Lesson 6

by Lorraine Gaudio

Lesson generated on August 21, 2025

Contents

1. Welcome back to R!

In lesson one, through four you learned about objects, vectors, functions and how to handle missing data in R. In this lesson, you will learn about subsetting, extraction, and insertion in data frames. You rarely analyse an entire data table at once. You nearly always pull out specific rows or columns, or write back cleaned-up values. Mastering R's bracket "[ ]" syntax is therefore foundational.

To begin Lesson 6, follow these steps:

1. Open your course project for RStudio
2. Create a new file. Today, let's try "R Markdown" (File > New File > R Markdown).
3. Type in the code provided in this document as you follow along with the video. Pause the video at anytime to answer assignment questions, dig deeper or add memo notes.

Lesson Overview

By the end of Lesson 6 you will be able to:

1. Remember – State R's basic subsetting template: x[rows, cols].
2. Understand – Describe the difference between $ and [ ] for data frames.
3. Apply – Use logical tests + which() to select rows.
4. Analyze – Summarize a subset to answer a question.
5. Evaluate – Choose an insertion (overwrite) vs. safe copy strategy

Keep these goals in mind as you move through each section.

2. Quick Warm￾up

Type the following code in a new code chunk and run.

```
# Review the `sample()` function from lesson 3.
set.seed(1)

args(sample)  # Check the arguments of sample()
```

Type the following code in a new code chunk and run.

```
trial_vec <- sample(x = c(1:3, NA, NA, NA, NA), size = 25, replace = TRUE)
trial_vec
```

Type the following code in a new code chunk and run. Create a memo note, demonstrate learning skill(s) used.

```
trial_vec <- sample(size = 25, replace = TRUE, x = c(1:3, NA, NA, NA, NA))
trial_vec
```

NOTICE: How does changing the order of arguments affect the output? Create a memo note, demonstrate learning skill(s) used.

Review is.na() and which() from lesson 4.

```
# Review is.na() and which() from lesson 4 and Nesting from lesson 3
is.na(trial_vec)
which(is.na(trial_vec))
which(!is.na(trial_vec))
```

Are you ready? Remember, you can summon help whenever you need it.

3. Build a Data Frame

In lesson 2, you learned how to create a data frame using the function data.frame() to combine three vectors of equal length. You created these vectors using the c() function. In this lesson, you will create the vectors using the ifelse() function. Type the following code in a new code chunk.

```
?ifelse
args("ifelse")  # □ Check the arguments of ifelse()
```

ifelse() is a vectorized function that allows you to create a new variable based on a condition. Type the following code in a new code chunk.

```
# Example: Create a new vector trial_vec2 based on trial_vec
trial_vec2 <- ifelse(trial_vec == 2, "two", "not two")
```

This creates a new vector where each element is "two" if the corresponding element in trial_vec is 2, and "not two" otherwise. It takes three arguments:

- condition: a logical test (e.g., trial_vec == 2)
- value_if_true: the value to assign if the condition is TRUE
- value_if_false: the value to assign if the condition is FALSE

Now, let's create a data frame using trial_vec and trial_vec2. Type the following code in a new code chunk and run.

```
# Create a data frame with trial_vec and trial_vec2
trial_df <- data.frame(trial_vec, trial_vec2)
```

```
# View(trial_df)  # □ Check the data frame with two columns
View(trial_df) # (opens a new tab in RStudio)
```

□ That was a practice round. Now, let's create a more complex data frame with two long vectors of equal length.

We'll simulate a small inventory called snack_box with two columns:

- Type – "Cereal" or "Marshmallow"
- Shape – shape names for marshmallows, placeholder for cereal

Type the following code in a new code chunk.

```
set.seed(8)
Type  <- sample(c(rep("Cereal", 3091), rep("Marshmallow", 287)))
Shape <- ifelse(Type == "Cereal", "Plain",
            sample(c("Heart", "Star", "Horseshoe", "Clover",
                   "Blue_Moon", "Pot_of_Gold", "Rainbow",
                   "Red_Balloon"), length(Type), replace = TRUE))
snack_box <- data.frame(Type, Shape)

# View the data
View(snack_box)
```

☐ Look deeper: Why does the code has line breaks with indentation? What is the advantage? Explain when to break and how far to indent. Create a memo note, demonstrate learning skill(s) used.

4. Bracket Syntax

□ Review Lesson 2 about indexing vectors x[]. Create a memo note, demonstrate learning skill(s) used. Create a memo note, demonstrate learning skill(s) used.

In this lesson we will use the same syntax to subset data frames.

- x is the object (vector, data frame, etc.)
- [] is the bracket operator for subsetting
- x[rows, cols] extracts specific rows and columns from x.
- rows and cols can be: indices (1, 3:5), names ("Shape"), or a logical vector / expression (Type == "Marshmallow").

Type the following code in a new code chunk and run.

```r
snack_box[1:3, ]  # □  Shows full row extraction
```

```r
# R's subsetting format:  x[rows, columns]
snack_box[1:3, 1]  # □  Shows first three rows of first column
snack_box[1:3, c(1, 2)]  # □  Shows first three rows of first and second
```

5. Subsetting

We'll start with trial_df from earlier. Type the following code in a new code chunk and run.

```
trial_df[trial_df$trial_vec == 2, ]
# Test direct subsetting (will include NA rows)
direct_subset <- trial_df[trial_df$trial_vec == 2, ]
print(direct_subset)
trial_df[trial_df$trial_vec == 2, ]
```

☐ Identify: which rows are extracted?

- TRUE (2 == 2)
- FALSE (1 == 2)
- FALSE (3 == 2)
- NA (NA == 2)

☐ **Goal: isolate only the marshmallows.**

☐ **Step 1** – create a logical test: [ ] vs. $. Type the following code in a new code chunk and run.

```
snack_box[snack_box$Type == "Marshmallow", ]
```

snack_box[snack_box$Type == "Marshmallow",] returns the rows where the condition is TRUE, along with all columns.

```
snack_box$Type == "Marshmallow"          # TRUE/FALSE for each row
```

snack_box$Type == "Marshmallow" returns a logical vector.

☐ **Step 2** – Store subset of snack_box data frame as a new data frame. Type the following code in a new code chunk and run.

```
marshmallows_only <- snack_box[snack_box$Type == "Marshmallow", ]
```

☐ Look deeper: How many *rows* did we keep? How could you check the number of rows in a data set? Create a memo note, demonstrate learning skill(s) used, including method you used to answer the question.

Type the following code in a new code chunk and run.

6. Extraction

Using which() is a strategy to handle potential NA values in your data. When the test could include NA, wrap it in which() to drop unknowns.

Type the following R script in your document script and run.

```r
rows_without_na <- which(trial_df$trial_vec == 2)

which_subset <- trial_df[rows_without_na, ]

print(which_subset)
```

□ Explore and Play: What is the difference between running print(which_subset) and which_subset? Create a memo note, demonstrate learning skill(s) used.

When you use a direct logical condition like trial_df$trial_vec == 2, any NA values in the Type column will result in NA in the logical vector. You get only the indices where the condition is TRUE, effectively dropping/ignoring any rows where the comparison resulted in NA. You get a result that excludes rows with NA in the Type column

6.1 $ vs. [ ]

Type the following code in a new code chunk and run.

```r
row_idx <- which(snack_box[,"Type"] == "Marshmallow" & snack_box[,"Shape"] == "Star")
```

The bracket notation [,"column_name"] uses matrix-style indexing.

You can access the same columns using $ notation where snack_box$Type replaces snack_box[,"Type"] and snack_box$Shape replaces snack_box[,"Shape"]. Type the following code in a new code chunk and run.

```r
row_idx2 <- which(snack_box$Type == "Marshmallow" & snack_box$Shape == "Star")
```

□ Common Pain Point: Using which() but forgetting to subset columns. Type the following code in a new code chunk and run.

```r
subset_star <- snack_box[row_idx, ]
head(subset_star)  # □  Practice Task 5 logic with known data
head(x = subset_star, n= 3)
```

☐ Explain the function head() and how it works with the n argument. Create a memo note, demonstrate learning skill(s) used.

6.2 Selecting Columns

☐ Method 1: Select columns by name using indexing.

Recall that columns can be chosen by index or name.

- By index: snack_box[, 2] (second column)
- By name : snack_box[, "Shape"]

```
head(subset_star[ , "Shape"])
```

You can combine row + column ideas: shapes of the first 10 marshmallows. Type the following code in a new code chunk and run.

```
snack_box[1:10, "Shape"]
```

☐ Why did we use head() in the first chunk and not in the second? Create a memo note, demonstrate learning skill(s) used.

☐ Method 2: Select columns by name using $. Type the following code in a new code chunk and run.

```
Cereal_Shapes <- snack_box$Shape[snack_box$Type == "Cereal"]
```

This creates a vector called Cereal_Shapes that contains only the shapes from rows where the type is "Cereal". Type the following code in a new code chunk and run.

```
Marshmallow_Shapes <- snack_box$Shape[snack_box$Type == "Marshmallow"]
```

It's particularly useful when you want to:

- Analyze the distribution of "Shapes" within each "Type".
- Compare properties between different categories.
- Create separate vectors for further analysis by category.

This creates a vector called Marshmallow_Shapes with only shapes from marshmallow rows.

☐ Bonus: Two conditions + column subset

Use snack_box: Pick shapes that are Red_Balloon and Type = Marshmallow. Type the following code in a new code chunk and run.

```r
marshmallow_data <- snack_box[snack_box$Type == "Marshmallow", ]

red_balloon_rows <- which(marshmallow_data$Shape == "Red_Balloon")

red_balloon_marshmallows <- marshmallow_data[red_balloon_rows, ]
```

7. Insertion

Insertion is the process of adding or modifying values in a data frame. You can insert new values or overwrite existing ones.

7.1 Overwrite Values

Suppose we decide to rename “Cereal” to “Grain” in the Type column.

☐ Good practice: make a backup first! Type the following code in a new code chunk and run.

```
box_safe <- snack_box          # copy
indices_cereal <- which(snack_box$Type == "Cereal")
box_safe[indices_cereal, "Shape"] <- "Grain"
```

☐ Check-in: Look at the format of that code: x[row, column] <- value. The <- operator is not just for assigning a value to a variable name. <- Grain is assigned as the new value "Grain" to replace the existing value "Cereal". The column name remains "Type".

```
# Verify the change
unique(box_safe$Shape)
```

☐ Comment: Why is snack_box unchanged while editing box_safe? Why is this good practice? Create a memo note, demonstrate learning skill(s) used.

8. Renaming Columns

Example 1: To rename a column, you can use the colnames() function. Type the following code in a new code chunk and run.

```
names(box_safe)

colnames(box_safe) <- c("Piece_Type", "Piece_Shape")
head(box_safe)
```

□ Identify: Which columns are renamed? □ Look deeper: Create a list of all the different ways you might identify the column names of a data frame? Create a memo note, demonstrate learning skill(s) used.

Example 2: You can also rename a specific column by index. Type the following code in a new code chunk and run.

```
box_safe2 <- snack_box     # □ Always copy first
names(box_safe2)[which(names(box_safe2) == "Type")] <- "Piece_Type"
names(box_safe2)[which(names(box_safe2) == "Shape")] <- "Piece_Shape"
```

9. Replace Missing Values

In lesson 4, you learned about handling missing values. Now, let's practice replacing missing values in a data frame. Recall that when a numeric vector has gaps, you can impute with the mean.

Type the following code in a new code chunk and run.

```r
# Before
trial_vec
```

```r
trial_vec[is.na(trial_vec)] <- mean(trial_vec, na.rm = TRUE)
```

The <- operator is not just for assigning a value to a variable name. Here the <- operator is used to assign the mean of the non-NA values'

Left side of the assignment: trial_vec[is.na(trial_vec)]

- x[ ] index of a vector: Recall we can use ":" like 1:3
- index value is is.na(trial_vec) making logical vector (TRUE/FALSE) marking which elements are NA.

Right side of the assignment: mean(trial_vec, na.rm = TRUE)

- mean() calculates the mean of the non-NA values in trial_vec

The assignment operator <- takes the value from the right side, places it into the specific positions identified on the left side.

```r
# After
trial_vec
```

R took all the non-missing values, calculated their mean and replaced all the NA values with this decimal mean

10. □ Assignment

Now it's your turn to practice creating and using vector objects. Follow the tasks below to complete part of the **technical skill practice assignment**.

1. Work through each task in order. Replace the ___ placeholder with your code or short written answer.
2. Run each completed line to be sure no errors appear and objects show in the Environment.
3. When finished, save your workspace and submit this R Markdown file (RMD) plus the .RData file.

10.1 Task 0

□ Setup: Load the built-in data set mtcars and take a quick look. Identify what the columns contain and meaning of that content.

```r
data("mtcars")   # already in memory but this keeps the workflow explicit
?mtcars          # help file for variable descriptions
View(mtcars)     # spreadsheet view (optional)
```

□ Comment: Explain R's basic subsetting template. Hint: It looks like data[rows , columns]

10.2 Task 1

□ Manual vs. Automatic

1. Create Manual_Cars: all rows where am == 1 (manual), all columns.

```r
Manual_Cars <- ___
```

2. Use nrow() to record how many manual cars there are (store in n_manual).

```r
n_manual <- ___ # numeric count
```

10.3 Task 2

□ MPG by Cylinder

Extract mpg for each cylinder group into separate objects. Hint: Repeat the bracket pattern

```
Four_Cyl_MPG  <- mtcars[mtcars$cyl == 4, "mpg"]
Six_Cyl_MPG   <- ___
Eight_Cyl_MPG <- ___
```

10.4 Task 3

□ Analyze Fuel Efficiency

1. Compute the mean mpg for each object.

```
mean_4  <- mean(___)
mean_6  <- mean(___)
mean_8  <- mean(___)
```

2. □ Comment: Which cylinder group is most fuel□ efficient? Answer in one word or number.

Most efficient: ___

10.5 Task 4

□ Safe vs. Direct Editing

1. Make a safe copy of mtcars called cars_safe.

```
cars_safe <- ___
```

2. Rename the 'am' column in cars_safe to "Transmission_Type".

```
colnames(cars_safe)[___] <- ___
```

3. □ Comment: Explain *why* working on cars_safe (a copy) is safer than editing mtcars directly.

Answer: ___

10.6 Task 5

□ Practice which()

Using cars_safe, extract the rows for V□ shaped engines (vs == 0) *and* horsepower (hp) above 150. Use which() for the row indices and keep only mpg, hp, and cyl columns. Name the object hi_power_v.

1. Create row_idx for the condition above.

- cars_safe$vs ? 0
- &
- cars_safe$hp ? 150
- cars_safe[row_idx , c("mpg","hp","cyl")]

```
row_idx     <- ___
```

2. Subset cars_safe with row_idx and columns mpg, hp, cyl. Name object hi_power_v.

```
hi_power_v  <- ___
head(hi_power_v)
```

10.7 Task 6

□ Insertion Example

Add a new column called "Efficiency_Class" to cars_safe. Label "High" if mpg >= 25, "Low" otherwise.

Hint: use ifelse() and cars_safe$mpg >= 25, "?", "?"

```
cars_safe$Efficiency_Class <- ___
```

11. Save and Upload

1. You will be submitting **both** the R Markdown and the workspace file. The workspace file saves all the objects in your environment that you created in this lesson. You can save the workspace by running the following command in a code chunk of the R Markdown document:

```r
save.image("Assignment6_Workspace.RData")
```

Or you can click the "Save Workspace" button in the Environment pane.

□ **Always save the R documents before closing.**

2. Find the assignment in this week's module in Canvas and upload **both** the RMD and the workspace file.

12. Today you practiced:

- Reading R’s subsetting template x[rows, cols].
- Extracting rows with logical tests and which().
- Selecting columns by index or name.
- Safely overwriting values (insertion) after making a copy.
- Renaming columns and imputing simple missing values.

□ Great job! Subsetting is the gateway to every data-cleaning task.