The conditional expression will return "Not all positives".

Here’s the reasoning:

* The vector x <- c(1, 2, -3, 4) contains the values 1, 2, -3, and 4.
* The condition all(x > 0) checks if all elements in the vector x are greater than 0.
* Since the third element -3 is not greater than 0, the condition all(x > 0) evaluates to FALSE.
* Therefore, the else block will be executed, printing "Not all positives".

Given that at least one entry of a logical vector x is TRUE, let’s analyze each expression:

1. **all(x)**: This expression checks if **all** elements in x are TRUE. Since we know only that at least one entry is TRUE, the rest could be FALSE. Therefore, all(x) could be either TRUE or FALSE, depending on the other entries. **Not always FALSE.**
2. **any(x)**: This expression checks if **any** element in x is TRUE. Given that at least one entry of x is TRUE, any(x) will be TRUE. **Not FALSE.**
3. **any(!x)**: This expression checks if **any** element in x is FALSE (since !x inverts the logical values). If at least one entry is TRUE, it is possible that other entries could be FALSE, making any(!x) potentially TRUE. **Not always FALSE.**
4. **all(!x)**: This expression checks if **all** elements in x are FALSE (since !x inverts the logical values). Since we know that at least one entry of x is TRUE, all(!x) cannot be TRUE because not all elements are FALSE. This expression is therefore **always FALSE**.

**Conclusion: all(!x) is the expression that is always FALSE when at least one entry of a logical vector x is TRUE.**

You can use the nchar function along with logical subsetting to assign the state abbreviations of state names longer than 8 characters to the object new\_names. Here’s a line of code that accomplishes this:

new\_names <- state.abb[nchar(state.name) > 8]

### Explanation:

* state.name: A built-in vector in R that contains the full names of all 50 U.S. states.
* state.abb: A built-in vector that contains the corresponding state abbreviations.
* nchar(state.name) > 8: This checks which state names have more than 8 characters.
* The logical condition nchar(state.name) > 8 is used to subset state.abb so that only the abbreviations corresponding to state names longer than 8 characters are assigned to new\_names.

Here’s how you can create the sum\_n function in R, which computes the sum of integers from 1 to a given value n. Then, you can use this function to determine the sum of integers from 1 to 5,000.

sum\_n <- function(n) {  
 sum(1:n)  
}  
  
# Determine the sum of integers from 1 to 5,000  
result <- sum\_n(5000)  
result

### Explanation:

* The function sum\_n takes a single argument n.
* sum(1:n) calculates the sum of the sequence of integers from 1 to n.
* You then call sum\_n(5000) to compute the sum of integers from 1 to 5,000.

### Result:

Running the code above will give you the sum of integers from 1 to 5,000, which is 12502500.

Here’s how you can create a function altman\_plot in R that takes two arguments, x and y, and plots the difference (x - y) against the sum (x + y):

altman\_plot <- function(x, y) {  
 # Calculate the difference and sum  
 diff <- x - y  
 sum <- x + y  
   
 # Create the plot  
 plot(sum, diff,   
 xlab = "Sum of x and y",   
 ylab = "Difference of x and y",   
 main = "Altman Plot",  
 pch = 19,   
 col = "blue")  
}  
  
# Example usage  
# altman\_plot(c(1, 2, 3, 4, 5), c(2, 2, 2, 2, 2))

### Explanation:

* **diff <- x - y**: Computes the difference between the two vectors.
* **sum <- x + y**: Computes the sum of the two vectors.
* **plot(sum, diff, ...)**: Creates a scatter plot with the sum on the x-axis and the difference on the y-axis. The plot is labeled with "Sum of x and y" on the x-axis, "Difference of x and y" on the y-axis, and "Altman Plot" as the title.
* **pch = 19, col = "blue"**: These arguments specify the plotting character (solid circle) and the color of the points (blue).

### Example:

If you were to use altman\_plot(c(1, 2, 3, 4, 5), c(2, 2, 2, 2, 2)), it would generate an Altman plot for these two vectors.

This function will help you visualize the relationship between the sum and difference of two numeric vectors, which is useful in various contexts, such as method comparison studies.

After running the code provided, the value of x remains 3.

Here’s the reasoning:

1. **Initial Assignment:**

* x <- 3
  + x is assigned the value 3 in the global environment.

1. **Function Definition:**

* my\_func <- function(y){  
   x <- 5  
   y + 5  
  }
  + Inside the function my\_func, x <- 5 assigns the value 5 to a local variable x. This local x is different from the global x.
  + The line y + 5 adds 5 to the argument y but does not affect the global variable x.

1. **Effect on Global x:**
   * Since the assignment x <- 5 is inside the function, it does not modify the x in the global environment.
   * The global x remains unchanged with the value 3.

So, after running the code, the value of x in the global environment is still 3.

Here’s how you can write a function compute\_s\_n in R to compute the sum ( S\_n = 1^2 + 2^2 + 3^2 + + n^2 ). Then, I’ll show you how to use this function to find the sum when ( n = 10 ).

compute\_s\_n <- function(n) {  
 sum((1:n)^2)  
}  
  
# Calculate the sum for n = 10  
result <- compute\_s\_n(10)  
result

### Explanation:

* **compute\_s\_n function**:
  + This function takes a single argument n.
  + It calculates the sum of the squares of the integers from 1 to n using the expression sum((1:n)^2).
* **Result for n = 10**:
  + When you call compute\_s\_n(10), the function computes the sum ( 1^2 + 2^2 + 3^2 + + 10^2 ).

### Result:

Running the code will give you the sum ( S\_{10} = 1^2 + 2^2 + 3^2 + + 10^2 ), which equals 385.

You can use a for loop to compute the sum ( S\_n = 1^2 + 2^2 + 3^2 + + n^2 ) for each value of ( n ) from 1 to 25, and store the results in the s\_n vector. Here’s how you can do it:

# Define an empty numerical vector of size 25  
s\_n <- vector("numeric", 25)  
  
# Use a for-loop to compute and store the results  
for (n in 1:25) {  
 s\_n[n] <- sum((1:n)^2)  
}  
  
# Print the result  
s\_n

### Explanation:

* **s\_n <- vector("numeric", 25)**: Initializes an empty numeric vector s\_n with 25 elements, all set to 0.
* **for (n in 1:25)**: Iterates over the sequence from 1 to 25.
* **s\_n[n] <- sum((1:n)^2)**: For each iteration, the sum ( S\_n = 1^2 + 2^2 + + n^2 ) is calculated and stored in the n-th position of the vector s\_n.

### Result:

After running this code, s\_n will contain the sum of squares for each n from 1 to 25. For example, s\_n[10] will store the sum ( 1^2 + 2^2 + + 10^2 = 385 ).

To perform the same computation using sapply, you can use it to apply a function over a sequence of numbers. Here’s how you can do it:

# Define a function to compute the sum of squares up to n  
sum\_squares <- function(n) {  
 sum((1:n)^2)  
}  
  
# Use sapply to compute and store the results for n from 1 to 25  
s\_n <- sapply(1:25, sum\_squares)  
  
# Print the result  
s\_n

### Explanation:

* **sum\_squares function**: Defines a function to calculate the sum of squares from 1 to n.
* **sapply(1:25, sum\_squares)**: Applies the sum\_squares function to each element of the sequence 1:25, and stores the results in the vector s\_n.

### Result:

After running this code, s\_n will contain the sum of squares for each integer from 1 to 25, similar to the previous approach. For example, s\_n[10] will be 385, which corresponds to ( 1^2 + 2^2 + + 10^2 ).

To plot ( S\_n ) versus ( n ) where ( S\_n ) is the sum of squares up to ( n ), and ( n ) ranges from 1 to 25, you can use the plot function in R. Here’s how you can do it:

# Define the function to compute the sum of squares up to n  
sum\_squares <- function(n) {  
 sum((1:n)^2)  
}  
  
# Use sapply to compute and store the results for n from 1 to 25  
s\_n <- sapply(1:25, sum\_squares)  
  
# Plot S\_n versus n  
plot(1:25, s\_n,   
 xlab = "n",   
 ylab = "S\_n",   
 main = "Plot of S\_n versus n",  
 pch = 19,   
 col = "blue")

### Explanation:

* **sapply(1:25, sum\_squares)**: Computes the sum of squares for each integer from 1 to 25 and stores the results in s\_n.
* **plot(1:25, s\_n, ...)**: Creates a scatter plot with n on the x-axis (from 1 to 25) and S\_n on the y-axis.
  + **xlab = "n"**: Labels the x-axis as "n".
  + **ylab = "S\_n"**: Labels the y-axis as "S\_n".
  + **main = "Plot of S\_n versus n"**: Sets the title of the plot.
  + **pch = 19**: Uses solid circles for the points.
  + **col = "blue"**: Colors the points blue.

Running this code will generate a scatter plot showing how ( S\_n ) changes as ( n ) increases from 1 to 25.

To confirm that the formula for the sum of squares ( S\_n = 1^2 + 2^2 + + n^2 ) is ( ), we can use this formula and compare it to the results obtained from the sum computation.

Here’s how you can do this in R:

1. **Define the formula function**:

sum\_squares\_formula <- function(n) {  
 n \* (n + 1) \* (2 \* n + 1) / 6  
}

1. **Compute the sum using both methods and compare**:

# Define the function to compute the sum of squares up to n using sapply  
s\_n\_computed <- sapply(1:25, sum\_squares)  
  
# Compute the sum using the formula  
s\_n\_formula <- sapply(1:25, sum\_squares\_formula)  
  
# Check if the computed values match the formula values  
all.equal(s\_n\_computed, s\_n\_formula)

### Explanation:

* **sum\_squares\_formula(n)**: Defines a function that computes ( S\_n ) using the formula ( ).
* **s\_n\_computed**: Contains the values of ( S\_n ) computed by summing squares for ( n ) from 1 to 25.
* **s\_n\_formula**: Contains the values of ( S\_n ) computed using the formula for ( n ) from 1 to 25.
* **all.equal(s\_n\_computed, s\_n\_formula)**: Checks if the two vectors are equal, confirming that the formula correctly computes the sum of squares.

If the function returns TRUE, it confirms that the formula ( ) correctly represents the sum of squares ( S\_n ).