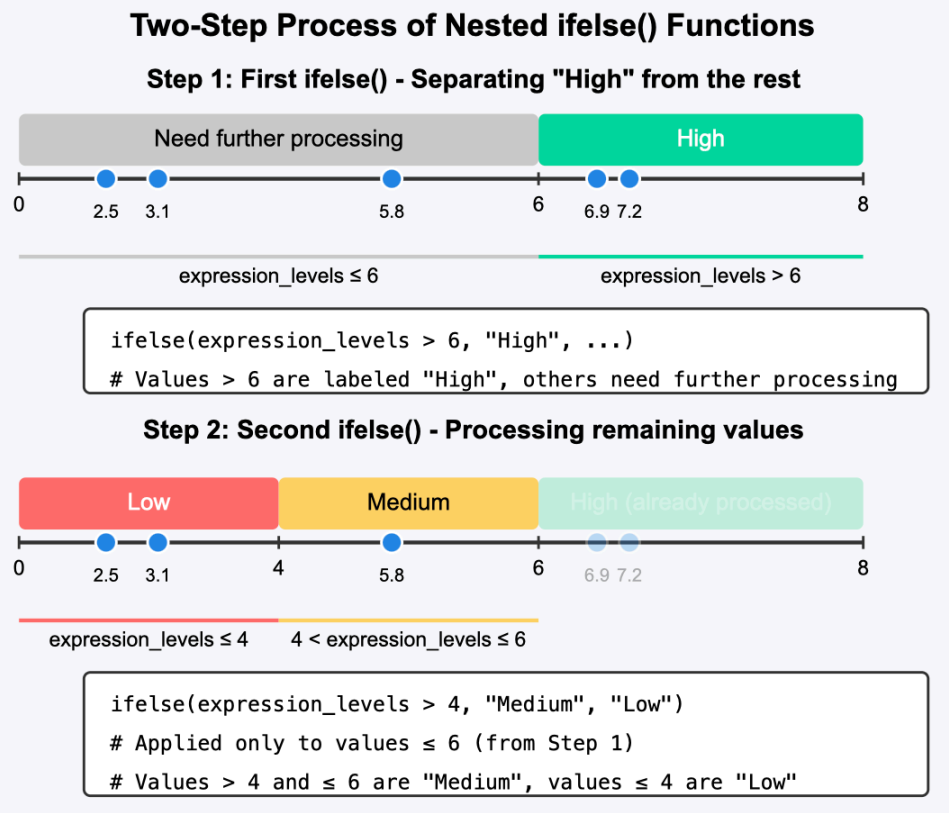


> **Note**
>
> `elseif` has 3 things inside the parentheses, right? The first one is the condition, the second one is the category we define if the condition is met, and the third thing is the other remaining category we want to assign if the condition is not met. So, it's usage is perfect to say if a gene/transcript is upregulated or downregulated (binary classification).

If we still want to categorize more than 2 categories using `elseif`, we need to use it in a nested way. See this example:

```r
# ifelse() for vectors
expression_levels <- c(2.5, 5.8, 7.2, 3.1, 6.9)
expression_category <- ifelse(expression_levels > 6,
                              "High",
                              ifelse(expression_levels > 4, "Medium", "Low"))
expression_category
```

```
[1] "Low"    "Medium" "High"   "Low"    "High"
```



> **Note**
>
> You remember the general structure of `elseif` loop, right? the second thing after the first `,` is the assigned category if the condition is met. So, we assigned it as `High` here in this example. But then after the second `,` there is a second `elseif` loop instead of a category. The second loop makes 2 more binary categories `Medium` and `Low`, and our task of assigning 3 categories is achieved.

`dplyr` package has a function named `case-when()` to help us use as many categories we want. The same task would be achieved like this:

```
# Requires dplyr package
#install.packages("dplyr") #decomment if you need to install the package
library(dplyr)
expression_levels <- c(2.5, 5.8, 7.2, 3.1, 6.9)
labels <- case_when(
```

```
  expression_levels > 6 ~ "High",
  expression_levels > 4 ~ "Medium",
  TRUE ~ "Low"  # Default case
)
labels
```

```
[1] "Low"    "Medium" "High"   "Low"    "High"
```

> **Note**
>
> Do you see the point how you would use the `elseif` loop if you wanted to write a function to make 4 or 5 categories? If not, pause and re-think. You need to see the point. But anyway, categorizing more than 2 is better using `if else` statement

2.8.3 `for` loop

```
genes <- c("BRCA1", "TP53", "MYC", "CDC2", "MBP")
expr <- c(8.2, 5.4, 11.0, 5.4, 13.0)

for (i in 1:length(genes)) {
  status <- if (expr[i] > 10) "High" else if (expr[i] > 6) "Moderate" else "Low"
  cat(genes[i], "has", status, "expression\n")
}
```

```
BRCA1 has Moderate expression
TP53 has Low expression
MYC has High expression
CDC2 has Low expression
MBP has High expression
```

In-class Task:

- Make a data frame using `genes` and `expr`.
- Add/flag the categories `High`, `Moderate` and `Low` you get using the `for` loop in a new column named `expression_level` or similar.

```
# Step 1: Vectors
genes <- c("BRCA1", "TP53", "MYC", "CDC2", "MBP")
expr <- c(8.2, 5.4, 11.0, 5.4, 13.0)
```

```r
# Step 2: Create a data frame
gene_df <- data.frame(gene = genes, expression = expr)

# Step 3: Add an empty column for expression level
gene_df$expression_level <- NA

# Step 4: Use for loop to fill in the expression_level column
for (i in 1:nrow(gene_df)) {
  gene_df$expression_level[i] <- if (gene_df$expression[i] > 10) {
    "High"
  } else if (gene_df$expression[i] > 6) {
    "Moderate"
  } else {
    "Low"
  }
}

# View the final data frame
print(gene_df)
```

```
   gene expression expression_level
1 BRCA1        8.2         Moderate
2  TP53        5.4              Low
3   MYC       11.0             High
4  CDC2        5.4              Low
5   MBP       13.0             High
```

2.8.4 `while` loop

Context: You are preparing biological samples (e.g., blood, DNA extracts) for analysis. You have a set of samples labeled `Sample 1` to `Sample 5`. You want to check each one in order and confirm that it's ready for analysis. Use a while loop to process the samples sequentially.

```r
i <- 1
while (i <= 5) {
  cat("Sample", i, "is ready for analysis\n")
  i <- i + 1
}
```

```
Sample 1 is ready for analysis
Sample 2 is ready for analysis
Sample 3 is ready for analysis
Sample 4 is ready for analysis
Sample 5 is ready for analysis
```

2.8.5 `next` and `break`

Context: You are screening biological samples (e.g., tissue or blood) in a quality control process. Some samples are good, some are suboptimal (not contaminated but poor quality), and some are contaminated (must be flagged and stop further processing). Use `next` to skip suboptimal samples and `break` to immediately stop when a contaminated sample is found.

```r
samples <- c("good", "bad", "good", "contaminated")

for (s in samples) {
  if (s == "contaminated") {
    print("Stop! Contaminated sample.")
    break
  }
  if (s == "bad") next
  print(paste("Processing", s))
}
```

```
[1] "Processing good"
[1] "Processing good"
[1] "Stop! Contaminated sample."
```

2.9 Writing Functions in R

2.9.1 Flag gene expression

```r
flag_expression <- function(value) {
  if (value > 10) {
    return("High")
  } else if (value > 5) {
    return("Moderate")
  } else {
```

```
    return("Low")
  }
}

flag_expression(8.3)
```

```
[1] "Moderate"
```

Apply to a vector

```
expr_values <- c(12.2, 4.4, 7.5)
sapply(expr_values, flag_expression)
```

```
[1] "High"     "Low"      "Moderate"
```

2.9.2 Function with multiple arguments

```
gene_status <- function(gene, expression, threshold = 6) {
  label <- ifelse(expression > threshold, "Up", "Down")
  return(paste(gene, "is", label, "regulated"))
}

gene_status("TP53", 8.1)
```

```
[1] "TP53 is Up regulated"
```

2.9.3 Return a list

```
calc_stats <- function(values) {
  return(list(mean = mean(values), sd = sd(values)))
}

calc_stats(c(4.2, 5.5, 7.8))
```

```
$mean
[1] 5.833333
```

```
$sd
[1] 1.823001
```

2.9.4 Practical Session

Check out this repo: `https://github.com/genomicsclass/dagdata/`

```
# Download small example dataset
download.file("https://github.com/genomicsclass/dagdata/raw/master/inst/extdata/msleep_ggp
              destfile = "msleep_data.csv")

# Load data
msleep <- read.csv("msleep_data.csv")
```

1. Convert 'vore' column to factor and plot its distribution.
2. Create a matrix of sleep data columns and add row names.
3. Find and handle any missing values.
4. Calculate mean sleep time by diet category (vore).
5. Identify outliers in sleep_total.

2.10 Summary of the Lesson

In this lesson, we covered:

1. **Factor Variables**: Essential for categorical data in biology (genotypes, treatments, etc.)

- Creation, levels, ordering, and visualization

2. **Subsetting Techniques**: Critical for data extraction and analysis

- Vector and data frame subsetting with various methods
- Using row names effectively for biological identifiers

3. **Matrix Operations**: Fundamental for expression data

- Creation, manipulation, and biological applications
- Calculating fold changes and other common operations

4. **Missing Values**: Practical approaches for real-world biological data

- Identification and appropriate replacement methods

5. **Data Transformation**: Making data suitable for statistical analysis

- Log, square root, and z-score transformations
- Outlier identification and handling

6. **Logical Operations**: For data filtering and decision making

- Conditions, combinations, and applications

   These skills form the foundation for the more advanced visualization techniques we'll cover in future lessons.

7. **List**: Fundamental for many biological data and packages' output.

- Properties, accessing, and applications

8. We will know more about conditionals, R packages to handle data and visualization in a better and efficient way.

2.11 Homework

1. **Matrix Operations**:

- Create a gene expression matrix with 8 genes and 4 conditions
- Calculate the mean expression for each gene
- Calculate fold change between condition 4 and condition 1
- Create a heatmap of your matrix

2. **Factor Analysis**:

- Using the `iris` dataset, convert Species to an ordered factor
- Create boxplots showing Sepal.Length by Species
- Calculate mean petal length for each species level

3. **Data Cleaning Challenge**:

- In the downloaded `msleep_data.csv`:
- Identify all columns with missing values
- Replace missing values appropriately
- Create a new categorical variable "sleep_duration" with levels "Short", "Medium", "Long"

4. **List challenge**:

- Make your own lists
- Replicate all the tasks we did
- You may ask AI to give you beginner-level questions but don't ask to solve the questions programmatically. Tell AI not to provide answers.

5. **Complete Documentation**:

- Write all code in R Markdown
- Include comments explaining your approach
- Push to GitHub

2.11.0.1 Due date: Friday 10pm BD Time