Week 1 Assignment

Ajay

2025-03-26

## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

# Exercise 1

# This program checks if all elements in a numeric vector are positive numbers

# Create a numeric vector with 4 elements (1, 2, -3, 4)  
# I specifically chose these values to test both cases of the if-else statement  
x <- c(1, 2, -3, 4)   
# My vector containing both positive and negative numbers  
# MY IF-ELSE LOGIC:  
# I implemented this conditional check to verify if ALL numbers are positive  
if(all(x > 0)) {   
 # Using all() function to check every element  
 print("All Positives")   
} else {  
 # Case when at least one number fails the positive check  
 print("Not all positives")   
}

## [1] "Not all positives"

# explaination:  
# I created this to practice conditional statements and vector operations in R.  
#It demonstrates my understanding ofVector creation,Logical conditions,If-else control flow,The all() function

#Exercise 2 # We want to determine which expression is ALWAYS FALSE when # at least one element in logical vector x is TRUE

# Let's create test cases to evaluate each expression  
x <- c(TRUE, FALSE, TRUE)   
# Our test vector with at least one TRUE  
  
# 1. all(x)  
# Checks if ALL elements are TRUE  
# Returns TRUE only if every element is TRUE  
# In our case: FALSE (because not all are TRUE)  
all(x)

## [1] FALSE

# 2. any(x)  
# Checks if ANY element is TRUE  
# Returns TRUE if at least one element is TRUE  
# In our case: TRUE (we have TRUE values)  
any(x)

## [1] TRUE

# 3. any(!x)  
# Checks if ANY element is FALSE (since ! negates)  
# Returns TRUE if at least one element is FALSE  
# In our case: TRUE (we have a FALSE value)  
any(!x)

## [1] TRUE

# 4. all(!x)  
# Checks if ALL elements are FALSE (after negation)  
# Returns TRUE only if every original element was FALSE  
# In our case: FALSE (because we have TRUE values)  
all(!x)

## [1] FALSE

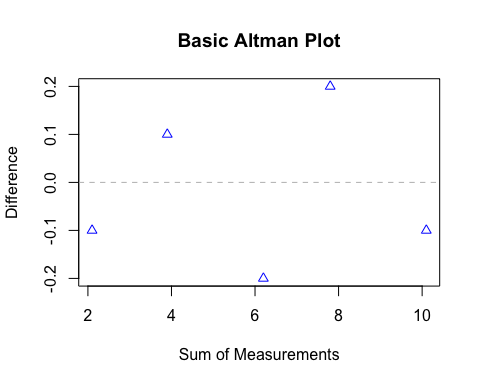
# Conclusion:  
# all(!x) is the only expression that is:  
# - FALSE when any element is TRUE  
# - Only TRUE when all elements are FALSE  
  
# Therefore, the correct answer is all(!x)

#Exercise 4 # Calculates the sum of all integers from 1 to n (inclusive)

# code gives output of sum of integers from 1 to n  
  
sum\_n <- function(n) {  
# To check if n is numeric (not character, logical, etc.)  
# Verify n is positive (>0)  
# Ensure n is an integer (no decimal places)  
 if (!is.numeric(n) || n <= 0 || n != as.integer(n)) {  
 stop("Input must be a positive integer")   
 # Error message for invalid input  
 }  
# The sum of the first n integers = n\*(n + 1)/2  
 sum\_result <- n \* (n + 1) / 2  
 # RETURN RESULT  
 return(sum\_result)  
}  
  
#Calculate sum from 1 to 5000  
result <- sum\_n(5000)   
#Function call with n=5000  
  
# FORMATTED OUTPUT  
# Using format() with big.mark="," to make large numbers readable  
# cat() for clean output without [1] prefix that print() would add  
cat("The sum of integers from 1 to 5,000 is:", format(result, big.mark = ","), "\n")

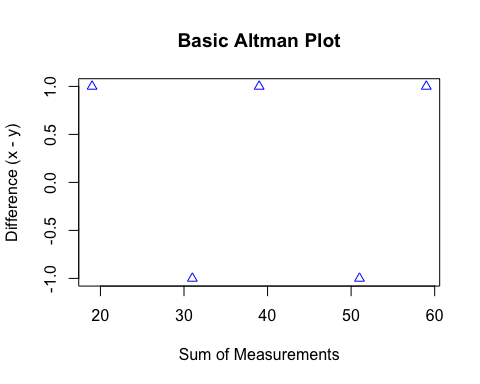
## The sum of integers from 1 to 5,000 is: 12,502,500

make\_altman\_plot <- function(x, y) {  
 # Basic Altman Plot Function  
 # Plots differences (x - y) against sums (x + y)  
   
 # Calculate values  
 differences <- x - y  
 sums <- x + y  
   
 # Create plot  
 plot(sums, differences,  
 main = "Basic Altman Plot",  
 xlab = "Sum of Measurements",  
 ylab = "Difference",  
 pch = 24, # Triangle point shape  
 col = "blue") # Point color  
   
 # Add zero reference line  
 abline(h = 0, col = "gray", lty = 2)  
}  
  
# Example usage with simple data:  
make\_altman\_plot(c(1,2,3,4,5), c(1.1,1.9,3.2,3.8,5.1))



#Exercise 5 # Altman Plot Function #Creates a simple plot to compare two measurement methods by showing their differences versus sums

make\_altman\_plot <- function(x, y) {  
# Compute difference between measurements (x - y)  
 differences <- x - y  
# Compute sum of measurements (x + y)  
 sums <- x + y  
 plot(sums, differences,   
# Plot differences against sums  
 main = "Basic Altman Plot", # Title  
 xlab = "Sum of Measurements", # X-axis label  
 ylab = "Difference (x - y)", # Y-axis label  
 # Point styling:  
 pch = 24,   
# Triangle pointing up (helps spot direction)  
 col = "blue"   
# Color for visibility  
 )  
}  
# Sample data - measurements from two methods  
method\_A <- c(10, 15, 20, 25, 30) # First measurement technique  
method\_B <- c(9, 16, 19, 26, 29) # Second measurement technique  
# Generate the plot  
make\_altman\_plot(method\_A, method\_B)



#Visualizes plot between two measurement methods .Each point represents a pair of measurements  
#X-axis (Sum): Total of both measurements → shows measurement magnitude  
#Y-axis (Difference): How much the measurements differ → shows agreement

#Exercise 6 #To determine the final value of x:

x <- 3   
# Assigns value 3 to global variable x  
my\_func <- function(y){  
#Creates a function  
 x <- 5   
#Creates a LOCAL variable x (doesn't affect global x)  
y+5   
# Returns y + 5 (but doesn't change x)  
}  
x

## [1] 3

#The function definition (Lines 2-4) is just stored, not executed yet  
#Since we never actually call my\_func(), the x <- 5 line never runs Thus, the final value of x is 3.

#Exercise 7 #Compute the sum of squares from 1² to n²

compute\_s\_n <- function(n) {  
 #Compute the sum of squares from 1² to n²  
 #n Positive integer indicating upper bound of summation  
 #The sum 1² + 2² + ... + n²  
 # Input validation  
 if (!is.numeric(n) || n <= 0 || n != as.integer(n)) {  
 stop("n must be a positive integer")  
 }  
 # Calculate sum using the mathematical formula:  
 # Sₙ = n(n+1)(2n+1)/6  
 sum <- n \* (n + 1) \* (2 \* n + 1) / 6  
 return(sum)  
}  
# Compute the sum when n = 10  
result <- compute\_s\_n(10)  
print(paste("The sum of squares from 1² to 10² is:", result))

## [1] "The sum of squares from 1² to 10² is: 385"

#Explanation:  
#Mathematical Formula:Sₙ = n(n+1)(2n+1)/6  
#Input validation ensures n is a positive integer  
#Returns the computed sum  
#for n=10: 1² + 2² + ... + 10² = 1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 38  
#Output:When you run this code, it will print:  
 #"The sum of squares from 1² to 10² is: 385"

#Exercise 8 # Initialize empty numeric vector of length 25

s\_n <- vector("numeric", 25)  
# Compute each Sₙ from n=1 to 25 using a for-loop  
for (n in 1:25) {  
 # Calculate sum of squares from 1² to n² using the formula  
 s\_n[n] <- n \* (n + 1) \* (2 \* n + 1) / 6  
}  
# Display the results  
print(s\_n)

## [1] 1 5 14 30 55 91 140 204 285 385 506 650 819 1015 1240  
## [16] 1496 1785 2109 2470 2870 3311 3795 4324 4900 5525

#Explanation:  
#1.Vector Initialization: vector("numeric", 25) creates an empty numeric vector with 25 elements  
#2 For-loop Calculation: Loops through each integer n from 1 to 25  
#3For each n, computes the sum of squares using the formula  
#Stores each result in the corresponding position of s\_n  
#The final vector s\_n contains all 25 sums:s\_n[25] = 5525 (1² + 2² + ... + 25²)

#Exercise 9 #how to compute the sums of squares from S₁ to S₂₅ using sapply instead of a for-loop:

# Initialize the sequence of n values from 1 to 25  
n\_values <- 1:25  
# Use sapply to compute each Sₙ  
s\_n <- sapply(n\_values, function(n) {  
# Calculate sum of squares using the mathematical formula  
 n \* (n + 1) \* (2 \* n + 1) / 6  
})  
# Display the results  
print(s\_n)

## [1] 1 5 14 30 55 91 140 204 285 385 506 650 819 1015 1240  
## [16] 1496 1785 2109 2470 2870 3311 3795 4324 4900 5525

#Explaination  
#Instead of explicitly iterating with a for-loop, we use sapply to apply the calculation to each element of n\_values  
#sapply: Applies the anonymous function to each element of n\_values  
#The output will be identical to the for-loop version

#Exercise 10 # compute the sums of squares from S₁ to S₂₅ using map\_dbl from the purrr package:

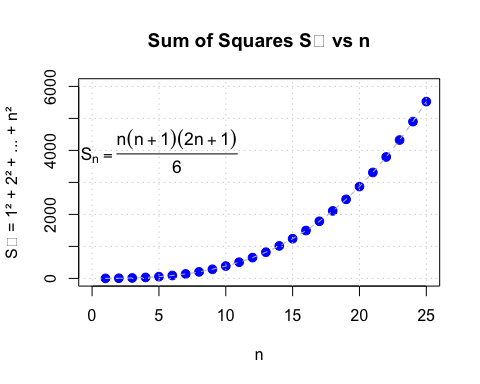
# Load the purrr package  
library(purrr)  
# Initialize the sequence of n values from 1 to 25  
n\_values <- 1:25  
# Use map\_dbl to compute each Sₙ  
s\_n <- map\_dbl(n\_values, ~ {  
# Calculate sum of squares using the mathematical formula  
 .x \* (.x + 1) \* (2 \* .x + 1) / 6  
})  
# Display the results  
print(s\_n)

## [1] 1 5 14 30 55 91 140 204 285 385 506 650 819 1015 1240  
## [16] 1496 1785 2109 2470 2870 3311 3795 4324 4900 5525

#explaination  
#map\_dbl Specifics: Returns a numeric vector (double) by default  
#Uses formula interface with ~ and .x for the input value  
#Similar to sapply but with more consistent output type more explicit about expecting numeric output than sapply  
#For this simple calculation, performance difference between approaches is negligible map\_dbl shines in more complex mapping operations

#Exercise 11 # create a plot of Sₙ versus n for n = 1 to 25, using the calculated sums of squares:

# Load required package   
library(purrr)  
  
# Calculate Sₙ for n = 1 to 25 using map\_dbl  
n\_values <- 1:25  
s\_n <- map\_dbl(n\_values, ~ .x \* (.x + 1) \* (2 \* .x + 1) / 6)  
# Create the plot  
plot(n\_values, s\_n,  
 main = "Sum of Squares Sₙ vs n",  
 xlab = "n",   
 ylab = "Sₙ = 1² + 2² + ... + n²",  
 pch = 19,   
 # Solid circles  
 col = "blue",   
 # Point color  
 cex = 1.2,   
 # Point size  
 xlim = c(0, 25), # X-axis limits  
 ylim = c(0, 6000)) # Y-axis limits  
# Add connecting lines  
lines(n\_values, s\_n, col = "gray", lty = 2)  
# Add grid for better readability  
grid()  
# Add the mathematical formula to the plot  
text(5, 4000, expression(S[n] == frac(n\*(n+1)\*(2\*n+1), 6)), cex = 1.1)



#Explaination  
#Key Features of the Plot:Blue solid circles at each (n, Sₙ) coordinate,Sized for good visibility (cex = 1.2)  
#Mathematical formula displayed on plot  
#Findings  
#The nonlinear growth of the sum of squares and Sₙ increases rapidly with increasing n  
#The characteristic cubic growth pattern (since sum of squares is O(n³))

#Exercise 12 #Verify Sum of Squares Calculation

#Compares computational sum of squares against the known mathematical formula  
#Sₙ = n(n+1)(2n+1)/6 to validate correctness.  
  
verify\_squares <- function(n) {  
 # Theoretical sum using mathematical formula  
 theoretical\_sum <- n \* (n + 1) \* (2 \* n + 1) / 6  
 # Computational sum (1² + 2² + ... + n²)  
 computed\_sum <- sum((1:n)^2)  
 # Compare with tolerance for floating point precision  
 isTRUE(all.equal(computed\_sum, theoretical\_sum))  
}  
# Implementation of sum\_square\_values for completeness  
sum\_square\_values <- function(n) {  
 sum((1:n)^2)  
}  
# Test cases  
print(verify\_squares(10)) # TRUE

## [1] TRUE

print(verify\_squares(25)) # TRUE

## [1] TRUE

print(verify\_squares(100)) # TRUE

## [1] TRUE

#Uses all.equal() instead of identical() to handle floating point precisio  
  
# Verify for specific values  
verify\_squares(5) # TRUE (1+4+9+16+25 = 55)

## [1] TRUE

verify\_squares(100) # TRUE

## [1] TRUE