

Week4RProjectCristian

Cristian Paraschiv

2025-06-04

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>.

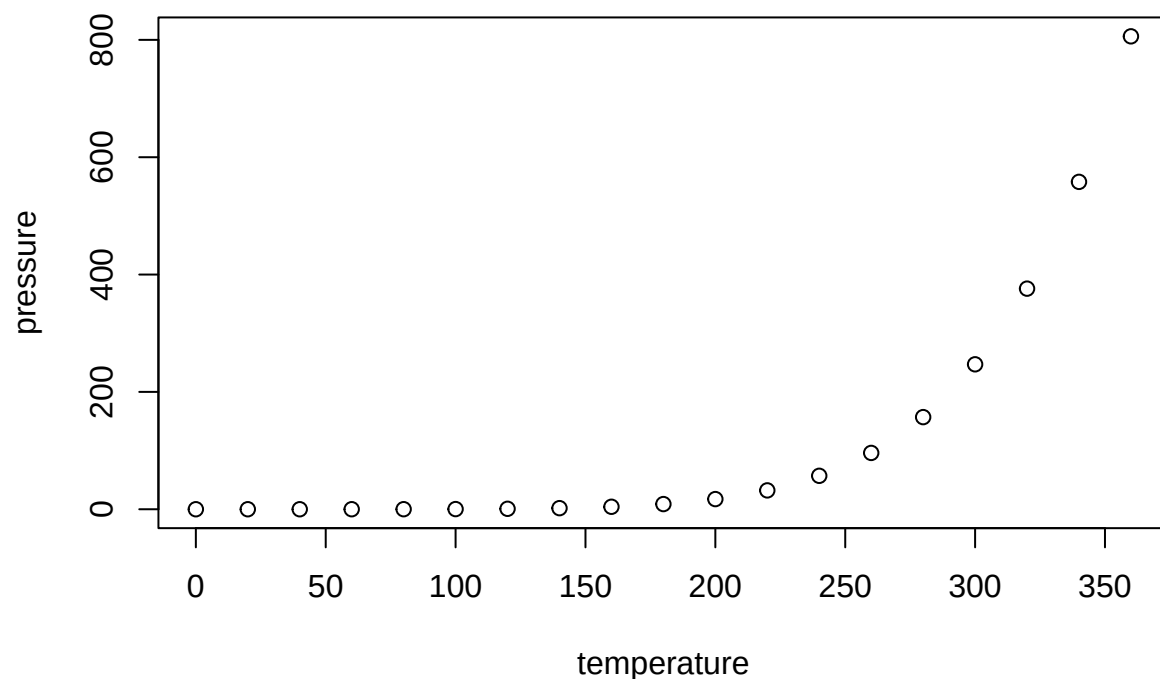
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed      dist
##  Min.   : 4.0    Min.   : 2.00
##  1st Qu.:12.0    1st Qu.: 26.00
##  Median :15.0    Median : 36.00
##  Mean   :15.4    Mean   : 42.98
##  3rd Qu.:19.0    3rd Qu.: 56.00
##  Max.   :25.0    Max.   :120.00
```

Including Plots

You can also embed plots, for example:



Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Activity 1: Write a script to classify numbers as positive, negative, or zero.

```
# Function to classify a number
classify_number <- function(x) {
  if (x > 0) {
    return("Positive")
  } else if (x < 0) {
    return("Negative")
  } else {
    return("Zero")
  }
}
```

```
# Example usage
classify_number(10)  # "Positive"
```

```
## [1] "Positive"
```

```
classify_number(-7)  # "Negative"
```

```
## [1] "Negative"
```

```
classify_number(0)    # "Zero"
```

```
## [1] "Zero"
```

Activity 2: Use a loop to sum numbers from 1 to 100.

```
sum_result <- 0

for (i in 1:100) {
  sum_result <- sum_result + i
}

print(sum_result)    # Should print 5050
```

```
## [1] 5050
```

```
sum_result <- 0
i <- 1

while (i <= 100) {
  sum_result <- sum_result + i
  i <- i + 1
}

print(sum_result)    # Should print 5050
```

```
## [1] 5050
```

Activity 3: Generate and print random numbers until one is greater than 0.9.

```
repeat {
  num <- runif(1)    # Generate a random number between 0 and 1
  print(num)

  if (num > 0.9) {
    break
  }
}
```

```
## [1] 0.916201
```

Activity 4: Rewrite a loop using vectorisation.

```
squares <- c()
for (i in 1:5) {
  squares[i] <- i^2
}
print(squares)
```

```
## [1] 1 4 9 16 25
```

```
x <- 1:5
squares <- x^2
print(squares)
```

```
## [1] 1 4 9 16 25
```

Function Definition

```
# Enhanced triple function with type checking
triple <- function(x) {
  if (is.numeric(x) || is.logical(x)) {
    return(x * 3)
  } else if (is.character(x)) {
    return(paste0(x, x, x)) # Repeat each character 3 times
  } else {
    warning("Unsupported data type. Please provide numeric, logical, or character data.")
    return(NULL)
  }
}
```

```
### Main Script ###
```

```
# Integer
a <- 5
print("Triple of a:")
```

```
## [1] "Triple of a:"
print(triple(a)) # Should return 15
```

```
## [1] 15
print("Original a remains:")
```

```
## [1] "Original a remains:"
print(a) # Still 5
```

```
## [1] 5
# Numeric vector
b <- c(2.1, 5.3, 7.9)
print("Triple of b:")
```

```
## [1] "Triple of b:"
print(triple(b)) # Returns 6.3, 15.9, 23.7
```

```
## [1] 6.3 15.9 23.7
print("Original b remains:")
```

```
## [1] "Original b remains:"
print(b)
```

```
## [1] 2.1 5.3 7.9
```

```

# Logical vector
c <- c(TRUE, FALSE, TRUE)
print("Triple of c:")

## [1] "Triple of c:"
print(triple(c)) # Returns 3, 0, 3

## [1] 3 0 3
print("Original c remains:")

## [1] "Original c remains:"
print(c)

## [1] TRUE FALSE TRUE
# Character vector
d <- c("A", "B", "C")
print("Triple of d:")

## [1] "Triple of d:"
print(triple(d)) # Returns "AAA", "BBB", "CCC"

## [1] "AAA" "BBB" "CCC"
print("Original d remains:")

## [1] "Original d remains:"
print(d)

## [1] "A" "B" "C"
### Function Definitions ###

# Function mean_mean(x)
mean_mean <- function(x)
{
  out <- mean(x) # Calculate using mean()
  out # Return out
}

# Function mean_loop(x)
mean_loop <- function(x)
{
  running_total <- 0 # Initialise for calculating sum
  element_total <- 0 # Initialise for counting number of elements

  for (i in x) {
    running_total <- running_total + i # Add each element
    element_total <- element_total + 1 # Increment count
  }
  out <- running_total / element_total # Calculate the mean
  out # Return out
}

```

```

# Function mean_function(x)
mean_function <- function(x)
{
  out <- sum(x) / length(x) # Calculate mean using vector functions
  out # Return out
}

### Main Script ###

# Create test vectors
int_test <- 1:5
float_test <- c(1.2, 2.5, 3.5, 4.5, 5.5)

# Test mean_mean
print("mean_mean results:")

## [1] "mean_mean results:"
print(mean_mean(int_test))

## [1] 3
print(mean_mean(float_test))

## [1] 3.44
# Test mean_loop
print("mean_loop results:")

## [1] "mean_loop results:"
print(mean_loop(int_test))

## [1] 3
print(mean_loop(float_test))

## [1] 3.44
# Test mean_function
print("mean_function results:")

## [1] "mean_function results:"
print(mean_function(int_test))

## [1] 3
print(mean_function(float_test))

## [1] 3.44

### Function Definitions ###

# Raise x to the power of y
raise <- function(x, y) x^y

# Repeat each element in a string vector 'n' times
echostring <- function(n, string) {
  return_vector <- NA

```

```

for (i in 1:length(string)) {
  for (j in 1:n) {
    return_vector <- c(return_vector, string[i])
  }
}
return(return_vector[-1]) # Remove the initial NA
}

### Main Script ###

# Test raise()
print("Testing raise():")

## [1] "Testing raise():"
print(raise(2, 3)) # 8

## [1] 8
a <- c(2, 3)
print(raise(a[1], a[2])) # 2^3 = 8

## [1] 8
b <- c(3, 4)
print(raise(a, b)) # c(2^3, 3^4) = c(8, 81)

## [1] 8 81
# Test echostring()
print("Testing echostring():")

## [1] "Testing echostring():"
outstring <- echostring(3, "Echo...")
print(outstring) # "Echo..." repeated 3 times

## [1] "Echo..." "Echo..." "Echo..."
c <- c("A string", "Another string")
outstring <- echostring(3, c)
print(outstring) # Each string repeated 3 times

## [1] "A string" "A string" "A string" "Another string"
## [5] "Another string" "Another string"

### Function Definitions ###

# Function: build_range(x, y, size)
# Description: Returns a numeric vector starting from x, increasing by 'size',
#              and ending at or before y
build_range <- function(x, y, size) {
  outvector <- seq(x, y, by = size)
  return(outvector)
}

# Function: print_squares(v, limit)
# Description: Prints squares of elements in vector v,
#              but only if the square is less than or equal to 'limit'

```

```

print_squares <- function(v, limit) {
  for (i in v) {
    square <- i^2
    if (square <= limit) {
      print(square)
    }
  }
}

# Function: print_limit(x, y, size, limit)
# Description: Combines build_range and print_squares.
#           Builds a range and prints squares under the limit
print_limit <- function(x, y, size, limit) {
  numberlist <- build_range(x, y, size)
  print_squares(numberlist, limit)
}

# Optional shorter version using nesting
# print_limit <- function(x, y, size, limit) {
#   print_squares(build_range(x, y, size), limit)
# }

### Main Script ###

# Test build_range
cat("Testing build_range(4, 13, 4):\n")

## Testing build_range(4, 13, 4):
print(build_range(4, 13, 4)) # Expected: 4 8 12

## [1] 4 8 12

# Test print_squares
cat("\nTesting print_squares(3:9, 40):\n")

##
## Testing print_squares(3:9, 40):
print_squares(3:9, 40) # Expected: 9, 16, 25, 36

## [1] 9
## [1] 16
## [1] 25
## [1] 36

# Test print_limit with integers
cat("\nTesting print_limit(2, 9, 3, 30):\n")

##
## Testing print_limit(2, 9, 3, 30):
print_limit(2, 9, 3, 30) # Expected: 4, 25

## [1] 4
## [1] 25

```

```
# Test print_limit with floats
cat("\nTesting print_limit(2.1, 15.6, 2.5, 35.2):\n")

##
## Testing print_limit(2.1, 15.6, 2.5, 35.2):
print_limit(2.1, 15.6, 2.5, 35.2)

## [1] 4.41
## [1] 21.16
# Expected: squares of 2.1, 4.6, 7.1 etc. if under 35.2
```