Basic R

Md Rasheduzzaman

2025-04-26

Data types, variables, vectors, data frame, functions

Table of contents

1 L1: 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
```

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

```
[1] 2
```

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

```
[1] 30
```

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

```
[1] 4
```

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

```
[1] 30
```

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

```
[1] 1.75
```

```
[1] 10
```

```
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
```

```r
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?

```r
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
```

```r
y / x
```

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

```r
log2(x)
```

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

```r
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
```

```r
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

```
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 L2: 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
```

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

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

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

```
  names gender age
1  Mina Female  15
```

```
3 Mithu Female   2
```

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

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

```
#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
```

Check your colleague's repo for the Q3.

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 assume 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.

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] 23.54667
```

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

```
[1] 23.86417
```

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

```
[1] 25.34333
```

Do you see the difference in the mean value now?

2.1.5 Some 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 Some 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
```

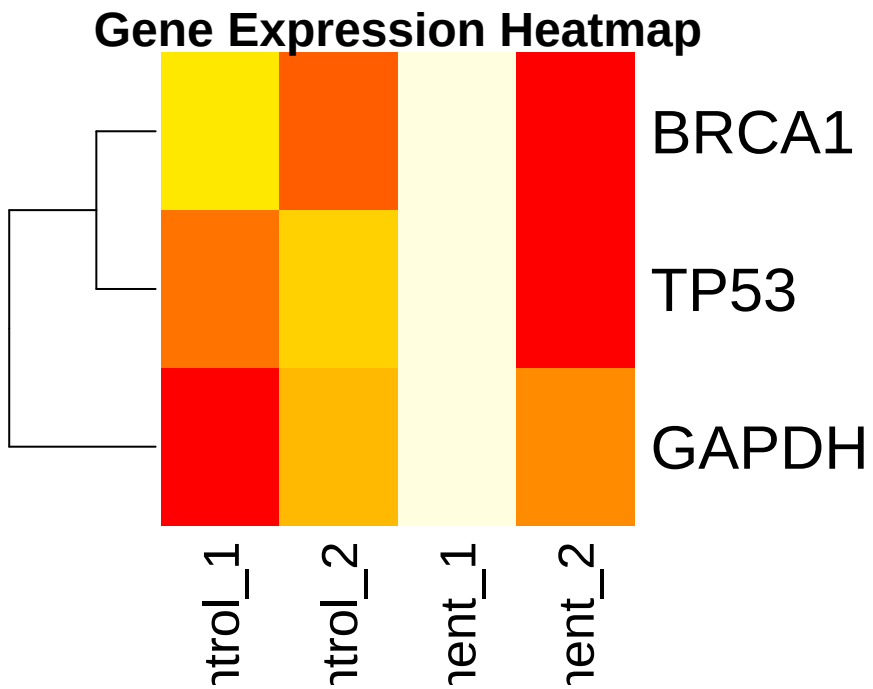
```r
# Mean expression per sample
sample_means <- colMeans(expr_matrix)
sample_means
```

```
  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")
```



2.1.7 Some 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?

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

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

```
# 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
```

```
# 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

```r
# 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"
```

Converting Between Structures

```r
# 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

```r
# 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:

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

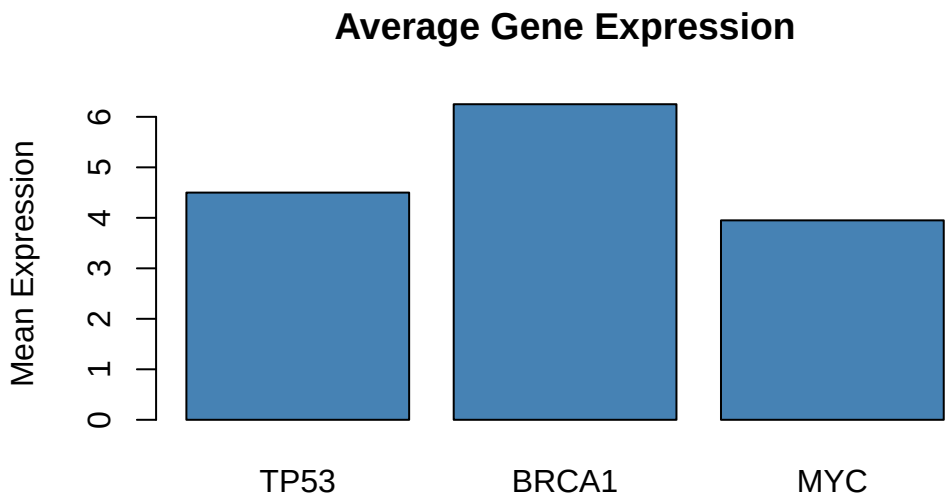
```
[1] 3 2
```

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

```
[1] 3
```

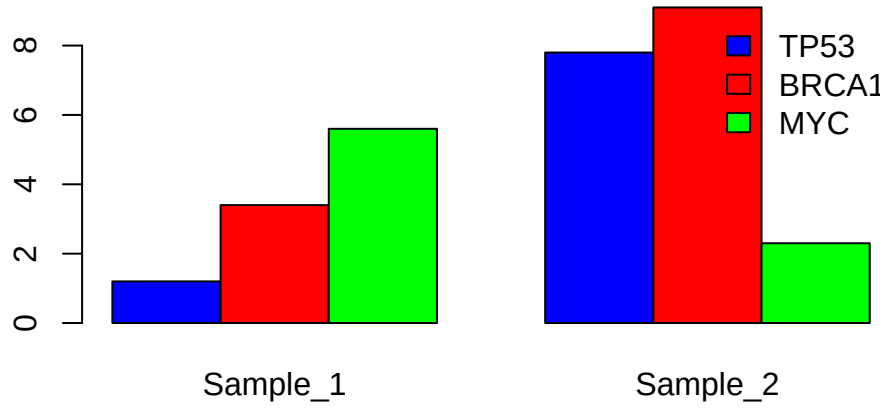
A. Gene-Centric (Mean Expression)

```
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)

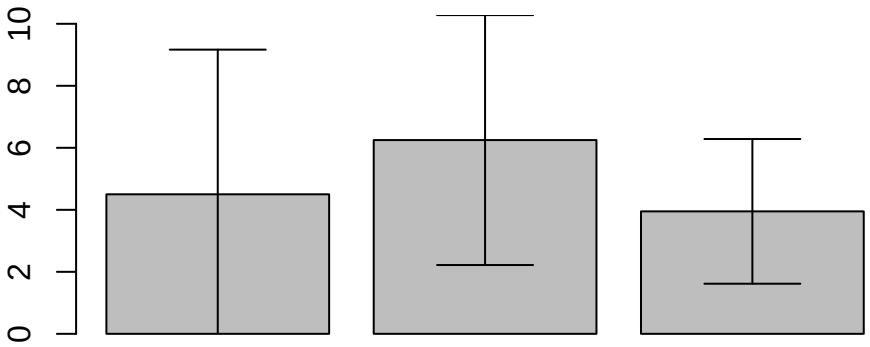
```
barplot(genomics_data$expression,
        beside = TRUE,
        names.arg = paste0("Sample_", 1:ncol(genomics_data$expression)),
        legend.text = genomics_data$gene_names,
        args.legend = list(x = "topright", bty = "n"),
        col = c("blue", "red", "green"),
        main = "Expression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

Important for categorical data

2.2.1 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)
origins_factor
```

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

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

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

```
# 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] TRUE
```

2.2.2 Factor Operations

```
# Count frequencies
table(origins_factor)
```

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

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

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

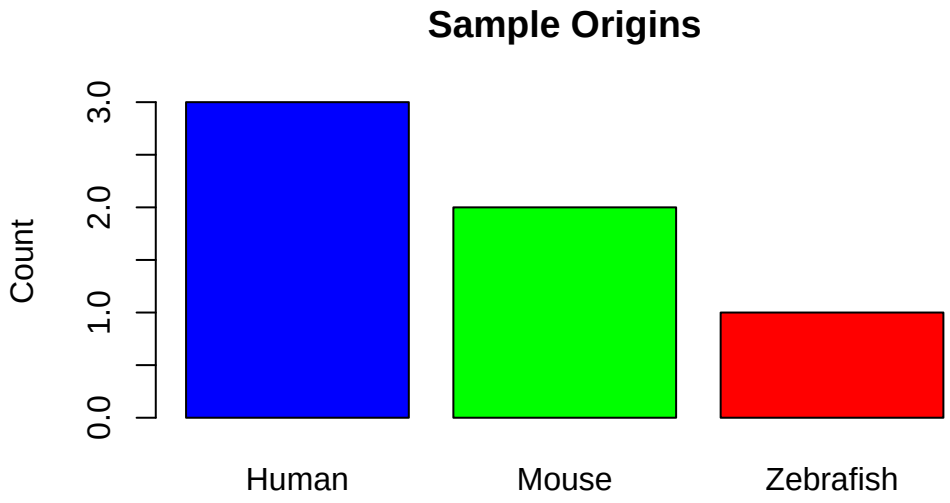
```
# Change reference level (important for statistical models)
origins_factor_relevel <- relevel(origins_factor, ref = "Mouse")
origins_factor_relevel
```

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

```
# Convert to character
as.character(origins_factor)
```

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

```
# Plot factors - Basic barplot
barplot(table(origins_factor),
        col = c("blue", "green", "red"),
        main = "Sample Origins",
        ylab = "Count")
```



More advanced plot with factors:

```r
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**
>
> 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.table(table(origins_factor)) 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:

```r
prop <- prop.table(table(origins_factor))
prop
```

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

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

```
Human
  0.5
```

```
    Mouse
0.3333333
```

```
Zebrafish
0.1666667
```

2.3 Subsetting Data

2.3.1 Vectors

```r
# 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
```

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

```
Sample_3
     2.1
```

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

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

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

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

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

```
Sample_6
     3.3
```

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

```
Sample_1 Sample_8
     3.2      2.9
```

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

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

```
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
```

2.3.2 Data Frames

```
# 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")
)

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
```

```
# 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
```

```
# Subsetting by column index
gene_df[, c(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
```

```
# 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
```

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

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

```
gene_df$mutation
```

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

```
# 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
```

```
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
```

```
# 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`

2.3.3 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 desired
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
```

```r
# Another approach: allow duplicates but with a warning
# Using row.names argument during creation (internally calls make.unique)
#dup_genes2 <- data.frame(
#  expression = c(5.2, 6.3, 5.2, 8.1),
#  mutation = c("Yes", "No", "Yes", "No"),
#  row.names = c("BRCA1", "BRCA1", "TP53", "EGFR")
#)
```

```
#dup_genes2  # Notice how R handles the duplicates automatically
```

2.4 Handling Missing/Wrong Values

2.4.1 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
```

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

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

```r
clinical_data$weight < 0
```

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

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

```
[1] 2 4
```

2.4.2 Fixing Data

```r
# 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
```

```r
# 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
```

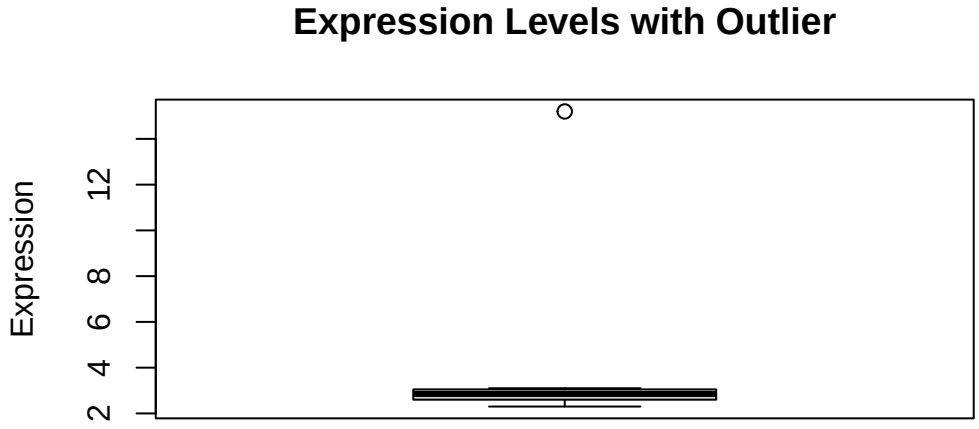
```
  patient_id age    bp   weight
1          1  25 120.0 65.00000
2          2  30 119.5 70.00000
3          3  30 115.0 67.66667
4          4  30 125.0 68.00000
5          5  40 118.0 67.66667
```

2.5 Data Transformation

2.5.1 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")
```



2.5.2 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
```

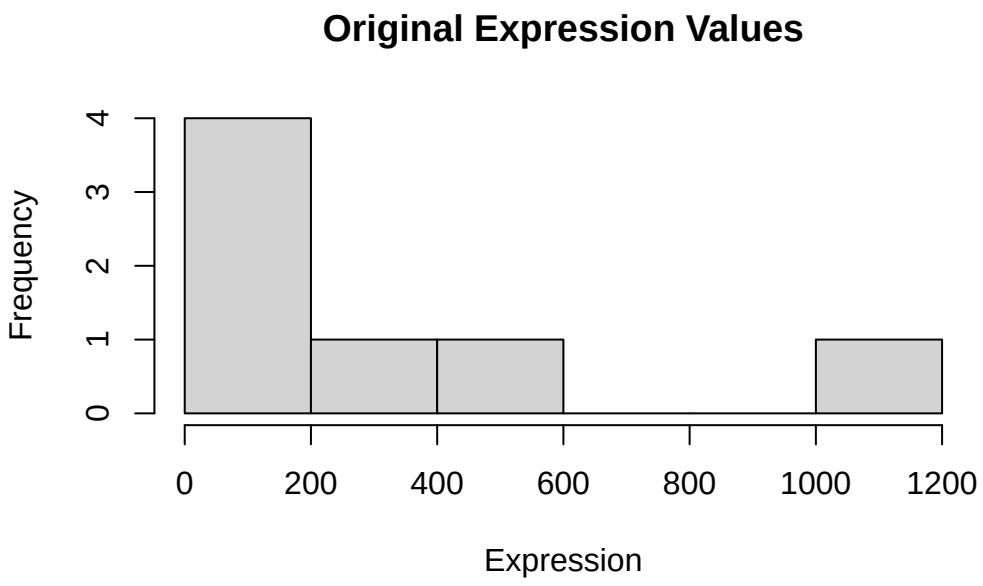
```
IQR_value <- IQR(expression_levels)
upper_bound <- data_summary["3rd Qu."] + 1.5 * IQR_value
lower_bound <- data_summary["1st Qu."] - 1.5 * IQR_value

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

```
[1] 15.2
```

2.5.3 Transforming Vectors

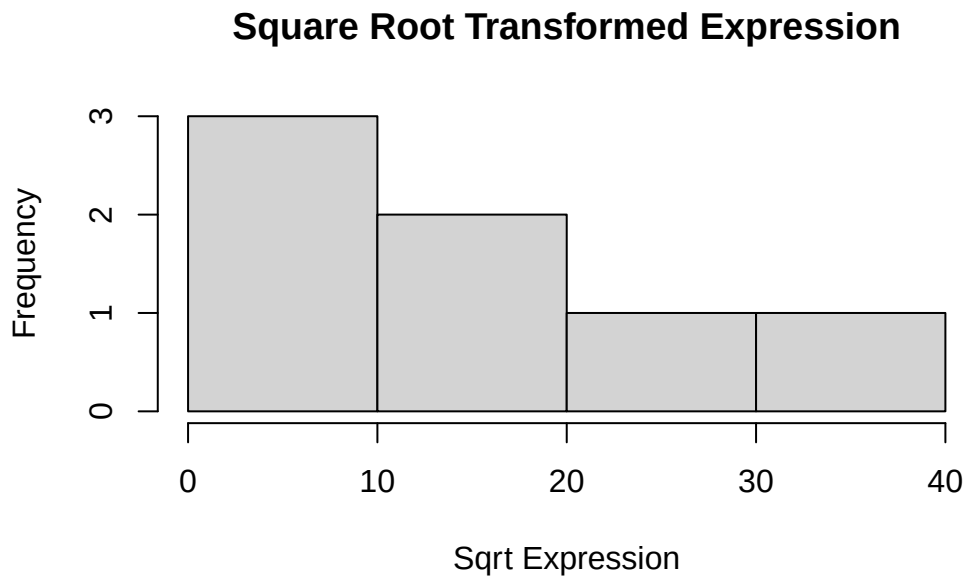
Mathematical transformations can normalize data, reduce outlier effects, and make data more suitable for statistical analyses.

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

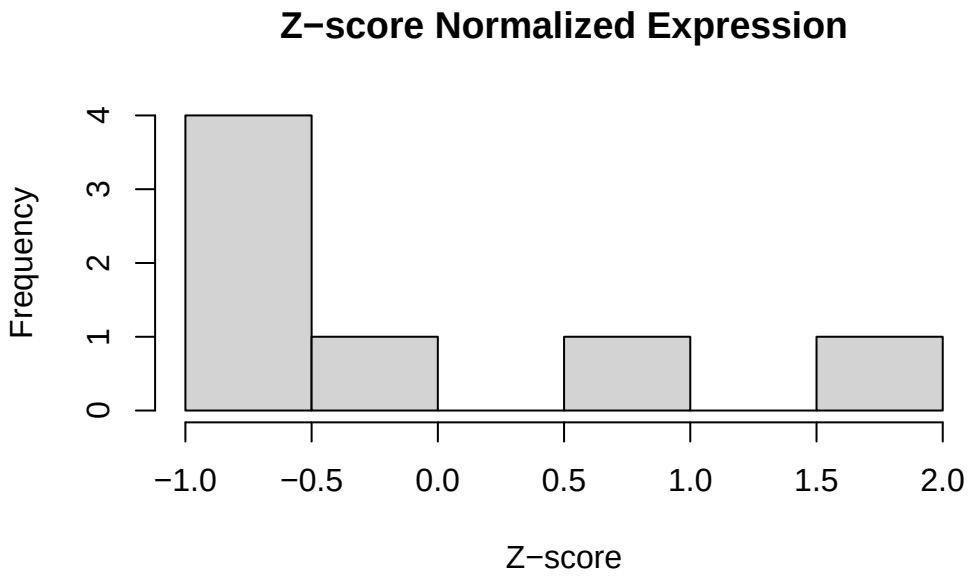



```r
# Log transformation (common in gene expression analysis)
log_exp <- log2(gene_exp)
hist(log_exp, main = "Log2 Transformed Expression", xlab = "Log2 Expression")
```


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

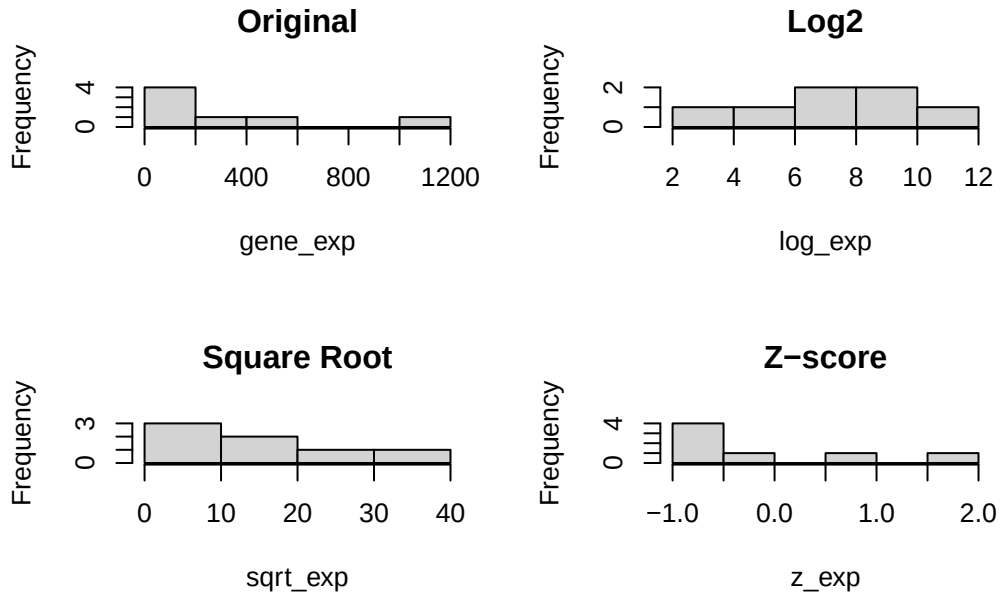



```r
# Z-score normalization (standardization)
z_exp <- scale(gene_exp)
hist(z_exp, main = "Z-score Normalized Expression", xlab = "Z-score")
```




```r
# Compare transformations
par(mfrow = c(2, 2))
hist(gene_exp, main = "Original")
```

```r
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
```

2.5.4 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
```

```r
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
```

2.5.5 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
```

```r
# 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
```

2.5.6 Logical Functions

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

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

```r
# %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
```

2.5.7 Conditionals

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

if(gene_value > 6) {
  cat("High expression\n")
} else if(gene_value > 4) {
  cat("Medium expression\n")
} else {
  cat("Low expression\n")
}
```

```
High expression
```

```
# 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"
```

2.6 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")
```

In-class Tasks:

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.

2.7 Summary of Today's Lesson

In today's class, 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. We will know more about conditionals, R packages to handle data and visualization in a better and efficient way.

2.8 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. **Complete Documentation**:
    - Write all code in R Markdown
    - Include comments explaining your approach
    - Push to GitHub

2.8.0.1 Due date: Friday 10pm BD Time