R Essential Functions

Table of contents

1 Miscellaneous Functions

1.1 `unique()`

- The `unique()` function removes duplicated elements from a vector or data frame.
- Example: `unique()`

```r
x = c(1, 2, 2, 3, 4, 4, 5)
unique(x)
```

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

1.2 `any()` and `all()`

- `any()` returns `TRUE` if any of the values are `TRUE`.

```r
v = c(FALSE, FALSE, TRUE)
any(v)
```

```
[1] TRUE
```

- `all()` returns `TRUE` if all of the values are `TRUE`.

```r
all(v)
```

```
[1] FALSE
```

1.3 `ifelse()`

- The `ifelse` function applies a function to elements of a vector depending on a condition.
- Example:

```r
numbers = 1:10
numbers
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
ifelse(numbers %% 2 == 0, "Even", "Odd")
```

```
 [1] "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even"
```

1.4 `cbind()` and `rbind()`

- `cbind()` combines vectors, matrices, or data frames by columns.
- `rbind()` combines vectors, matrices, or data frames by rows.

```r
A = matrix(1:4, ncol=2)
A
```

1	3
2	4

```r
B = matrix(5:8, ncol=2)
B
```

5	7
6	8

```r
cbind(A, B)
```

1	3	5	7
2	4	6	8

```r
rbind(A, B)
```

1	3
2	4
5	7
6	8

2 Set Functions

- Set is a collection of distinct elements.
- Set functions perform operations on sets of elements.

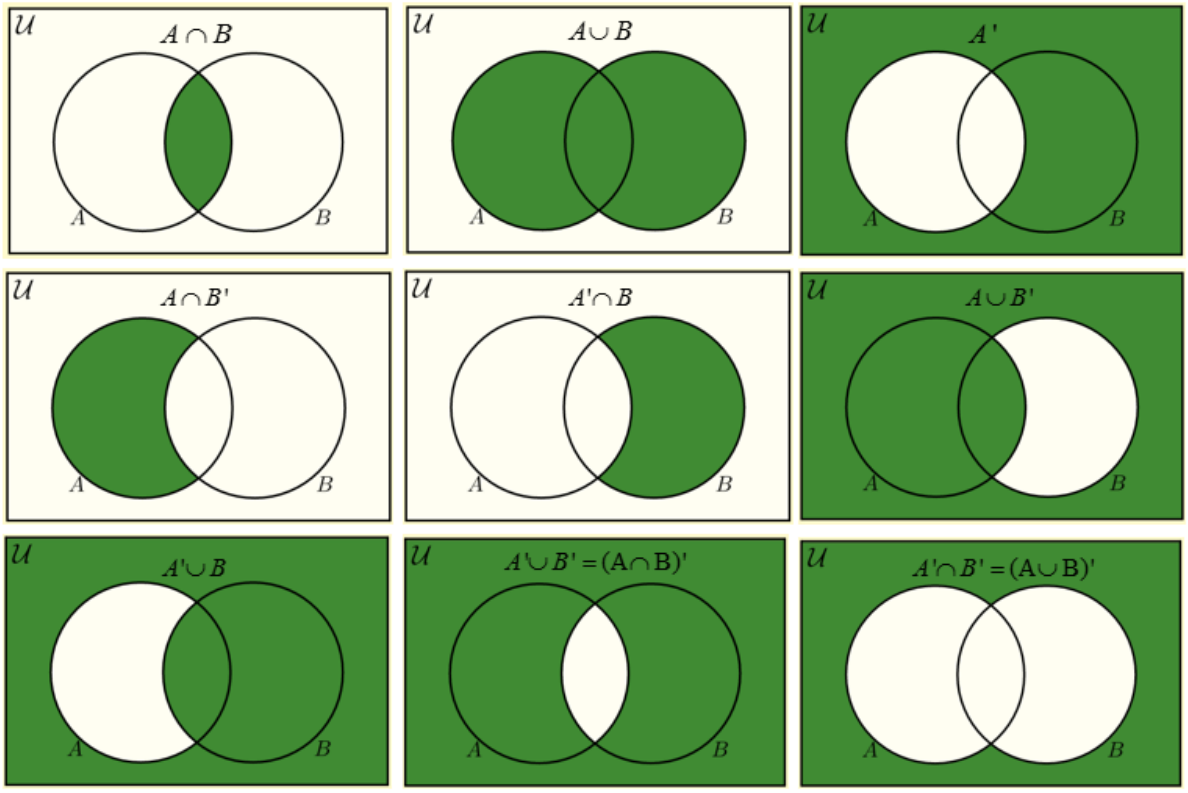


Figure 1: Set Theory

2.1 Union ($A \cup B$)

- The set of all elements in A, or in B, or in both.
- $A \cup B = \{x | x \in A \text{ or } x \in B\}$
- `union()`
- Example:

```
A = c(1, 2, 3, 4)
B = c(3, 4, 5, 6)
union(A, B)
```

```
[1] 1 2 3 4 5 6
```

2.2 Intersection ($A \cap B$)

- The set of all elements that are both in A and B.
- $A \cap B = \{x | x \in A \text{ and } x \in B\}$
- `intersect()`
- Example:

```
A = c(1, 2, 3, 4)
B = c(3, 4, 5, 6)
intersect(A, B)
```

```
[1] 3 4
```

2.3 Set Difference ($A - B$)

- The set of all elements that are in A but not in B.
- $A - B = \{x | x \in A \text{ and } x \notin B\}$
- `setdiff()`
- Example:

```
A = c(1, 2, 3, 4)
B = c(3, 4, 5, 6)
setdiff(A, B)
```

```
[1] 1 2
```

2.4 Subset ($A \subseteq B$)

- A is a subset of B if every element of A is also an element of B.
- $A \subseteq B \iff (\forall x)(x \in A \implies x \in B)$
- Example:

```r
A = c(1, 2, 3, 4)
all(A %in% c(1, 2, 3, 4, 5))
```

```
[1] TRUE
```

```r
all(A %in% c(1, 2, 3))
```

```
[1] FALSE
```

2.5 Set Equality ($A = B$)

- Two sets are equal if they have exactly the same elements.
- $A = B \iff (A \subseteq B)$ and $(B \subseteq A)$
- `setequal()`
- Example:

```r
A = c(1, 2, 3, 4)
B = c(3, 4, 5, 6)
setequal(A, B)
```

```
[1] FALSE
```

3 Random Sampling

Simple Random Sample (SRS): is a subset of a population, chosen in such a way that every possible sample of a given size has an equal chance of being selected. This method ensures that each individual or item within the population has an equal probability of being included in the sample, and the selection process is entirely by chance, without any bias.

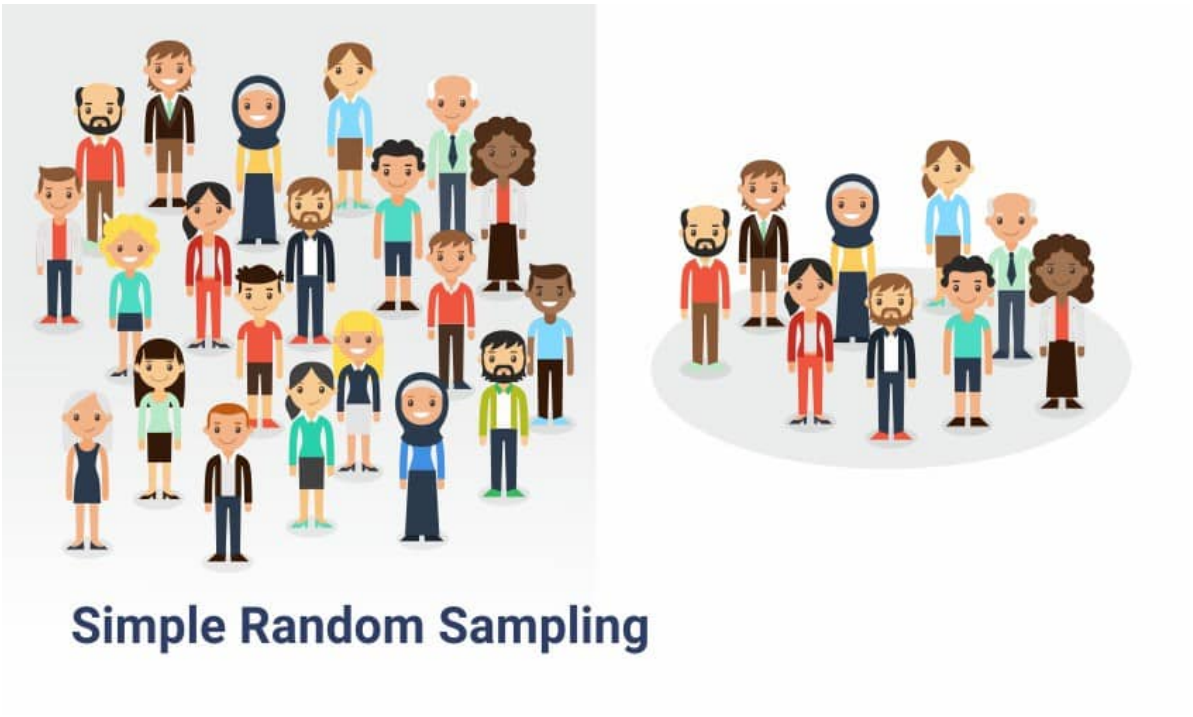


Figure 2: Random Sampling

3.1 With *vs.* Without Replacement

Sampling With Replacement (SWR): In this method, after an individual or item is selected for the sample, it is placed back into the population before the next selection is made, allowing for the possibility of being chosen more than once. This method is particularly useful when dealing with small population sizes or when it's important to maintain the same population size for each draw.

Sampling Without Replacement (SWOR): Contrary to SWR, in Sampling Without Replacement, once an individual or item is selected, it is not placed back into the population, and hence, cannot be selected again. This method is often utilized when the population size is large, or when maintaining the same population size for each draw is not crucial.

3.2 The `sample()` function

- The `sample()` function draws random samples from a vector.
- Syntax:

```r
sample(x, size, replace = FALSE, prob = NULL)
```

- Example:

```r
sample(1:10, 5)
```

```
[1] 2 7 6 4 9
```

3.3 Example: Coin Flip Simulation

3.3.1 Part 1: Single Simulation

Write an R function `coin_flip()` that simulates flipping a coin. The function should return `H` (for head) or `T` (for tail).

```r
coin_flip = function() {
  flip = sample(c("H", "T"), size = 1)
  return (flip)
}

coin_flip()
```

```
[1] "H"
```

3.3.2 Part 2: Multiple Simulations

Now, extend your function to perform multiple simulations of coin flips and return the number of heads and tails.

```r
coin_flip = function(n) {
  flips = sample(c("H", "T"), size = n, replace = TRUE)
  return(table(flips))
}

coin_flip(5)
```

H	T
3	2

3.3.3 Part 3: Analysis

Analyze the results of your multiple simulations. What do you observe as the number of flips increases?

```
coin_flip(10)/10
```

H	T
0.4	0.6

```
coin_flip(100)/100
```

H	T
0.5	0.5

```
coin_flip(1000)/1000
```

H	T
0.519	0.481

```
coin_flip(10000)/10000
```

H	T
0.5012	0.4988

4 The Apply Functions

The apply functions in R provide a concise and efficient way to apply a **function** to the elements of data structures such as vectors, lists, data frames, or matrix.

4.1 Comaprsion between the Apply Functions

Apply functions provide a concise way to apply a **function** to data.

Function	Description	Usage	Example
`apply()`	Applies a function over the margins of an array or matrix.	`apply(X, MARGIN, FUN, ...)`	`apply(matrix(1:9, nrow = 3), 1, sum)`
`lapply()`	Applies a function to each element of a list, returning a list.	`lapply(X, FUN, ...)`	`lapply(list(1:5, 6:10), sum)`
`sapply()`	Similar to `lapply()`, but tries to simplify the result.	`sapply(X, FUN, ..., simplify = TRUE)`	`sapply(list(1:5, 6:10), sum)`

4.2 Example: Calculating summary statistics

4.2.1 Creating a numeric *named* `list`

```
numeric_list = list(a = 1:5, b = 3:7, c = 10:14)
numeric_list
```

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

$b
[1] 3 4 5 6 7

$c
[1] 10 11 12 13 14
```

4.2.2 Applying `mean()` using `lapply()`

```
lapply(numeric_list, mean)
```

```
$a
[1] 3

$b
[1] 5

$c
[1] 12
```

4.2.3 Applying `sum()` using `sapply()`

```r
sapply(numeric_list, sum)
```

```
 a  b  c
15 25 60
```

5 The `sweep()` Function

The `sweep()` function in R allows you to perform operations on arrays by "sweeping" out values of a summary statistic across margins.

5.1 Syntax of `sweep()`

```
sweep(x, MARGIN, STATS, FUN = "-", ...)
```

- `x`: the array to sweep out statistics from.
- `MARGIN`: the margin to apply the sweep on.
- `STATS`: the summary statistic to be used.
- `FUN`: the function to apply.

5.2 Example: Centering

5.2.1 Sample matrix

```
mat = matrix(1:9, nrow = 3)
mat
```

1	4	7
2	5	8
3	6	9

5.2.2 Calculate column means

```
col_means = apply(mat, 2, mean)
col_means
```

```
[1] 2 5 8
```

5.2.3 Center the matrix by subtracting column means

```
centered_mat = sweep(mat, 2, col_means)
centered_mat
```

-1	-1	-1
0	0	0
1	1	1

5.3 Example: Scaling

5.3.1 Calculate max for each column

```
col_maxs = apply(mat, 2, max)
col_maxs
```

```
[1] 3 6 9
```

5.3.2 Scale the matrix by dividing by column maxs

```
scaled_mat = sweep(mat, 2, col_maxs, FUN = "/")
scaled_mat
```

0.3333333	0.6666667	0.7777778
0.6666667	0.8333333	0.8888889
1.0000000	1.0000000	1.0000000

6 The Z-scores

The Z-score of an observation is a metric that indicates how many standard deviations an element is from the mean of the whole set.

$z = \frac{x-\mu}{\sigma}$

where:

- x is the raw score,
- μ is the mean of the population, and
- σ is the standard deviation of the population.

Note: The Z-score is **unitless** i.e., having no units of measurement

6.1 Steps to Calculate Z-scores

1. Calculate means and standard deviations for each column.
2. Center data by subtracting mean values using `sweep()`.
3. Divide by standard deviation to standardize using `sweep()`.

6.2 Example: Z-score Calculation

6.2.1 Create a sample matrix with random data

```
data_matrix = matrix(rnorm(100), ncol=10)
data_matrix
```

0.7427667 1.2792899 - - 1.0033053 1.6909852 - - - -
0.3972483 1.4102493 1.4804755 1.3073921 1.6217032 0.1852155
- 1.3440336 0.6992811 0.1651069 - - 1.0150338 0.2403331 - -
0.5840396 0.0336841 0.7146305 0.4598288 0.4911908
0.4494523 0.3546111 0.2559209 - 0.5644226 - - 0.5426797 0.0196132 -
0.8997363 0.4735368 0.4738774 0.6225893
0.9801887 0.8319361 - - - - - - 0.2977715 0.4494763
0.7394885 0.1062452 0.4504511 0.8845715 0.9356131 0.9341220
0.3612463 0.7993764 - - 0.4429866 0.2446596 0.3842856 - 0.9344708 0.1658653
1.7879747 0.4703643 0.7743875
- - 0.6427011 - - 0.1607766 - 0.1593613 0.6410808 -
0.7430468 0.1051577 1.3034051 1.6665589 0.0344166 0.4082167
- - - - 0.4717387 0.2872116 - 0.3111099 0.4885885 0.3835364
0.5450866 0.1806004 0.1937679 0.4065021 0.0019967
- 1.0921123 - - - - 0.5524030 - 0.8636177 1.4845153
1.1299224 2.1901255 0.4027841 1.7143056 0.6274684 0.9941545
0.0721744 - - - 1.0377706 - 0.2930979 - 0.0265363 -
0.1894894 0.5883799 0.3006362 0.7158386 2.1456503 1.9899200
- 0.1813360 0.1144744 0.2905148 0.5734126 0.8745530 - - - -
0.4180377 0.9769801 2.1016062 0.2492819 1.2804782

6.2.2 Calculate column means

```
col_means = apply(data_matrix, 2, mean)
col_means
```

```
[1] -0.08143046  0.54074478 -0.41846074 -0.48443009  0.02286367 -0.01578597
[7] -0.16585391 -0.70038286  0.09408649 -0.24942172
```

6.2.3 Calculate column standard deviations

```
col_sds = apply(data_matrix, 2, sd)
col_means
```

```
[1] -0.08143046  0.54074478 -0.41846074 -0.48443009  0.02286367 -0.01578597
[7] -0.16585391 -0.70038286  0.09408649 -0.24942172
```

6.2.4 Center the matrix by subtracting column means

```
data_matrix_centered = sweep(data_matrix, 2, col_means, FUN = "-")
data_matrix_centered
```

0.8241972 0.7385451 0.0212124 - 0.9804416 1.706771 - - - 0.0642062
0.9258192 1.3146216 0.6070092 1.715789
- 0.8032888 1.1177418 0.6495370 - - 1.1808877 0.940716 - -
0.5026092 0.0565477 0.6988445 0.5539153 0.241769
0.5308828 - 0.6743816 - 0.5415589 - - 1.2430626 - -
0.1861337 0.4153062 0.4577508 0.3080235 0.0744733 0.3731676
1.0616191 0.2911913 - 0.3781849 - - - - 0.2036850 0.6988980
0.3210278 0.4733148 0.8687856 0.7697592 0.233739
0.4426768 0.2586317 - 0.0140658 0.4201230 0.2604455 0.5501395 - 0.8403843 0.415287
1.3695139 0.074004
- - 1.0611619 - - 0.1765626 0.1314373 0.8597442 0.546994 -
0.6616163 0.6459025 0.8189750 1.6894226 0.158795
- - 0.2246928 0.0779280 0.4488750 0.3029976 0.1638572 1.0114928 0.3945020 0.6329581
0.4636562 0.721345
- 0.5513675 - 0.0816460 - - 0.7182570 - 0.7695312 1.733937
1.0484919 1.7716648 1.7371693 0.6116824 0.293771
0.1536048 - - 0.1837938 1.0149070 - 0.4589519 - - -
0.7302342 0.1699191 0.7000526 1.4452675 0.0675502 1.740498
- - 0.5329352 0.7749449 0.5505490 0.890339 - - - -
0.3366072 0.3594088 0.8111262 1.4012233 0.3433684 1.031056

6.2.5 Divide by the standard deviation to get z-scores

```
z_scores = sweep(data_matrix_centered, 2, col_sds, FUN = "/")
z_scores
```

1.1753344 1.2231931 0.0220736 - 0.9769302 2.054128 - - - 0.0665090
1.6274702 1.6753808 0.6152849 2.263041
- 1.3304229 1.1631192 1.1418018 - - 1.5049476 0.953541 - -
0.7167385 0.0563452 0.8410714 0.7305867 0.250440
0.7570577 - 0.7017597 - 0.5396193 - - 1.2600098 - -
0.3082784 0.7300545 0.5509110 0.3925515 0.0982266 0.3865517
1.5139066 0.4822769 - 0.6648000 - - - - 0.2686505 0.7239648
0.3340607 0.4716196 1.0455985 0.9809970 0.236925

0.6312728	0.4283509	-1.4251127	0.0247259	0.4186183	0.3134507	0.7011091	-0.0750136	1.1084252	0.4301818
-0.9434883	-1.0697566	1.1042423	-1.4396520	-1.6833720	0.2124961	0.1675065	0.8714655	0.7214584	-0.1644904
-0.6611902	-1.1947063	0.2338148	0.1369873	0.4472674	0.3646629	0.2088230	1.0252829	0.5203285	0.6556599
-1.4951867	0.9131859	-1.8435898	0.1435231	-1.7309477	-0.7361703	0.9153614	-0.2977768	1.0149735	1.7961267
0.2190459	-1.2094284	-0.1768174	0.3230858	1.0112721	-0.8425254	0.5848977	-1.4649715	-0.0890954	-1.8029232
-0.4800138	-0.5952600	0.5545709	1.3622528	0.5485772	1.0715384	-1.0337160	-1.4203269	-0.4528859	-1.0680366

6.3 Example: Data Frame Normalization Using `sweep()`

Normalize a data frame `df` (with columns `X1`, `X2`, `X3` each containing 10 random integers between 1 and 100) by subtracting the median and dividing by the interquartile range of each column.

6.3.1 Creating a data frame

```
set.seed(123)
df = data.frame(X1 = sample(1:100, 10),
                X2 = sample(1:100, 10),
                X3 = sample(1:100, 10))

df
```

X1	X2	X3
31	90	7
79	91	42
51	69	9
14	99	83
67	57	36
42	92	78
50	9	81
43	93	43
97	72	76

X1	X2	X3
25	26	15

6.3.2 Calculating medians

```
medians = sapply(df, median) # or apply(df, 2, median)
medians
```

```
  X1   X2   X3
46.5 81.0 42.5
```

6.3.3 Calculating interquartile ranges

```
iqrs = sapply(df, IQR) # or apply(df, 2, IQR)
iqrs
```

```
   X1    X2    X3
29.25 31.75 57.25
```

6.3.4 Normalizing the data frame

```
normalized_df = sweep(sweep(df, 2, medians, "-"), 2, iqrs, "/")
normalized_df
```

X1	X2	X3
-0.5299145	0.2834646	-0.6200873
1.1111111	0.3149606	-0.0087336
0.1538462	-0.3779528	-0.5851528
-1.1111111	0.5669291	0.7074236
0.7008547	-0.7559055	-0.1135371
-0.1538462	0.3464567	0.6200873
0.1196581	-2.2677165	0.6724891
-0.1196581	0.3779528	0.0087336
1.7264957	-0.2834646	0.5851528

X1	X2	X3
-0.7350427	-1.7322835	-0.4803493

6.4 Example: Generating and Analyzing Height Data

Generate a dataset that simulates the heights (in centimeters) of 1000 individuals. Assume an average height of 170 cm and a standard deviation of 10 cm. Follow the following steps:

- Create the dataset using a normal distribution with the given mean and standard deviation using the `rnorm()` function.
- Calculate the Z-scores for the entire dataset.
- Determine how many standard deviations away from the mean is a height of 185 cm.

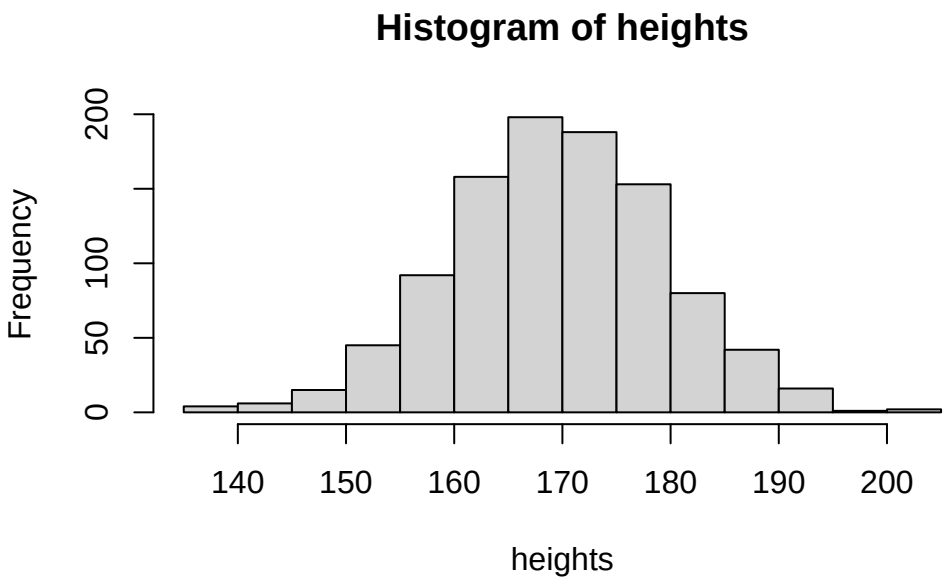
6.4.1 Generating the height dataset

```
heights = rnorm (1000, 170, 10)
head(heights)
```

```
[1] 167.8413 166.6509 159.1430 169.1458 180.7061 168.5461
```

6.4.2 Displaying the distribution of heights

```
hist(heights)
```



6.4.3 Calculating the Z-scores

```
zscores = (heights - mean(heights))/sd(heights)
head(zscores)
```

```
[1] -0.17243735 -0.29236253 -1.04868498 -0.04103338  1.12352467 -0.10144587
```

6.4.4 Displaying the distribution of Z-scores

```
hist(zscores)
```

6.4.5 Calculating the Z-score of a specific height

```
zscore = (185 - mean(heights))/sd(heights)
zscore
```

```
[1] 1.55608
```

6.4.6 Displaying the Z-score of a specific height

```
hist(zscores)
abline(v=zscore, col="red", lwd=2)
```

7 Comparison between `apply()` and `sweep()`

Feature	`apply()`	`sweep()`
Purpose	Apply a function over the margins of an array or matrix to summarize or transform it.	Apply arithmetic operations to an array "sweeping" out array summaries.
Usage	Used for summarizing data with a function over specified margins (rows or columns)	Used for adjusting data using a summary statistic for operations like centering or scaling.
Functionality	Used to apply a wide range of functions for summarizing or transforming data across dimensions	Used to perform arithmetic operations using a summary statistic and is often used after summarizing data with apply().
Arguments	`apply(X, MARGIN, FUN, ...)` where `X` is the array, MARGIN specifies rows(1) or columns(2), and FUN is the function to be applied.	`sweep(x, MARGIN, STATS, FUN = "-", ...)` where `x` is the array, `MARGIN` specifies the dimension, STATS is the summary statistic, and `FUN` is the arithmetic function to be applied.

Feature	`apply()`	`sweep()`
Return Value	Returns an array, matrix, or list with the results of the function application, which may be of a different dimension from the input.	Returns an adjusted array with the same dimensions as the input, with element-wise arithmetic operations performed.
Exclusive Actions	- Can return different structures (vector, array, list) based on the function and margin.- Can work with higher-dimensional arrays beyond matrices.	- Directly performs arithmetic *sweep* operations using a summary statistic.- Ideal for data adjustments after using `apply()` to calculate the summary statistic.
Limitations	- Cannot directly adjust data using a summary statistic; additional steps are required to integrate the summary before or after using `apply()`.	- Not designed for summarizing data; it requires pre-calculated statistics to perform the sweep operation.
Flexibility	- Can use any function, including user-defined ones, for summarization or transformation.- More general-purpose in data manipulation.	- Limited to arithmetic *sweep* operations; custom functions must conform to the expected input and output format of `sweep()`.
Common Use Case	- Computing aggregate statistics like means, sums, etc., across rows or columns.- General data manipulation tasks requiring the application of a function.	- Standardizing or normalizing data.- Centering data by subtracting the mean or dividing by a standard deviation after calculating these with `apply()`.

8 Testing Data Types

- R provides functions to test the data type of a variable.

8.1 Examples:

```r
is.character("Hello")
```

```
[1] TRUE
```

```r
is.numeric(10)
```

```
[1] TRUE
```

```r
is.na(NA)
```

```
[1] TRUE
```

```r
is.vector(c(1, 2, 3))
```

```
[1] TRUE
```

```r
is.matrix(matrix(1:4, ncol=2))
```

```
[1] TRUE
```

```r
is.data.frame(data.frame(x=1:4, y=4:1))
```

```
[1] TRUE
```

```r
is.factor(factor(c("a", "b", "a")))
```

```
[1] TRUE
```