Lesson 6 Point Boost

by Lorraine Gaudio

Lesson generated on September 28, 2025

BOISE STATE UNIVERSITY

Contents

1 ⊠ Level Up in Lesson 6 **2**

2 Quick Warm-up **3**

3 Quick Set-up **4**

4 Build a Data Frame **6**

4.1 trial_vec2 6

4.2 Themes 7

4.3 ⊠ Task 1 8

5 Numeric Column **10**

5.1 ⊠ Task 2 11

6 $ vs. **[ ]** **13**

6.1 Subsetting 13

6.2 Extraction 15

6.3 Insertion 18

6.4 Renaming Columns 18

7 Replace Missing Values **20**

1. ▯ Level Up in Lesson 6

In lesson 6 you learn about subsetting, extraction, and insertion in data frames. You explored how to use R's bracket "[ ]" syntax is therefore foundational. In this Technical Skill and Learning Point Boost, you will do a side-by-side comparision of $ and [ ]. Addiionally, we we review ideas from previous lessons.

To begin Lesson 6 Point Boost, follow these steps:

1. Open your course project for RStudio
2. Create a new file. Today, let's use ▯ "R Markdown" again (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 learning skill prompts by digging deeper or add memo notes.
4. Complete the assignment questions in this same document.
5. Submit the RMD and RData file to the Canvas assignment for this lesson.

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 ifelse() to create a themed dataset.
4. ▯ Analyze – Summarize a subset to answer a question.
5. ▯ Evaluate – Choose an insertion (overwrite) vs. safe copy strategy

2. Quick Warm-up

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

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

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

```
## function (x, size, replace = FALSE, prob = NULL)
## NULL
```

3. Quick Set-up

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

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

```
## [1]  1 NA NA  1  2 NA NA  3 NA  2  3  3  1 NA NA  2 NA NA  2 NA  1 NA NA NA  1
```

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

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

```
## [1]  1 NA NA  1  2 NA NA  3 NA  2  3  3  1 NA NA  2 NA NA  2 NA  1 NA NA NA  1
```

☐ NOTICE: Does changing the order of arguments affect the output? Why?

☐ Break Things! Can you change the order of the arguments without assigning the argument name (e.g,TRUE, 25, c(1:3, NA, NA, NA, NA)) instead of replace = TRUE, size = 25, x = c(1:3, NA, NA, NA, NA))? Create a memo note, demonstrate learning skill(s) used.

We will be using trial_vec throughout the lesson as a simple example. Be sure the you have a working trial_vec object in your environment.

```
print(trial_vec)
```

```
## [1]  1 NA NA  1  2 NA NA  3 NA  2  3  3  1 NA NA  2 NA NA  2 NA  1 NA NA NA  1
```

Learning Skill: Review is.na() and which() from lesson 4 and Nesting from lesson 3

```
# Part 1
is.na(trial_vec)
```

```
##  [1] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
## [13] FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
## [25] FALSE
```

□ Explain: What type of object is returned? What is its length relative to trial_vec?

```
# Part 2
which(is.na(trial_vec))
```

```
##  [1]  2  3  6  7  9 14 15 17 18 20 22 23 24
```

□ Explain: What are these numbers: values from the vector or positions in the vector? How do these numbers relate to the TRUE entries from part (1)?

```
# Part 3
which(!is.na(trial_vec))
```

```
##  [1]  1  4  5  8 10 11 12 13 16 19 21 25
```

□ Explain: Interpret these numbers the same way as in part (2). Create a memo note, demonstrate learning skill(s) used. ___

4. 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 Lesson 6, we created a bird band dataset using the ifelse() function. To level up, explore the ifelse() function. Using this skill, you will create a dataset that contains your own theme.

Type the following code in a new code chunk.

```
?ifelse # Open the Help for the ifelse() function
```

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

```
## function (test, yes, no)
## NULL
```

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.

4.1 trial_vec2

```
# 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)
```

□ Check the data frame trial_df.

```
# head(trial_df)
head(trial_df) # Prints only for first 6 rows of trial_df
```

trial_vec	trial_vec2
1	not two
NA	NA
NA	NA
1	not two
2	two
NA	NA

☐ Comment: Describe trial_df.

4.2 Themes

In Lesson 6, we created a bird band dataset. In this point boost activity, you will create a dataset that contains your own theme. Here's what we did in the lesson.

1. Dataset theme: Bird Band Observations ☐

This theme simulates data from a field biologist observing a population of birds, where most are untagged, but a small number have been previously captured and marked with colored leg bands. This is a very common practice in ecology and behavioral studies.

We simulated a small inventory called field_sightings with two columns:

- Status – Describes whether the sighted bird is "Untagged" (the common case) or "Tagged" (the rare case).
- Band_Color – If a bird is untagged, the value is "None." If it is tagged, the value is one of several possible band colors.

Cope and paste the following code in a new code chunk.

```
set.seed(8)
Status  <- sample(c(rep("Untagged", 4168), rep("Tagged", 302)))
Band_Color <- ifelse(Status == "Untagged", "None",
            sample(c("Red", "Blue", "Green", "Yellow",
                  "Silver", "Black", "Orange",
                  "White"), length(Status), replace = TRUE))
field_sightings <- data.frame(Status, Band_Color)

# View the data
View(field_sightings)
```

Here is another example of a theme.

2. Anthropology: Archaeological Pottery Shards □

This theme is based on an archaeological dig where researchers are cataloging pottery fragments (shards). Most shards are plain, undecorated earthenware, but a few special ones feature distinct decorative motifs.

Column 1 (Find_Type): Differentiates between "Plain_Earthenware" (common) and "Decorated_Ware" (rare).

Column 2 (Motif): Plain shards are "Undecorated." Decorated ones can have various patterns like "Incised_Lines" or "Cross_Hatching."

```r
set.seed(8)
Find_Type <- sample(c(rep("Plain_Earthenware", 1642), rep("Decorated_Ware", 158)))
Motif <- ifelse(Find_Type == "Plain_Earthenware", "Undecorated",
            sample(c("Incised_Lines", "Cross_Hatching", "Punctate",
                "Cord_Marked", "Stamped_Circle", "Wavy_Line",
                "Polychrome_Geometric", "Zoned_Red"),
              length(Find_Type), replace = TRUE))
sherd_inventory <- data.frame(Find_Type, Motif)
# View the data
head(sherd_inventory)
```

Find_Type	Motif
Plain_Earthenware	Undecorated
Plain_Earthenware	Undecorated
Decorated_Ware	Incised_Lines
Plain_Earthenware	Undecorated
Plain_Earthenware	Undecorated
Plain_Earthenware	Undecorated

4.3 □ Task 1

Create you own, more complex data frame, with two long vectors of equal length. Be creative!

1. Describe Your Dataset

First, compose a written description of your dataset's theme. Your description should clearly explain the scenario you are simulating and what each of your two columns represents. Use the bird and pottery examples above as a guide.

Theme: You are a *profession* collecting data on Subject_Type. Most of the Subject_Type you find are "Common_Type", but occasionally you find a "Rare_Type". For the common species, you just note its "Common_Attribute". For the "Rare_Type", you identify its "Specific_Attribute".

Column 1 (Subject_Type): "Common_Type" vs. "Rare_Type"

Column 2 (Attribute_Name): "Common_Attribute" vs. a list of "Rare_Type"'s "Specific_Attribute" names like "Attribute_1", "Attribute_2", etc. (Your list can have a range of lengths).

2. Use the R template below to create your dataset in R.

```
# Step 1: Create the first vector for the main category
# This vector, `Subject_Type`, will contain many "Common_Type" items
# and a few "Rare_Type" items.
Subject_Type <- sample(c(rep("Common_Type", 1250), rep("Rare_Type", 95)))


# Step 2: Create the second vector for the sub-category
# This vector, `Attribute_Name`, gives a specific detail. The detail depends
# on whether the item is common or rare.
Attribute_Name <- ifelse(Subject_Type == "Common_Type", "Common_Attribute",
                    sample(c("Attribute_1", "Attribute_2", "Attribute_3",
                          "Attribute_4", "Attribute_5", "Attribute_6"),
                       length(Subject_Type), replace = TRUE))


# Step 3: Combine the vectors into a data frame
# Give your final data frame a descriptive name by replacing `your_dataframe`.
your_dataframe <- data.frame(Subject_Type, Attribute_Name)


# Step 4: View the first few rows of your data frame
head(your_dataframe)
```

Subject_Type	Attribute_Name
Common_Type	Common_Attribute
Common_Type	Common_Attribute
Common_Type	Common_Attribute
Common_Type	Common_Attribute
Rare_Type	Attribute_3
Common_Type	Common_Attribute

5. Numeric Column

In your practice dataset, you'll want to add numeric values. Here is an example using the Bird Band Observations □ theme.

Tagged Birds: The Release Cohorts

In conservation, it's common for birds bred in captivity to be released into the wild in groups at different times. Each group is called a cohort, and they are often marked with a specific color band to identify which release program and year they belong to.

- Age and Band Color are Linked: The code establishes a direct link between a bird's band color and its age. The cohort_means vector defines the average age, in months, for each colored cohort at the time of the field study.
  - Red=14: The red-banded birds are the youngest cohort, with an average age of 14 months. They were likely released most recently.
  - White=56: The white-banded birds are the oldest cohort, with an average age of 56 months (about 4.5 years). They were released much earlier than the red-banded group.
- Simulating Age: For each tagged bird, the code generates a random age from a normal distribution centered on the mean age for its specific color. This creates a realistic spread of ages within each cohort. All tagged birds are simulated to be at least 12 months old, indicating they are adults that were part of these established release programs.

Untagged Birds: The Wild Population

The untagged birds represent the general wild population with unknown origins.

- Unknown and Younger: Since their history is unknown, they are simulated as a younger group. The code assigns them a random age between 0 and 11 months, with the average being around 6 months.
- Biological Assumption: This reflects a natural population, which typically includes a large number of juveniles and sub-adults that have not yet been captured and banded.

Cope and paste the following code in a new code chunk.

```r
# cohort_means
cohort_means <- c(Red=14, Blue=20, Green=26, Yellow=32, Silver=38, Black=44, Orange=50, White=56)
cohort_means
```

```
##    Red   Blue  Green Yellow Silver  Black Orange  White 
##     14     20     26     32     38     44     50     56 
```

Cope and paste the following code in a new code chunk. The code below will be explained in later lessons.

```r
# Random assignment within normal bell curve
field_sightings$Age_Months <- round(ifelse(field_sightings$Status=="Untagged",
  pmin(pmax(rnorm(nrow(field_sightings), 6, 3), 0), 11),
  pmax(12, rnorm(nrow(field_sightings),
             mean = replace(cohort_means[field_sightings$Band_Color],
                         is.na(cohort_means[field_sightings$Band_Color]), 20),
             sd = 3))))

View(field_sightings)
```

5.1 ￿ Task 2

1. Describe Your Numeric Column

First, write a brief description of what your new numeric column represents. Explain how its values relate to the two categories in your first column (Common_Type vs. Rare_Type), similar to the explanation for the bird example.

2. Fill Out the Template

Use the R code template below to add the numeric column to your data frame. Read the comments (#) to understand which parts you need to change.

```r
# Step 1: Create the vector of means for your rare items
# This named vector links each of your "Rare_Type" attributes to an average numeric value.
# TODO: The names ("Attribute_1", "Attribute_2", etc.) MUST EXACTLY MATCH the rare
#       attribute names you created in Task 1.
# TODO: Replace the numbers (14, 20, 26, etc.) with the mean values that make
#       sense for your theme.
attribute_means <- c(Attribute_1=14, Attribute_2=20, Attribute_3=26,
               Attribute_4=32, Attribute_5=38, Attribute_6=44)
attribute_means
```

```
## Attribute_1 Attribute_2 Attribute_3 Attribute_4 Attribute_5 Attribute_6
##          14          20          26          32          38          44
```

```r
# Step 2: Add the new numeric column to your data frame

# This code creates the new column using ifelse(). It assigns one set of numeric
# values to your "Common_Type" and a different set to your "Rare_Type".

# TODO: Replace `your_dataframe` with the name of your data frame from Task 1.
# TODO: Replace `Numeric_Column_Name` with a descriptive name for your new column.
# TODO: Make sure `Subject_Type == "Common_Type"` matches the names from your data frame.

# For the "Common_Type" (the first rnorm line):
# TODO: Change the mean (6) and standard deviation (3) to fit your theme.
# TODO: Change the min (0) and max (11) values for the common items.

# For the "Rare_Type" (the second rnorm line):
# TODO: Change the minimum value (12) for the rare items. This should be higher
#       than the max value for your common items.
# TODO: Make sure the code uses `attribute_means[your_dataframe$Attribute_Name]` to look up the correct mean.

your_dataframe$Numeric_Column_Name <- round(ifelse(your_dataframe$Subject_Type == "Common_Type",
  pmin(pmax(rnorm(nrow(your_dataframe), 6, 3), 0), 11),
  pmax(12, rnorm(nrow(your_dataframe),
                 mean = replace(attribute_means[your_dataframe$Attribute_Name],
                                is.na(attribute_means[your_dataframe$Attribute_Name]), 20),
                 sd = 3))))

# Step 3: View your updated data frame
head(your_dataframe)
```

Subject_Type	Attribute_Name	Numeric_Column_Name
Common_Type	Common_Attribute	3
Common_Type	Common_Attribute	8
Common_Type	Common_Attribute	5
Common_Type	Common_Attribute	10
Rare_Type	Attribute_3	28
Common_Type	Common_Attribute	2

6. $ vs. [ ]

6.1 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, ]
```

	trial_vec	trial_vec2
NA	NA	NA
NA.1	NA	NA
5	2	two
NA.2	NA	NA
NA.3	NA	NA
NA.4	NA	NA
10	2	two
NA.5	NA	NA
NA.6	NA	NA
16	2	two
NA.7	NA	NA
NA.8	NA	NA
19	2	two
NA.9	NA	NA
NA.10	NA	NA
NA.11	NA	NA
NA.12	NA	NA

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

```
##       trial_vec trial_vec2
## NA           NA       <NA>
## NA.1         NA       <NA>
## 5             2        two
## NA.2         NA       <NA>
## NA.3         NA       <NA>
## NA.4         NA       <NA>
## 10            2        two
## NA.5       NA     <NA>
## NA.6       NA     <NA>
## 16          2     two
## NA.7       NA     <NA>
## NA.8       NA     <NA>
## 19          2     two
## NA.9       NA     <NA>
## NA.10      NA     <NA>
## NA.11      NA     <NA>
## NA.12      NA     <NA>
```

```
trial_df[trial_df$trial_vec == 2, ]
```

	trial_vec	trial_vec2
NA	NA	NA
NA.1	NA	NA
5	2	two
NA.2	NA	NA
NA.3	NA	NA
NA.4	NA	NA
10	2	two
NA.5	NA	NA
NA.6	NA	NA
16	2	two
NA.7	NA	NA
NA.8	NA	NA
19	2	two
NA.9	NA	NA
NA.10	NA	NA
NA.11	NA	NA
NA.12	NA	NA

☐ Identify: which rows are extracted?

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

☐ **Goal: isolate only the tagged birds.**

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

```
field_sightings[field_sightings$Status == "Tagged", ]
```

field_sightings[field_sightings$Status == "Tagged",] returns the rows where the condition is TRUE, along with all columns.

```
field_sightings$Status == "Tagged"          # TRUE/FALSE for each row
```

field_sightings$Status == "Tagged" returns a logical vector.

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

```
tagged_only <- field_sightings[field_sightings$Status == "Tagged", ]
```

 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.

```
n_tagged <- nrow(tagged_only)  #  Check number of rows
print(n_tagged)
```

```
## [1] 302
```

And 302 makes sense because of this code that helped create the dataset: Status <- sample(c(rep("Untagged", 4168), rep("Tagged", 302)))

6.2 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.

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

which_subset <- trial_df[rows_without_na, ]

print(which_subset)
```

```
##    trial_vec trial_vec2
## 5          2        two
## 10         2        two
## 16         2        two
## 19         2        two
```

□ 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 trial_vec 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 trial_vec column.

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

```
row_idx <- which(field_sightings[,"Status"] == "Tagged" & field_sightings[,"Band_Color"] == "Blue")
```

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

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

```
row_idx2 <- which(field_sightings$Status == "Tagged" & field_sightings$Band_Color == "Blue")
```

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

```
subset_blue <- field_sightings[row_idx, ]
head(subset_blue)  # □ Practice Task 5 logic with known data
```

	Status	Band_Color	Age_Months
66	Tagged	Blue	13
284	Tagged	Blue	25
430	Tagged	Blue	16
675	Tagged	Blue	25
700	Tagged	Blue	21
751	Tagged	Blue	25

```
head(x = subset_blue, n= 3)
```

	Status	Band_Color	Age_Months
66	Tagged	Blue	13
284	Tagged	Blue	25
430	Tagged	Blue	16

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

6.2.1 Selecting Columns

☐ Method 1: Select columns by name using indexing.

Recall that columns can be chosen by index or name.

- By index: field_sightings[, 2] (second column)
- By name : field_sightings[, "Band_Color"]

```
head(subset_blue[ , "Band_Color"])
```

```
## [1] "Blue" "Blue" "Blue" "Blue" "Blue" "Blue"
```

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

```
field_sightings[1:10, "Band_Color"]
```

```
##  [1] "None" "None" "None" "None" "None" "None" "None" "None" "None" "None"
```

☐ 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.

```
Untagged_Birds <- field_sightings$Band_Color[field_sightings$Status == "Untagged"]
```

This creates a vector called Untagged_Birds that contains only the band colors from rows where the status is "Untagged". Type the following code in a new code chunk and run.

```
Tagged_Birds <- field_sightings$Band_Color[field_sightings$Status == "Tagged"]
```

It's particularly useful when you want to:

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

This creates a vector called Tagged_Birds with only band colors from tagged birds rows.

☐ Bonus: Two conditions + column subset

Use field_sightings: Pick band colors that are White and Status = Tagged. Type the following code in a new code chunk and run.

```r
tagged_data <- field_sightings[field_sightings$Status == "Tagged", ]

white_rows <- which(tagged_data$Band_Color == "White")

white_tagged <- tagged_data[white_rows, ]
```

6.3 Insertion

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

6.3.1 Overwrite Values

Suppose we decide to rename “Untagged” to “Wild_Hatch” in the Status column.

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

```r
# Part 1
reintroduction <- field_sightings          # copy
indices_untagged <- which(field_sightings$Status == "Untagged")
reintroduction[indices_untagged, "Status"] <- "Wild_Hatch"
```

☐ 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. <- Wild_Hatch is assigned as the new value "Wild_Hatch" to replace the existing value "Untagged". The column name remains "Status".

```r
# Verify the change
unique(reintroduction$Band_Color)
```

```
## [1] "None"   "Orange" "Silver" "Green"  "Black"  "Blue"   "White"  "Red"
## [9] "Yellow"
```

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

6.4 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.

```r
# Part 2
names(reintroduction)
```

```
## [1] "Status"     "Band_Color" "Age_Months"
```

```
colnames(reintroduction) <- c("Band_Status", "Breeding_Cohort")
head(reintroduction)
```

Band_Status	Breeding_Cohort	NA
Wild_Hatch	None	11
Wild_Hatch	None	5
Wild_Hatch	None	5
Wild_Hatch	None	5
Wild_Hatch	None	10
Wild_Hatch	None	3

 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.

```
reintroduction2 <- field_sightings     #  Always copy first
names(reintroduction2)[which(names(reintroduction2) == "Status")] <- "Band_Status"
names(reintroduction2)[which(names(reintroduction2) == "Band_Color")] <- "Breeding_Cohort"
```

```
head(reintroduction2)
```

Band_Status	Breeding_Cohort	Age_Months
Untagged	None	11
Untagged	None	5
Untagged	None	5
Untagged	None	5
Untagged	None	10
Untagged	None	3

7. 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.

```
# Before
trial_vec
```

```
## [1]  1 NA NA  1  2 NA NA  3 NA  2  3  3  1 NA NA  2 NA NA  2 NA  1 NA NA NA  1
```

```
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.

```
# After
trial_vec
```

```
##  [1] 1.000000 1.833333 1.833333 1.000000 2.000000 1.833333 1.833333 3.000000
##  [9] 1.833333 2.000000 3.000000 3.000000 1.000000 1.833333 1.833333 2.000000
## [17] 1.833333 1.833333 2.000000 1.833333 1.000000 1.833333 1.833333 1.833333
## [25] 1.000000
```

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