Introduction to Data Analysis in R

Catherine Barber

2022-09-08

Goal and Learning Outcomes

Goal: The goal of this lesson is for you to learn to navigate RStudio and conduct basic data analysis tasks in R.

Learning outcomes: During this lesson, you will demonstrate your ability to...

- Create a project
- Install and load packages
- Obtain help on functions
- Examine a dataframe's structure
- Create and append a vector
- Index a dataframe
- Calculate summary statistics for the entire sample and for subgroups
- Create and interpret basic plots (histogram and boxplot)

Overview of R

R is an open-source language and programming environment designed to facilitate statistical analysis and graphic representation of data (R Project, n.d.). It was originally developed by Ross Ihaka and Robert Gentleman from the University of Auckland, and it has been managed by the R Core Team since 1997 (Dalgaard, 2008).

R has many benefits, including its extensive visualization and computing capacities, flexibility, and open-source format; these features make R cost-effective and superb for highly customized analyses. R compares favorably to other tools, including Python, SAS, and SPSS, for both explanatory and predictive data science tasks (Kromme, 2017).

Challenges include a steep learning curve, particularly compared to GUI-based tools. However, numerous free resources exist to help you learn R.

R Lingo

There are a few terms to learn as you become familiar with R lingo:

- Function: chunk of code that produces a result.
- Argument: specification about the data that a function should be performed on and details of how that function should be performed.
- Value: datapoint (such as a number or string).
- Variable: container for one or more values (alternatively, measured dimension in a dataset).
- Call: execute a command in the code (as in "call a function on an argument").

- Object: data structure, such as a vector, matrix, or dataframe.
- Package: collection of codes that enhance R's capabilities.
- Data type: quality of the values in a variable, e.g., logical (Boolean: TRUE, FALSE, NA), integer (whole numbers), numeric (decimals or doubles), and character (string).

In this document, input (i.e., a command) is shown in a `shaded box`. Output (i.e., a result) is shown in a clear box and is preceded with ## (and sometimes a number in brackets, which you can ignore).

RStudio

This lesson assumes that you have downloaded the latest version of R and RStudio. If you need assistance, please view one of the following resources from YouTube (developed by Nick Paterno through openintro.org):

- How to install R and RStudio on Windows
- How to install R and RStudio on Mac OS X

RStudio is an integrated development environment used to work with R. Its GUI features make it user-friendly.

To begin, open RStudio on your device and examine the environment. The first time you use RStudio, you will probably see three panes:

- a command console, where commands are executed
- the global environment, which summarizes the objects currently stored in R's working memory during a session
- a "miscellaneous" pane, which includes a list of files in the working directory, a viewing area where plots appear, a list of packages available in the library, and a help tab.

A fourth pane, which you can open with Ctrl/Cmd+Shift+N or by clicking New File and R Script, is the script editor.

Starting with the Basics

Exercise 1: Create, Run, and Save a Project

In keeping with good workflow practices, your first task is to create, run, and save a project.

1. Create a new project by clicking File then New Project.

- Choose New Directory and click New Project.
- Name the directory something memorable (e.g., first_R_project).
- Select a sub-directory where the new directory will reside (e.g., Desktop).
- Click Create Project.

2. Check the working directory.

- Input `getwd()`.
- The output will be a path to the working directory you created for the project.
- You can also check the path in the Files pane.

3. Use the command console.

- In the console pane, input the following commands and hit Enter/Return after each line.
- You should obtain the output shown directly below each of the commands.

```
2 * 2
```

```
## [1] 4
```

```
3 / 4
```

```
## [1] 0.75
```

```
10 ^ 10
```

```
## [1] 1e+10
```

```
print("I love statistics!")
```

```
## [1] "I love statistics!"
```

- Now try the following command.

```
print(Good morning.)
```

```
## Error: <text>:1:12: unexpected symbol
## 1: print(Good morning.
##                  ^
```

What happened? Why did you get an error? Hint: Compare this command to the previous one. What is the latter missing?

Quotation marks! A string must be enclosed in quotation marks for the function to recognize it as an argument.

4. Use the script editor.

- Click File, then New File, then R Script (or use Ctrl+Shift+N).
- Input your choice of three new commands and hit Enter/Return.
- What happened? Nothing! Unlike in the command console, Enter/Return is not sufficient to run the code in the script editor.
- Now try Ctrl/Cmd+Enter at the end of the last line. What happened? The last command is executed!
- After practicing a bit more, save the script. It becomes part of your R project and is ready for later use.

Tips for RStudio

- Use the broomstick icon to clean up panes.
- Use the script editor rather than the command console for input.
- Comment on your code using #.
- Look for errors in your code.
- Punctuation and capitalization matter! Common issues:
  - Not closing parentheses/brackets.
  - Not including quotation marks when needed.
  - Misspelling names of functions or variables.

Exercise 2: Install and Load a Package

Although base R contains many powerful functions, R is greatly enhanced by dozens of packages that have been developed by programmers worldwide. When you find R code online, you will often discover that you need to install an R package before the code will run. To install, use the function `install.packages()` with the package name in quotation marks inside the parentheses.

Let's practice by installing and loading the tidyverse package, which is actually a family of packages including the popular dplyr and ggplot2, among others. Install the package using `install.packages("tidyverse")`. R will evaluate this command and return output showing that the package has successfully installed.

Next, load the package using the `library()` function.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6      v purrr   0.3.4
## v tibble  3.1.8      v dplyr   1.0.10
## v tidyr   1.2.0      v stringr 1.4.1
## v readr   2.1.2      v forcats 0.5.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

R will evaluate this command and return output showing the tidyverse sub-packages loaded from the R library are now available to use. You may see some warning messages indicating the version of R under which the tidyverse package was created, and any conflicts between earlier and later versions of the package.

Now that you have loaded a package, let's look at R's built-in datasets. Some of these are in base R, while other datasets are in a specific package. Call `data()` and see what happens.

A new pane labeled R data sets opens in the script editor. Note that the datasets are grouped by package.

Exercise 3: Get Help in R

You are almost ready to start analyzing data. There are a couple of important functions that you will find useful for getting help in R. The first is `help()`, which can be called on any function. Alternatively, you can use a question mark before the function name (no space or parentheses). For example, both `help(mean)` and `?mean` will return help documentation on the function `mean()`.

Another useful function is `args()`. Try calling args on mean and see what happens.

```
args(mean)
```

```
## function (x, ...)
## NULL
```

The output is a reminder of the arguments that the function requires. In this case, `mean()` requires a single argument: the variable (here labeled x) whose mean you want to calculate. Note that the ellipsis (. . .) indicates that the function can take additional arguments, but these are not required. Ignore the word "NULL" on the second line of output, as it is not relevant to the current discussion.

practice_script.R* × R data sets ×

```
Data sets in package 'datasets':

AirPassengers        Monthly Airline Passenger Numbers
                     1949-1960
BJsales              Sales Data with Leading Indicator
BJsales.lead (BJsales)
                     Sales Data with Leading Indicator
BOD                  Biochemical Oxygen Demand
CO2                  Carbon Dioxide Uptake in Grass Plants
ChickWeight          Weight versus age of chicks on different
                     diets
DNase                Elisa assay of DNase
EuStockMarkets       Daily Closing Prices of Major European
                     Stock Indices, 1991-1998
Formaldehyde         Determination of Formaldehyde
HairEyeColor         Hair and Eye Color of Statistics Students
Harman23.cor         Harman Example 2.3
Harman74.cor         Harman Example 7.4
Indometh             Pharmacokinetics of Indomethacin
InsectSprays         Effectiveness of Insect Sprays
JohnsonJohnson       Quarterly Earnings per Johnson & Johnson
                     Share
LakeHuron            Level of Lake Huron 1875-1972
LifeCycleSavings     Intercountry Life-Cycle Savings Data
Loblolly             Growth of Loblolly pine trees
Nile                 Flow of the River Nile
Orange               Growth of Orange Trees
OrchardSprays        Potency of Orchard Sprays
PlantGrowth          Results from an Experiment on Plant
                     Growth
```

Figure 1: Sample Datasets in R

Exercise 4: Access a Dataset

Now you are ready to access and work with a dataset! Data can contain various types of data: character (also known as string), integer (whole numbers), numeric (also known as double or float, which are numbers that contain decimals), and logical (also known as Boolean). It is important to know what type of data each variable in a dataset contains, as this will affect what analyses you can perform.

For this lesson, you will use a dataset that "lives" in R.

Scenario: A Strange Experiment in Plant Biology

Figure 2: Carnivorous Plant; Image by Etheriel, Pixabay open license

Imagine you are interning in a plant biology lab. You are working on a strange experiment in which you have been feeding a rare species of carnivorous plant three types of food. Plant specimens in the control group receive standard plant food for their species–insects and such. Plant specimens in treatment group 1 receive cat food. And plant specimens in treatment group 2 receive dog food.

In this very bizarre experiment, the plants have been growing outrageously big. Your team has completed the experimental phase and recorded the data in a spreadsheet called PlantGrowth.

Input the dataset name and examine the output.

```
PlantGrowth
```

```
##    weight group
## 1    4.17  ctrl
## 2    5.58  ctrl
## 3    5.18  ctrl
## 4    6.11  ctrl
## 5    4.50  ctrl
## 6    4.61  ctrl
## 7    5.17  ctrl
## 8    4.53  ctrl
## 9    5.33  ctrl
## 10   5.14  ctrl
## 11   4.81  trt1
## 12   4.17  trt1
## 13   4.41  trt1
## 14   3.59  trt1
## 15   5.87  trt1
## 16   3.83  trt1
## 17   6.03  trt1
```

```
## 18   4.89  trt1
## 19   4.32  trt1
## 20   4.69  trt1
## 21   6.31  trt2
## 22   5.12  trt2
## 23   5.54  trt2
## 24   5.50  trt2
## 25   5.37  trt2
## 26   5.29  trt2
## 27   4.92  trt2
## 28   6.15  trt2
## 29   5.80  trt2
## 30   5.26  trt2
```

This command simply prints the entire dataset in the console. This can be handy for small datasets like the one you are working with in this scenario, but for large datasets, you may need a different way to view the data, which will be discussed shortly.

Next, call `help(PlantGrowth)` to examine the metadata.

The documentation in the Help pane provides you with information about the dataset. You can see that the sample size is 30, and there are two variables: weight (a continuous variable) and group (a categorical variable or factor).

Note that `help()` provides real background information about the dataset. In the case of `PlantGrowth`, the experiment involved measuring the weight (presumably in ounces or grams) of dried plants. However, in keeping with the scenario, imagine that the plants are live, their weight is measured in pounds, and the experimental conditions are as previously described.

Now call `view(PlantGrowth)`.

This command causes a small spreadsheet-style pane to open in the script editor area. This can be handy if you have many variables or if you want to look at how the dataset is organized in spreadsheet form.

R Object #1: The Dataframe

Recall that an object is a data structure in R. The first object to discuss is the dataframe, which is a two-dimensional structure of rows and columns. If you are used to spreadsheets and other rectangular formats for data, this will be familiar to you.

The dataframe is the most complex object in R because it includes features of other objects while also having its own unique features. For example, it can contain different data types.

The key to a clean dataframe is for each row to represent an observation and each column to represent a variable (that is, something that was measured).

Assigning Data to a Named Dataframe

Assignment is an important concept in R. It involves giving a name to an object and making it re-usable in codes. Here, you will assign the `PlantGrowth` dataset to a dataframe object using the assignment operator `<-` and the `data.frame()` function, with the dataset name as the argument.

Tips for Assignment and Dataframes

- Do not put a space between < and - in the assignment operator.

- When choosing an object name, don't start with a number, keep the names memorable, and use underscores or hyphens instead of spaces.
- In the script editor, you can use Alt+ - as a shortcut to create the assignment operator.
- There are a couple of key arguments to be aware of in `data.frame()`:
  - `check.names = TRUE` tells R to check the dataset column headings for the names of the variables. This argument is true by default (and thus can be omitted from the command). However, if you don't want R to use the column headings as variable names, simply change this argument to `check.names = TRUE`.
  - `stringsAsFactors = FALSE` tells R **not** to treat string variables as factors. This argument is false by default, but if you want R to treat all string variables as factors, change this argument to `stringsAsFactors = TRUE`.

Exercise 5: Assign Data to a Dataframe

In this exercise, you will assign the `PlantGrowth` dataset to an object named `plants`, which will be a dataframe. The `PlantGrowth` dataset column headings will be the names of the variable, so that argument can be omitted. However, you want R to treat the group column containing string data as a factor, so you will add that argument to the command.

```
plants <- data.frame(PlantGrowth, stringsAsFactors = TRUE)
```

Note that you won't see anything in the console output other than the command itself. However, in the Global Environment pane, you should see a new object named `plants`, with a bit of information that you already knew from your preliminary steps: the dataframe contains 30 observations of 2 variables. It's good practice to check the Global Environment after creating and assigning objects to make sure you haven't made any errors.

Exercise 6: Examine the Dataframe

Inputting the dataframe name provides the first few rows and columns (or the entire dataset, if small). Try inputting `plants` and see what happens. You should see all of the data in the console.

Next, look at the structure of the object with `str()`.

```
str(plants)
```

```
## 'data.frame':    30 obs. of  2 variables:
##  $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
##  $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1 ...
```

The output confirms that `plants` is a dataframe consisting of two variables (indicated with $). In this case, the output is no surprise–after all, you created the dataframe object! But if you are working with imported data or with borrowed code, this function can be handy.

Note that `str()` also provides information about the variables: weight is numeric and group is factor. Remember how you set `stringsAsFactors = TRUE` when you created the `plants` dataframe? The result is that R converted the string data in the groups variable to factors. This is very useful for categorical data, such as group membership.

You now have a good sense of your dataframe. You have the weights of 30 carnivorous plants and the experimental group each plant was assigned to.

Just as you were starting to think about analyzing your data, your lab partner informs you that some new data have just been collected on each plant and that you need to add those data to your dataset. Yikes! Do you have to start all over?

No, thank goodness. You can just create a new R object: a vector.

R Object #2: The Vector

A vector (or, more technically, an atomic vector) is a series of values. It is the most basic R object and contains a single data type: logical, integer, numeric (double), or character.

The most common way to create a vector is with the concatenation function `c()`. You assign the vector to a name and enter the data in parentheses, with each datapoint separated by a comma. Note that if you have character data, you must enter the data with quotation marks around each string.

Note: Some functions return a vector of values as the output, as you will see shortly.

Exercise 7: Create a Vector and Append It to a Dataframe

Let's get back to the scenario. Your lab partner noted that the new data represent the number of whole leaves on each plant. Because you only have 30 observations, you can easily create a vector containing these observations. Your lab partner hands you a piece of paper on which the leaves data are recorded.

Plant #	Leaves	Plant #	Leaves
1	1	16	15
2	2	17	14
3	3	18	13
4	4	19	12
5	5	20	11
6	6	21	10
7	7	22	9
8	8	23	8
9	9	24	7
10	10	25	6
11	11	26	5
12	12	27	4
13	13	28	3
14	14	29	2
15	15	30	1

Figure 3: Table of Leaves Data

You could enter the data using `leaves <- c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,15,14,13,12,11,10,9,8,7,6,5,4,`

This would create a vector of the values and assign it to `leaves`.

However, consistent with the strange nature of the experiment, you discover as you look at the data that the number of leaves seems to follow a pattern. Plants 1-15 have 1 through 15 leaves, in ascending order,

while plants 16-30 have 1 through 15 leaves in descending order. In the real world, your data will probably not follow a known sequence, so you would need to input values separated by commas.

Nonetheless, in the current scenario, you can capitalize on the sequence inherent in the data by using a colon when creating the vector. The colon is placed between the first value and the last value in a sequence and tells R to select all values between (and inclusive of) the first and the last.

In the following input, parentheses around the entire command will call the function and return the output in a single command.

```
(leaves <- c(1:15, 15:1))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 15 14 13 12 11 10  9  8  7  6
## [26]  5  4  3  2  1
```

The output displays the leaves vector in the command console, and you also see "leaves" appear in the Global Environment. Note that the data type for leaves is integer because you did not include any decimals in the vector values. That makes sense, as your lab partner only counted whole leaves, not fractions of leaves.

Now you are ready to append this vector to your dataframe by using the `cbind()` function:

```
(plants <- cbind(plants, leaves))
```

```
##    weight group leaves
## 1    4.17  ctrl      1
## 2    5.58  ctrl      2
## 3    5.18  ctrl      3
## 4    6.11  ctrl      4
## 5    4.50  ctrl      5
## 6    4.61  ctrl      6
## 7    5.17  ctrl      7
## 8    4.53  ctrl      8
## 9    5.33  ctrl      9
## 10   5.14  ctrl     10
## 11   4.81  trt1     11
## 12   4.17  trt1     12
## 13   4.41  trt1     13
## 14   3.59  trt1     14
## 15   5.87  trt1     15
## 16   3.83  trt1     15
## 17   6.03  trt1     14
## 18   4.89  trt1     13
## 19   4.32  trt1     12
## 20   4.69  trt1     11
## 21   6.31  trt2     10
## 22   5.12  trt2      9
## 23   5.54  trt2      8
## 24   5.50  trt2      7
## 25   5.37  trt2      6
## 26   5.29  trt2      5
## 27   4.92  trt2      4
## 28   6.15  trt2      3
## 29   5.80  trt2      2
## 30   5.26  trt2      1
```

Note that the output appears in the console and that the object `plants` has changed in the Global Environment. The dataframe now has three variables: weight, group, and leaves. Use caution when assigning an object to the name of an existing object, as it will overwrite that object–which you may or may not want. In the current scenario, you want `plants` to contain all three variables, so it's okay to overwrite.

Tips for Vectors

- For vectors containing strings, always put quotation marks around the strings.
- Be sure you understand the properties of your objects before working with them. For example, if your lab partner had accidentally given you 29 values rather than 30 and you created a vector with 29 values, R would recycle the vector and start over with the first value when it reached observation 30 in your dataframe.

Indexing a Dataframe

Indexing (also known as subsetting) involves selecting part of the dataframe. You can select a single observation or multiple observations, based on your criteria.

The general format for a single observation is `df_name[r,c]`, where df_name is the dataframe name, r is the row number, and c is the column number for the observation. Note that the row number and column number are placed inside single square brackets and are separated by a comma. In R, indexing begins with 1, so the first row would be 1 and the first column would be 1. Let's practice.

Example

Your lab supervisor rushes in and tells you that plant #4 seems particularly hungry today. She asks you for the plant's weight measurement. Input the code to find the plant's weight.

```
plants[4,1]
```

```
## [1] 6.11
```

Inputting `plants[4,1]` tells R to return the fourth row, first column–namely, the fourth plant's weight: 6.11 pounds!

Exercise 8: Select All Values in a Column

Now you want to move beyond a single observation to select multiple observations. Start by selecting all values in a column. There are three methods for doing this:

1. Double brackets and the column number

```
plants[[1]]
```

```
##  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
## [16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

2. Double brackets and the column "name"

Figure 4: Carnivorous Plant; image by Ben Paul, Pixabay open license

```
plants[["group"]]
```

```
##  [1] ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt1
## [16] trt1 trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2
## Levels: ctrl trt1 trt2
```

3. $ between the dataset name and the column name

```
plants$leaves
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 15 14 13 12 11 10  9  8  7  6
## [26]  5  4  3  2  1
```

Looking at the output of the three commands, what do you notice? All three of these methods return a vector of values.

Thinking ahead, you decide to store weight as a vector because you might want to perform some calculations on weight later. Re-run the first code `plants[[1]]` and assign it to an object named `weight` using the assignment operator `<-`. Place parentheses around the entire code to view the output in the command console.

```
##  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
## [16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

This assignment returns the vector of weights you have saved as `weight`. Be sure to check the Global Environment to confirm that the object `weight` appears there as well.

Exercise 9: Select Multiple Columns

You are ready for a more complex task. The graduate fellow in your lab has been walking amongst the plants and noting any that seem unusual. He thinks plant #14 seems a little small and wants to see the weight and group for this plant. You will need to subset row 14, columns "weight" and "group." This involves specifying the row number and then using concatenation for the names of the columns.

```
plants[14, c("weight", "group")]
```

```
##    weight group
## 14   3.59  trt1
```

The output is a tiny dataframe–one row and two columns, representing the weight (3.59) and the group (trt1) for plant 14.

The fellow notices another plant that is smaller than those around it. He asks you to subset the data for these two small plants: 14 and 16. Because you need multiple rows, you will use concatenation for the row numbers. In addition, because you want all columns (weight, group, and leaves), you can simply place a comma after the row number and leave the column area blank. This tells R that you want all columns.

```
plants[c(14, 16),]
```

```
##    weight group leaves
## 14   3.59  trt1     14
## 16   3.83  trt1     15
```

Once again, the output is a tiny dataframe, with two rows and three columns. The data are interesting– although these two plants seem to have smaller weights, they have quite a few leaves.

The fellow decides to look at all of the plants in terms of their weight and number of leaves. for now, group is not important. You don't need to specify row (because you want all rows), so you leave that area blank and just specify the two columns you want, using concatenation and quotation marks.

```
plants[,c("leaves", "weight")]
```

```
##    leaves weight
## 1       1   4.17
## 2       2   5.58
## 3       3   5.18
## 4       4   6.11
## 5       5   4.50
## 6       6   4.61
## 7       7   5.17
## 8       8   4.53
## 9       9   5.33
## 10     10   5.14
## 11     11   4.81
## 12     12   4.17
## 13     13   4.41
## 14     14   3.59
## 15     15   5.87
## 16     15   3.83
## 17     14   6.03
## 18     13   4.89
## 19     12   4.32
## 20     11   4.69
## 21     10   6.31
## 22      9   5.12
## 23      8   5.54
## 24      7   5.50
## 25      6   5.37
## 26      5   5.29
## 27      4   4.92
## 28      3   6.15
## 29      2   5.80
## 30      1   5.26
```

Notice that in this command you changed the order of leaves and weight so that leaves would appear as the first column of output. Once again, you have a dataframe. Any time you index more than one column, the results will be returned as a dataframe.

Parsing Index Punctuation

Let's pause for a moment to discuss indexing and punctuation. This can be one of the trickier aspects to understand, so we will delve a bit into what is going on behind the scenes in R. Feel free to follow along by inputting the commands and looking at the output.

The most basic case is indexing a single observation for a single variable. For example, if you input:

```
plants[1,1]
```

```
## [1] 4.17
```

R returns a single value: 4.17. Technically, this single value is a vector of plant weight for the first row.

The next most basic case is indexing all observations on a single variable. For example, if we input:

```
plants[[1]]
```

```
##  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
## [16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

R returns a vector of plant weights. R interprets the command as "subset all values in the weight column (i.e., column 1)." Remember that you can put the column name (rather than the number) in quotation marks as an alternative.

Another way to index all observations on a single variable and return a vector is to include a comma but not specify the row:

```
plants[,1]
```

```
##  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
## [16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

R interprets this command as "subset all values in the weight column" as well. This is a good example of the truism that there are many ways to do the same thing in R; choose the method that works best for you.

Once you start indexing multiple variables, R will always return a dataframe. For example, if you input:

```
plants[1,]
```

```
##   weight group leaves
## 1   4.17  ctrl      1
```

R interprets this as "subset all columns and their values for the first row." Thus, you get a tiny dataframe with one row and three columns.

Finally, you could simply put the column number in single brackets:

```
plants[1]
```

```
##    weight
## 1    4.17
## 2    5.58
## 3    5.18
## 4    6.11
## 5    4.50
## 6    4.61
## 7    5.17
```

```
## 8    4.53
## 9    5.33
## 10   5.14
## 11   4.81
## 12   4.17
## 13   4.41
## 14   3.59
## 15   5.87
## 16   3.83
## 17   6.03
## 18   4.89
## 19   4.32
## 20   4.69
## 21   6.31
## 22   5.12
## 23   5.54
## 24   5.50
## 25   5.37
## 26   5.29
## 27   4.92
## 28   6.15
## 29   5.80
## 30   5.26
```

This is a special case. The lack of a comma makes R treat the index differently. It interprets the command as "subset the weights column, including the column structure." Because the column exists within the dataframe structure, the output will be a dataframe with numbered rows and the values for the indexed column, in this case plant weights.

Tips for Indexing

- To keep things simple, use the $ method to specify columns whenever the columns are named.
- Remember that a comma at the end of an index will always return a dataframe.

Using a Criterion to Select Rows

So far, you have been subsetting based on plant number. But what if you want to select observations that meet a certain criterion, such as group membership or numeric value for one or more measures? You can combine index elements to indicate the criterion.

Here are the steps:

- Use $ to indicate the column to which the criterion applies (e.g., plants$weight).
- Use == " " to specify the value of a column with a character data type (e.g., == "ctrl"). The double equal sign in the command indicates a conditional that means "is exactly the same as," which R uses to classify observations as TRUE or FALSE.
- Include a comma at the end of the index (inside the brackets) to return all columns.
- Option: Include a vector of column numbers or "names" to return specified columns.

Let's practice.

Exercise 10a: Select Rows Based on a Criterion

You want to look at all of the data for the control group–the plants that ate the usual plant food (flies, etc.).

Input:

```
plants[plants$group=="ctrl",]
```

```
##    weight group leaves
## 1    4.17  ctrl      1
## 2    5.58  ctrl      2
## 3    5.18  ctrl      3
## 4    6.11  ctrl      4
## 5    4.50  ctrl      5
## 6    4.61  ctrl      6
## 7    5.17  ctrl      7
## 8    4.53  ctrl      8
## 9    5.33  ctrl      9
## 10   5.14  ctrl     10
```

The output returns all columns for the observations in the ctrl (i.e., control) group.

What would happen if you left off the comma?

```
plants[plants$group=="ctrl"]
```

```
## Error in `[.data.frame`(plants, plants$group == "ctrl"): undefined columns selected
```

Why did you receive an error message? When you use == to tell R which rows you want based on a column criterion, R looks at the specified column (in this case, "group") and assigns TRUE to all observations that meet the criterion (i.e., group == "ctrl"). R then looks for the comma to indicate which columns you want to include in the output.

If you have a comma at the end with no specified columns (as was true in the first command), R will return all columns for those groups. However, if you leave off the comma, R returns an error message saying that you have not defined the columns to select for the output.

Exercise 10b: Select Rows Based on a Criterion

Let's try another criterion. Your lab partner is running a small experiment on plants that have an even number of leaves, and they want to know which plants meet that criterion. The criterion is a little more complex, as you will need to provide a formula for R to determine which plants have an even number of leaves. Fortunately, the modulus operator %% does the trick.

The modulus calculates the remainder in a division equation. For example, $2\%\%2 = 0$, because 2 is divisible by 2 with no remainder. Thus, if you want all values that are even (i.e., divisible by 2 with no remainder), you can set the modulus to 0. Try this input:

```
plants[plants$leaves %% 2 == 0, c("weight", "leaves")]
```

```
##    weight leaves
## 2    5.58      2
## 4    6.11      4
```

```
## 6    4.61      6
## 8    4.53      8
## 10   5.14     10
## 12   4.17     12
## 14   3.59     14
## 17   6.03     14
## 19   4.32     12
## 21   6.31     10
## 23   5.54      8
## 25   5.37      6
## 27   4.92      4
## 29   5.80      2
```

The output is a dataframe of all plants with an even number of leaves, along with their weights and the actual number of leaves. Keep in mind that if you had left off the column names and just included a comma at the end of the index within brackets, R would return all columns for those even-leaved plants.

Exercise 10c: Select Rows Based on a Criterion

Your supervisor is back and wants to look at the weight data for the group that was fed cat food, namely trt1. She asks you to generate a dataframe with the numbers of the plants in group trt1 and their weights. Input:

```
plants[plants$group == "trt1", "weight"]
```

```
##  [1] 4.81 4.17 4.41 3.59 5.87 3.83 6.03 4.89 4.32 4.69
```

Notice that the output is a vector rather than a dataframe. This is because you only selected one column; remember that if you select multiple rows but only one column, R will return a vector. If you want a dataframe instead, you need to change a default argument in the index.

When indexing, R automatically drops whatever it perceives to be "redundant" information, such as row numbers, when you select only one column. That's why you get a vector (rather than a dataframe) when you index a single column. Most of the time, that's what you want! But sometimes, you actually need to know the row numbers. Try adding drop = FALSE as an argument to the previous input.

```
plants[plants$group == "trt1", "weight", drop = FALSE]
```

```
##    weight
## 11   4.81
## 12   4.17
## 13   4.41
## 14   3.59
## 15   5.87
## 16   3.83
## 17   6.03
## 18   4.89
## 19   4.32
## 20   4.69
```

Now you have the dataframe your supervisor has requested!

Exercise 11: Select Observations Using Other Conditionals

You have practiced selecting criteria based on the conditional ==, but you can also use other conditionals such as > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to), & (and), and | (or).

Review this command: `plants[plants$weight > 5,]`

Before you input the command, see if you can parse it and determine what will happen.

Now input the command and view the results:

```
plants[plants$weight > 5,]
```

```
##    weight group leaves
## 2    5.58  ctrl      2
## 3    5.18  ctrl      3
## 4    6.11  ctrl      4
## 7    5.17  ctrl      7
## 9    5.33  ctrl      9
## 10   5.14  ctrl     10
## 15   5.87  trt1     15
## 17   6.03  trt1     14
## 21   6.31  trt2     10
## 22   5.12  trt2      9
## 23   5.54  trt2      8
## 24   5.50  trt2      7
## 25   5.37  trt2      6
## 26   5.29  trt2      5
## 28   6.15  trt2      3
## 29   5.80  trt2      2
## 30   5.26  trt2      1
```

Did you predict that R would return a dataframe of all plants with a weight > 5, including all columns? Good!

Now let's test your understanding of indexing. What would happen if you input the same command but add a 1 after the comma? Guess, then try it!

```
##  [1] 5.58 5.18 6.11 5.17 5.33 5.14 5.87 6.03 6.31 5.12 5.54 5.50 5.37 5.29 6.15
## [16] 5.80 5.26
```

Did you predict that R would return a vector of weights (i.e., column 1)? If so, you are getting a hang of indexing!

Try one more. What do you think would happen if you omit the comma? Guess, and then try it!

```
## Error in `[.data.frame`(plants, plants$weight > 5): undefined columns selected
```

Did you remember that R would return an error message because the lack of a comma indicates that no columns were selected? Great work.

More Tips for Indexing

- Think through what you are selecting and how you will use that selection.
  - Consider whether you want the output to be a vector or a dataframe.
- A dataframe column is not automatically stored as an object.
  - For example, if you call `print(group)`, R will return an error because "group" is not stored as an object outside of the dataframe plants.
  - To perform a function on a column as a variable, use the $ method.
  - Example: call `mean(plants$leaves)` and see what happens. R will return the mean (average) number of leaves for the entire sample because you appropriately specified something R can understand, namely the column `plants$leaves`.
  - Note that in the current scenario, you created a vector named leaves, so you could also call `mean(leaves)`. However, this will only work if the variable exists as an object in the global environment. Otherwise, use the $ method.
- R does not automatically save indexed values.
  - To have access to those values in the future, you must assign them to a named object.
  - Example: call `(leaves_mean <- mean(plants$leaves))` and see what happens in the console and in the Global Environment.
  - You should now have a vector named leaves_mean with one value–8, which is the mean of the variable "leaves."

Indexing on Multiple Criteria Using with()

Let's cover one more aspect of indexing: selecting data based on more than one criterion. In this case, you can combine indices with the `with()` function. The general format is `df_name[with(df_name, variable1 == & variable2 ==),]`. This will return all rows that meet the specified criteria; because you have placed a comma at the end, R will return all columns for those rows. Let's try it.

Exercise 12: Combine Two Indices

Your supervisor asks you to identify all plants in the two treatment groups with weights less than 5 lbs.

Input:

```
sm_trt <- plants[with(plants, weight < 5 & (group == "trt1" | group == "trt2")),]
```

The punctuation in this command is complex! You are specifying that, within the `plants` dataframe, you want all rows with a weight of < 5 and group membership of either trt1 or trt2, and that you want all columns in the output.

You'll need to pay close attention to parentheses. When working with conditionals, R treats parentheses as a cue to perform the operation within the parentheses first. So, R first looks for all rows with a group label of trt1 or trt2. Then, within that subset, it looks for plants with a weight < 5.

Now call `sm_trt` to view the results.

```
##    weight group leaves
## 11   4.81  trt1     11
## 12   4.17  trt1     12
## 13   4.41  trt1     13
## 14   3.59  trt1     14
```

```
## 16   3.83  trt1     15
## 18   4.89  trt1     13
## 19   4.32  trt1     12
## 20   4.69  trt1     11
## 27   4.92  trt2      4
```

The output is a dataframe of the smaller plants (i.e., < 5 lbs) in the two treatment groups–just what your supervisor requested.

Exercise 13: Design an Index to Answer a Question

- Consider what you have learned about indexing so far.

- Formulate a question about the dataframe.
- Design an index to answer that question.

- Assign the index to a variable and run the code.
- Examine the output and summarize your answer.

Here is an example:

Which plants in group trt1 have the largest and smallest weights?

```
plants[plants$group == "trt1", 1, drop = FALSE]
```

```
##    weight
## 11   4.81
## 12   4.17
## 13   4.41
## 14   3.59
## 15   5.87
## 16   3.83
## 17   6.03
## 18   4.89
## 19   4.32
## 20   4.69
```

Summary: Within treatment group 1, plant #14 has the smallest weight (3.59), while plant #17 has the largest weight (6.03).

Summary Statistics

Researchers are often interested in summary statistics, such as the mean and standard deviation, for the entire sample and for various subgroups. R provides functions for calculating these statistics quickly.

Exercise 14: Calculate Summary Statistics

Recall that in Exercise 8, you created a vector called `weight` that contains the weights of all of the plants. Call the following:

```
mean(weight)
```

```
## [1] 5.073
```

```
median(weight)
```

```
## [1] 5.155
```

```
sd(weight)
```

```
## [1] 0.7011918
```

```
min(weight)
```

```
## [1] 3.59
```

```
max(weight)
```

```
## [1] 6.31
```

R returns the results of these statistical calculations: mean, median, standard deviation, minimum, and maximum for the entire sample. This is often a good starting point for describing a dataset.

Tips for Object Names vs. Strings

- If you are calling a statistic (or any function) on an object in the Global Environment, you do not need to use quotation marks around the object name. Quotation marks around a word indicate a string. In the current exercise, you are technically calling functions on the object `weight`, not on the column "weight" within the `plants` dataset, even though `weight` and "weight" contain the same data.
- This is a good reminder to choose object names carefully.

Exercise 15: Summarizing Based on Indices

More often than not, you will also want to calculate summary statistics for various subgroups in your dataset, such as experimental groups in the current study.

Your lab supervisor wants the mean and standard deviation of each plant group's weight. Let's think through how to obtain these statistics. Here is one pipeline:

- Index the dataset to find the set of weights for each group.
- Use assignment `<-` to name (and keep track of) each group's set of weights.
- Find the mean and sd for each group.

```
control <- plants[plants$group == "ctrl", "weight"]
treat1 <- plants[plants$group == "trt1", "weight"]
treat2 <- plants[plants$group == "trt2", "weight"]
mean(control)
```

```
## [1] 5.032
```

```
sd(control)
```

```
## [1] 0.5830914
```

```
mean(treat1)
```

```
## [1] 4.661
```

```
sd(treat1)
```

```
## [1] 0.7936757
```

```
mean(treat2)
```

```
## [1] 5.526
```

```
sd(treat2)
```

```
## [1] 0.4425733
```

This series of commands returns six values–one mean and one standard deviation per group.

But... this method is quite inefficient! Is there another option? Yes! `tapply()`

Exercise 16a: Use tapply() to Repeat a Function

The function `tapply()` applies a specified function to each group specified in the argument. For example, your supervisor asked you to find the mean for each experimental group. Recall that the mean is a function that takes one required argument: a vector of values. Thus, in this case, `tapply()` takes three arguments:

- The first argument is the vector you will apply the function to (in this example, `plants$weight`, which is a vector of weights).
- The second argument is the factor that specifies how `tapply()` should group the data before applying the function (in this example, `plants$group`, which you specified as a factor when you originally created the dataframe by setting `stringsAsFactors=TRUE`).
- The third argument is the function to be applied (in this example, mean).

Let's try it:

```
(means <- tapply(plants$weight, plants$group, mean))
```

```
##  ctrl  trt1  trt2
## 5.032 4.661 5.526
```

The output is a one-dimensional array of means–one per group. This is slightly different from a vector, as the values have attributes (namely, the group name). Notice that R stored this new object, `means`, in the Global Environment.

Exercise 16b: Use tapply() to Repeat a Function

Now find the standard deviation for each group using `tapply()` and assign these values to an object named sds. Check your results against the output below.

```
##      ctrl      trt1      trt2
## 0.5830914 0.7936757 0.4425733
```

Did you obtain the correct output? If not, input `(sds <- tapply(plants$weight, plants$group, sd))` to obtain the correct results.

Tips for tapply()

- Make sure that the function you want to apply (such as means) is suitable for the vector of values you have selected. For example, if you had another categorical variable besides group, you would not call `mean()` on that variable.
- `tapply()` requires a factor variable in your dataset.

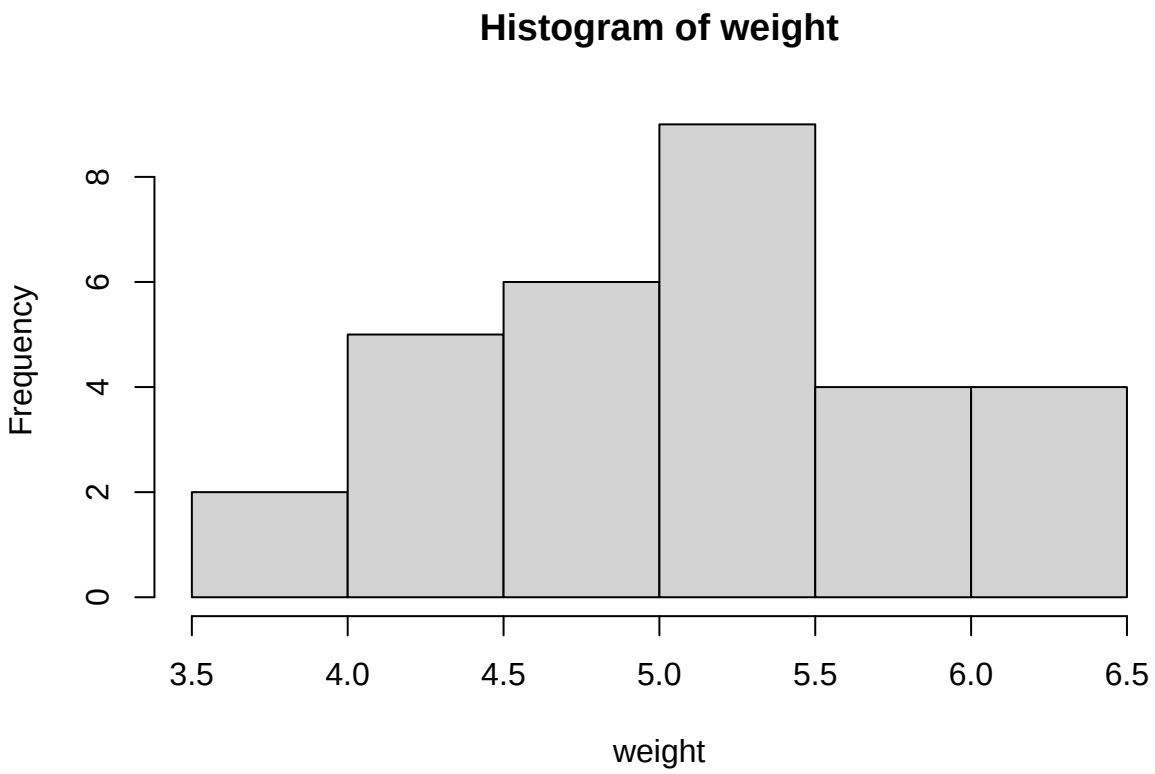
Basic Plots

So far, you have worked with descriptive statistics. Now you are ready to do some basic plotting. R has built-in plot functions that are adequate for initial data analysis. For example, you can create a histogram, boxplot, or scatterplot with base R. However, there are many packages (such as ggplot2 in the tidyverse) that are designed to create fantastic plots, so don't stop with base R!

Exercise 17a: Create a Histogram

A histogram puts data into "bins" based on their values and plots the frequency of values in each bins. The most basic format is `hist(df_name)`; however, there are additional optional arguments that will change the look of the graph. First try a basic histogram of the vector `weight`.

```
hist(weight)
```



R returns a basic histogram that indicates that most of the plant weights are between 4.0 and 5.5 pounds.

Note that when you don't specify any additional arguments, R uses the defaults. For example, the default number of breaks (which is R's term for bins) is based on Sturge's Rule, which provides a good representation of the data when the data are not heavily skewed and when there are ample (but not too many) observations, such as 30-200 (Glen, n.d.).

This default may be fine, but you should try a few different break values to see how the plot changes. Add the argument `breaks = x` where x is the approximate number of breaks; note that R may adjust this number somewhat, based on an internal algorithm.

In addition, the basic plot has a slightly strange y axis–the tallest column surpasses the maximum y value. You can fix that by adding the argument `ylim = c(a,b)` where a is the minimum (0 is strongly recommended) and b is the maximum.

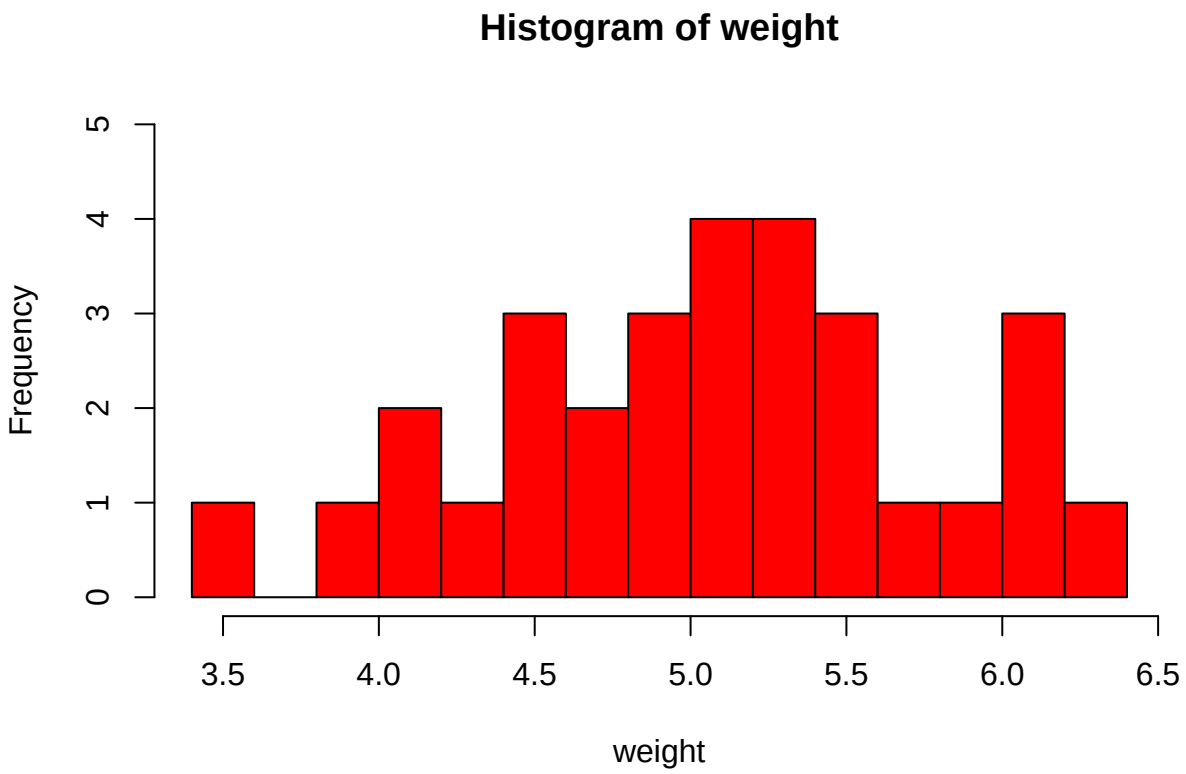
Finally, if gray really isn't doing much for your plot, you can add color with col = "colorname".

Let's try it!

Exercise 17b: Create a Histogram

Add arguments to the original input:

```
hist(weight, breaks = 10, ylim = c(0,5), col = "red")
```

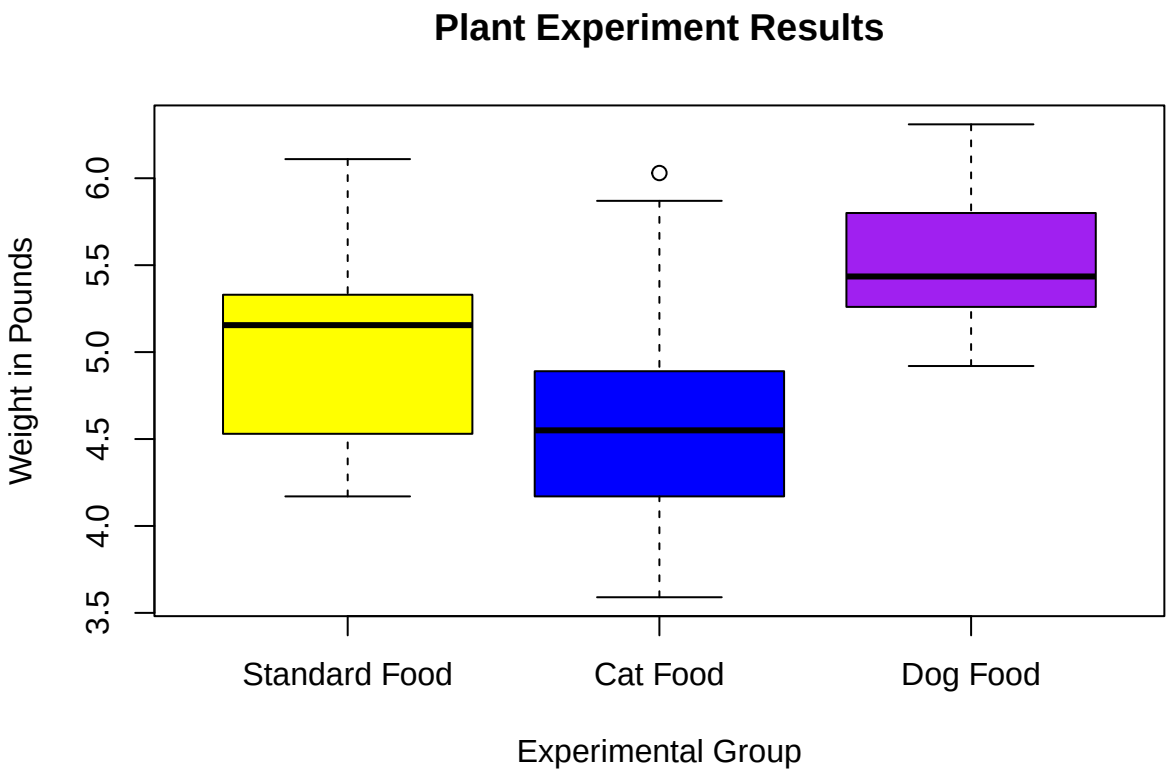


Now the y axis is more appropriate–none of the bars exceed the upper limit. The larger number of bins also gives you more information about the actual distribution of the weights. However, the choice of red may be a little intense for this plot and may not be suitable for accessibility. Consider using the R Color Cheat Sheet for different color ideas and their R codes.

Exercise 18: Create a Boxplot

A boxplot can be useful for comparing groups' summary statistics. The basic formula for a boxplot is `boxplot(formula, data = df_name)` but you can add arguments to customize the plot. Let's try it.

```
boxplot(weight~group, data = plants,
        xlab = "Experimental Group",
        ylab = "Weight in Pounds",
        main = "Plant Experiment Results",
        notch = FALSE, varwidth = FALSE,
        col = c("yellow", "blue", "purple"),
        names = c("Standard Food", "Cat Food", "Dog Food"))
```



The output shows three boxes: one per group. The horizontal bar indicates the median (50th percentile) of the group, while the upper and lower limits of each box indicate the interquartile range (75th and 25th percentiles, respectively). The dotted lines are "whiskers," which show the upper and lower limits (maximum and minimum, statistically defined as Q3 + (1.5 * IQR) and Q1 - (1.5 * IQR), respectively). Any dots outside the whiskers are outliers.

From this plot, it appears that the median of treatment group 2 (that is, the dog food group) is larger than the medians of the other two groups, though all three groups have a fair degree of variability. Perhaps the dog food diet is making these plants heavier? This is an interesting hypothesis that you might want to test statistically!

Summary and Quick Guide

This lesson has covered many functions! Here is a quick guide to each function's purpose.

- `getwd()` indicates working directory
- `print()` prints output
- `install.packages()` installs packages
- `library()` loads packages
- `data()` lists data built into R and its packages
- `help()` gets help on a function
- `args()` shows a function's arguments
- `view()` displays the dataset
- `str()` provides an object's structure
- `data.frame()` changes structure to a dataframe

- `c()` concatenates values into a vector
- `cbind()` binds a column to a dataframe
- `[]` indexes/subsets a dataframe
- `&` alternate notation for indexing
- `with()` for indexing on more than one variable
- `mean()` arithmetic mean of a set of values
- `median()` median of a set of values
- `sd()` standard deviation of a set of values
- `min()` minimum value in a set of values
- `max()` maximum value in a set of values
- `tapply()` repeats a function across groups
- `hist()` generates a histogram
- `boxplot()` generates a boxplot

References and Recommended Reading

- Dalgaard, P. (2008). *Introductory statistics with R.* Springer
- Glen, S. (n.d.). *Choose bin sizes for histograms in easy steps + Sturge's rule.* https://www.statisticshowto.com/choose-bin-sizes-statistics/
- Kromme, J. (2017, March 18). *Python & R vs. SPSS & SAS.* https://www.r-bloggers.com/2017/03/python-r-vs-spss-sas/
- R-Project for Statistical Computing. (n.d.). https://www.r-project.org
- Teetor, P. (2011). *R cookbook.* O'Reilly.
- Wickham, H., & Grolemund, G. (2017). *R for data science.* O'Reilly.

Contact Information

Thank you for participating in this lesson. If you have questions, please reach out to cb88@rice.edu.