CDFW Introduction to R Programming

Liz Siemion

2025-05-13

Contents

Chapter 1

CDFW Introduction to R Programming

1.1 Overview

I reference course materials from USU's ecology center workshop, and Dr. Simona Picardi's Reproducible data science website.

1.2 Workshop Goals

This course is intended to teach the basic programming fundamentals of R and how to use RStudio. I hope to provide you with the skills needed to continue learning R on your own and to incorporate R into your analyses and workflows.

1.3 Workshop Content

1. Into to R and R-Studio
2. Packages
3. File paths
4. Importing and exporting data files (csv, xlsx, mdb, rds)]
5. Working with R objects
6. Data types (character, numeric, logical, etc.) and classes (scalar, vector, data frames, matrices, lists, etc.)
7. Functions (arguments, function workflow)
8. Data organization and manipulation (base R and tidyverse)

9. Indexing
10. Basic plotting 11 Basic statistics & statistical summaries
11. Introduction to debugging (how to ask the right questions)
12. Making readable R code (commenting, outlining, etc.)

Chapter 2

Intro to R and R-Studio

2.1 Learning R

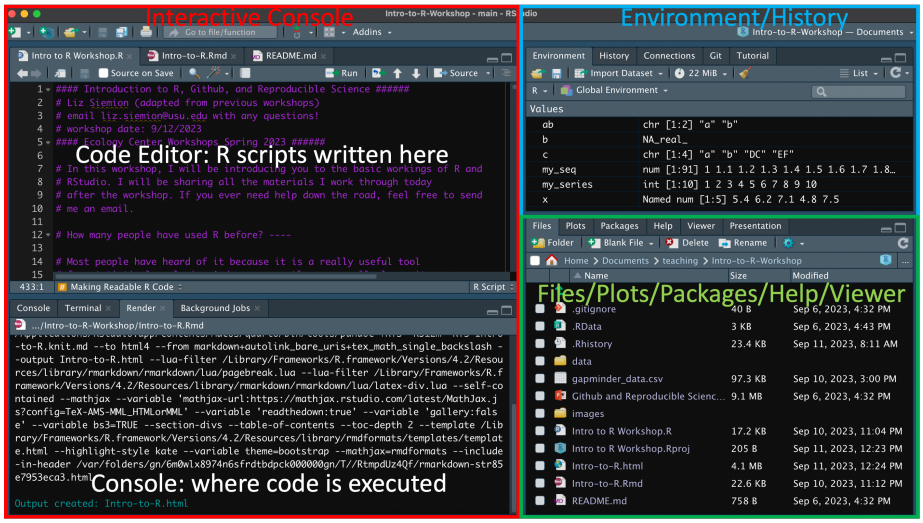
I first started learning R in an undergraduate Statistics course. I remember thinking it was amazing for doing complex calculations, but I honestly had no idea how the programming worked. I just knew that using certain code would produce output that I needed. This is common for a lot of people. We learn how to run whatever analyses, but miss out on basic fundamentals. 90% of the time when an analysis won't run, it isn't a statistics problem, but a programming problem (i.e. the data isn't assigned the right data type, the data is being indexed wrong, etc). Knowing how to program can also help with reproducibility. The goal of this course is to help you develop basic programming skills, separate from any kind of analyses, and steps for troubleshooting in R.

2.2 What is R?

R is both the name of the programming language and the software used for data storage and manipulation.

RStudio is the interface. You will need to download R before downloading RStudio. RStudio allows you to keep track of your code, working directory, and environment all in one place. I keep R and RStudio in my applications folder on my macbook and on my C drive on PCs.

When you first open RStudio, you will be greeted by three panels:



1) **The Interactive R Console (entire left)**

- R scripts appear in the top left. This is where you write your code. When you execute this code, it appears in the Console (bottom left). You can run code from the console, but it will be forgotten once you close R. It's best to write a saveable script in the code editor where you can send lines of code to the console to be executed.

2) **Environment/History (tabbed in upper right)**

- Environment: collection of objects (i.e. variables, data frames, functions, etc.) that we define
- History: this contains every line of code executed during the session

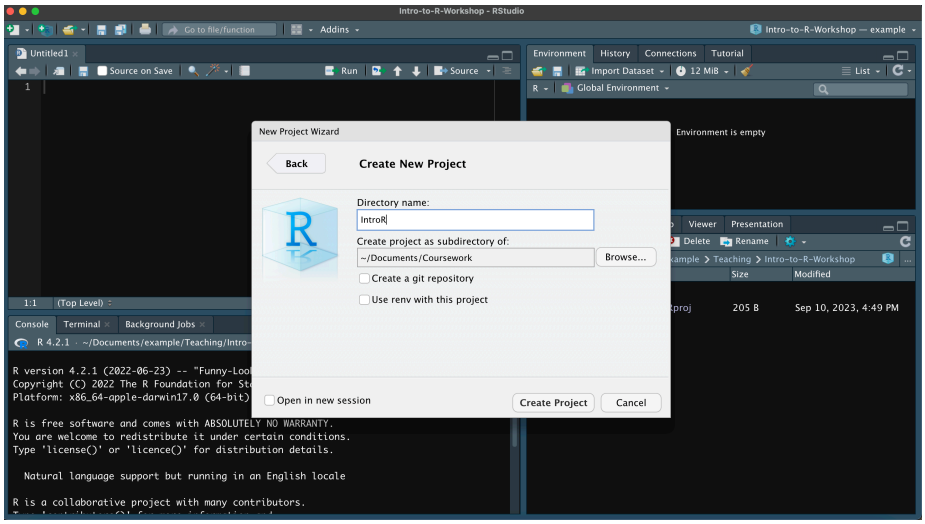
3) **Files/Plots/Packages/Help/Viewer (tabbed in lower right)**

- **Files** shows all the files in your working directory. A **working directory** is essentially the default folder that R is reading data from/putting output into. We will go through setting the working directory below. You can also create, delete, and rename files and folders from this tab.
- **Plots** displays plots that you generate
- **Packages** displays any packages you have downloaded and installed in R. If there is a check mark next to a package, it means you've loaded it into your current R session
- **Help** will show you a description of functions. To get a function description, simply run `?` followed by the function name. For example, `?setwd()` will show me the documentation for the `setwd()` function.

2.3 RStudio Projects

RProjects are a self-contained, portable work space where you can have your data, code, and output all in one place. It's essentially all the things that you need for your analyses all in one place. RProjects are great to use for reproducibility because they are easy to share. Let's go through how to set up an RProject.

Steps for Making an RProject



1) Open the "File" menu from the upper left.
2) Then select "New Project".
3) Select "New Directory".
4) Select "Empty Project".
5) Enter a directory name
6) Select the "Create Project" button.

Every time you open this project the working directory will always be set to the folder `Intro-to-R-Workshop`! Projects are useful because you don't have to assign the working directory. The working directory is automatically set to the folder that your RProject is located.

For example, if my RProject is located in C:/Teaching/IntroR/, my working directory is also located C:/Teaching/IntroR/.

2.4 Using Comments

Use the `#` to tell R you are making a comment. Comments are used to explain code and allow someone unfamiliar with your code to follow more easily. Commenting can also be used to prevent R from running specific lines of code since

R ignores anything that follows the `#` mark.

```
# 567*5 tells R that 567*5 is a comment, and so R knows not to execute this line of co
```

Sometimes you want to comment out large sections of code, and this can be done using `control + shift + c` on windows or `command + shift + c` on a macbook.

2.5 Installing Packages

Sometimes you will need/want to use tools that are not built into the baseR code. You can download these tools from R repositories as `packages`.

```
install.packages("tidyverse") # install a new package
library(tidyverse) # call a package you previously downloaded
```

If there are functions with the same name in 2 different packages, R will arbitrarily mask one of them. To get around this, simply specify which package you intend to use the function from. For example, the 'filter' function exists in both tidyverse and dplyr. If I specifically want to use the tidyverse version, I could run `dplyr::filter()`.

Chapter 3

File paths

What about if we want to manually set the working directory directory or tell R to load a file from a specified location? How would we do that? We need to first know the file path…

3.1 What is a file path?

File paths are addresses to different locations or files on a computer. Your computer uses a system of nested folders.

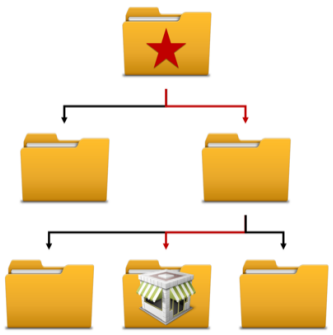


Figure 3.1: Nested file system on your computer – image from R for epidemiology (https://www.r4epi.com)

File paths are addresses to different locations or files within this nested framework. They represents the order of nested folders that the computer must go through to find that particular item. Each folder is separated by a slash. We can use `Absolute` or `Relative` file paths in R to locate files. Knowing the file

path is important when you need to set your working directory. See this website for a detailed explanation.

3.2 Different separators between operating systems

Different operating systems use different separators between folders of a file path.

- On windows, it is `\`
- On Mac/Linux, it is `/`

R uses the `/` separator, so in windows, remember to either use a backward slash `\\` or change all `\` to `/` between your folders of your file path.

3.3 Absolute and Relative file paths

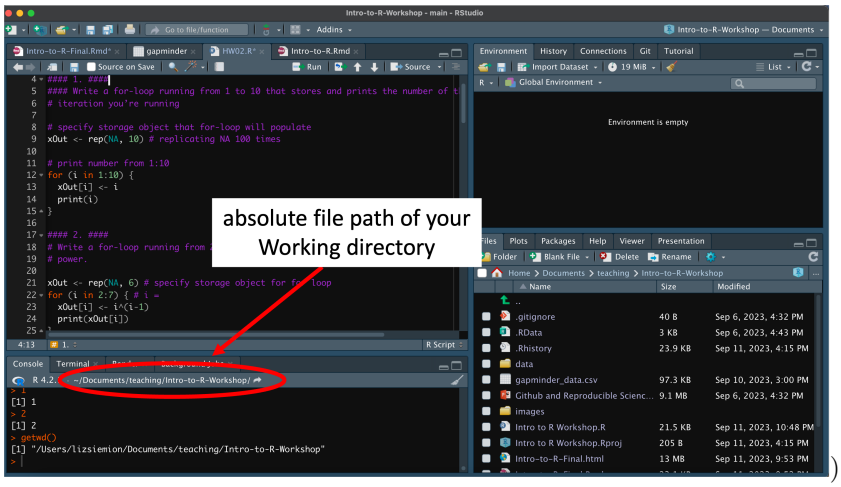
We can use `Absolute` or `Relative` file paths to give R directions to where we want to go.

Absolute Paths: describe where a file is located relative to the root directory of the computer. This can be done on windows through right clicking the file path in windows explorer and selecting copy as text, or right clicking a file, holding the option key and selecting copy as path name on a macbook.

- Windows example: `C:/Users/Documents/Teaching/IntroR/data/Intro-to-R-Workshop.csv`
- Macbook example: `/Users/lizsiemion/Documents/Teaching/IntroR/Intro-to-R-Workshop.cs`

Relative Paths: describe file location with respect to the current working directory. This just means that the file path starts with the location of the home directory.

- Windows example: `IntroR/data/Intro-to-R-Workshop.csv`
- Macbook example: `IntroR/data/Intro-to-R-Workshop.csv`



)

It can be a bit cumbersome to work with absolute file paths. Since R Projects automatically sets the working directory as the project folder, we can use relative paths without any sort of additional set-up. Using relative paths also makes our code more readable, and easier to share and maintain. If we want to set our working directory manually, we can either use absolute or relative file paths. I recommend not changing the working directory within your script, as this can limit reproducibility.

```
# check working directory
getwd()
# Assign working directory to new location using absolute path
setwd("/Users/lizsiemion/Documents/teaching/Intro-to-R-Workshop")
# Again, I do not recommend changing the working directory from an R project.
```

3.4 Navigating outside the working directory

Let's say we want to load a csv file into R that is outside of our working directory subfolders. How might we do that with absolute or relative paths from our current working directory?

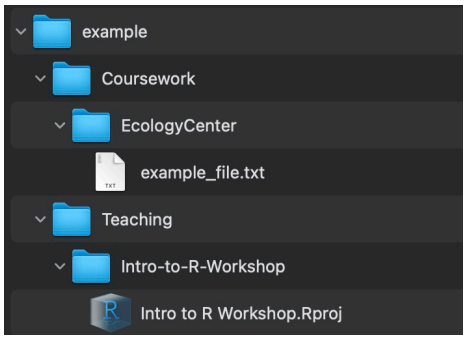
The absolute path is the full file path from our computer's root directory. If we want to use the relative path, we need tell R to go up a given number of parent folder levels from a working directory, and then to the given location within that parent folder. This can be accomplished using `../` syntax.

`./` tells R to go to the folder of the working directory

`../` tells R to go to the parent folder of the working directory

`../../` tells R to go to the parent folder of the parent folder of the working directory

Let's look at an example. Say our folder structure resembles the structure below and our RProject is located in the Intro-to-R-Workshop folder.



- **Step 1:** How many parent levels do we need to move up?
  - Looks like we need to move up 1 level to the teaching folder `../`, and another level up to the Documents folder `../`.
- **Step 2:** Now that we are in the Documents folder, what is the relative path to to the `example_file.txt`?
  - We need to go into the Coursework folder, and then the EcologyCenter folder, where `example_file.txt` is ultimately located.

    `Coursework/EC-tidyverse-workshop-main/example_file.txt`
- By combining steps 1 and 2, we've create the relative file path and can load the `example_file.txt` into our environment with `read.csv()`.

```
read.csv("../../Coursework/ecologycenter/example_file.txt")
```

Chapter 4

Using R

4.1 Use R as a calculator

Remember, order of operations matters. The order is the same as you learned back in school.

From highest to lowest precedence:

- Parentheses: `(`, `)`
- Exponents: `^` or `**`
- Divide: `/`
- Multiply: `*`
- Add: `+`
- Subtract: `-`

```
1 + 100
```

```
## [1] 101
```

```
3 + 5 * 2 # performs multiplication before addition
```

```
## [1] 13
```

```
(3 + 5) * 2 # performs addition in () before multiplication
```

```
## [1] 16
```

```
2/10000
```

```
## [1] 2e-04
```

```
2^3 # text after a code line is called a "comment"
```

```
## [1] 8
```

4.2 Use R to compare things

To compare things in R, we use logical operators. Below is a brief list.

Summary of logical operators == is equal to > greater than < less than >= greater than or equal <= less than or equal ! not | or %in% is contained

```
1 == 9  # equality (note two equals signs, read as "is equal to")
```

```
## [1] FALSE
```

```
1 != 1  # inequality (read as "is not equal to")
```

```
## [1] FALSE
```

```
1 < 2  # less than
```

```
## [1] TRUE
```

```
1 <= 1  # less than or equal to
```

```
## [1] TRUE
```

Notice how R evaluates each of these lines of code as TRUE or FALSE. We are essentially asking R if the above comparison is TRUE or FALSE.

4.3 Use R to assign objects

Objects are a bit of an abstract concept. All you really need to know for now is that objects are things that we make in R that can take on a variety of structures with different data types, and when we assign them a name, they get saved in our global environment. They are data structures with associated data attributes.

Object assignment allows us to assign a variable name to the object for later use. This helps prevent writing redundant code. We assign an object to a variable using the assignment arrow <- or the = so that R knows that x is an object that we can use. So when we run, x <- 6, it reads "make x contain 6." It's recommended to use the <- since the = can get mixed up with assigning values to function arguments (more on this later). Once we assign an object to a variable, it is stored in our global environment (upper right hand panel of R studio).

```
x <- 1/40 # here we are telling R to assign 1/40 to the variable x so that it recogniz
x
```

```
## [1] 0.025
```

```
x = 24 # variables can easily be re-assigned/over-written
y <- x * 2
```

```
rm(y) # you can also remove objects from the environment
```

4.3.1 Variable names

Variable names can contain letters, numbers, underscores and periods. They CANNOT start with a number OR contain spaces (**at all**). Remember that R is case sensitive.

A few different conventions for longer variable names:

- periods.between.words
- underscores_between_words
- camelCaseToSeparateWords

Your choice of convention is up to you, *JUST BE CONSISTENT*.

4.4 Data Types (Modes)

Below are 6 main classes of common data types: `numeric`, `integer`, `complex`, `logical`, `character`, and `factor`

```
class(1.11) # numeric: any real number
class(1L) # integer: any integer. The L suffix forces the number to be an integer
class(TRUE) # logical: binary TRUE or FALSE
# You can have data that look essentially the same, but have different classes.
class('1') # character: words; "" denote words
class(1) # numeric; any real number
class(factor("1a")) # factor: denotes categorical variables, they can be words or numbers
```

You can coerce to a desired data type, as long as they follow the rules using the functions as.

Hierarchy from general to specific: Logical -> Numeric -> Character

1. logical (least general, cannot turn turn character or numeric into a logical type without correctly specifying the value)
2. numeric (can read integer and logical types as numbers)
3. character (most general: anything can be turned into a character by adding "quotes")

```
# Convert from numeric to integer
a <- 45.6
class(a)
```

```
## [1] "numeric"
```

```
# Convert from numeric to character
a_character <- as.character(a)
class(a_character)
```

```
## [1] "character"
```

80% of the time when something isn't running, it's because the data isn't the right type. In the example below, we throw an error because R cannot make a character class into a numeric class. All columns in a data frame need to be the same type as well.

```
# convert from character to numeric
b <- "banana"
b_numeric <- as.numeric(b)
```

```
## Warning: NAs introduced by coercion
```

4.5 Data Structure Classes

Remember earlier when we talked about objects as data with attributes? Well, there are many different ways to store data. The most common ways are in vectors, data frames, and lists.

4.5.1 Scalar

A vector has one element with length 1.

```
x <- 3
```

4.5.2 Vector

A vector in R is essentially a collection of items **of the same basic data type**. Each 'thing' in the vector is called an element. If you don't choose the data type, it'll default to logical; or, you can declare an empty vector of whatever type you like. You can also make vectors with explicit contents using the concatenate function `c()`.

4.6 Lists

A list in R is essentially an object with data that can be in different data types/modes.

```
# list with numeric, character, and logical classes
my_list <- list(1, "banana", TRUE)
your_list <- list(2, "apple", FALSE)
my_list
```

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

```
## [[2]]
## [1] "banana"
##
## [[3]]
## [1] TRUE
```

The $ is called the operator, and it is used for indexing named elements in a list. It allows you to access part of a data object for extracting or subsetting data.

```
names(my_list) <- c("x", "y", "z")
my_list$x
```

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

If you wanted to index a specific element in the list, you could also use brackets

```
# combine two vectors into a list
big_list <- list(your_list, my_list)
big_list
```

```
## [[1]]
## [[1]][[1]]
## [1] 2
##
## [[1]][[2]]
## [1] "apple"
##
## [[1]][[3]]
## [1] FALSE
##
##
## [[2]]
## [[2]]$x
## [1] 1
##
## [[2]]$y
## [1] "banana"
##
## [[2]]$z
## [1] TRUE
```

```
# index second list [[2]] and second element [2] of second list
big_list[[2]][2]
```

```
## $y
## [1] "banana"
```

4.7 Data Frames

A data frame in R is a like a list or generalized matrix but with the constraints that:

(1) all list elements are vectors (i.e. they have 1 mode),
(2) all vectors have the same length
(3) all columns (the list elements) have names

Essentially, imagine each column in the dataframe as a vector, and the dataframe is just a big list of all those vectors. Unlike actual lists however, in dataframes all of the columns/vectors MUST be the same length and have names.

4.8 Playing with different data types

Let's mess with some vectors:

```
my_vector <- vector(length = 3)
my_vector  # this is a logical vector
```

```
## [1] FALSE FALSE FALSE
```

```
num_vector <- c(1, 2, 3, 4, 5) # numeric vector
# what happens when we add elements of different data types to a vector?
combine_vector <- c(2, "banana", TRUE)
combine_vector
```

```
## [1] "2"      "banana" "TRUE"
```

```
class(combine_vector)
```

```
## [1] "character"
```

R interprets the whole vector as character. It can't make "banana" into a number but it can turn 2 and TRUE into text strings.

You can also assign NA values to a vector of defined length as well. # R is able to handle missing values, and these missing values are given NA. When you read in an csv file with empty cells, R will assign these values as NA. A 0 is not the same as NA, since R treats 0 as a numeric data class.

```
x <- rep(NA, 10)
x[1] <- 0
# test if zero is NA
is.na(x)
```

```
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The concatenate function `c()` will also append an existing vector or create a list...

```
# cm
ab <- c('a', 'b')
ab
```

```
## [1] "a" "b"
```

```
c <- c(ab, 'DC', "EF")
c
```

```
## [1] "a"  "b"  "DC" "EF"
```

Or we can make a series of numbers…

```
my_series <- 1:10 # just integers
my_series
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# make series of numbers from 1 to 10 by increments of 0.1
my_seq <- seq(from = 1, to = 10, by = 0.1)
my_seq
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4
## [16]  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9
## [31]  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3  5.4
## [46]  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
## [61]  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4
## [76]  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7  9.8  9.9
## [91] 10.0
```

4.9 Functions

We used the function seq() above to make a series of numbers. So what is a function? A function is a defined script that is used to accomplish a particular task. Functions use an input to give a desired output. Every function has arguments that determine what kind of inputs are needed to make the function run. The inputs in a function are called arguments.

- Arguments: information that goes inside the parenthesis to tell the function what to do. For example, when we used the `seq` function above, the arguments are `from = 1`, `to = 10`, and `by = 0.1`.
- Pass: We pass a value to a function argument. Above, we pass the value 1 to the argument `from`, and the value 10 to the argument `to` and the value 0.1 to the argument `by`
- Return: This is the terminology to say that the function gives us a an output. So with the `seq(from = 1, to = 10, by = 0.1)`, the function returns a sequence of numbers

The order that we put the arguments into the function matters. It can be helpful

to define the arguments in the function. You can use the help function to ensure that your arguments are correct. `?seq`

4.10 Indexing

So now that we've created dummy vectors to play with, how do we get at its contents?

To extract elements of a vector we can give their corresponding index (square brackets `[]` are used for indexing). R uses 1-based indexing, so the first element of a vector, list, or dataframe is accessed using the index 1 (Python and C use 0-based indexing).

```
my_seq[1] # extract first element out of the my_seq vector
```

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

```
my_series[4] # extract 4th element out of the my_series vector
```

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

```
my_seq[c(2:4)] # extract elements 2 to 4 from the my_seq vector
```

```
## [1] 1.1 1.2 1.3
```

We can also extract elements by using their name instead of extracting by index. Names are another attribute that you can give to data. They are often useful so that you don't have to type out a lot of numbers.

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5) # create vector
names(x) <- c("a", "b", "c", "d", "e") # assign names to vector
# we can also name a vector 'on the fly'
x <- c(a = 5.4, b = 6.2, c = 7.1, d = 4.8, e = 7.5)
x[c("a", "c")]
```

```
##   a   c
## 5.4 7.1
```

Since comparison operators (e.g. >, <, ==) evaluate to logical vectors, we can also use them to succinctly subset vectors: the following statement gives the same result as the previous one.

```
x[x > 7] # this statement first evaluates x>7, generating a logical vector
```

```
##   c   e
## 7.1 7.5
```

```
# c(FALSE, FALSE, TRUE, FALSE, TRUE), and then selects the elements of x
# corresponding to the TRUE values.
```

If we use a negative number as the index of a vector, R will return every element except for the one specified:

```
x[-2]
```

```
##   a   c   d   e
## 5.4 7.1 4.8 7.5
```

```
# we can skip multiple elements
x[c(-1, -5)]
```

```
##   b   c   d
## 6.2 7.1 4.8
```

```
# or
x[-c(1,5)]
```

```
##   b   c   d
## 6.2 7.1 4.8
```

```
# a common mistake would be to ask R x[-1:3] # but there isn't a negative first row
# But remember the order of operations.
# : is really a function. It takes its first argument as -1,
# and its second as 3, so generates the sequence of
# numbers: c(-1, 0, 1, 2, 3).
x[-(1:3)]
```

```
##   d   e
## 4.8 7.5
```

```
# To remove elements from a vector, we need to assign the result
# back into the variable:
x <- x[-4]
x
```

```
##   a   b   c   e
## 5.4 6.2 7.1 7.5
```

Chapter 5

Debugging Code

Even when you know R well, you will inevitably find yourself in a situation where you don't understand a package, have an error, don't know the right function to use, or need ideas of how to write the correct code.

USE GOOGLE! Using a google search will most often give you results that can answer your questions. Most packages have a documentation page that you can google if you don't understand how it works too. For example, I could google, "how do I use the seq function in R to return a sequence of numbers by 0.5?" It's important to note that you are using the programming language R, and the objective that you want to take.

Never be afraid to google if you run into trouble! The website stack-overflow is your friend.

Chapter 6

Data Organization

Let's download some data and look at a dataframe

```
mort <- read.csv("./docs/data/BighornMortTable.csv")
```

If we look at the data, we can see that each column is a specific data type, and we can index it with the operator.

```
mort$Animal_ID[1]
```

```
## [1] "M267"
```

We can use the functions `head()` and `tail()` to look at the first or last 6 rows of the data frame.

```
head(mort, 3)
```

```
##   NecropDateDt NecropDate DeadDateDt DeadDate Date1stHeardDt Date1stHeard
## 1    31-Jan-25   20250131  22-Jan-25 20250122      30-Jan-25     20250130
## 2    21-Jan-25   20250121  25-Jun-23 20230625                           NA
## 3    05-Jan-25   20250105  02-Dec-24 20241202      02-Dec-24     20241202
##   AgeCarcass BestDead AnimalYear      Recorder           Herd Animal_ID     Sex
## 1          9       NA         NA     C_Massing Mt. Williamson      M267    Male
## 2        583       NA         NA M_Christopher     Mt. Baxter      M266    Male
## 3         33 20241202       2024      L_Greene      Mt. Gibbs      M265 Unknown
##   Age    AgeEstMethod GPS_CollarSerialNo_Date_FK CollarSN VHF_freq
## 1   6      Horn Rings
## 2  11      Horn Rings
## 3   0 Size of Remains
##                                                    Location  InvestReason
## 1 On Shepherd Pass trail, 1 mi up from TH, ~6800ft elevation Opportunistic
## 2                                                Sand Mtn Opportunistic
## 3                                           Gibbs, S side  Lion Cluster
```

```
##    UTM_E   UTM_N ElevDEM CarcassCon Cause_Mort CausMortAcc
## 1 384668 4067085      NA   Consumed       Lion     Certain
## 2 384854 4085215      NA   Consumed    Unknown
## 3 306232 4193944      NA   Consumed       Lion     Certain
##
## 1                      skeleton intact, ribs clipped, eaten, most meat eaten but not
## 2 old carcass, bleached bones, still close together, complete skull attached to spi
## 3                                                          rumen and lower leg, s
##
## 1                                                                              rum
## 2 could be lion, clipped ribs, broken pelvis, leg bones chewed, all bones clost toge
## 3                                                                        lion cl
##
## 1 Lion could have scavenged dead sheep. Sheep on trail, not below cliff. Collected s
## 2
## 3
##   CauseMortRevised RevisionReason PhotoTaken Houndsmen RumenIntact TracksFound
## 1                                          Y                    Y           N
## 2                                          Y                    N           N
## 3                                          Y                    Y
##   ScatFound DragMarks Latrine Femur Tooth Mandible Skull LionID
## 1                 N             N     N        N     N
## 2         N       N       N     N     N        N     N
## 3                                                        267
##   KillLocUsingGPSData LionClusterID
## 1                   N
## 2                   N
## 3                   Y  267_20241202
```

Similarly, `[[` will act to extract a single column:

```
mort[["AnimalYear"]][500]
```

```
## [1] 2016
```

We could also use brackets to index specific elements of the dataframe. Remember that when you index a dataframe using square brackets, the order is rows, columns.

```
# [rows, columns]
mort[1,2] # grab element at first row and second column
```

```
## [1] 20250131
```

```
mort[1,1:6] # grab row 1 in columns 1-6
```

```
##   NecropDateDt NecropDate DeadDateDt DeadDate Date1stHeardDt Date1stHeard
## 1    31-Jan-25   20250131  22-Jan-25 20250122      30-Jan-25     20250130
```

```
mort[1:6,5] # grab row 1-6 in column 5
```

```
## [1] "30-Jan-25" ""          "02-Dec-24" ""          "08-May-24" "08-Mar-24"
```

```
mort[1:6,1:6] # grab row 1-6 in columns 1-6
```

```
##   NecropDateDt NecropDate DeadDateDt DeadDate Date1stHeardDt Date1stHeard
## 1    31-Jan-25   20250131  22-Jan-25 20250122      30-Jan-25     20250130
## 2    21-Jan-25   20250121  25-Jun-23 20230625                           NA
## 3    05-Jan-25   20250105  02-Dec-24 20241202      02-Dec-24     20241202
## 4    08-Sep-24   20240908  26-Feb-17 20170226                           NA
## 5    09-Jul-24   20240709  11-Mar-24 20240311      08-May-24     20240508
## 6    17-Jun-24   20240617  08-Mar-24 20240308      08-Mar-24     20240308
```

```
head(mort[,1:3], 5) # grab all rows from columns 1-3, show me the first 5 rows
```

```
##   NecropDateDt NecropDate DeadDateDt
## 1    31-Jan-25   20250131  22-Jan-25
## 2    21-Jan-25   20250121  25-Jun-23
## 3    05-Jan-25   20250105  02-Dec-24
## 4    08-Sep-24   20240908  26-Feb-17
## 5    09-Jul-24   20240709  11-Mar-24
```

You can coerce lists to data.frames, assuming they follow the data frame rules.

```
test.list <- list(1, "banana", TRUE)
class(test.list)
```

```
## [1] "list"
```

```
my_seq <- as.data.frame(test.list)
class(my_seq)
```

```
## [1] "data.frame"
```

R is a natively vectorized language. Meaning it automatically performs vector operations. All of this starts to make sense if you're having trouble subsetting with `$` or even `[]` or `[[]]`.

Chapter 7

Exploring Data Frames

Let's check out the data! Data exploration is one of the most important parts of working in R and it's the first thing you should always do when looking at new data!

```
# str() shows us the structure of the data, including the data mode,
# the dimensions of # the dataframe including the data mode, the dimensions of
# the dataframe, and a few observations
str(mort)
```

```
## 'data.frame':    1061 obs. of  45 variables:
##  $ NecropDateDt              : chr  "31-Jan-25" "21-Jan-25" "05-Jan-25" "08-Sep-24" ...
##  $ NecropDate                : int  20250131 20250121 20250105 20240908 20240709 20240617 2024
##  $ DeadDateDt                : chr  "22-Jan-25" "25-Jun-23" "02-Dec-24" "26-Feb-17" ...
##  $ DeadDate                  : int  20250122 20230625 20241202 20170226 20240311 20240308 2023
##  $ Date1stHeardDt            : chr  "30-Jan-25" "" "02-Dec-24" "" ...
##  $ Date1stHeard              : int  20250130 NA 20241202 NA 20240508 20240308 20230925 2024012
##  $ AgeCarcass                : int  9 583 33 NA 120 NA 254 123 120 21 ...
##  $ BestDead                  : int  NA NA 20241202 20170226 20240311 20240308 20230925 2024012
##  $ AnimalYear                : int  NA NA 2024 2016 2023 2023 2023 2023 2023 2023 ...
##  $ Recorder                  : chr  "C_Massing" "M_Christopher" "L_Greene" "L_Greene" ...
##  $ Herd                      : chr  "Mt. Williamson" "Mt. Baxter" "Mt. Gibbs" "Mt. Langley" ..
##  $ Animal_ID                 : chr  "M267" "M266" "M265" "S217" ...
##  $ Sex                       : chr  "Male" "Male" "Unknown" "Male" ...
##  $ Age                       : chr  "6" "11" "0" "5" ...
##  $ AgeEstMethod              : chr  "Horn Rings" "Horn Rings" "Size of Remains" "Horn Rings" .
##  $ GPS_CollarSerialNo_Date_FK: chr  "" "" "" "" ...
##  $ CollarSN                  : chr  "" "" "" "" ...
##  $ VHF_freq                  : chr  "" "" "" "" ...
##  $ Location                  : chr  "On Shepherd Pass trail, 1 mi up from TH, ~6800ft elevatio
##  $ InvestReason              : chr  "Opportunistic" "Opportunistic" "Lion Cluster" "opportunis
```

```
##  $ UTM_E                    : int  384668 384854 306232 378981 299699 352085 39988
##  $ UTM_N                    : int  4067085 4085215 4193944 4046150 4216718 4142608
##  $ ElevDEM                  : int  NA NA NA NA NA NA NA NA NA NA ...
##  $ CarcassCon               : chr  "Consumed" "Consumed" "Consumed" "Scavenged" ..
##  $ Cause_Mort               : chr  "Lion" "Unknown" "Lion" "Unknown not Predation"
##  $ CausMortAcc              : chr  "Certain" "" "Certain" "Certain" ...
##  $ ConditionNotes           : chr  "skeleton intact, ribs clipped, eaten, most meat
##  $ Evidence                 : chr  "rumen intact, ribs clipped, hair plucked." "cou
##  $ Comments                 : chr  "Lion could have scavenged dead sheep. Sheep on
##  $ CauseMortRevised         : chr  "" "" "" "" ...
##  $ RevisionReason           : chr  "" "" "" "" ...
##  $ PhotoTaken               : chr  "Y" "Y" "Y" "" ...
##  $ Houndsmen                : chr  "" "" "" "" ...
##  $ RumenIntact              : chr  "Y" "N" "Y" "" ...
##  $ TracksFound              : chr  "N" "N" "" "" ...
##  $ ScatFound                : chr  "" "N" "" "" ...
##  $ DragMarks                : chr  "N" "N" "" "" ...
##  $ Latrine                  : chr  "" "N" "" "" ...
##  $ Femur                    : chr  "N" "N" "" "n" ...
##  $ Tooth                    : chr  "N" "N" "" "n" ...
##  $ Mandible                 : chr  "N" "N" "" "n" ...
##  $ Skull                    : chr  "N" "N" "" "n" ...
##  $ LionID                   : chr  "" "" "267" "" ...
##  $ KillLocUsingGPSData      : chr  "N" "N" "Y" "n" ...
##  $ LionClusterID            : chr  "" "" "267_20241202" "" ...
```

```
length(mort) # gives the number of columns
```

```
## [1] 45
```

```
nrow(mort) # to get the number of rows
```

```
## [1] 1061
```

```
ncol(mort) # number of columns
```

```
## [1] 45
```

```
dim(mort) # or both at once, dim()
```

```
## [1] 1061   45
```

```
colnames(mort) # names of the columns
```

```
##  [1] "NecropDateDt"              "NecropDate"
##  [3] "DeadDateDt"                "DeadDate"
##  [5] "Date1stHeardDt"            "Date1stHeard"
##  [7] "AgeCarcass"                "BestDead"
##  [9] "AnimalYear"                "Recorder"
```

```
## [11] "Herd"                          "Animal_ID"
## [13] "Sex"                           "Age"
## [15] "AgeEstMethod"                  "GPS_CollarSerialNo_Date_FK"
## [17] "CollarSN"                      "VHF_freq"
## [19] "Location"                      "InvestReason"
## [21] "UTM_E"                         "UTM_N"
## [23] "ElevDEM"                       "CarcassCon"
## [25] "Cause_Mort"                    "CausMortAcc"
## [27] "ConditionNotes"                "Evidence"
## [29] "Comments"                      "CauseMortRevised"
## [31] "RevisionReason"                "PhotoTaken"
## [33] "Houndsmen"                     "RumenIntact"
## [35] "TracksFound"                   "ScatFound"
## [37] "DragMarks"                     "Latrine"
## [39] "Femur"                         "Tooth"
## [41] "Mandible"                      "Skull"
## [43] "LionID"                        "KillLocUsingGPSData"
## [45] "LionClusterID"
```

It's a good time to ask ourselves if the structure R is reporting matches our intuition:

- Does the data type for each column make sense?
- If not, we need to sort out those issues now so we
- Don't run into issues down the road. Once we're happy with our data types, we can really start digging into our data!

We can also examine individual columns:

```
class(mort$AnimalYear)
```

```
## [1] "integer"
```

```
class(mort$Animal_ID)
```

```
## [1] "character"
```

```
str(mort$Cause_Mort)
```

```
##  chr [1:1061] "Lion" "Unknown" "Lion" "Unknown not Predation" "Lion" ...
```

7.1 Data Frame Manipulation

Let's create a new column to hold information on whether bighorn age is < 5:

```
below_five <- mort$Age < 5
head(below_five)
```

```
## [1] FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

We've simply created a vector with TRUE/FALSE values; but we can add this to our dataframe using:

```
mort <- cbind(mort, below_five)
head(cbind(mort, below_five), 3)
```

```
##   NecropDateDt NecropDate DeadDateDt DeadDate Date1stHeardDt Date1stHeard
## 1    31-Jan-25   20250131  22-Jan-25 20250122      30-Jan-25     20250130
## 2    21-Jan-25   20250121  25-Jun-23 20230625                         NA
## 3    05-Jan-25   20250105  02-Dec-24 20241202      02-Dec-24     20241202
##   AgeCarcass BestDead AnimalYear      Recorder           Herd Animal_ID     Sex
## 1          9       NA         NA     C_Massing Mt. Williamson      M267    Male
## 2        583       NA         NA M_Christopher     Mt. Baxter      M266    Male
## 3         33 20241202       2024      L_Greene      Mt. Gibbs      M265 Unknown
##   Age    AgeEstMethod GPS_CollarSerialNo_Date_FK CollarSN VHF_freq
## 1   6      Horn Rings
## 2  11      Horn Rings
## 3   0 Size of Remains
##                                                  Location  InvestReason
## 1 On Shepherd Pass trail, 1 mi up from TH, ~6800ft elevation Opportunistic
## 2                                                Sand Mtn Opportunistic
## 3                                           Gibbs, S side  Lion Cluster
##    UTM_E   UTM_N ElevDEM CarcassCon Cause_Mort CausMortAcc
## 1 384668 4067085      NA   Consumed       Lion     Certain
## 2 384854 4085215      NA   Consumed    Unknown
## 3 306232 4193944      NA   Consumed       Lion     Certain
##
## 1                      skeleton intact, ribs clipped, eaten, most meat eaten but not
## 2 old carcass, bleached bones, still close together, complete skull attached to spin
## 3                                                          rumen and lower leg, s
##
## 1                                                                              rum
## 2 could be lion, clipped ribs, broken pelvis, leg bones chewed, all bones clost toge
## 3                                                                        lion cl
##
## 1 Lion could have scavenged dead sheep. Sheep on trail, not below cliff. Collected s
## 2
## 3
##   CauseMortRevised RevisionReason PhotoTaken Houndsmen RumenIntact TracksFound
## 1                                          Y                    Y           N
## 2                                          Y                    N           N
## 3                                          Y                    Y
##   ScatFound DragMarks Latrine Femur Tooth Mandible Skull LionID
## 1                 N            N     N        N     N
## 2         N       N       N    N     N        N     N
## 3                                                  267
```

```
##   KillLocUsingGPSData LionClusterID below_five below_five
## 1                   N                   FALSE      FALSE
## 2                   N                    TRUE       TRUE
## 3                   Y  267_20241202       TRUE       TRUE
```

```
head(mort, 3)
```

```
##   NecropDateDt NecropDate DeadDateDt DeadDate Date1stHeardDt Date1stHeard
## 1    31-Jan-25   20250131  22-Jan-25 20250122      30-Jan-25     20250130
## 2    21-Jan-25   20250121  25-Jun-23 20230625                           NA
## 3    05-Jan-25   20250105  02-Dec-24 20241202      02-Dec-24     20241202
##   AgeCarcass BestDead AnimalYear      Recorder           Herd Animal_ID     Sex
## 1          9       NA         NA     C_Massing Mt. Williamson      M267    Male
## 2        583       NA         NA M_Christopher     Mt. Baxter      M266    Male
## 3         33 20241202       2024      L_Greene      Mt. Gibbs      M265 Unknown
##   Age    AgeEstMethod GPS_CollarSerialNo_Date_FK CollarSN VHF_freq
## 1   6      Horn Rings
## 2  11      Horn Rings
## 3   0 Size of Remains
##                                                    Location  InvestReason
## 1 On Shepherd Pass trail, 1 mi up from TH, ~6800ft elevation Opportunistic
## 2                                                 Sand Mtn Opportunistic
## 3                                            Gibbs, S side  Lion Cluster
##    UTM_E   UTM_N ElevDEM CarcassCon Cause_Mort CausMortAcc
## 1 384668 4067085      NA   Consumed       Lion     Certain
## 2 384854 4085215      NA   Consumed    Unknown
## 3 306232 4193944      NA   Consumed       Lion     Certain
##
## 1                      skeleton intact, ribs clipped, eaten, most meat eaten but not all, legs
## 2 old carcass, bleached bones, still close together, complete skull attached to spine, couple
## 3                                                          rumen and lower leg, snow may ha
##
## 1                                                                                rumen intact,
## 2 could be lion, clipped ribs, broken pelvis, leg bones chewed, all bones clost together, but
## 3                                                                      lion cluster, bro
##
## 1 Lion could have scavenged dead sheep. Sheep on trail, not below cliff. Collected scat in pos
## 2
## 3
##   CauseMortRevised RevisionReason PhotoTaken Houndsmen RumenIntact TracksFound
## 1                                          Y                    Y           N
## 2                                          Y                    N           N
## 3                                          Y                    Y
##   ScatFound DragMarks Latrine Femur Tooth Mandible Skull LionID
## 1                 N             N     N        N     N
## 2         N       N       N     N     N        N     N
```

```
## 3                                                                 267
##   KillLocUsingGPSData LionClusterID below_five
## 1                   N                   FALSE
## 2                   N                    TRUE
## 3                   Y  267_20241202       TRUE
```

Note that if we try to add a vector of below_average with a different number of entries then the number of rows in our data frame, it would fail:

```
below_five <- c(TRUE, TRUE, TRUE, TRUE, TRUE)
head(cbind(mort, below_five))
```

The sequence TRUE,TRUE,FALSE is repeated over all the gapminder rows. Let's overwite the content of gapminder with our new data frame.

```
below_five <-  as.logical(mort$Age<5)
gapminder <- cbind(mort, below_five)
```

How about adding rows?

```
# first let's make a mortality dataframe of 3 columns
mort_short <- mort[,c("Animal_ID", "Age", "Cause_Mort")]
new_row <- list('S781', 3, "Lion")
# note that we have to make the row a list, since each column has a different data typ
mort_new <- rbind(mort_short, new_row)
tail(mort_new) # lets you look at the last 6 lines of data
```

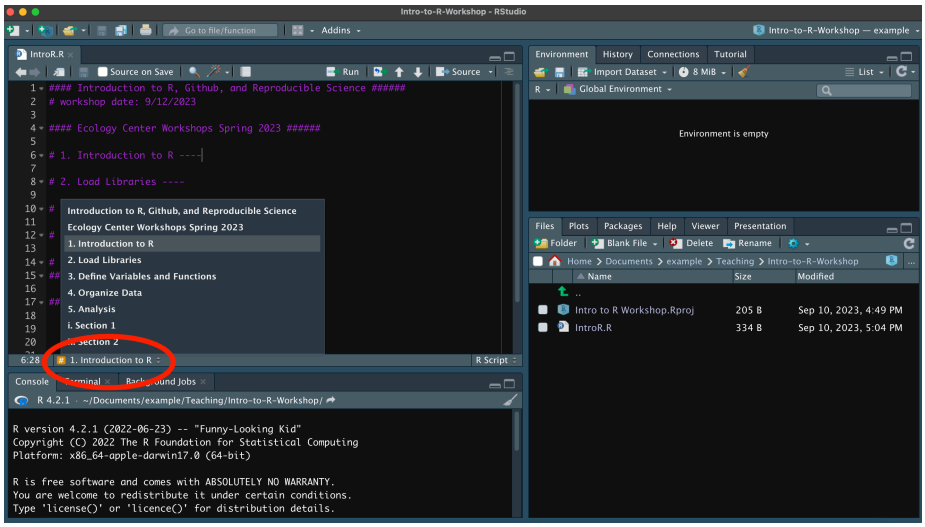
Chapter 8

Making Readable R Code

8.1 Outlining R Scripts

R scripts can get long, and it can get a bit unwieldy to navigate between different sections of your script. One neat way to outline your script is by using comment headers as bookmarks. This can be accomplished by either putting `----` or `####` after a commented line.

```
# 1. Section ---- ## a. Subsection #### ### i. Sub-subsection ----
```



8.2 Making Readable R Code

1) Start each script with a brief description of what it does, who wrote it, and the last date it was edited.
2) Then load all required packages, define functions, and set any variables used consistently throughout your script.

- Example: If you are going to be exporting many plots during the course of your script, you could set a variable to the output folder. Setting variables allows you to not have repeat code. `output_plots <- "output/plots/"`
- You can source functions from a separate script using `source()`. Remember what working directory you are in when sourcing a script.

4) Use comments to mark off sections of code.
5) Outline your script
6) Name and style code consistently. Spacing can make code more readable, but it's really up to personal preference.
7) Break code into small, discrete pieces.
8) Think about the best way to organize your directory given your objectives. I often have sub folders for data, output, and docs.
9) Start with a clean environment instead of saving the work space.
10) Consider using version control (i.e. Github).