Paper 224-2023

Open-Source Essentials HOW – R and RStudio®

Wendy Christensen, Ph.D., University of Colorado Anschutz Medical Campus

ABSTRACT

In this Open-Source Essentials Hands-On Workshop, attendees will learn the basics of using the R/RStudio® environment to create and import data sets, manipulate data, combine data sets, and produce summary statistics. Previous experience with these topics in SAS® or other software is useful but not required, and no previous experience with R or RStudio is assumed – all levels of experience are welcome. Attendees of this interactive workshop will access workshop material and complete hands-on exercises in RStudio through Posit Cloud (free account required). Posit Cloud requires an internet-connected computer and access to a web browser; no installations required. After the workshop, materials will be available through Github.

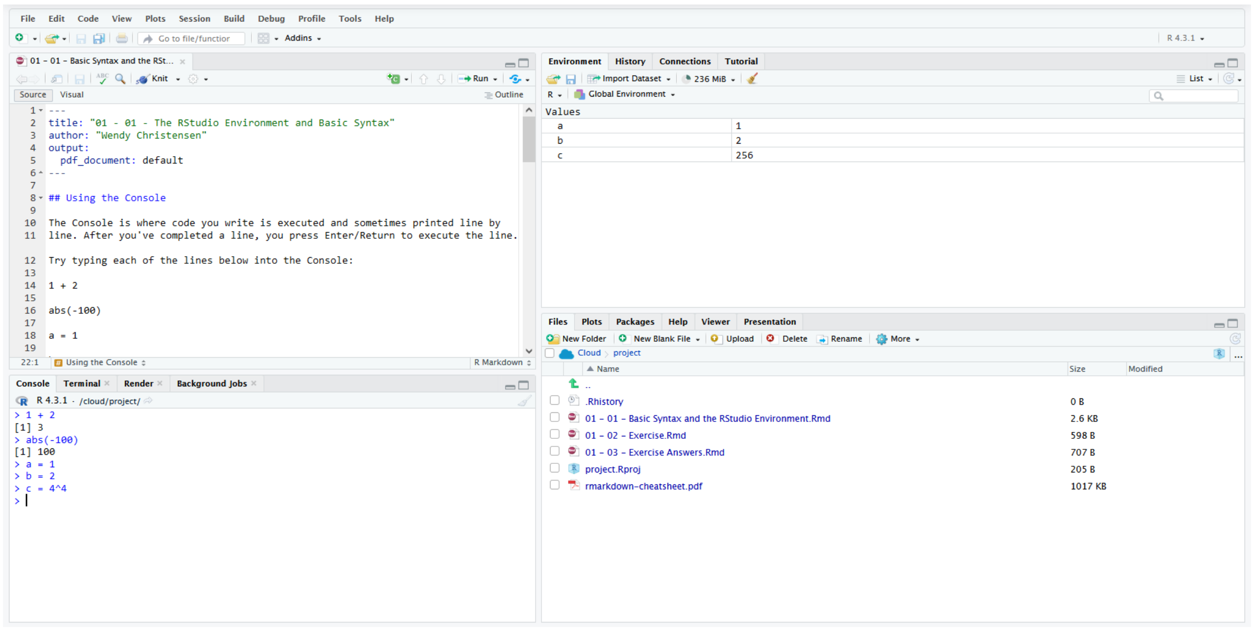
INTRODUCTION

People who work with data today have many choices of software and services for managing, analyzing, and visualizing data. R is a programming language and an open-source software environment that is easy to install on Windows, macOS, and Linux computers. RStudio® is an open-source integrated development environment for R (and other programming languages like Python®) offered by Posit, PBC. The company itself was formerly known as RStudio but changed its name to Posit in 2022. Although R and RStudio are separate programs and must both be installed to use RStudio, many R users today primarily use R through RStudio because it offers a convenient and powerful environment in which to write, execute, and share R code. In this workshop, we will use a cloud version of RStudio hosted on Posit Cloud so we can start working with code as soon as possible.

Much like SAS, it is not possible to master R/RStudio in 90 minutes. The goal of this workshop is for attendees to gain familiarity with how core data tasks – writing R syntax, reading in data, manipulating data, combining data sets, and obtaining summary statistics – can be completed using RStudio. Materials used for this workshop can be found here: https://github.com/wendychristensen/WUSS-2023-Open-Source-Essentials-R-Rstudio. If you access the workshop material on Github, you will first need to install R and RStudio onto your computer to complete the exercises.

THE RSTUDIO ENVIRONMENT AND BASIC SYNTAX

The RStudio environment, a picture of which can be seen in Figure 1, consists of four panes. All panes can be resized by clicking and dragging on the sides of the panes, and they can be maximized and minimized. It is possible to further customize the RStudio environment (Tools -> Global Options -> Pane Layout).



```
---
title: "01 - 01 - The RStudio Environment and Basic Syntax"
author: "Wendy Christensen"
output:
  pdf_document: default
---

## Using the Console

The Console is where code you write is executed and sometimes printed line by
line. After you've completed a line, you press Enter/Return to execute the line.

Try typing each of the lines below into the Console:

1 + 2

abs(-100)

a = 1

```

```
> 1 + 2
[1] 3
> abs(-100)
[1] 100
> a = 1
> b = 2
> c = 4^4
> 
```

```
Values
  a        1
  b        2
  c        256
```

Name	Size	Modified
..		
.Rhistory	0 B	
01 – 01 – Basic Syntax and the RStudio Environment.Rmd	2.6 KB	
01 – 02 – Exercise.Rmd	598 B	
01 – 03 – Exercise Answers.Rmd	707 B	
project.Rproj	205 B	
rmarkdown–cheatsheet.pdf	1017 KB	

Figure 1. The RStudio® Environment. Going clockwise, the top-left is the Source pane, the top-right is the Environments pane, the bottom-right is the Output pane (currently showing the Files tab), and the bottom-left is the Console.

The **Source pane** at the top-left of the environment is where code documents are displayed. Multiple code documents can be open at the same time, with each document shown as a tab in the Source pane. There are many types of code documents that can be executed in the RStudio environment, such as R script files (.r), R Markdown files (.rmd), and Quarto files (.qmd). The coding documents for this workshop are all R Markdown files.

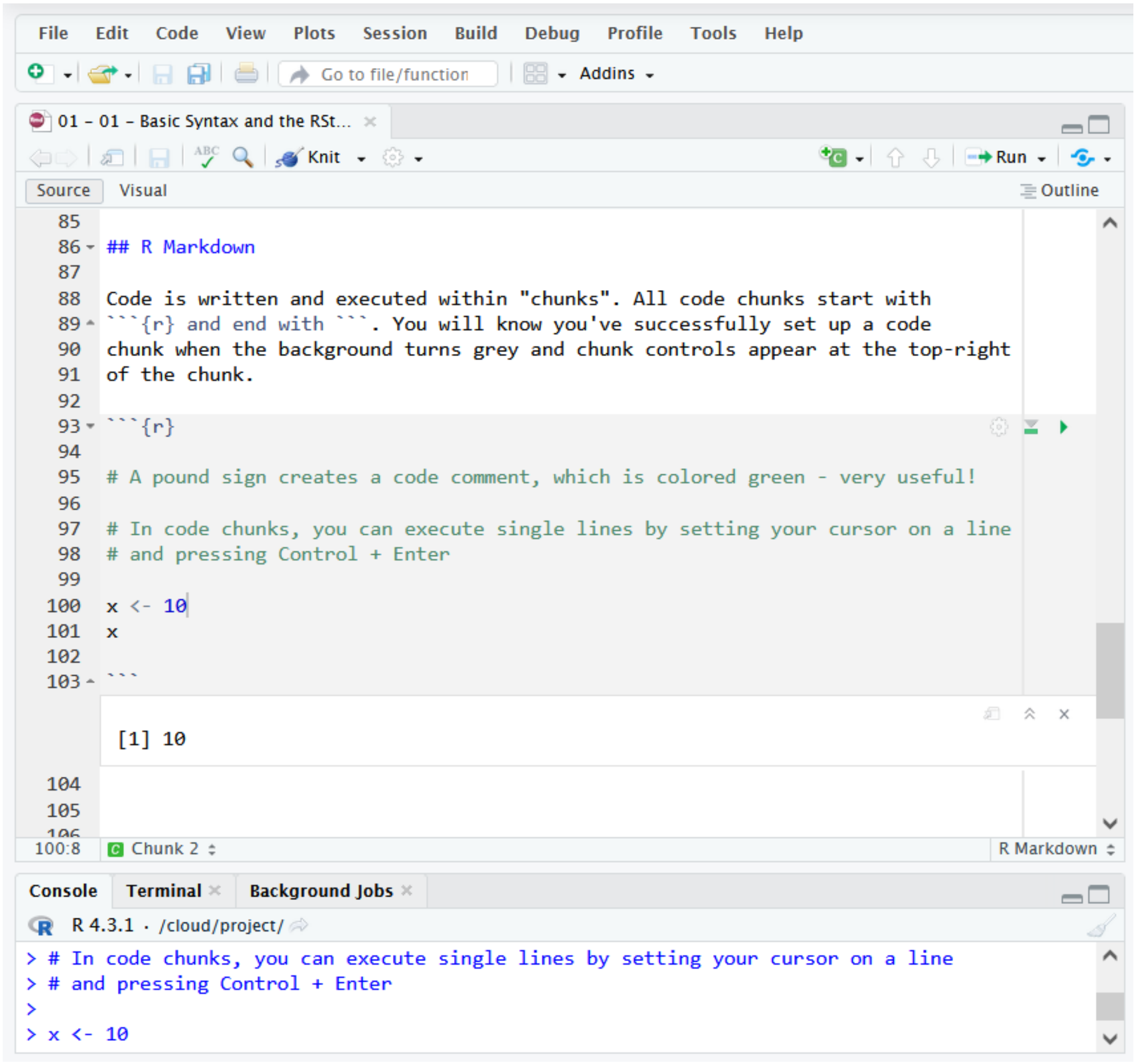
The **Environment pane** at the top-right is where information about the RStudio environment is displayed. The Environment tab shows all the objects (data and values, which will be discussed in later sections) that have been saved to memory during the current session. Data objects can be clicked to get more information about variable types and to display the data. To remove all saved objects, click the broom icon at the top of the environment tab. There are other tabs in the Environment pane, but their functionality will not be used as part of this workshop.

The **Output pane** at the bottom-right is a multifunction pane with six tabs. The Files tab shows the contents of the current directory. When using RStudio through Posit Cloud, each project is itself a self-contained directory that shows all code documents and uploaded files (e.g., data sets for importing) saved to the project. When using RStudio locally, any files and folders in the directory will be shown. The functionality of two other tabs, Packages and Help, will be demonstrated in later sections. The Plots tab displays plots as the name suggests, but whether a plot will appear there or somewhere else depends on how the code was submitted. If the source of the code for the plot is an R script file or was directly typed into the Console, the resulting plot will appear in the Plots tab.

The **Console pane** at the bottom-left interactively executes R code. One can type directly in the Console and press Enter/Return to execute single lines of code, or code documents can be used to execute multiple lines of code with one click or press.

USING R MARKDOWN FILES

All the code documents in this workshop are R Markdown files, which allow for code execution, narrative text, and formatting (via markdown) to co-exist in the same document. Figure 2 shows the core components of an R Markdown file: narrative text (with optional formatting) and “code chunks”.



```
 85
 86 ## R Markdown
 87
 88 Code is written and executed within "chunks". All code chunks start with
 89 ```{r} and end with ```. You will know you've successfully set up a code
 90 chunk when the background turns grey and chunk controls appear at the top-right
 91 of the chunk.
 92
 93 ```{r}
 94
 95 # A pound sign creates a code comment, which is colored green - very useful!
 96
 97 # In code chunks, you can execute single lines by setting your cursor on a line
 98 # and pressing Control + Enter
 99
100 x <- 10
101 x
102
103 ```

 [1] 10

104
105
106
100:8
```

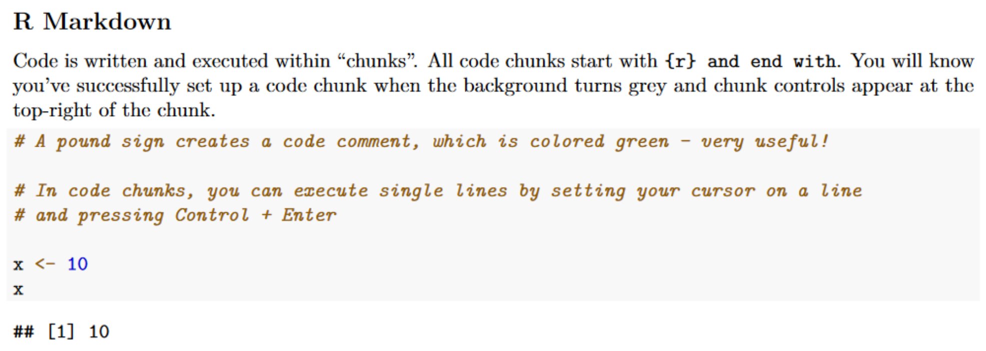
```
> # In code chunks, you can execute single lines by setting your cursor on a line
> # and pressing Control + Enter
>
> x <- 10
```

Figure 2. An R Markdown file showing narrative text with header formatting via markdown, a code chunk with code and comments, and the results of the code executed from the code chunk.

Code chunks start with ` ```{r} ` and end with ` ``` `. These starting and ending pieces are referred to as "chunk delimiters". Code chunks are visually distinguished in the R Markdown document by a grey background. Each code chunk has three icons that appear in the upper-right corner. The most important of these is the rightmost icon, the green "play button". Pressing it will execute all code in the chunk. Code can also be executed line-by-line by highlighting the code to be executed and pressing Control + Enter. If the code results in anything being printed or otherwise displayed, the result will be shown directly below the code chunk as shown in Figure 2. To erase the result, click on the X icon at the top-right.

Narrative text with formatting via markdown is shown in Figure 2 between lines 86 and 91. The ability to write narrative text in the same document in which code is written and executed is convenient for things like documenting code, explaining processes, and providing interpretations of any results. Pound signs (#) indicate headers, which are useful for separating code into sections with headers. A single pound sign creates a first-level heading (the largest size) and six pound signs create a sixth-level heading (the smallest size). The size of the header text does not change in the R Markdown document itself, but someone reading the code document can tell based on the number of pound signs which level a heading should be understood to be. Thus, headers are an example of R Markdown formatting that is useful for enhancing presentation both within the R Markdown file and when the file is exported.

There are many additional formatting touches that can be added to an R Markdown file, making R Markdown documents also good for creating reports and manuscripts that can be exported to other file formats. This exporting process is called "knitting", and the menu to choose the file type to knit to is found by clicking the icon at the top of the R Markdown document that is a ball of blue yarn with a knitting needle sticking out of it. The result of knitting the code shown in Figure 2 is shown in Figure 3.



R Markdown

Code is written and executed within "chunks". All code chunks start with {r} and end with. You will know you've successfully set up a code chunk when the background turns grey and chunk controls appear at the top-right of the chunk.

```
# A pound sign creates a code comment, which is colored green - very useful!

# In code chunks, you can execute single lines by setting your cursor on a line
# and pressing Control + Enter

x <- 10
x
```

```
## [1] 10
```

Figure 3. The PDF export of the code in Figure 2.

It is important to review the knitted files to check that everything exported as intended. For example, the two ``` included in line 89 as part of the text narrative disappeared. I can tell that R Markdown interpreted those as being misplaced code because the font is different for the words "{r} and end with", so I would need to reconsider how I included the ``` in the narrative if the intended end product were an exported PDF. The intended end product it the RMD file itself, so no need to fix it.

BASIC SYNTAX

The basic syntax of R is the same whether one types it into a code document or directly into the R Console. R syntax follows some key rules:

- Outside of character strings, whitespace does not matter.
- Everything is case-sensitive. For example, R treats `A` and `a` as completely different objects.
- Statements and names must start with a letter, not a number or a punctuation mark. Numbers and punctuation may be used in non-starting places in statements and names.
- Lines end when Enter/Return is used to create a new line or if a semi-colon is used. It is vastly more common to see the former, which is reflected in this workshop.
- Comments start with at least one pound sign and end when Enter/Return is used to create a new line. Multiline comments are made by having each comment line start with a pound sign.
- When assigning values to objects, `<-` is the most preferred assignment operator. The equals sign (`=`) is often functionally equivalent, but there are edge cases where the behavior of `<-` and `=` will be different. For example, even though `A <- 2` and `A = 2` will both assign the value of 2 to object A, it is considered better practice to use the first exclusively because doing so eliminates risk of encountering an edge case where the equal sign will be interpreted differently than intended.

FUNCTIONS, PACKAGES, AND READING IN DATA

Functions are ubiquitous in R, and almost everything one will ever want to do in R will involve at least one function. **Functions** are a set of pre-defined statements that accomplish some kind of task when given arguments by the user. An example of a very simple function is the abs() function, which takes any number or number-assigned object as an argument and returns the number's absolute value. The function's name is "abs", and arguments are included in the parentheses. The abs() function takes just one argument. If I wanted to, say, get the absolute value of -11, I would enter it as an argument in the parentheses like so: `abs(-11)`. If the function takes more than one argument, which will be the case for functions we use later, each argument is separated by a comma.

R comes with many functions that can always be called, which are sometimes referred to as "base R functions". The abs() function is such a function; you do not have to do anything before calling the function. R's set of available functions can be greatly expanded by downloading packages or writing your own functions, the former of which we will do in this workshop. **Packages** contain functions that other users have created and made available to other users.

To use functions from a package, one must first download and install the package. One way (of several) to install packages is to use the install.packages() function. This requires you to know the exact name of the package you want to install. One package we will soon need is the haven package, which contains a function to read in SAS7BDAT files. To install the haven package, execute the following line:

```
install.packages("haven")
```

Recall that R is case-sensitive, so using "Install.packages" or "Haven" are not equivalent to the above line and will result in an error message if used. The quotes are required to identify the package's name; if omitted, an error message saying that the named object cannot be found will appear.

A package only needs to be installed once on a computer or Posit Cloud project, but the package's functions will not be available to call in a given session until the package is loaded into memory. This is done with the library() function, like so:

```
library("haven")
```

Interestingly, the quotes are not required in the argument entered into library() function like they are in the install.packages() function, but I've included them for consistency. Once the above line is executed in a given R session, all the functions in the haven package will be callable in any code document for the duration of the session. A session starts when R/RStudio is opened and ends when R/RStudio is closed.

GETTING DATA

Data is usually obtained by either manually creating a data set by entering values into rows and columns or by importing an existing data set.

Manually creating a data set

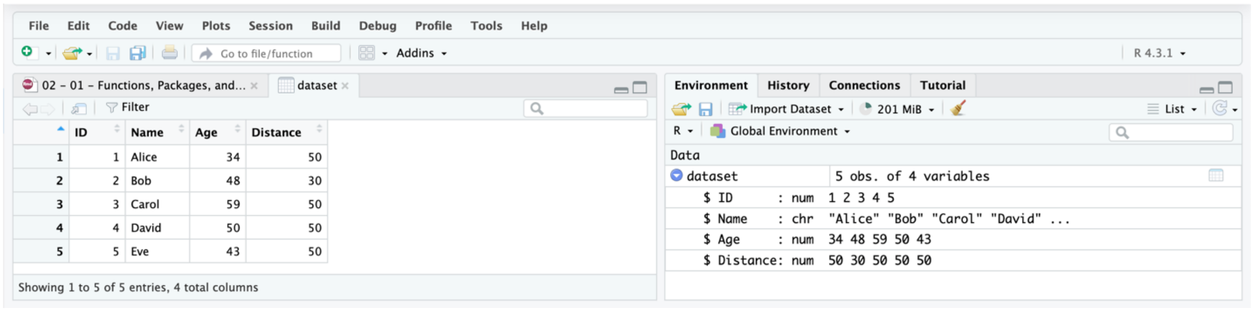
The simplest form that data can take is a vector, which is a single row or column composed of individual elements. Vectors can be manually created in R using the c() function, in which each element must be separated by a comma. To create a vector object named B that contains the elements 1, 2, 3 and 4, the code to do so is like so:

```
B <- c(1,2,3,4)
```

Typically, though, data sets contain both rows and columns. The rows are usually observations, and the columns are usually variables with names. There are many ways to manually create new data sets with both rows and columns in R. I will show you how to do so with the data.frame() function, which allows for manually entering data and giving names to the columns simultaneously by using named vectors as arguments for the function.

```
dataset <- data.frame(ID = c(1,2,3,4,5), Name = c("Alice", "Bob", "Carol",
"David", "Eve"), Age = c(34, 48, 59, 50, 43), Distance = c(50, 30, 50, 50,
50))
```

To better understand how the nested functions work together, let's walk through what each piece does. The result of the line is a data object named dataset. There are four arguments, which correspond to four created variables (i.e., named columns): ID, Name, Age, and Distance. Each argument is itself a named vector with five elements, corresponding to five observations for each variable. Clicking on the dataset object in the Environment pane will display its contents, which are shown in Figure 4.


File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
R 4.3.1
02 – 01 – Functions, Packages, and...
dataset
Filter
Environment History Connections Tutorial
Import Dataset
201 MiB
List
Global Environment

	ID	Name	Age	Distance
1	1	Alice	34	50
2	2	Bob	48	30
3	3	Carol	59	50
4	4	David	50	50
5	5	Eve	43	50

Showing 1 to 5 of 5 entries, 4 total columns

```
Data
dataset            5 obs. of 4 variables
   $ ID      : num  1 2 3 4 5
   $ Name    : chr  "Alice" "Bob" "Carol" "David" ...
   $ Age     : num  34 48 59 50 43
   $ Distance: num  50 30 50 50 50
```

Figure 4. Display of manually created data set (named dataset).

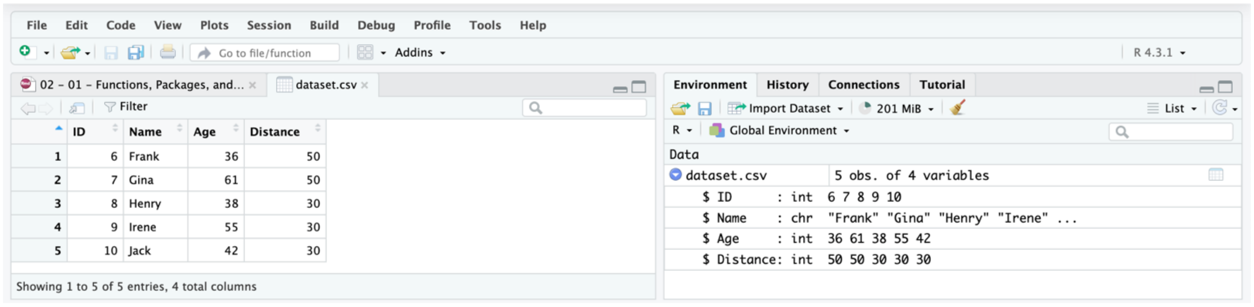
Importing data files

R has functions that support importing many kinds of data files. A common file type for data sets is CSV, which stands for "Comma-Separated Values". The read.csv() function is a base R function that reads in a CSV file and saves it as a data object. This function requires just one argument, but its contents depend on where the file to be imported is. If the code document and the data file are in the same directory, which will always be the case on Posit Cloud, then the argument is the quoted name of the file with the file extension included, like so:

```
dataset.csv <- read.csv("datafile.csv")
```

If one is using a locally installed version of RStudio, then the current working directory in RStudio could be changed to match the location of the file – in that case, the code would be the same. Is it also possible to include the whole file path along with the file name and extension, which will always work no matter what the current working directory is. An example of this is available in the Github version of the workshop materials.

As shown in Figure 5, the imported data set from the CSV file has the same structure but different values than the one that was manually created. Note that the ".csv" part of the data object name does not mean that it is still a CSV file; rather, I have used punctuation in a non-starting place in the data object name to help us remember where the data originated.


File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
R 4.3.1
02 – 01 – Functions, Packages, and...
dataset.csv
Filter
Environment History Connections Tutorial
Import Dataset
201 MiB
List
Global Environment

	ID	Name	Age	Distance
1	6	Frank	36	50
2	7	Gina	61	50
3	8	Henry	38	30
4	9	Irene	55	30
5	10	Jack	42	30

Showing 1 to 5 of 5 entries, 4 total columns

```
Data
dataset.csv        5 obs. of 4 variables
   $ ID      : int  6 7 8 9 10
   $ Name    : chr  "Frank" "Gina" "Henry" "Irene" ...
   $ Age     : int  36 61 38 55 42
   $ Distance: int  50 50 30 30 30
```

Figure 5. Display of imported CSV file (named dataset.csv).

There are other types of files that R cannot read in using base R functionality but can if certain packages are installed. The haven package has functions that import data file types like SAS7BDAT (from SAS), SAV (from SPSS), and DTA (from Stata). Once the haven package is installed and loaded into memory, the read_sas() function can be used to read in SAS7BDAT files. Apart from the name of the function (read_sas) and the file type (.sas7bdat), the syntax is identical to that of read.csv().

```
dataset.sas <- read_sas("datafile.sas7bdat")
```

Like the dataset.csv data object, the dataset.sas data object has the same structure but different values than the manually created data set.

MANIPULATING DATA SETS

Once data has been brought into statistical software, the next step is often not analysis but rather some degree of data manipulation to get the data into the best form for analysis. Data manipulation is an expansive topic that could easily be a 90-minute workshop itself, so the focus of this one will be on some of the most common data manipulation tasks undertaken before moving on to analysis. There are often many ways to accomplish the same task in R, so the methods shown in this section are just one way to accomplish these data manipulation tasks.

ADDING AND REMOVING ROWS AND COLUMNS

Because data files usually consist of rows and columns, it's natural to think of this data structure as a matrix. Matrices have a convenient row-and-column notation style (rows x columns) that makes it easy to identify any element, row, or column in a matrix. R allows users to use row-and-column notation to identify and perform operations on the elements, rows, columns of a data set.

The data set shown in Figure 4 has 5 rows and 4 columns. Thus, it is also a 5 x 4 matrix. Each column has a number according to its position from left to right. The Name variable, because it's the second column from the left, is column 2. Each row has a number according to its position from top to bottom. The row where David's name appears is the fourth row from the top, so that row is row 4. David's name appears in row 4 and column 2, so the location of that element is 4 x 2. This is the R code to select these rows, columns, and elements:

- `dataset[,2]` - Selects Name column (column 2)
- `dataset[4,]` - Selects David's row (row 4)
- `dataset[4,2]` - Selects David's name (row 4, column 2)

In the case of columns, it's also possible to substitute the name of the variable for the number of the column. For example, `dataset[,”Name”]` is equivalent to `dataset[,2]`.

To remove a column from a data object, this notation can be combined with a minus sign to indicate that the column should be dropped. For example, if the ID column isn't useful, it could be dropped like so:

```
dataset[,-”ID”]
```

The same can be done to a remove a row from a data set. For example, if David's information should be removed from the data set, his row can be removed like so:

```
dataset[-4,]
```

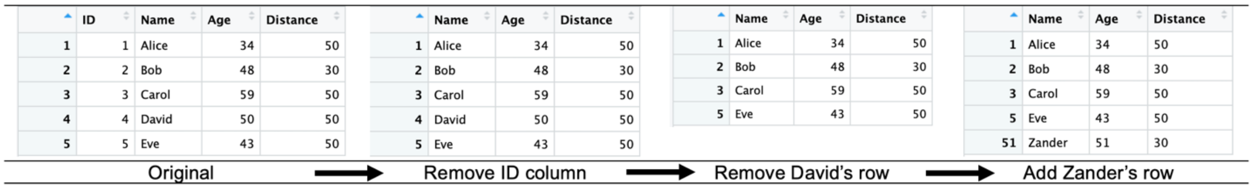
By running these two lines, The ID column has been removed and David's row has been removed. The data object named dataset now looks like what's shown in the third pane in Figure 6.

To add a column or a row, a vector with the appropriate number of elements can be added as a new row or a new column using the rbind() and the cbind() functions, respectively. The arguments for these functions are the data objects or vectors that one wants to bind together, each of which is separated by a comma. If the goal is to add something to an existing data set, then the result should be assigned to the same name as the original data set. For example, if I wanted to add a new entry to the data set with the Name, Age, and Distance columns, I could create a vector with the new information and use the rbind() function to add the row like so:

```
new.entry <- c(“Zander”, 51, 30)

dataset <- rbind(dataset, new.entry)
```

The result of this is shown in Figure 6 in the last pane. To review what's been accomplished in this section, we have removed the ID column from the original data set, removed a particular row, and then added a vector containing a new observation into the original data set.



	ID	Name	Age	Distance
1	1	Alice	34	50
2	2	Bob	48	30
3	3	Carol	59	50
4	4	David	50	50
5	5	Eve	43	50

Original ⟶

	Name	Age	Distance
1	Alice	34	50
2	Bob	48	30
3	Carol	59	50
4	David	50	50
5	Eve	43	50

Remove ID column ⟶

	Name	Age	Distance
1	Alice	34	50
2	Bob	48	30
3	Carol	59	50
5	Eve	43	50

Remove David's row ⟶

	Name	Age	Distance
1	Alice	34	50
2	Bob	48	30
3	Carol	59	50
5	Eve	43	50
51	Zander	51	30

Add Zander's row

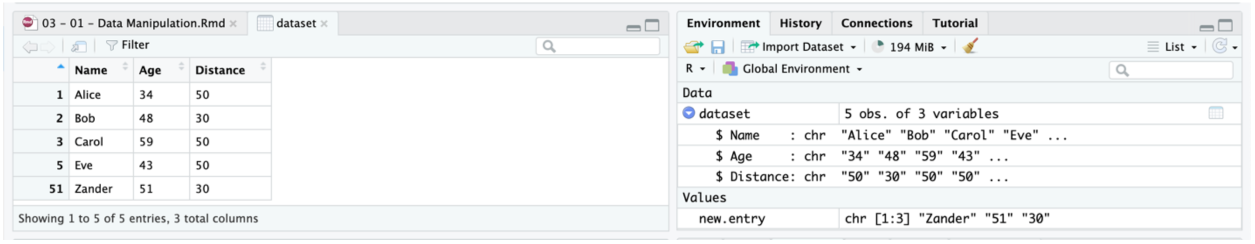
Figure 6. Progression of row and column removal/addition.

SETTING VARIABLE TYPES

So far, the focus has been on the contents of the data set. It is also important to pay attention to **variable types**. Variable types are of interest at the data manipulation stage because some analyses cannot be conducted on certain variable types. For example, it is not possible to take a mean of something that is not a number. There are many types of variables that R recognizes, among the most common of which are numeric, integer, character, and factor. Numeric and integer are slightly different in the sense that the latter does not have decimals and the former does, but they are functionally equivalent. Character variables, also known as string variables, are those that have letters in them. Factor is special variable type that creates factor levels (i.e., categories) for each unique value that exists in the variable. Character and factor variable types are often functionally equivalent, but there are situations where the results will differ somewhat depending on the choice between character and factor types.

Figure 4 additionally displays information about the variables as R understands them in the Environment pane. When a data object is created, R will assign a variable type based on the nature of the values in each variable. The ID, Age, and Distance variables were assigned the "num" designation, which means the variable type is numeric. The Name variable was assigned the "chr" designation, which means the variable is character. In this case, R chose reasonable variable types when the data set was first imported.

However, something happened to those variable types when the new row was added in the previous section.

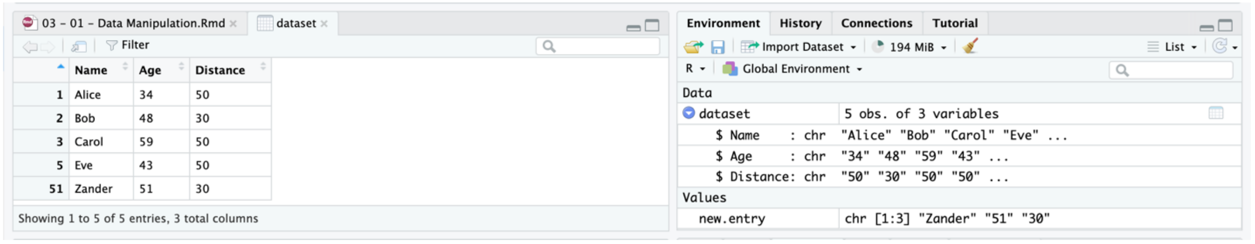


	Name	Age	Distance
1	Alice	34	50
2	Bob	48	30
3	Carol	59	50
5	Eve	43	50
51	Zander	51	30

Showing 1 to 5 of 5 entries, 3 total columns

```
Data
dataset          5 obs. of 3 variables
   $ Name    : chr "Alice" "Bob" "Carol" "Eve" ...
   $ Age     : chr "34" "48" "59" "43" ...
   $ Distance: chr "50" "30" "50" "50" ...
Values
  new.entry      chr [1:3] "Zander" "51" "30"
```

Figure 7 shows the Environment pane for the data object named dataset after the new entry was added. The Name, Age, and Distance variables are now all character variable types. If any of the elements in a vector have characters (i.e., Zander's name), all elements in the vector will be considered character elements. Thus, when this vector was bound to dataset as a new row, R re-typed the Age and Distance variables as character variables.



	Name	Age	Distance
1	Alice	34	50
2	Bob	48	30
3	Carol	59	50
5	Eve	43	50
51	Zander	51	30

Showing 1 to 5 of 5 entries, 3 total columns

```
Data
dataset          5 obs. of 3 variables
   $ Name    : chr "Alice" "Bob" "Carol" "Eve" ...
   $ Age     : chr "34" "48" "59" "43" ...
   $ Distance: chr "50" "30" "50" "50" ...
Values
  new.entry      chr [1:3] "Zander" "51" "30"
```

Figure 7. The variable types shown in the Environment pane are all character.

To change a variable to a numeric, character, or factor variable type, use the as.numeric(), as.character(), and as.factor() functions. These functions can be combined with the row-and-column notation discussed in the previous section to target specific variables in an existing data set. The Age and Distance variables can be changed back to numeric like so:

```
dataset[,"Age"] <- as.numeric(dataset[,"Age"])

dataset[,"Distance"] <- as.numeric(dataset[,"Distance"])
```

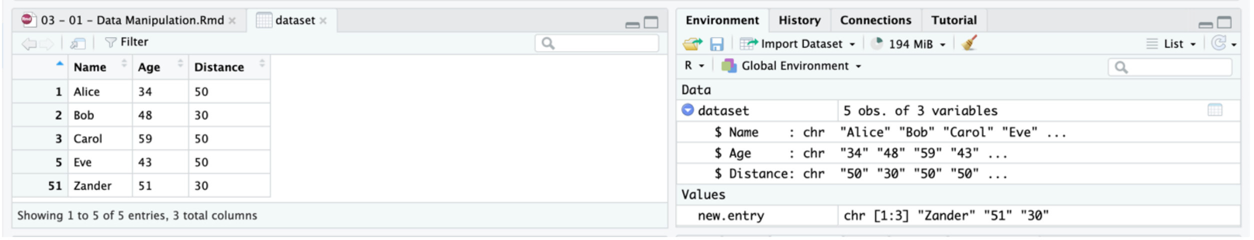
CREATING NEW VARIABLES BASED ON EXISTING VARIABLES

Another common data manipulation task is to create new variables based on the values of existing variables. One way to do this is to use the ifelse() function, which takes three arguments. The first argument is the condition on which the new variable should be created, the second is what value the new variable will take if the condition is true, and the third is what value the new variable will take if the condition is false.

Many different types of conditional statements can be made by using one or more **comparison operators**. Here is a list of the comparison operators as implemented in R:

- `==` (equal to)
- `!=` (not equal to)
- `>` (greater than)
- `>=` (greater than or equal to)
- `<` (less than)
- `<=` (less than or equal to)

Continuing with the data set shown in



	Name	Age	Distance
1	Alice	34	50
2	Bob	48	30
3	Carol	59	50
5	Eve	43	50
51	Zander	51	30

Showing 1 to 5 of 5 entries, 3 total columns

```
Data
dataset          5 obs. of 3 variables
   $ Name    : chr  "Alice" "Bob" "Carol" "Eve" ...
   $ Age     : chr  "34" "48" "59" "43" ...
   $ Distance: chr  "50" "30" "50" "50" ...
Values
  new.entry        chr [1:3] "Zander" "51" "30"
```

Figure 7, the Age variable could be used to create a new categorical variable that indicates which age category applies to each person. For example, if I wanted people who are less than 50 years old to be considered "Senior" and people 50 or more years old to be "Master", here is how I could do that with the ifelse() function:

```
dataset$AgeLevel <- ifelse(dataset$Age < 50, "Senior", "Master")
```

This line creates a new variable in the dataset data object named AgeLevel, as shown in Figure 8.

Sometimes a new variable needs to be created based on the values of multiple variables and not just one. This can be done using **logical operators**. A logical operator connecting two variables means that the condition of both variables will be evaluated in the ifelse() function. The logical operators I use the most are AND, OR, and NOT. In R, AND is represented by `&` (the ampersand key), OR is represented by `|` (the pipe key), and NOT is represented by `!` (the exclamation point key).

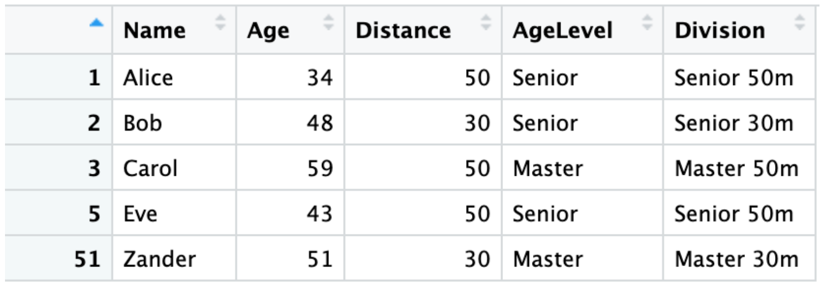
No matter the form of the conditional statement, a single ifelse() function can make one binary true/false evaluation. This is fine if the desired outcome is a new two-category variable, but what if three or more categories are desired? To have three or more categories, ifelse() functions can be nested to allow for as many condition evaluations as needed. The total number of ifelse() functions needed to create a new categorical variable is one fewer than the desired number of categories.

Let's say that I want to create a four-category variable, Division, based on the values of Age and Distance. The values Division should take are "Senior 30m", "Senior 50m", "Master 30m", and "Master 50m". A combination of comparison operators, logical operators, and nested ifelse() functions that will do so looks like this:

```
dataset$Division <- ifelse(dataset$Age < 50 & dataset$Distance == 30,
"Senior 30m", ifelse(dataset$Age < 50 & dataset$Distance == 50, "Senior
50m", ifelse(dataset$Age >= 50 & dataset$Distance == 30, "Master 30m",
"Master 50m")))
```

This is a dense statement, so let's walk through how it works. The AND (`&`) logical operator is used to make both the Age and Distance conditional statements be simultaneously evaluated. Because four categories will be created, there are three ifelse() functions. Recall that the ifelse() function has three arguments: the conditional statement, the resulting value if the condition is true, and the resulting value if the condition is false. Thus, the appropriate place to put another ifelse() function is in that third argument. The first ifelse() function in the line above sets the condition for "Senior 30m". The second ifelse() statement sets the condition for "Senior 50m". The third ifelse() function, the last one, sets the condition for "Master 30m" if it's true and to "Master 50m" if false. This covers all four categories I want to create. The result of the above line of code can be seen in Figure 8.

To review what's been accomplished in this section, we have created two new variables using a combination of ifelse() functions, conditional operators, and logical operators.



	Name	Age	Distance	AgeLevel	Division
1	Alice	34	50	Senior	Senior 50m
2	Bob	48	30	Senior	Senior 30m
3	Carol	59	50	Master	Master 50m
5	Eve	43	50	Senior	Senior 50m
51	Zander	51	30	Master	Master 30m

Figure 8. Two new variables have been created based on existing variables using the ifelse() function, conditional operators, and logical operators.

SUBSETTING OBSERVATIONS

Subsetting data refers to selecting a subset of observations or a subset of variables in a data set. In the case of subsetting variables, the row-and-column notation discussed in a previous section can be expanded to keeping or removing multiple variables. In R, one can use a colon as a shortcut for denoting multiple adjacent rows or columns. For example, rather than typing out 1, 2, 3, 4, 5, one can instead type 1:5 to get the same result. Going back to the data set shown in Figure 8, different variables can be subsetted like so:

```
first.three.vars <- dataset[,c(1:3)]

first.last.vars <- dataset[,c(1,5)]

middle.three.vars <- dataset[,-c(1,5)]
```

The first line creates a new data object named first.three.vars that contains only the Name, Age, and Distance variables. Because these were adjacent variables, the colon notation was used to save some typing. The second line creates first.last.vars, a new data object containing only Name and Division. The last line creates middle.three.vars, which contains Age, Distance, and AgeLevel. Although the columns changed in these data sets, the content of their rows did not. The leading commas within the brackets are very important to include, as that is how R knows that columns should be kept or dropped.

A similar process can be used to keep or remove rows, like so:

```
first.three.obs <- dataset[c(1:3),]

first.last.obs <- dataset[c(1,5),]

middle.three.obs <- dataset[-c(1,5),]
```

Here, the first line creates a new data object that contains just the first three rows (Alice, Bob, and Carol). The second contains just the first and last rows (Alice and Zander), and the third contains the middle three rows (Bob, Carol, and Eve). The position of the commas in the brackets are again important because the trailing commas identify that rows should be kept or dropped.

It is also common to subset rows based on their contents. One way to do this is the subset() function, which has two arguments, The first argument is the name of the data set to be subsetted, and the second argument is the condition statement describing which rows should be kept.

Again going back to the data set shown in Figure 8, we can subset just the people whose AgeLevel was Senior and whose Distance was 50 like so:

```
Seniors50m <- subset(dataset, AgeLevel == ‘Senior’ & Distance == 50)
```

The resulting data set contains just Alice and Eve, who are the two people in dataset who meet both conditions.

COMBINING DATA SETS

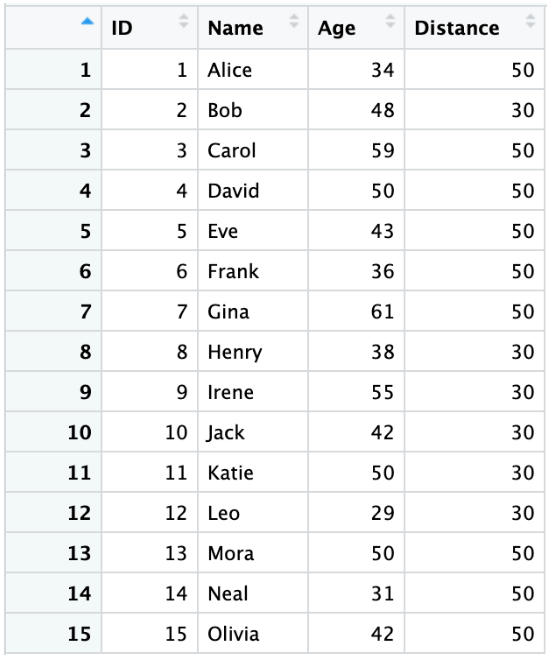
Another step that is sometimes necessary before proceeding to analysis is combining two or more data sets. In this workshop, we’ll practice three: concatenation, merging records one-to-one, and merging records many-to-one.

CONCATENATING DATASETS

The simplest case of combining data is concatenating multiple data sets on top of one another or side-to-side. If the variables across the data sets to be concatenated are the same, the rbind() function that we previously used to bind single row vectors can also be used to concatenate the rows of multiple data sets. For example, to concatenate all three data sets that were created in the Getting Data section, the following code would do so:

```
dataset.rbind <- rbind(dataset, dataset.csv, dataset.sas)
```

The result of this code is shown in Figure 9. All three data sets have the ID, Name, Age, and Distance variables, and the rbind() function concatenated them in the order of the arguments entered. Alice to Eve were in the manually created data set, Frank to Jack were in the CSV file, and Katie to Olivia were in the SAS7BDAT file.



	ID	Name	Age	Distance
1	1	Alice	34	50
2	2	Bob	48	30
3	3	Carol	59	50
4	4	David	50	50
5	5	Eve	43	50
6	6	Frank	36	50
7	7	Gina	61	50
8	8	Henry	38	30
9	9	Irene	55	30
10	10	Jack	42	30
11	11	Katie	50	30
12	12	Leo	29	30
13	13	Mora	50	50
14	14	Neal	31	50
15	15	Olivia	42	50

Figure 9. Result of concatenating three data sets with the same variables.

To concatenate side-by-side, the cbind() function can be used like so:

```
dataset.cbind <- cbind(dataset, dataset.csv, dataset.sas)
```

The result of this is a data set with 12 columns and 5 rows. The first four columns are the ID, Name, Age, and Distance variables from dataset because that was the first argument. The second four columns are those same variables from dataset.csv, and the third four columns are from dataset.sas. These had the same variable names, but R does not allow variables to have the same name within a data set. As part of the side-to-side concatenation, R adjusted the variable names that appeared in columns 5 through 12. Specifically, the variables from dataset.csv became ID.1, Name.1, Age.1, and Distance.1, and the variables from dataset.sas became ID.2, Name.2, Age.2, and Distance.2.

If the data sets had different variables or different numbers of rows, then the rbind() and cbind() functions would not have worked. If concatenating data sets with different variables is of interest, the bind_rows() and bind_cols() functions in the dplyr package can do this.

Straightforward concatenation is somewhat unusual in practice, though. It is much more common to combine data sets through merging.

MERGING ONE-TO-ONE

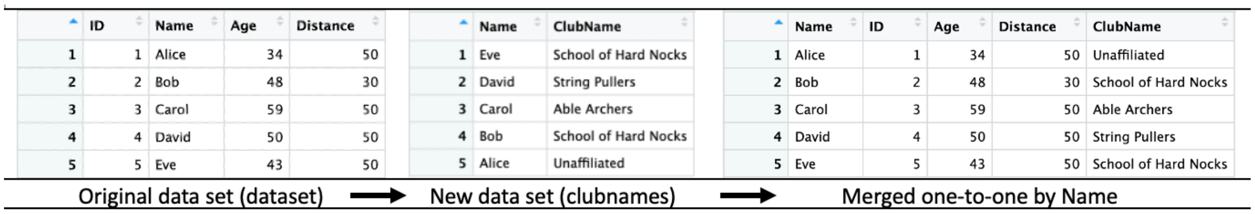
At the surface, a one-to-one merge resembles a side-to-side concatenation because both entail combining two data sets side-to-side. The difference is that concatenation does this in a cut-and-paste fashion with no regard to the contents of either data set. In contrast, a one-to-one merge works by first looking for observations in the two data sets that should be kept together because they share a source. This shared source is sometimes referred to as a “primary key”, and it often takes the form of something like an ID number or a name. For example, a data set containing student grades and a different data set containing student demographic information would both likely include student ID numbers, unique to each student, as a variable. The student ID number is an example of a primary key because it can be used to merge the grades and demographic information of students who are in both data sets in such a way that Student 123’s grades are in the same row as Student 123’s demographics in the merged data set.

One way to merge data one-to-one in R is the merge() function, which needs three arguments. The first two are the names of the data sets to be merged. The third argument tells the merge() function on which variable the two data sets should be merged.

To preserve space, we will again return to the five-observation data set that was manually created in the Getting Data section. We will merge this data set one-to-one with a new data set named clubnames. The contents of both data sets are shown in the first and second panes of Figure 10. The primary key between these data sets is the Name variable. The following code will perform the one-to-one merge:

```
dataset.one.to.one <- merge(dataset, clubnames, by = “Name”)
```

The result of this code is shown in the third panel in Figure 10. There is no duplication of the Name variable because the merge() function used it to combine the data sets. The combined data set kept the order of the rows from the first argument, but notice that pre-sorting the clubnames data set to match the ordering of the rows in dataset was not necessary.



	ID	Name	Age	Distance
1	1	Alice	34	50
2	2	Bob	48	30
3	3	Carol	59	50
4	4	David	50	50
5	5	Eve	43	50

	Name	ClubName
1	Eve	School of Hard Nocks
2	David	String Pullers
3	Carol	Able Archers
4	Bob	School of Hard Nocks
5	Alice	Unaffiliated

	Name	ID	Age	Distance	ClubName
1	Alice	1	34	50	Unaffiliated
2	Bob	2	48	30	School of Hard Nocks
3	Carol	3	59	50	Able Archers
4	David	4	50	50	String Pullers
5	Eve	5	43	50	School of Hard Nocks

Figure 10. One-to-one merge of two data sets.

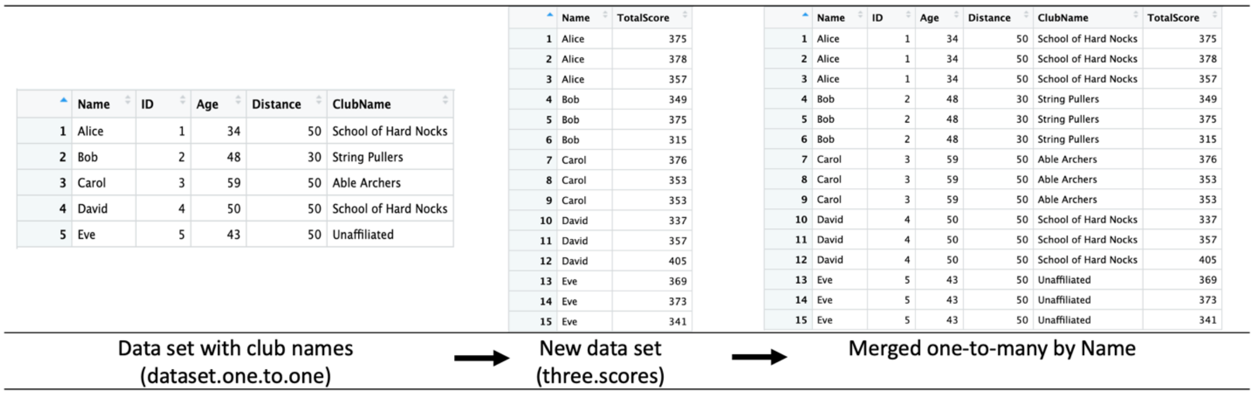
MERGING ONE-TO-MANY

A one-to-many merge is very similar to the one-to-one merge in how it works; records from two data sets are combined based on a primary key. The difference between the two is in the number of rows to be matched between the two constituent data sets. In a one-to-one merge, each observation in the first data set is matched to just one row from the second data set. In a one-to-many merge, each observation in the first data set is matched with more than one row from the second data set. Thinking about the student grades example in the previous section, it is likely that students have grades from many courses. If each individual course grade were stored in separate rows, one per class, then merging with the student demographics data set would result in as many rows for a given student as that student has grades. The demographic information would be repeated in each row for that student.

Continuing with the combined data set from the last section, we will now consider another data set named three.scores, shown in the second pane of Figure 11. This data set contains Alice, Bob, Carol, David, and Eve’s three most recent tournament scores. Each score is its own row (TotalScore) and each row has the participant’s name (Name). We can once again use the merge() function in a similar fashion as in a one-to-one merge

```
dataset.one.to.many <- merge(dataset.one.to.one, three.scores, by = “Name”)
```

The result of this one-to-many merge is shown in Figure 11. The new data set has 15 rows, three for each of the five participants.



	Name	ID	Age	Distance	ClubName
1	Alice	1	34	50	School of Hard Nocks
2	Bob	2	48	30	String Pullers
3	Carol	3	59	50	Able Archers
4	David	4	50	50	School of Hard Nocks
5	Eve	5	43	50	Unaffiliated

	Name	TotalScore
1	Alice	375
2	Alice	378
3	Alice	357
4	Bob	349
5	Bob	375
6	Bob	315
7	Carol	376
8	Carol	353
9	Carol	353
10	David	337
11	David	357
12	David	405
13	Eve	369
14	Eve	373
15	Eve	341

	Name	ID	Age	Distance	ClubName	TotalScore
1	Alice	1	34	50	School of Hard Nocks	375
2	Alice	1	34	50	School of Hard Nocks	378
3	Alice	1	34	50	School of Hard Nocks	357
4	Bob	2	48	30	String Pullers	349
5	Bob	2	48	30	String Pullers	375
6	Bob	2	48	30	String Pullers	315
7	Carol	3	59	50	Able Archers	376
8	Carol	3	59	50	Able Archers	353
9	Carol	3	59	50	Able Archers	353
10	David	4	50	50	School of Hard Nocks	337
11	David	4	50	50	School of Hard Nocks	357
12	David	4	50	50	School of Hard Nocks	405
13	Eve	5	43	50	Unaffiliated	369
14	Eve	5	43	50	Unaffiliated	373
15	Eve	5	43	50	Unaffiliated	341

Figure 11. One-to-many merge of two data sets.

SUMMARIZING DATA

The first step of almost any analytic workflow is to get to know your data by computing summary statistics and generating useful descriptive visualizations.

SUMMARIZING CONTINUOUS VARIABLES

Creating summaries of continuous variables usually involves summary statistics about the central tendency of the data (e.g., the mean and median) and the variability of the data (e.g., the minimum value, the maximum value, the first and third quartiles, the interquartile range, standard deviance, and variance). The R functions for these are as follows:

- Mean: `mean()`
- Median: `median()`
- Minimum value: `min()`
- Maximum value: `max()`
- First quartile value: `quantile(variable-name, 0.25)`
- Third quartile value: `quantile(variable-name, 0.75)`
- Interquartile range: `IQR()`
- Standard deviation (from a sample): `sd()`
- Variance: `var()`

As shown in the dataset.one.to.one data object shown Figure 11, the Age variable is a continuous variable, so it's sensible to compute these measures for that variable. The code and the result are shown below:

- `mean(dataset.one.to.one$Age)` Returns 46.8
- `median(dataset.one.to.one$Age)` Returns 48
- `min(dataset.one.to.one$Age)` Returns 34 (Alice's age)
- `max(dataset.one.to.one$Age)` Returns 59 (Carol's age)
- `quantile(dataset.one.to.one$Age, 0.25)`Returns 43
- `quantile(dataset.one.to.one$Age, 0.75)` Returns 50
- `IQR(dataset.one.to.one$Age)`Returns 7
- `sd(dataset.one.to.one$Age)` Returns 9.20326
- `var(dataset.one.to.one$Age)` Returns 84.7

Through packages, R has incredible capacity for producing graphics; I highly recommend the R Graphics Cookbook (https://r-graphics.org) if you want to get a sense for the variety of visualizations R can do. When getting to know one's data, though, quick and simple functions are useful because of their simplicity. The plot() function can be used to produce a simple scatter plot of a single continuous variable.

```
plot(dataset.one.to.one$Age)
```

The result of this is shown in Figure 12. The plot would be more interesting if there were more than five observations, but it gives a good visual sense for the statistics that were computed previously. To plot two continuous variables, add the second continuous variable (separated by a comma) as a second argument to the plot() function. The first argument in the plot() function is set as the Y-axis, and the second is set as the X-axis. When only one argument is included as in Figure 12, that single variable is plotted on the Y-axis and an Index, corresponding to the order of the observations in the data set, is added to the X-axis.

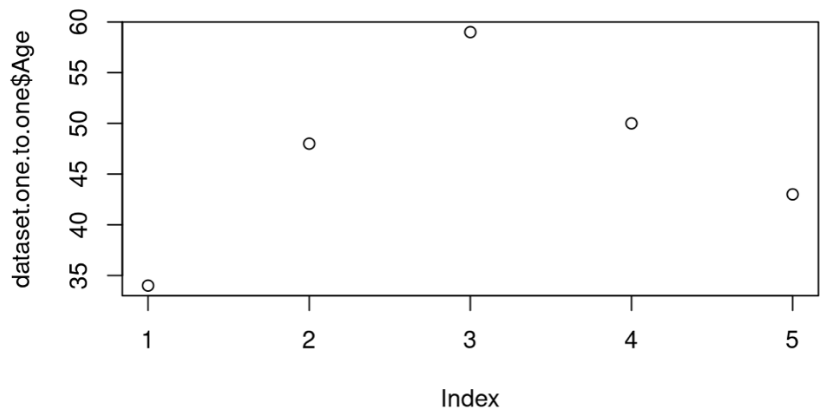


Figure 12. Result of using the plot() function to visualize age

Histograms are also a common choice of visualization when getting to know one's data, which can be created in R using the hist() function. Like the plot() function, the hist() function requires just one argument: the variable for which you want to create a histogram.

```
hist(dataset.one.to.one$Age)
```

The result of this is shown in Figure 13. Again, this histogram would be more interesting with more than five observations, but it gives a reasonable sense of the spread of the data given its size.

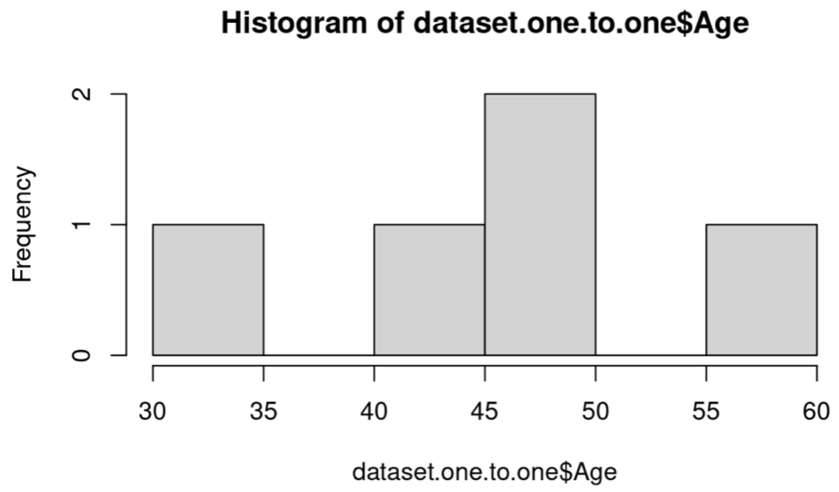


Figure 13. Result of using the hist() function to visualize age.

Another common choice of visualization when getting to know one's data is the boxplot, which can be created in R using the boxplot() function. Like the plot() and hist() functions, the boxplot() function requires just the variable in question as an argument.

```
boxplot(dataset.one.to.one$Age)
```

The result of this is shown in Figure 14. We know from the summary statistics that the median is 48, which is where the bolded line is. The first and third quartiles, 43 and 50 respectively, are the edges of the box. The whiskers of the plot represent the values that are less than the first quantile and greater than the third quantile.

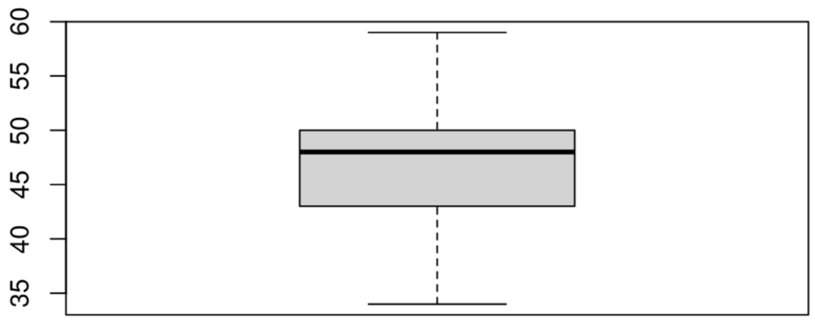


Figure 14. Result of using the boxplot() function to visualize age.

The plot(), hist(), and boxplot() functions can all produce more refined visualizations than shown here by adding additional arguments, which you can learn about by reading the documentation for these functions. For example, to read the documentation for the plot() function, type `?plot()` into the Console.

SUMMARIZING CATEGORICAL VARIABLES

A common method of summarizing categorical variables is the frequency table, which displays counts and/or proportions of the number of observations in different categories. One way to look at frequencies in R is the table() function. The table() function requires just one argument, which will produce a row of counts for a single categorical variable. One categorical variable in the dataset.one.to.many data object is the ClubName variable, a frequency table for which can be created like so:

```
table(dataset.one.to.one$ClubName)
```

The result of this code is shown in the top pane of Figure 15. It shows that two participants identified their club as “School of Hard Nocks” and one participant each identified their club as “Able Archers” and “String Pullers”, and one participant reported no club affiliation.

Contingency tables, also called cross-tabulation tables, are frequency tables in which two categorical variables are crossed. The levels of one variable make up the rows, and the levels of the other variable make up the columns, and counts appear wherever both conditions in a cell apply to an observation. When using the table() function to create a contingency table, the first argument designates the rows and the second designates the columns.

```
table(Club = dataset.one.to.one$ClubName, “Distance in M” =
dataset.one.to.one$Distance)
```

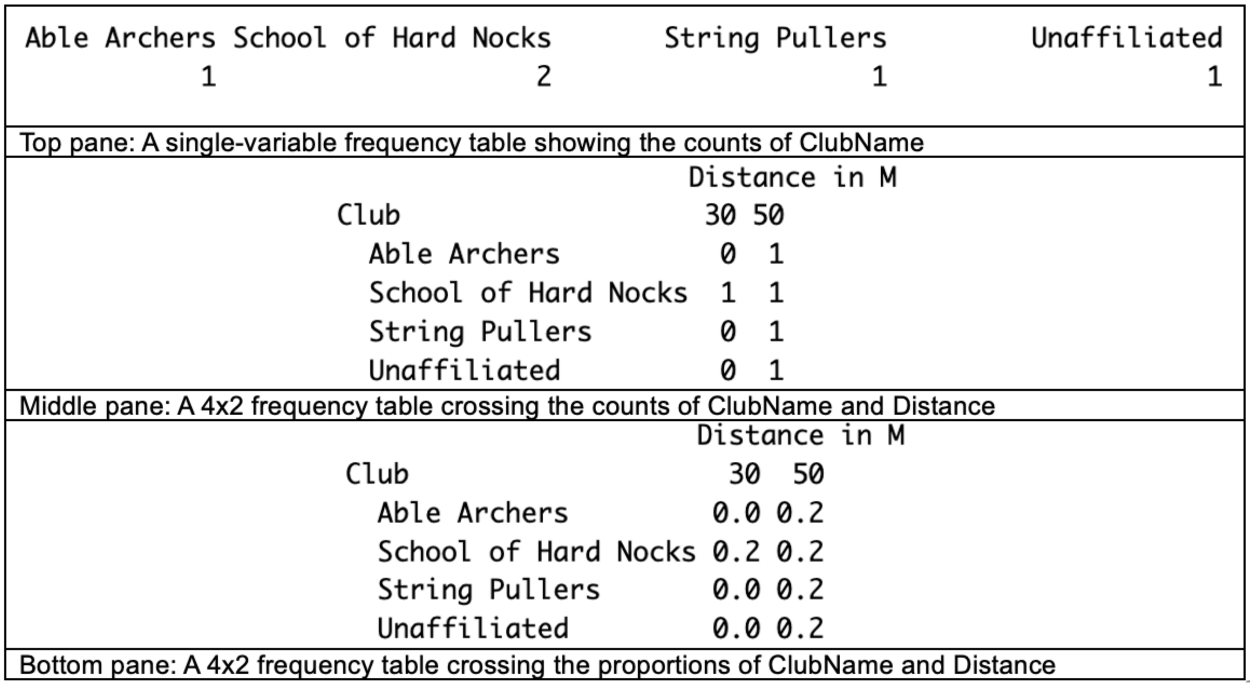
The above code produces the contingency table shown in the middle pane in Figure 15. In addition to adding Distance as a second variable, this code also demonstrates a convenient option for labeling the axes of the contingency table. The row axis label, Club, did not require quotes because it was a single word. The column axis label, Distance in M, needed quotes because the label was more than one word.

It is sometimes useful to see proportions of a total rather than raw counts. In R, the proportions() function computes the proportion of the total for each cell in the contingency table. It is possible to put a table directly into the proportions() function by copying the code used to make the table into it, but it is sometimes easier to save a table as an object and then enter that object as an argument into the function like so:

```
Club.Distance.table <- table(Club = dataset.one.to.one$ClubName, “Distance
in M” = dataset.one.to.one$Distance)

proportions(Club.Distance.table)
```

The result of the above code is shown in the bottom pane in Figure 15. Each cell that was not zero had just one participant within, so all the non-zero proportions are 0.20 (or 20%).



```
Able Archers School of Hard Nocks       String Pullers         Unaffiliated
           1                  2                    1                  1
```

Top pane: A single-variable frequency table showing the counts of ClubName

```
                                          Distance in M
                    Club                   30 50
                      Able Archers          0  1
                      School of Hard Nocks  1  1
                      String Pullers        0  1
                      Unaffiliated          0  1
```

Middle pane: A 4x2 frequency table crossing the counts of ClubName and Distance

```
                                          Distance in M
                    Club                   30  50
                      Able Archers         0.0 0.2
                      School of Hard Nocks 0.2 0.2
                      String Pullers       0.0 0.2
                      Unaffiliated         0.0 0.2
```

Bottom pane: A 4x2 frequency table crossing the proportions of ClubName and Distance

Figure 15. Single-variable and contingency tables using the table() function.

CONCLUSION

Today's data worker has many choices for managing and analyzing data. R and RStudio are both free, open-source, and easy to install locally or use on Posit Cloud, making them readily available to try and use. By participating in this hands-on workshop, you have experienced all parts of a full data workflow in R/RStudio: obtaining data, manipulating data, combining data, and summarizing data.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Wendy Christensen, Ph.D.
University of Colorado Anschutz Medical Campus
Wendy.Christensen@cuanschutz.edu
https://github.com/wendychristensen/WUSS-2023-Open-Source-Essentials-R-Rstudio