CDFW Introduction to R Programming

Liz Siemion & Dave German

2025-05-21

Contents

Chapter 1

Overview

1.1 Workshop Goals

This course introduces the foundational concepts of programming in R and using RStudio, with a focus on producing reproducible and well-documented workflows. Our goal is to equip you with the skills to continue learning R independently and to integrate R programming into your analyses and workflows.

Artwork by Allison Horst

1.2 Workshop Content

1. Intro to R and R-Studio
2. Directory Structure
3. File paths
4. Tidy Data
5. Packages
6. Working with R objects
7. Data types (factor, numeric, character, and logical)
8. Data classes (scalar, vector, data frames, matrices, lists)
9. Functions (arguments, function workflow)
10. Indexing
11. Importing and exporting data files (csv, xlsx, mdb, rds)]
12. Data organization and manipulation (baseR and tidyverse)
13. Vectorized Operations
14. Basic plotting
15. Control Structures
16. For-loops

17. Basic statistics & statistical summaries
18. Debugging
19. Making readable R code

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

Chapter 2

Intro to R and R-Studio

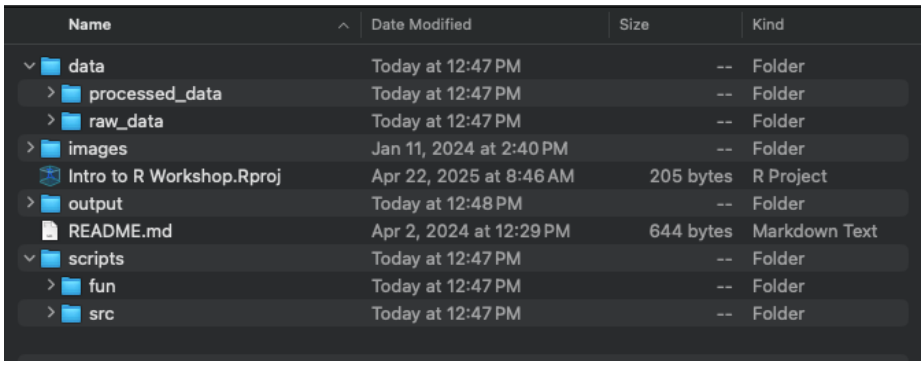
2.1 What is R?

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

RStudio is the Integrated Development Environment (IDE) for the R programming language that makes writing, running, and organizing R code more efficient. RStudio provides a centralized interface where you can manage your code, working directory, data, output, and environment all in one place. On my MacBook, I keep R and RStudio in the Applications folder, and on Windows, I store them on the C drive. You will need to download R before downloading RStudio.

2.2 Directory Structure

Before creating an RStudio Project, it's important to think about how your project is structured, as this will shape your directory (i.e. folder) organization. I typically create a separate folder for each analysis, and within each, I include subfolders for data, output, and scripts. Other folders might include figures, results, and documents. You may want to consider housing two projects in the same directory if they utilize the same data. **All files need to be readable by the computer and should not contain white spaces, punctuation, or special characters.** I generally follow the same naming conventions for my files (e.g., camelCase, snake_case, kebab-case, PascalCase).



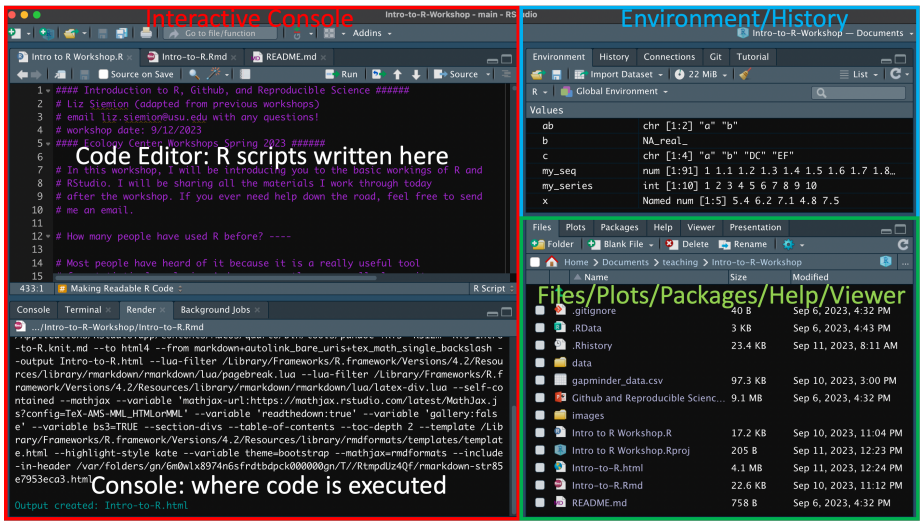
After setting up your directory structure, the next step is to create an RStudio Project in R Studio.

EXERCISE 1: CREATING A WORKING DIRECTORY

1. Create a new working directory named `IntroR`. You can put this folder in whichever location makes sense for you on your computer.
2. Create subfolders named `data`, `output`, and `scripts`

2.3 R Studio

First let's go over the basics of RStudio. When you first open RStudio, you will be greeted by three panels:



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

- The top-left panel in RStudio is the code editor, where you write and save your code. When you run a line of code from the code editor (i.e. script), the output appears in the Console (bottom-left panel). While you can type and run code directly in the Console, it won't be saved when you close R unless you explicitly save your R history. That's why it's best to write your code in the code editor—so you have a permanent, editable copy—and send lines to the Console to execute as needed.

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

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

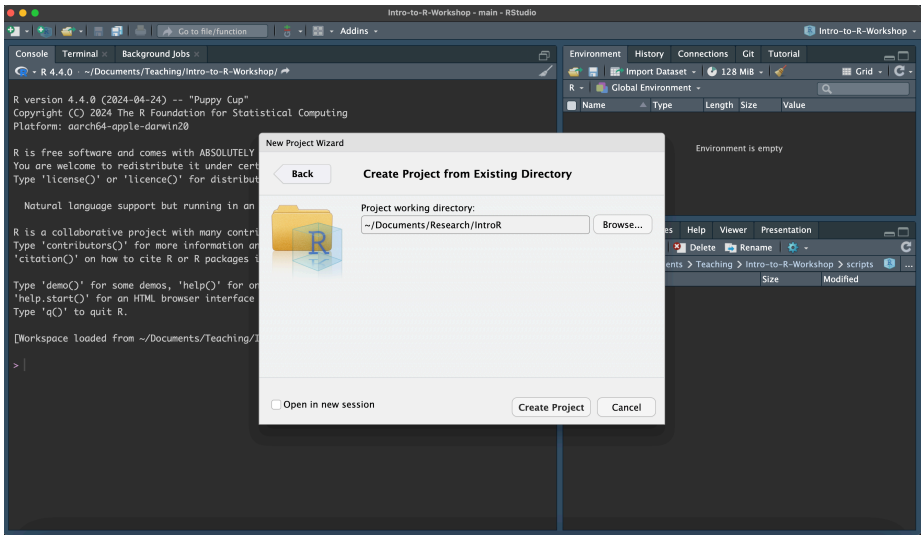
3) **Files/Plots/Packages/Help/Viewer (tabbed in lower right)**

- **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 figures 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.
- **Viewer** displays interactive or web-based content
- **Presentation** displays slide-style documents created using R Markdown

2.4 RStudio Projects

Now that we understand the basics of RStudio, let's create an RStudio Project that will live in the directory folder of the project. RStudio Projects are a self-contained, portable work space where you can have your data, code, and output all in one place. RStudio Projects are also great to use for reproducibility because they are self-contained and easy to share with collaborators. This means you can compress the entire RStudio Project into a ZIP file and share it with a collaborator, who should then be able to run your code and reproduce the same results. Let's go through how to set up an RStudio Project.

Steps for Making an RProject



1) Open the `File` menu from the upper left.
2) Select `Existing Directory` since we have already set up our project's directory
3) Navigate to the directory folder
4) Select `Create Project`

Each time you open this RStudio project, the working directory is automatically set to the `IntroR` folder, where the project is saved. This means you don't need to manually set the working directory, as RStudio Projects handle it for you, helping keep your files and code organized and consistent.

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

EXERCISE 2: CREATE AN RSTUDIO PROJECT

1. Create an RStudio PRoject in your `IntroR` folder.
2. Set a new editor theme. To change the editor theme, go to **Tools > Global Options**, click on the **Appearance tab**, and choose a new **editor theme** from the list. Each theme will be previewed in the right window.

Chapter 3

File paths

To manually set the working directory or load a file from a specific location, it's important to understand how file paths work.

3.1 What is a file path?

Your computer organizes files using a system of nested folders (i.e. directory structure), where each folder can contain other folders or files.

Image from R for Epidemiology (https://www.r4epi.com)

File paths are addresses to different locations (e.g. files, documents) 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 \ or `//`
- On Mac/Linux, it is `/`

R uses the / as the folder separator, even on Windows. So if you copy a file path from File Explorer (which uses), be sure to either replace all backslashes () with forward slashes (/), or use double backslashes (\) so R can read the path correctly.

3.3 Absolute and Relative file paths

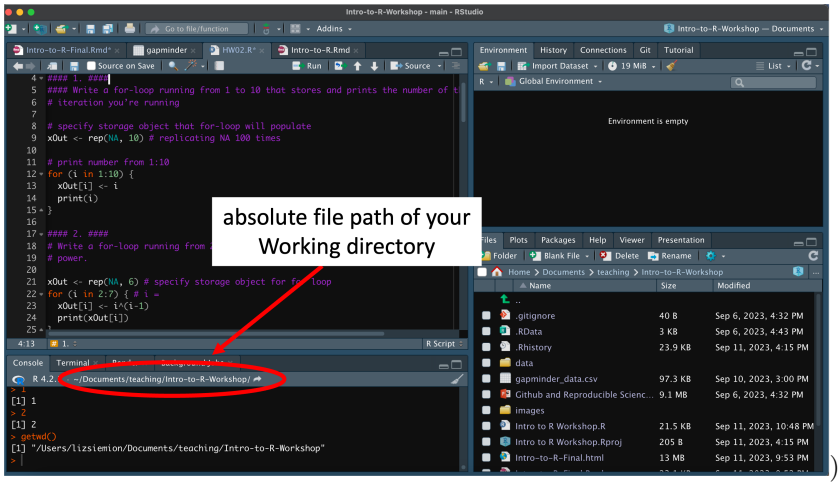
We can use `Absolute` or `Relative` file paths to give R directions to where we want to go.

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.
```

Artwork by Allison Horst

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?

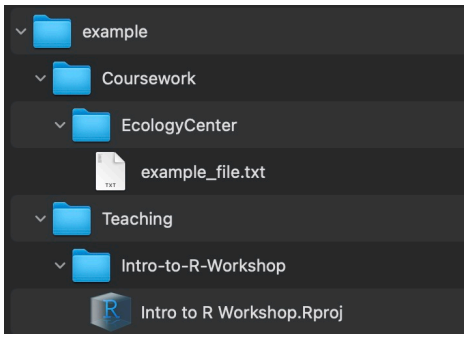
The absolute path is the full file path from our computer's root directory. If we want to use the relative path, we need tell R to go up a given number of parent folder levels from a working directory, and then to the given location within that parent folder. This can be accomplished using `../` syntax.

`./ 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()`.

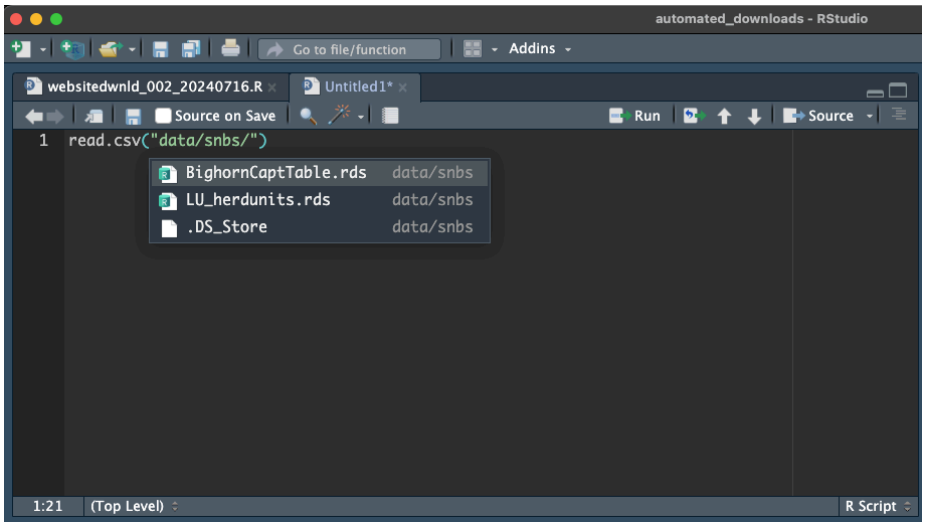
```
read.csv("../../Coursework/ecologycenter/example_file.txt")
```

3.5 Using the Tab Shortcut in RStudio

The Tab shortcut in RStudio is a powerful shortcut that helps you write code faster and with fewer errors. It's especially helpful for auto-completing file paths, function names, object names, and more. When you're loading a file (e.g., using read.csv()), you can use the Tab key to help you find and insert the correct relative file path.

For example: `read.csv("data/`

Then press the Tab key. A drop down list will appear showing the contents of the data/ folder (if it exists). You can then use the arrow keys to select a file and select Enter to insert it into the code. The Tab key shortcut helps avoid typos and ensures you're referencing the correct file.



EXERCISE 1: FILE PATHS

1. Enter `getwd()` into the console to return the absolute file path
2. Use a relative file path to load the bighorn capture table using read.csv("TYPE_RELATIVE_FILE_PATH"). The read.csv() function tells R to load the CSV file into your Environment. Make sure the file path is placed inside quotes and within the parentheses. Tip: Try using the Tab key after typing the opening quote to help you navigate and generate the correct relative file path.

Chapter 4

Tidy Data

4.1 Tidy Data

Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency, and collaboration by Julia Lowndes and Allison Horst

4.2 Document your code

It's good practice to include comments and documentation explaining what your code is doing and why. This makes it easier to understand your code, which can be useful if your code is complex or you haven't worked with it in awhile. Good documentation helps with debugging, collaboration, and keeping your workflow organized and reproducible. RMarkdown and README files are useful methods for documenting your progress.

4.3 Rows for Observations, Columns for Variables

When organizing your data, it is best to keep it in long format, where each row represents a new observation and each column represents a new variable.

4.4 Problemmatic Practices

- Avoid putting units in cells
- Do not use white spaces in column names
- Recognize the difference between zero and a missing value
- Ensure that you consistently format your values in your spreadsheets

Chapter 5

Using R

5.1 Installing Packages

Sometimes you will want to use tools that are not built into the baseR code. You can download these tools from R repositories as `packages`. A package is a collection of functions, data, and documentation designed to accomplish a particular task. An R package can help with tasks such as data visualization, statistical modeling, or data cleaning. You can install an R package through the RStudio interface, or through entering code into the console.

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

EXERCISE

Load Tidyverse package using RStudio

1. Select `Packages`
2. Select `Install`
3. Type `tidyverse`
4. Select `Install`
5. Bonus: Install tidyverse in the code editor using the example code above.

We will go over what tidyverse is later on in this course

If two different packages contain functions with the same name, R will mask one with the other based on the order they are loaded. To avoid confusion, you can explicitly tell R which package to use with the `package::function()` format. For example, the `filter()` function exists in both `stats` and `dplyr` packages.

If you specifically want the version from `dplyr`, you can write `dplyr::filter()` in your code. We will talk more about this later on.

5.2 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.

EXERCISE

1. Create a new script (**File > New File > R Script**)
2. In the first line, type `# Intro R`
3. On line 2, type `1 + 1`
4. Select `Run` in the upper right corner of the console. You can also use the shortcut `control + enter` on Windows and `command + enter` on a Macbook. If the keyboard shortcuts aren't working, navigate to **Tools > Modify Keyboard Shortcuts** , type **Run Current Line or Section** into the search bar, double-click the shortcut, remove the current key combination, and enter your preferred shortcut.

5.3 Functions

A function is a defined script that is used to accomplish a particular task. Functions use an input to give a desired output. R provides built-in functions, such as `mean()`, `sum()`, and `plot()`, which perform commonly used operations. Every function has arguments that determine what kind of inputs are needed to make the function run.

- Arguments: information that goes inside the parenthesis to tell the function what to do. For example, when we used the `seq` function, the arguments are `from`, `to`, and `by`. Many arguments have a default value, which is a value that is automatically used for an argument if you don't provide your own. For example, in `seq()`, the default value for the `from` argument is 1.

- Pass: We pass a value to a function argument. We can 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 an output. So with the `seq(from = 1, to = 10, by = 0.1)`, the function returns a sequence of numbers

```
function_name <- function(arg1, arg2) { # arg1 and arg2 are inputs
  # Code that processes the inputs
  result <- arg1 + arg2
  return(result)  # return() provides the output of the function
}
```

The order of arguments in a function call matters, especially when you don't name them. To avoid confusion, it's often helpful to explicitly name the arguments. You can use the help function `?` to check the correct usage of a function. For example, to learn more about the `rep()` function, you can execute the command `?rep` in your Console.

Additionally, you can define your own functions to solve specific problems and use them repeatedly in your code without having to repeat the code multiple times. For example, if you need to convert elevation from meters to feet in several places in your script, you can write a function to do it once and then use that function whenever you need it. Functions are an essential part of R programming, as they allow for cleaner, more efficient, and modular code. Although we won't be covering custom functions in this course, it's important to note that creating your own functions is a common and useful practice in programming.

```
# Write a function to convert a column of elevation in meters to feet
convert_elevation_to_feet <- function(elev_m) { # vector of elevation in meters is input (argumer
  elev_ft <- elev_m * 3.28084 # code to process elevation in meters to feet
  return(elev_ft) # return output of function
}
```

EXERCISE

1. Use the `?` function to look up the seq() function.
2. Why does `seq(from = 1, by = 3)` return a sequence successfully, while `seq(from = 40, by = 3)` results in an error? What is causing the difference in behavior? Hint: look up `seq` using the help function. Are there any default values? How might that affect the behavior of the function?

5.4 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:

Operator	Symbol(s)	Description
Parentheses	`( )`	Controls the order of operations
Exponents	`^`, `**`	Raises a number to a power
Divide	`/`	Division
Multiply	`*`	Multiplication
Add	`+`	Addition
Subtract	`-`	Subtraction

```
1 + 100 # performs addition
```

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

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

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

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

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

```
2/10000 # division
```

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

```
2 ^ 3 # exponentiation: 2 raised to the power of 3
```

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

5.5 Use R to compare things

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

Operator	Meaning
`==`	is equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`!`	not
`|`	or

Operator	Meaning
`%in%`	is contained in

Let's go through a few examples of using logical operators. 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. We will go over `%in%` later in this class.

```
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
```

EXERCISE

1. In your code editor, enter the following lines of code. Then run them to view each line of code's output in your console.
   a. 67 * 9
   b. 7 + 9 + 10
   c. 4399 - 871 * (9 + 1)
2. Evaluate whether the following logical conditions are TRUE or FALSE
   a. `(380*3) == (190*6)`
   b. `567 > 890`
   c. `30 >= (27 + 1 + 2)`
3. Bonus: What is wrong with the following code?
   a. Type `87(9 + 1)` into your console
   b. Execute that specific line of code by placing the cursor on that line and selecting `Run`.
   c. What error do you see? What do you think is happening? How might you fix this?

5.6 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. **Objects are data structures with associated data attributes.**

Object assignment lets us assign a name to a value, making it easier to use later. This helps avoid repeating code. We assign a value to a variable using the assignment arrow `<-` or the `=` so that R recognizes the name as a reference to the object. 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. Once we assign an object to a variable, it is stored in our global environment (upper right hand panel of RStudio).

```
x <- 1/40 # here we are telling R to assign 1/40 to the variable x so that it recogniz
x <-  24 # variables can easily be re-assigned/over-written
y <- x * 2 # We can use existing objeects in expressions to create new objects

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

5.6.1 Variable names

Variable names can contain letters, numbers, underscores and periods. **They CANNOT start with a number OR contain any spaces.** Recall 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*.

`EXERCISE`

1. Assign `1+56` to a variable called `x1_a`
2. Assign `sqrt(24)` to a variable called `x1_b`. Use `?sqrt` to learn more about the `sqrt()` function. What is the name of the argument of the `sqrt()` function?
3. Bonus: Chained assignments. What happens when you execute `x3.c <- y3.c <- 9/10` into your console? Are `x3.c` and `y3.c` different or equal values?

5.7 Data Types (Modes)

There are 6 main classes of common data modes (i.e. data types): `numeric`, `character`, `logical`, `integer`, `complex`, and `factor`. Data modes refers to the basic type of data stored in an object. For example, numeric mode stores numbers (integers or decimals), and character mode stores text. We typically only use `factor`, `numeric`, `character`, and `logical` data modes.

Mode	Description	Example
factor	Categorical data (special class of integer with labels stored as characters)	Levels: "Sheep", "Lion", "Deer" (internally stored as: 1, 2, 3)
numeric	Numbers (includes integers and decimals)	3.14, 42
character	Text strings	"Lion180"
logical	TRUE or FALSE values	TRUE, FALSE

To ask R what `class` a data mode or object is, we use the `class()` function.

```
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.<desired data type>`

Coercion Hierarchy (from general to specific): Character → Numeric → Logical

1. character most general: anything can be turned into a character by adding "quotes"
   - e.g. as.character(TRUE) → "TRUE"
2. numeric: can read integer and logical types as numbers. Cannot reliably coerce character strings unless they represent valid numbers
   - e.g. "3.14" → 3.14, TRUE → 1; FALSE → 0; "S438" → NA
3. logical: most specific, cannot turn turn character or numeric into a logical type without correctly specifying the value. Coercing a character or invalid numeric value to logical produces an NA
   - e.g. as.logical("sheep" → NA); as.logical(3.14 → TRUE)
4. Factor: (special case) internally stored as integers with labels. To get the original numeric value, you must coerce factor to character first, then to

numeric. Direct coercion from factor to numeric returns internal integer codes (not labels)

```
# create numeric object
a <- 45.6
class(a)
```

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

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

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

```
# Create factored object
lion <- factor(c("194", "180"), levels = c("194", "180")) # The levels must match the
lion # print value into console
```

```
## [1] 194 180
## Levels: 194 180
```

```
as.character(lion) # coerced to character
```

```
## [1] "194" "180"
```

```
as.numeric(lion) # coerced to numeric
```

```
## [1] 1 2
```

You can also coerce from factor to character to numeric in one call

```
as.numeric(as.character(lion))
```

```
## [1] 194 180
```

A common mistake in R is using data of the wrong type. In the example below, an error occurs because R cannot convert a character value into a numeric one. It's important to ensure that the data you're working with is of the correct type.

```
# convert from character to numeric
L262 <- "Lion262"
L262_numeric <- as.numeric(L262)
```

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

```
L262_numeric
```

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

EXERCISE

1. Assign `"S437"` to a variable called `sheep` (`sheep <- "S437"`). Make sure to include the `""`.
2. What type of data class is sheep? This can be figured out by doing `class(sheep)`
3. Create an object with a value of `45`. Coerce to a character class using `as.character()`.
4. Create the following objects: `S1 <- as.numeric(as.character(1))` and `S2 <- as.logical(as.numeric("1"))`.
5. Do the values change? Why or why not? Think about this answer in the context of the coercian hierarchy.
6. Bonus: What happens when you execute `sheep <- S437` without `""`? Explain the error.

5.8 Other Object Types

R includes many other object types that we won't cover in detail during this course. For instance, spatial object types are used to represent and work with geographic data. Spatial objects can take on different data types depending on if they are from the `sp` or `sf` package.

5.9 Data Structure Classes

Remember when we talked about objects as data with attributes? Well, R offers several ways to store data, depending on what kind of structure you need. The most common types are vectors, data frames, and lists. Each of these can store different types of information and are useful in different contexts when working with data. Think of them as different types of containers for data that are designed to hold and organize data in specific ways. We will be discussing `Scalar`, `Vector`, `Data Frames`, `Matrices`, and `Lists`. Below is a summary of each of these data structures, and we will also go through each of these individually.

Data Structure	Description	Can Contain Different Types?	Dimensions	Example
Scalar	A single value (a vector of length 1)	No	1 (length = 1)	`x <- 42`
Vector	A sequence of elements of the same type	No	1 (length > 1)	`c(1, 2, 3)`

Data Structure	Description	Can Contain Different Types?	Dimensions	Example
Data Frame	Table-like structure with columns of equal length	Yes (by column)	2 (rows × cols)	`data.frame(a = 1:3, b = c("x", "y", "z"))`
Tibble	A modern, enhanced version of a data frame from the **tidyverse**, with stricter handling and improved printing	Yes (by column)	2 (rows × cols)	`tibble(a = 1:3, b = c("x", "y", "z"))`
Matrix	2D array with all elements of the same type	No	2 (rows × cols)	`matrix(1:6, nrow = 2)`
List	A collection of elements that can be of different types	Yes	1 (named or not)	`list(name = "A", age = 25, scores = c(1,2,3))`

5.9.1 Scalar

A scalar object contains only one element (length is 1). An element is an individual value or item within a data structure

```
x <- 3
```

EXERCISE

1. Create a scalar object named my_scalar and assign it the value 87

5.9.2 Vector

A scalar object is a special case of a vector with a length of 1. A vector in R is essentially a collection of elements **of the same basic data type**. Each 'thing' in the vector is called an element. You can combine values into one vector with different values using the concatenate function `c()`.

```
my.deer <- c("GDL256", "RVD1011", "CDB567")
```

You can also create an empty vector. If you don't specify a data type, R will default to logical, or, you can declare an empty vector of whatever type you like.

```
my_vector <- vector(length = 3, mode = "numeric")
my_vector  # this is a logical vector
```

```
## [1] 0 0 0
```

```
num_vector <- c(101, 222, 323, 435, 556) # numeric vector
```

What happens when we add elements of different data types to a vector?

```
combine_vector <- c(211, "CDB678", TRUE)
combine_vector
```

```
## [1] "211"    "CDB678" "TRUE"
```

```
class(combine_vector)
```

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

R coerces all the elements in "combine_vector" to character. It can't make `"CDB678"` into a number but it can turn `211` and `TRUE` into text strings (think about the coercion hierarchy)

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.

```
# create a vector of 10 NA values
x <- rep(NA, 10)
# set the first element in x to be the number 0
x[1] <- 0
# test if zero is NA
is.na(x)
```

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

```
# R returns a logical vector for TRUE if the value is NA and
# FALSE for non-NA values
```

We can also make a series of numbers using `:` or the `seq()` function.

```
my_series <- 1:10 # a vector of integers of length 10
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
```

EXERCISE

1. Make a character vector of lion ids into a vector called `my_lion`
2. Make a vector of deer ages into a vector called `deer_ages`
3. Create a vector with variable name of your choice with a sequence of numbers from 1 to 100 using both the `seq()` and `:` method
4. BONUS 1: Explain the logic behind coercian that occurs in making this vector. `a.new.vector <- c(FALSE, TRUE, 0)`. Check the class of `a.new.vector` using the `class()` function.
5. BONUS 2: Create the following object: `animals <- factor(c("sheep", "lion", "deer", "deer"), levels = c("sheep", "lion", "deer"))`. Use the `class()` and `str()` functions to examine the `animals` vector. Use the `?` function to understand what `str()` does. What values are assigned to "sheep", "lion", and "deer"? Now convert (coerce) this vector of factors to show each level's label value instead of the internal integer code.

5.9.3 Lists

A list in R is essentially an object with data that can be in different data types/modes. Many functions in R return outputs as lists.

```
# 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
```

We can also append to a list with c() like we did with a vector. However, c() will now treat each element in the vector (here "Lion171" & "S222") as separate list elements to append.

```
my_new_list <- c(my_list, c("Lion171", "S222")) # here we overwrite the original
# "my_list" list to include two additional list elements of "Lion171" and "S222".
my_new_list
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "banana"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] "Lion171"
##
## [[5]]
## [1] "S222"
```

If we want to add one additional list element that contain Lion171 and S222, we have to use the `list()` function in front of concatenate.

```
my_new_list2 <- c(my_list, list(c("Lion171", "S222"))) # here we overwrite the original
```

Notice that my_new_list contains 6 elements (Lion171 and S222 are put in different lists) and my_new_list contains 5 elements (Lion171 and S222 are put into the same list).

Sometimes we want to retrieve the values from a certain element in a list. We do this with the $ operator. The $ operator is used for indexing named elements in a list. It allows you to access part of a data object for extracting or subsetting data.

```
my_lions <- list("L211", c("L194", "S789"), "L180")
names(my_lions) # are names automatically assigned? Nope, so let's assign names with the names()
```

```
## NULL
```

```
names(my_lions) <- c("Lion211", "LionsAndSheep", "Lion180")
my_lions$LionsAndSheep
```

```
## [1] "L194" "S789"
```

If you want to index a specific element in the list, you can also use brackets.

```
# index second list [[2]] and second element [2] of the second list
my_lions[[2]][2]
```

```
## [1] "S789"
```

EXERCISE

1. Create a list of length 3 with at least two different data types.
2. Use the `class` and `str` functions to explore the contents of your list.
3. Print the list you create in your Script into the Console.

5.9.4 Matrix

A matrix in R is a two-dimensional data structure that stores elements of the same type (usually numbers). You can think of it like a spreadsheet or a table with rows and columns, but every value must be the same type (e.g., all numeric or all character). Matrices are structured in terms of rows and columns. For example, a 3x4 matrix has 3 rows and 4 columns.

Matrices are useful when you want to perform mathematical operations on tables of numbers, like multiplying rows and columns, or applying functions across rows or columns.

```
MyMatrix <- matrix(1:6, nrow = 2, ncol = 3)
MyMatrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

EXERCISE

1. Create a 3x1 matrix representing deer population estimates for Round Valley. Assign it the a variable name of your choice. Each row should represent the total number of individuals in the fawn, yearling, and adult age classes. The column represents a single year.
2. What does the dim function do? Use `?dim`. Notice dim does not work on vectors.
3. Use the dim function to print the dimensions of your matrix into the console. `dim(<your matrix object>)`

5.9.5 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. Data frames are also also structured in terms of rows and columns.

```
# Create vectors for each column
sheep_id <- c("S601", "S602", "S603")
sex <- c("F", "M", "F")
age <- c(4, 6, 3)
capt_date <- as.Date(c("2024-03-15", "2024-03-16", "2024-03-18"))
capt_loc <- c("Acrodectes Peak", "Sawmill Canyon", "Goodale Canyon")
# Combine into a data frame
capt_dat <- data.frame(
  ID = sheep_id,
  Sex = sex,
  Age = age,
  CaptureDate = capt_date,
  CaptureLoc = capt_loc
)

# View the data frame
print(capt_dat)
```

```
##     ID Sex Age CaptureDate      CaptureLoc
## 1 S601   F   4  2024-03-15 Acrodectes Peak
## 2 S602   M   6  2024-03-16  Sawmill Canyon
## 3 S603   F   3  2024-03-18  Goodale Canyon
```

We can use the `$` operator on data frames, just like we do with lists, to extract a specific column by name. For example, `capt_dat$CaptureLoc` will return all the values in the `CaptureLoc` column and print them in the Console.

```
capt_dat$CaptureLoc
```

```
## [1] "Acrodectes Peak" "Sawmill Canyon"  "Goodale Canyon"
```

EXERCISE

1. Create a data frame with three columns and three rows to represent deer

mortalities. The columns should be named "deer_id", "COM", and "Age". Assign (`<-`) it a variable name of your choice.

- "deer_id" and "COM" should be character
- "Age" should be numeric

2. What are the dimensions of this data frame? What function can you use to check this?
3. Use the `nrow()` and `ncol()` functions to display the number of rows and columns in your data frame
4. Print the `Sex` column using the `$` operator. Recall that the syntax is `<df>$<column_name>`

5.10 Indexing

Indexing in R refers to the process of accessing specific elements from data structures like vectors, lists, or data frames. The type of indexing you use depends on the structure of your data and the task at hand. To extract elements from a vector, use their index inside square brackets `[]`.

Indexing Method	**Description**	**Example**
Positional Indexing	Use numeric positions (1-based) to access elements in a vector, list, or data frame.	`my_vector[2]` returns the 2nd element.
Named Indexing	Access elements by their name, making the code more readable.	`my_vector["b"]` returns the value associated with `b`.
Logical Indexing	Use a logical vector (TRUE/FALSE) to subset elements. Only elements corresponding to TRUE are selected.	`my_vector[c(TRUE, FALSE, TRUE)]` returns elements where TRUE is present.
Range Indexing	Extract a range or sequence of elements using a colon or vector of indices.	`my_vector[2:4]` returns elements from positions 2 to 4.
Negative Indexing	Exclude elements by using negative indices.	`my_vector[-2]` excludes the 2nd element.

Indexing Method	Description	Example
which() Indexing	Find the indices of elements that satisfy a condition, and use those indices for subsetting.	`which(my_vector > 20)` returns indices where the values are greater than 20.

5.10.1 Numerical Indexing

This method involves using the position of elements in a vector, list, or other data structures. R uses 1-based indexing, meaning the first element has an index of 1, not 0 (as in some other programming languages like Python or C).

```
lion_list <- c("194", "228", "267", "272", list("228", "223"))
lion_list[1] # extract first element out of the lion_list vector
```

```
## [[1]]
## [1] "194"
```

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

```
## [[1]]
## [1] "272"
```

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

```
## [[1]]
## [1] "228"
##
## [[2]]
## [1] "267"
##
## [[3]]
## [1] "272"
```

```
lion_list[[5]][1] # From the 5th list, extract the first element
```

```
## [1] "228"
```

5.10.2 Named Indexing

You can also extract elements by name rather than by index. Using names for indexing is particularly useful when you want to avoid relying on numeric positions, which may change as new data is added.

```
horn_length <- c(10, 30.4, 90, 50.8, 30) # create vector of horn lengths
names(horn_length) <- c("S4", "S567", "S489", "S488", "S89") # assign sheep names to vector
```

```
# we can also name a vector 'on the fly'
horn_length <- c(S4 = 10, S567 = 30.4, S489 = 90, S488 = 50.8, S89 = 30)
horn_length[c("S489", "S488")]
```

```
## S489 S488
## 90.0 50.8
```

5.10.3 Logical Indexing

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. This involves using logical vectors (TRUE/FALSE) to index elements. The logical vector must be of the same length as the data structure you're indexing. Only elements corresponding to TRUE are selected.

```
horn_length[horn_length > 30] # this statement first evaluates horn_length > 7, genera
```

```
## S567 S489 S488
## 30.4 90.0 50.8
```

5.10.4 Range or Sequence Indexing

You can extract a range or sequence of elements using a colon : to specify a range or a vector of indices.

```
horn_length[1:3]
```

```
##   S4 S567 S489
## 10.0 30.4 90.0
```

5.10.5 Negative Indexing

Negative indexing is used to exclude elements from a data structure. When you use a negative index, R excludes the element at that specific position. Note that negative indexing only applies to numeric positions, not named elements.

```
horn_length[-2]
```

```
##   S4 S489 S488  S89
## 10.0 90.0 50.8 30.0
```

```
# we can skip multiple elements
horn_length[c(-1, -2)]
```

```
## S489 S488  S89
## 90.0 50.8 30.0
```

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

```
## S567 S489 S488
## 30.4 90.0 50.8
```

```
# 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).
horn_length[-(1:3)]
```

```
## S488  S89
## 50.8 30.0
```

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

```
##   S4 S567 S489  S89
## 10.0 30.4 90.0 30.0
```

5.10.6 Which Indexing

The which() function returns the indices of elements that satisfy a given condition (useful in combination with logical operations).

```
to_remove <- which(horn_length > 30)
to_remove # index of elements to remove from the to_remove vector
```

```
## S567 S489
##    2    3
```

```
horn_length[to_remove]
```

```
## S567 S489
## 30.4 90.0
```

EXERCISE

1. Create the following object. new_deer <- c("RVD1", "CDB222", "EWR343", "WWR526")
2. Print the first three elements of new_deer into the Console using the sequence indexing
3. Remove the second element ("CDB222") using negative indexing, and update the new_deer object.

4. Use the `which()` function to identify the index of the element equal to "WWR526" and print the result.
5. Create the following object. `age <- c(43, 44, 33, 22)`
6. Use logical indexing to print the elements of `age` that are less than 44. Display the result in the Console.

5.10.7 Indexing different data types

Different object types in R require distinct indexing syntax, as each type has its own structure and behavior. For instance, vectors are indexed using [], while lists require [[]] to extract individual elements. Data frames can be subsetted using both [] and [[]], but the syntax varies depending on whether you're accessing rows, columns, or individual elements. Understanding the proper indexing method for each object type is crucial for effectively manipulating and extracting data in R.

Data Structure	Indexing with `[]`	Indexing with `[[]]`	Description
Vector	`my_vector[2]` - Accesses the 2nd element.	Not applicable (use `[]` for vectors).	`[]` returns a subset (vector), and `[]` is used to extract individual elements by their position.
Data Frame	`my_df[2, ]` - Accesses the 2nd row (returns a data frame).	`my_df[[2]]` - Accesses the 2nd column (returns a vector).	`[]` keeps the result as a data frame, while `[[]]` extracts a specific column or element.
Matrix	`my_matrix[2, 3]` - Accesses the element in the 2nd row and 3rd column.	Not applicable (use `[]` for matrices).	`[]` is used to index both rows and columns, but `[[]]` is not used with matrices.
List	`my_list[2]` - Returns the 2nd element of the list (as a list).	`my_list[[2]]` - Returns the 2nd element of the list (as the actual object, not as a list).	`[]` returns a sublist, while `[[]]` extracts the actual object at that index.
List of Lists	`my_list_of_lists[2]` - Returns the 2nd list in the list of lists (as a list).	`my_list_of_lists[[2]]` - Returns the 2nd list (the entire list, not just an element).	`[]` returns a sublist, while `[[]]` extracts the entire second list when using a list of lists.

Data Structure	Indexing with []	Indexing with [[]]	Description
Nested List	`my_list[[2]][[2]]` - Returns the 2nd element of the 2nd list in the list of lists (after first subsetting).	`my_list[[2]][[2]]` - Extracts the 2nd element from the 2nd list in the list.	`[]` can be used to subset lists, and `[[]]` extracts elements within the nested list structure.

EXERCISE

1. Create the following object.

- ```
  elk_df <- data.frame(
  elk_id = c("E001", "E002", "E003", "E004", "E005"),
  weight = c(350, 410, 290, 500, 375),
  herd = c("North", "South", "East", "West", "North"))
  ```

2. Print rows 1 to 3 and the second column of `elk_df` to the Console. Recall that to index data frames, you use df[rows, columns].
3. Remove the elk ids "E001" and "E002" from the data frame using negative indexing.
4. Use which() indexing to subset the `elk_df` to include only elk from the "North" herd. Assign this to a new object called "elk_north.
5. BONUS: Use a which() statement to subset `elk_df` to elk in the "North" and "East" herds

---

5.11 BaseR Plotting

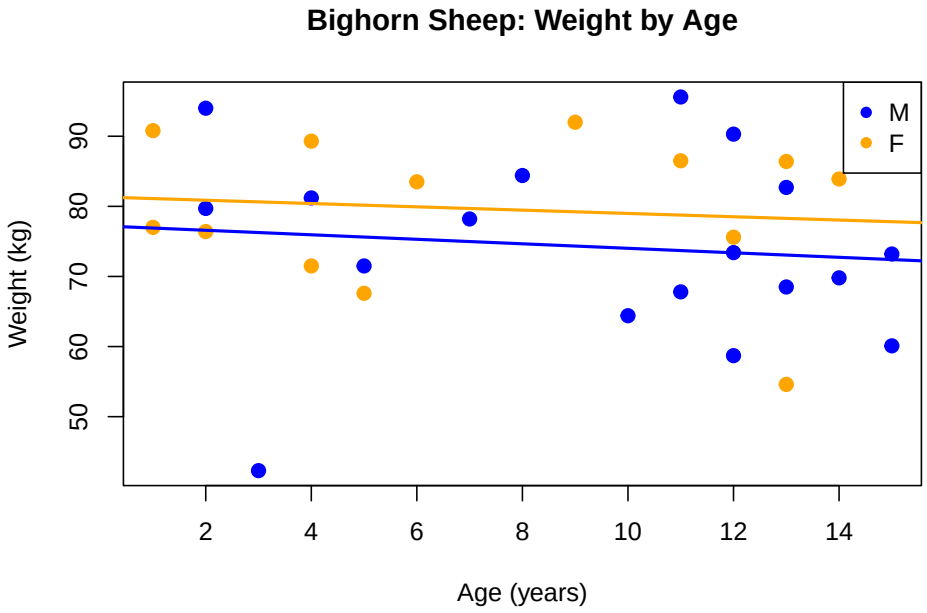
BaseR plotting is an efficient way to visualize a dataset.

```
# Create a fake data frame for bighorn sheep
bighorn_df <- data.frame(
  ID = paste0("S", 500:530),       # ID for each sheep
  Age = sample(1:15, 31, replace = TRUE),  # Random ages between 1 and 10
  Weight = round(rnorm(31, mean = 75, sd = 15), 1),  # Random weights with mean of 75 kg and sd o
  Sex = factor(sample(c("M", "F"), 31, replace = TRUE), labels = c("M", "F")),      # Random sexes
  Herd = sample(c("North", "South", "East", "West"), 31, replace = TRUE)  # Random herd names
)

# Define grey tones based on levels of the 'Sex' factor
grey_tones <- c("blue", "orange")  # grey50 for males, grey80 for females
```
```

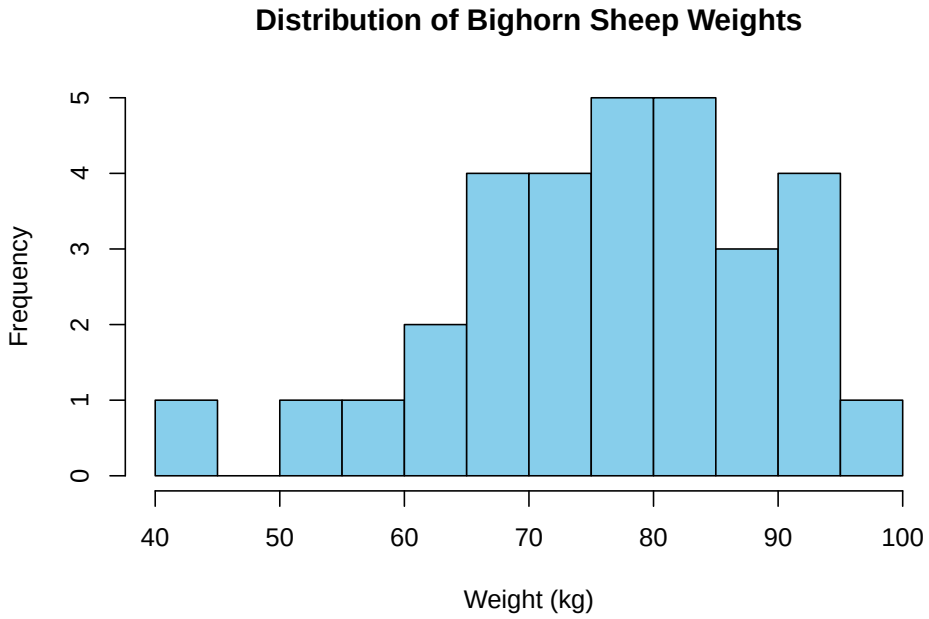
```r
# create a scatterplot
plot(x = bighorn_df$Age, # x-axis
     y = bighorn_df$Weight, # y-axis
     main = "Bighorn Sheep: Weight by Age", # title
     xlab = "Age (years)", # x-axis label
     ylab = "Weight (kg)", # y-axis label
     col = grey_tones[bighorn_df$Sex],  # point color (sex needs to be a factor)
     pch = 19,              # solid circle
     cex = 1.2)             # slightly larger points
# Add separate trend lines for each Sex group
# Males
abline(lm(Weight ~ Age, data = subset(bighorn_df, Sex == "M")), col = "blue", lwd = 2)
# Females
abline(lm(Weight ~ Age, data = subset(bighorn_df, Sex == "F")), col = "orange", lwd = 2

# Add a legend
legend("topright",
       legend = levels(bighorn_df$Sex),  # Use the levels of the factor for legend
       col = c("blue", "orange"),      # Different tones for trend lines
       pch = 16) # solid circle point type
```




```r
hist(bighorn_df$Weight,
     main = "Distribution of Bighorn Sheep Weights",
     xlab = "Weight (kg)",
     ylab = "Frequency",
     col = "skyblue",         # fill color of the bars
```

```
     border = "black",        # outline color of the bars
     breaks = 10)             # number of bins (adjust as needed)
```



EXERCISE

1. Create the following object:

```
# Create a fake data frame for bighorn sheep
bighorn_df <- data.frame(
  ID = paste0("S", 500:530),       # ID for each sheep
  Age = sample(1:15, 31, replace = TRUE),  # Random ages between 1 and 10
  Weight = round(rnorm(31, mean = 75, sd = 15), 1),  # Random weights with mean of 75 kg and sd o
  Sex = sample(c("M", "F"), 31, replace = TRUE),      # Random sexes (Male, Female)
  Herd = sample(c("North", "South", "East", "West"), 31, replace = TRUE)  # Random herd names
)
```

2. Create a histogram to visualize the age distribution of bighorn sheep.

- Set the bins to be maroon.
- Add a black border to the bins.
- Include a title and labels for the x and y axes.
- Adjust the breaks (number of bins) as needed

Chapter 6

Tidyverse

6.1 Why Tidyverse?

The `tidyverse` is a collection of R packages designed to simplify and streamline data science tasks. It promotes the concept of "tidy data," where each column is a variable and each row is an observation, which simplifies analysis and visualization. Core tidyverse packages include `ggplot2` for plotting, `dplyr` for data manipulation, and `tidyr` for reshaping data.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages -- tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts --------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

Notice the Conflicts from loading the tidyverse package.

One of the key strengths of the tidyverse is its readable syntax, especially with the use of the pipe operator `(%>%)`, which allows users to chain together multiple steps in a clear, logical order.

```
# example of using a piped operation
sheep <- filter(x = sheep, herds %in% c("Bx", "Cn")) %>%
  mutate(Yr_Month = paste(Year, Month, sep = "_"))
```

Note that I had to specify the object `sheep` in the `filter()` function, but not in the `mutate()` function. The `%>%` operator tells R to pass the sheep data frame through the pipeline, so it's automatically used in the subsequent function.

Learning tidyverse provides efficient tools and workflows for data cleaning, exploration, and analysis. One of the most commonly used packages within the `tidyverse` is `dplyr` (data plier), which provides a set of powerful and intuitive functions for manipulating data frames. With dplyr, you can easily filter rows, select columns, create new variables, group data, and summarize it—all using clear, readable code.

Key Features:

- Consistent syntax across packages, making code easier to write and read.
- Tidy data structure, where each variable is a column and each observation is a row.
- Readable workflows using the pipe operator (%>%) to chain commands clearly.
- Integrated tools for the entire data science process—importing, cleaning, transforming, visualizing, and modeling data.
- User-friendly functions with intuitive names and logical default argument values.
- Open-source and widely supported, with ample online learning resources
- Provides tools for data manipulation, exploration, and visualization.
- Uses piping (%>%) to create readable, fluent workflows (through the magrittr package)
- The Tidyverse's `Tibble` data structure offers a modern, consistent alternative to data frames, with improved subsetting behavior, better printing, and stricter data handling

6.2 Tibbles

The tidyverse uses a modern type of data frame structure called a `Tibble` that has a `tbl` class, a `tbl_df` class and a `data.frame` class. You will see tibbles as outputs from most of the tidyverse and dplyr packages.

```
# Create a tibble manually
my_tibble <- tibble(
  name = c("S432", "S678", "S573"),
  age = c(4, 2, 6)
)

class(my_tibble)
```

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

There are three key differences between tibbles and data frames: printing, subsetting, and recycling rules.

6.2.1 Improved printing

Unlike data frames, tibbles only print the first 10 rows by default and only show the columns that fit on your screen, producing more readable output. The data types of each column are displayed underneath the column names, making it easier to understand your data quickly. It also prints an abbreviated description of the column type, and uses font styles and colors for highlighting.

In addition, numbers are displayed with three significant figures by default, and a trailing dot indicates the existence of a fractional component. You can control the default appearance with options.

```
my_tibble
```

```
## # A tibble: 3 x 2
##   name    age
##   <chr> <dbl>
## 1 S432      4
## 2 S678      2
## 3 S573      6
```

6.2.2 Subsetting

Tibbles are stricter and more consistent than baseR data frames when it comes to subsetting. When subsetting with tibbles using `[`, the result is always another tibble, even if you're selecting a single column. This makes the behavior more predictable, especially in pipelines.

```
my_tibble[1]      # Returns a tibble with one column
```

```
## # A tibble: 3 x 1
##   name
##   <chr>
## 1 S432
## 2 S678
## 3 S573
```

```
my_tibble[[1]]    # Returns a vector (just the values from column name)
```

```
## [1] "S432" "S678" "S573"
```

In contrast, subsetting with data frames using `[` may return a data frame or a vector, depending on the context:

```
my_df <- data.frame(
  name = c("S432", "S678", "S573"),
  age = c(4, 2, 6))
my_df[1]      # Returns a data frame
```

```
##   name
```

```
## 1 S432
## 2 S678
## 3 S573
```

```
my_df[, 1]    # Returns a vector
```

```
## [1] "S432" "S678" "S573"
```

```
my_df[1, ]    # Returns a data frame (one row)
```

```
##   name age
## 1 S432   4
```

Tibbles are also more strict with the `$` operator: Tibbles do not allow partial matching. If the column name you use doesn't match exactly, it will return `NULL` and may show a warning. As a result, tibbles can be safer than data frames, especially with data sets with similar column names.

```
my_tibble$na
```

```
## Warning: Unknown or uninitialised column: `na`.
```

```
## NULL
```

Data frames allow partial matching, which can lead to unexpected results:

```
my_df$name
```

```
## [1] "S432" "S678" "S573"
```

```
my_df$na
```

```
## [1] "S432" "S678" "S573"
```

6.2.3 Recycling

Only values of length 1 are recycled when constructing a tibble. The first column with length different to one determines the number of rows in the tibble, conflicts lead to an error.

In R, recycling happens when you combine vectors of different lengths—shorter vectors are automatically repeated to match the longer ones. While this can be convenient, it can also lead to unintended results if you're not careful.

In a data frame recycling often happens silently, even if the lengths don't align exactly:

```
my_df <- data.frame(
  name = c("S432", "S678", "S573", "S698"),
  age = c(4, 2))
my_df
```

```
##   name age
## 1 S432   4
## 2 S678   2
## 3 S573   4
## 4 S698   2
```

Here, the shorter `age` vector `age = c(4, 2)` is silently recycled to fill the 4 rows of the data frame.

Tibbles are stricter about recycling:

- Tibbles only allow recycling if the shorter vector is length 1.
- If the lengths don't match and can't be safely recycled, tibbles will return an error—helping you catch mistakes early.

```
# This works - length 1 value recycled
my_tibble <- tibble(
  name = c("S432", "S678", "S573", "S698"),
  age = c(2))

# This fails - length 2 can't be recycled to length 4
my_tibble <- tibble(
  name = c("S432", "S678", "S573", "S698"),
  age = c(4, 2))
```

```
## Error in `tibble()`:
## ! Tibble columns must have compatible sizes.
## * Size 4: Existing data.
## * Size 2: Column `age`.
## i Only values of size one are recycled.
```

```
# Error: Tibble columns must have consistent lengths
```

6.2.4 Tibble vs. Data Frame — Key Differences

Below is a table summarizing the differences between data frames and tibbles. You can also see `vignette("invariants")` for a detailed comparison between tibbles and data frames.

Feature	Base R Data Frame	Tibble (from `tibble` package)
Printing	Prints entire dataset, which can flood console	Prints first 10 rows and fits columns to screen width
Column types shown?	No	Yes — shows column types under names
Type conversion on import	Often converts strings to factors (`read.csv`)	Does not convert strings to factors by default

Feature	Base R Data Frame	Tibble (from `tibble` package)
Partial column name match	Allows it (`df$na` may match `name`)	Disallows it — column name must match exactly
Subsetting with `[`	Can return a vector or data frame depending on syntax	Always returns a tibble (more consistent)
Recycling of short vectors	Silent, may lead to subtle bugs	Gives a warning if lengths don't match
Row names	Supports row names	Does not use row names
Integration	Basic support for base R functions	Seamless with tidyverse tools

6.3 Tidyverse Functions

`dplyr`, a main package of the `tidyverse`, consists of five main functions:

1. `filter()`: Subset the data based on specific conditions. **This is for selecting rows.**

```
# create data frame object
snbs <- data.frame(AID = c("S488", "S500", "S468"),
                   Sex = c("M", "F", "F"),
                   Herd = c("Ca", "Gb", "Wh"),
                   RU = c("NRU", "NRU", "CRU"))
# filter to females
females <- filter(snbs, Sex == "F")
head(females, 1)
```

```
##    AID Sex Herd  RU
## 1 S500   F   Gb NRU
```

2. `select()`: Choose specific columns from the data. **This is for selecting columns**

```
# select AID and RU columns
snbs_s <- select(snbs, AID, RU)

# remove one column with negative indexing
snbs_s <- select(snbs, -Sex, -RU)
head(snbs_s, 1)
```

```
##    AID Herd
## 1 S488   Ca
```

3. `mutate()`: Modify or create new variables in the data.

```
snbs_s <- mutate(snbs, AID_Sex = paste(AID, Sex, sep = "_")) %>%
  select(AID, Sex, AID_Sex)
head(snbs_s, 1)
```

```
##    AID Sex AID_Sex
## 1 S488   M  S488_M
```

4. `group_by()`: Group data based on one or more variables for grouped operations.

```
snbs_group <- group_by(snbs, RU)
snbs_group
```

```
## # A tibble: 3 x 4
## # Groups:   RU [2]
##   AID   Sex   Herd  RU
##   <chr> <chr> <chr> <chr>
## 1 S488  M     Ca    NRU
## 2 S500  F     Gb    NRU
## 3 S468  F     Wh    CRU
```

5. `summarize()`: Generate summary statistics from the data. Common summary statistics include: `n()`, `mean()`, `mode()`, `range()`, `median()`, `sd()`.

```
snbs_summ <- summarize(snbs_group, n = n())
snbs_summ
```

```
## # A tibble: 2 x 2
##   RU        n
##   <chr> <int>
## 1 CRU       1
## 2 NRU       2
```

6. `arrange()`: Sort the data in a specific order.

```
snbs_sorted <- arrange(snbs, Herd)
# Sorts character values in alphabetical order by default

snbs_sorted_desc <- arrange(snbs, desc(Herd))
# Sorts character values in descending alphabetical order
```

You can perform many operations in one call using the pipe operator:

```
snbs_tidy <- filter(snbs, Sex == "F") %>%
  mutate(AID_Sex = paste(AID, Sex, sep = "_")) %>%
  select(-RU) %>%
  group_by(Herd) %>%
  summarize(n = n()) %>%
  arrange(Herd)
```

EXERCISE 1

1. Recreate the following object:

- ```
  elk_df <- data.frame(
  elk_id = c("E001", "E002", "E003", "E004", "E005"),
  weight = c(350, 410, 290, 500, 375),
  herd = c("North", "South", "East", "West", "North"))
  ```

2. Use `filter()` to subset weights > 375
3. Create a new data frame that only contains the elk_id and herd column
4. Create a new data frame that is grouped by herd. Calculate the number of observations per herd. Arrange the final dataset in alphabetical order.

6.4 Tidyverse vs BaseR

There are many different ways to accomplish the same task in R (e.g. using tidyverse or baseR syntax). Whichever syntax you choose depends on what you're more comfortable with. Personally, I like using a mix of both. Tidyverse code is intuitive, but for really large data sets, baseR works faster. It's all about finding what works best for you and your data.

Purpose	**Tidyverse (`dplyr`)**	**Base R Equivalent**
Subset rows	`filter()`	`subset()` or logical indexing: `df[df$col == value, ]`
Select columns	`select()`	`df[, c("col1", "col2")]`
Create/modify columns	`mutate()`	`df$new_col <- ...`
Group data	`group_by()`	`tapply()`, `aggregate()`, or `split()`
Summarize data	`summarize()` (or `summarise()`)	`aggregate()` or `tapply()` with `mean()`, `sum()`, etc.

6.5 ggplot

Artwork by Allison Horst

ggplot is a powerful and flexible R package used for data visualization. It is part of the tidyverse and is built on the Grammar of Graphics—a layered approach to building plots that allows you to combine data, visual elements, and aesthetics step by step. You can customize nearly every part of a ggplot, including adding titles, axis labels, and legends; adjusting colors, shapes, and sizes; applying
  ```

different themes; adding trend lines or annotations; and creating multi-panel layouts using faceting.

Key Features

1. Layered plotting system: Build plots by adding layers (e.g., points, lines, bars).
2. Aesthetic mapping: Easily map variables to visual properties like color, size, and shape.
3. Customizable: Control themes, labels, scales, and more.
4. Works seamlessly with tidy data and other tidyverse tools.

6.5.1 Components of ggplot

When using ggplot2 in R, your code is built in layers, with each part adding a new component to the plot. Here's a breakdown of the key components you'll commonly use:

- Data: The dataset you're visualizing.
- Aesthetics (aes()): Defines how variables in the data are mapped to visual properties like position, color, or size.
- Geometries (geom_()): Determine the type of plot (e.g. points, lines, bars, etc.)

Each ggplot() call begins with the data and aesthetic mappings, and layers like geom_() or theme_() are added using the + operator. Let's go through the main parts step by step.

First we will create a fake lion dataset:

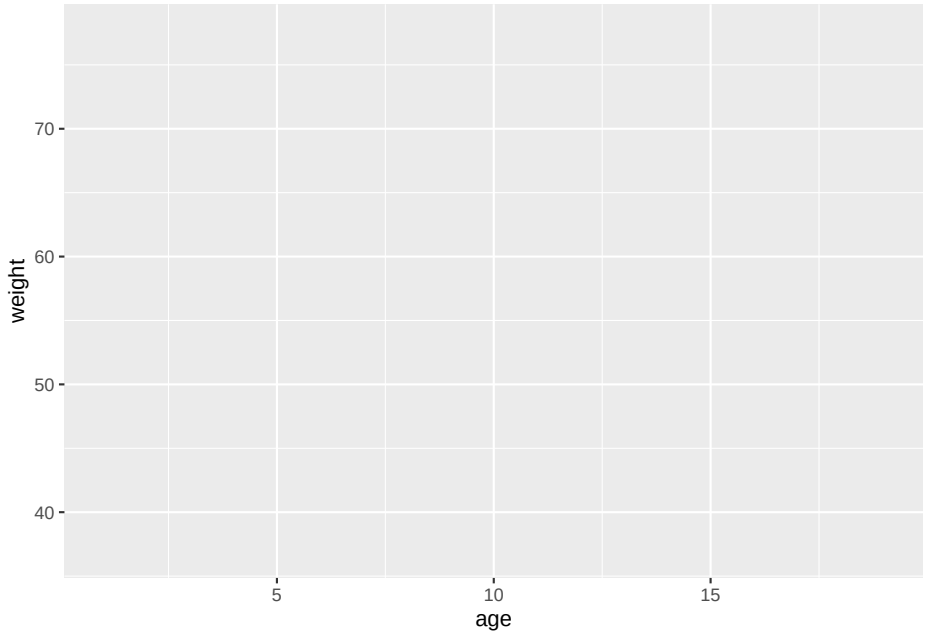
```
# Create fake lion data
lions <- data.frame(
  id = paste0("Lion", 200:219),
  age = sample(1:20, 20, replace = TRUE),          # age in years
  weight = round(rnorm(20, mean = 60, sd = 10), 1), # weight in kg
  sex = sample(c("M", "F"), 20, replace = TRUE),
  region = sample(c("Northern", "Southern", "Central", "Olancha"), 20, replace = TRUE)
)

head(lions, 3)
```

```
##        id age weight sex   region
## 1 Lion200  13   61.2   F Northern
## 2 Lion201   5   75.1   M Southern
## 3 Lion202   8   69.8   F Southern
```

1. The main ggplot() function

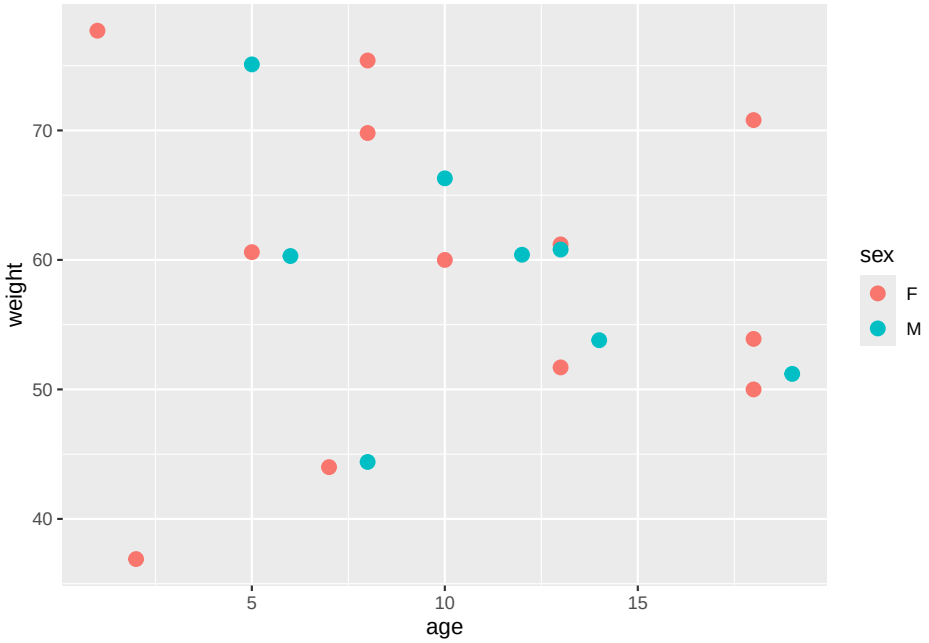
```
ggplot(data = lions, aes(x = age, y = weight, color = sex))
```



- ggplot(data = lions): This is where we tell ggplot what data we're using. In this case, it's the lions data frame.
- aes(x = age, y = weight, color = sex): This part is called the aesthetic mapping (or aes for short). It tells ggplot which variables to plot:
  - x = age: We're plotting age on the x-axis.
  - y = weight: We're plotting weight on the y-axis.
  - color = sex: We want to color the points by the sex variable. This way, male and female lions will have different colors.

2. Adding Points with geom_point()

```
ggplot(data = lions, aes(x = age, y = weight, color = sex)) +
geom_point(size = 3)
```

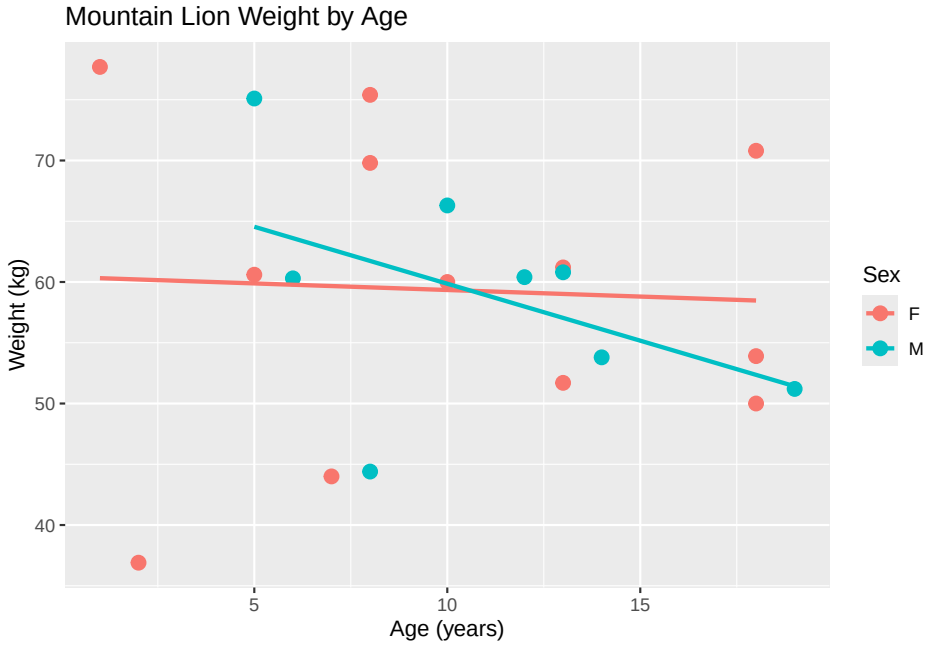


* geom_point(): This part tells ggplot to plot the data as points (a scatter plot). * size = 3: This adjusts the size of the points on the plot to make them more visible.

3. Adding a trendline and labels

```
ggplot(data = lions, aes(x = age, y = weight, color = sex)) +
  # Add scatterplot
  geom_point(size = 3) +
  # Adding trend line
  geom_smooth(method = "lm", se = FALSE, aes(color = sex), linetype = "solid") +
  labs(
    title = "Mountain Lion Weight by Age",
    x = "Age (years)",
    y = "Weight (kg)",
    color = "Sex"
  )
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



- geom_smooth: creates a trendline for each sex
- `labs()`: This function is used to add titles and labels to the plot.
  - title: Adds a title to the plot.
  - x: Labels the x-axis (Age).
  - y: Labels the y-axis (Weight).
  - color: Adds a label for the color legend, which shows the different colors for male and female lions.

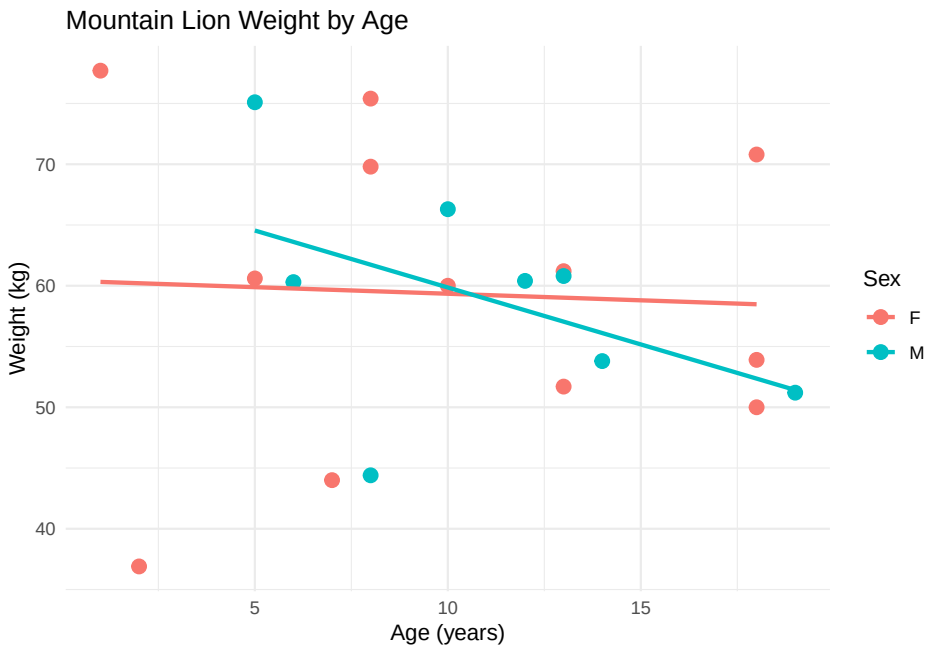
5. Final Touches with `theme_minimal()`

- theme_minimal(): This is a simple and clean theme for the plot. It removes unnecessary background grids and makes the plot look more visually appealing. There are many other different themes such as `theme_class`, `theme_dark`, and `theme_light`.

```
ggplot(data = lions, aes(x = age, y = weight, color = sex)) +
  # Add scatterplot
  geom_point(size = 3) +
  # Adding trend line
  geom_smooth(method = "lm", se = FALSE, aes(color = sex), linetype = "solid") +
  labs(
    title = "Mountain Lion Weight by Age",
    x = "Age (years)",
    y = "Weight (kg)",
    color = "Sex"
  ) +
```

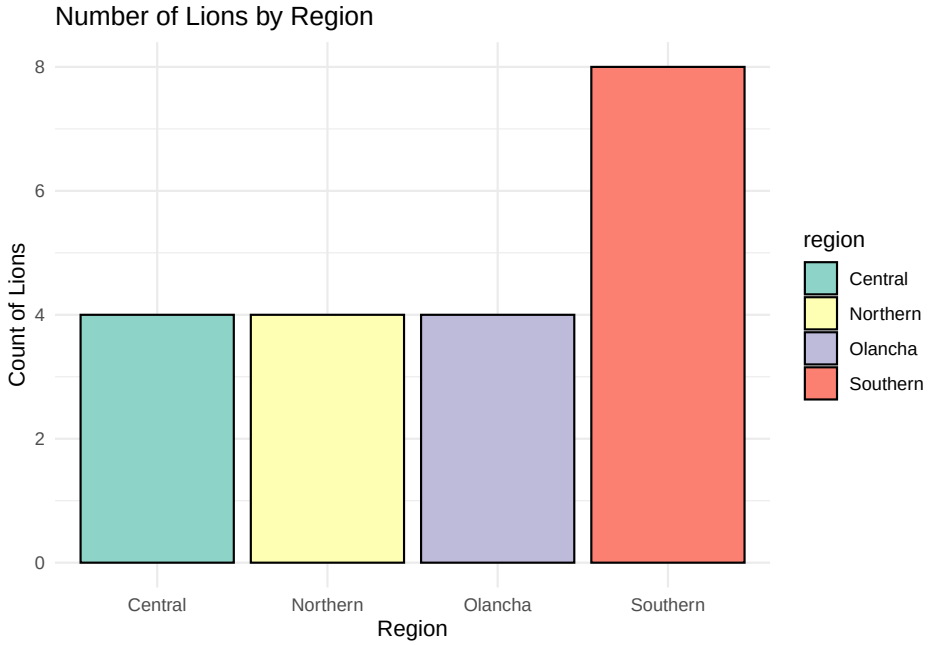
```
  theme_minimal()
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Here is an example of making a barplot that is colored by different region.

```
# Create the bar plot, colored by region
ggplot(data = lions, aes(x = region, fill = region)) +
  geom_bar(color = "black") +  # Bar border color
  labs(
    title = "Number of Lions by Region",
    x = "Region",
    y = "Count of Lions"
  ) +
  theme_minimal() +
  scale_fill_brewer(palette = "Set3")  # Using a color palette for better visuals
```



6.5.2 Different plots in ggplot

Plot Type	ggplot2 Function	Description
Scatter Plot	`geom_point()`	Shows relationship between two continuous variables
Line Plot	`geom_line()`	Displays trends over time or ordered categories
Bar Chart (counts)	`geom_bar()`	Plots the count of categories (x only)
Bar Chart (values)	`geom_col()`	Plots values directly (requires both x and y)
Boxplot	`geom_boxplot()`	Visualizes the distribution and spread by group
Histogram	`geom_histogram()`	Shows distribution of a single continuous variable

---

EXERCISE

1. Create the following object:

```
# Create a fake data frame for bighorn sheep
bighorn_df <- data.frame(
```

```
  ID = paste0("S", 500:530),       # ID for each sheep
  Age = sample(1:15, 31, replace = TRUE),  # Random ages between 1 and 10
  Weight = round(rnorm(31, mean = 75, sd = 15), 1),  # Random weights with mean of 75 kg and sd o
  Sex = sample(c("M", "F"), 31, replace = TRUE),     # Random sexes (Male, Female)
  Herd = sample(c("North", "South", "East", "West"), 31, replace = TRUE)  # Random herd names
)
```

2. Create a histogram to visualize the weights of bighorn sheep.

- Set the bins to be blue.
- Add a black outline to the bins.
- Include a title as well as labels for the x and y axes.
- Apply a theme to enhance the plot's appearance.

6.6 Tidyverse Analysis Example

Exercise – Using a data set in R called penguins_raw to perform a simple analysis

Explore the data set: * Look at the first 5 rows (head) * Look at the data structure (str) * How many samples? (nrow) * Get a list of the column names (names) * How many species of penguins (unique)

Question we are interested in: **We want to understand how body mass varies by species and sex**

6.6.1 Load libraries

First let's load the required libraries.

```
library(tidyverse)
library(palmerpenguins)
```

```
## Warning: package 'palmerpenguins' was built under R version 4.4.1
```

6.6.2 Examine data frame

Let's look at the first 10 rows of the penguins_raw data frame.

```
head(penguins_raw)
```

```
## # A tibble: 6 x 17
##   studyName `Sample Number` Species          Region Island Stage `Individual ID`
##   <chr>               <dbl> <chr>            <chr>  <chr>  <chr> <chr>
## 1 PAL0708                 1 Adelie Penguin ~ Anvers Torge~ Adul~ N1A1
## 2 PAL0708                 2 Adelie Penguin ~ Anvers Torge~ Adul~ N1A2
## 3 PAL0708                 3 Adelie Penguin ~ Anvers Torge~ Adul~ N2A1
```

```
## 4 PAL0708                4 Adelie Penguin ~ Anvers Torge~ Adul~ N2A2
## 5 PAL0708                5 Adelie Penguin ~ Anvers Torge~ Adul~ N3A1
## 6 PAL0708                6 Adelie Penguin ~ Anvers Torge~ Adul~ N3A2
## # i 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
## #   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
## #   `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
## #   `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>, Comments <chr>
```

Let's examine the structure of the penguins_raw data frame

```
str(penguins_raw)
```

```
## tibble [344 x 17] (S3: tbl_df/tbl/data.frame)
##  $ studyName          : chr [1:344] "PAL0708" "PAL0708" "PAL0708" "PAL0708" ...
##  $ Sample Number      : num [1:344] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Species            : chr [1:344] "Adelie Penguin (Pygoscelis adeliae)" "Adelie Pe
##  $ Region             : chr [1:344] "Anvers" "Anvers" "Anvers" "Anvers" ...
##  $ Island             : chr [1:344] "Torgersen" "Torgersen" "Torgersen" "Torgersen"
##  $ Stage              : chr [1:344] "Adult, 1 Egg Stage" "Adult, 1 Egg Stage" "Adult
##  $ Individual ID      : chr [1:344] "N1A1" "N1A2" "N2A1" "N2A2" ...
##  $ Clutch Completion  : chr [1:344] "Yes" "Yes" "Yes" "Yes" ...
##  $ Date Egg           : Date[1:344], format: "2007-11-11" "2007-11-11" ...
##  $ Culmen Length (mm) : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 .
##  $ Culmen Depth (mm)  : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 .
##  $ Flipper Length (mm): num [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
##  $ Body Mass (g)      : num [1:344] 3750 3800 3250 NA 3450 ...
##  $ Sex                : chr [1:344] "MALE" "FEMALE" "FEMALE" NA ...
##  $ Delta 15 N (o/oo)  : num [1:344] NA 8.95 8.37 NA 8.77 ...
##  $ Delta 13 C (o/oo)  : num [1:344] NA -24.7 -25.3 NA -25.3 ...
##  $ Comments           : chr [1:344] "Not enough blood for isotopes." NA NA "Adult no
##  - attr(*, "spec")=
##   .. cols(
##   ..   studyName = col_character(),
##   ..   `Sample Number` = col_double(),
##   ..   Species = col_character(),
##   ..   Region = col_character(),
##   ..   Island = col_character(),
##   ..   Stage = col_character(),
##   ..   `Individual ID` = col_character(),
##   ..   `Clutch Completion` = col_character(),
##   ..   `Date Egg` = col_date(format = ""),
##   ..   `Culmen Length (mm)` = col_double(),
##   ..   `Culmen Depth (mm)` = col_double(),
##   ..   `Flipper Length (mm)` = col_double(),
##   ..   `Body Mass (g)` = col_double(),
##   ..   Sex = col_character(),
##   ..   `Delta 15 N (o/oo)` = col_double(),
```

```
##   ..   `Delta 13 C (o/oo)` = col_double(),
##   ..   Comments = col_character()
##   .. )
```

Let's look at how many rows are in the penguins_raw data frame

```
nrow(penguins_raw)
```

```
## [1] 344
```

Let's look at the unique species in the `Species` column of the penguins_raw data frame using the `unique()` function.

```
unique(penguins_raw$Species)
```

```
## [1] "Adelie Penguin (Pygoscelis adeliae)"
## [2] "Gentoo penguin (Pygoscelis papua)"
## [3] "Chinstrap penguin (Pygoscelis antarctica)"
```

Now let's take a look at the column names in the penguins_raw dataset.

```
dput(names(penguins_raw))
```

```
## c("studyName", "Sample Number", "Species", "Region", "Island",
## "Stage", "Individual ID", "Clutch Completion", "Date Egg", "Culmen Length (mm)",
## "Culmen Depth (mm)", "Flipper Length (mm)", "Body Mass (g)",
## "Sex", "Delta 15 N (o/oo)", "Delta 13 C (o/oo)", "Comments")
```

In the previous code `dput(names(penguins_raw))`, R follows its usual order of operations by evaluating the innermost function first. So:

1. `names(penguins_raw)` retrieves the column names of the dataset as a character vector.
2. `dput()` then takes that result and prints it in a format that can be copied and reused as R code.

The dput() function (short for "dump put") converts an object into valid R code.

6.6.3 Manipulate data frame

Subsample the penguins_raw to create a new data set called p_measure.

First we will reassign the penguins_raw dataset to a new object called p_measure

```
p_measure <- penguins_raw
```

Let's select the 3rd, 5th, and 9th through 14th columns. Below are examples using both Base R and tidyverse approaches.

```
# base R method
p_measure <- p_measure[,c(3, 5, 9:14)]
```

```
# tidyverse method
# p_measure<-select(p_measure, Species, Island, "Date Egg", "Culmen Length (mm)",
#                   "Culmen Depth (mm)", "Flipper Length (mm)", "Body Mass (g)", Sex)
```

- We remove the following columns (studyName, Sample Number, Region, Stage, Clutch Completion, Delta 15 N (o/oo), Delta 13 C (o/oo) and Comments
- We keep the following columns (Species, Island, Date Egg Culmen Length (mm), Culmen Depth (mm), Flipper Length(mm), Body Mass (g), Sex)

Now let's rename the column names to eliminate spaces. Below are examples using the baseR and tidyverse methods.

```
# baseR method
names(p_measure)<-c("species", "island", "date", "bill_len", "bill_dep", "flipper_len"
                    "body_mass", "sex")
# tidyverse method
# p_measure<-rename(p_measure, species = Species, island = Island, date = "Date Egg",
#                   bill_len = "Culmen Length (mm)", bill_dep = "Culmen Depth (mm)",
#                   flipper_len = "Flipper Length (mm)", body_mass = "Body Mass (g)",
#                   sex = Sex)
```

Now we'll use the pipe operator (`%>%`) to add a column called `year` using mutate `(year=year(date))`. The`mutate()` function is used to add or modify columns in a data frame, and the `year()` function extracts the year component from a date. We then remove the date column using `select()`.

```
p_measure <- p_measure %>%
    mutate(year = year(date)) %>%
    dplyr::select(-3) %>%
    dplyr::filter(!is.na(body_mass) & !is.na(sex))
```

Let's look at our new data frame using `head()`

```
head(p_measure)
```

```
## # A tibble: 6 x 8
##   species             island bill_len bill_dep flipper_len body_mass sex    year
##   <chr>               <chr>     <dbl>    <dbl>       <dbl>     <dbl> <chr> <dbl>
## 1 Adelie Penguin (Py~ Torge~     39.1     18.7         181      3750 MALE   2007
## 2 Adelie Penguin (Py~ Torge~     39.5     17.4         186      3800 FEMA~  2007
## 3 Adelie Penguin (Py~ Torge~     40.3     18           195      3250 FEMA~  2007
## 4 Adelie Penguin (Py~ Torge~     36.7     19.3         193      3450 FEMA~  2007
## 5 Adelie Penguin (Py~ Torge~     39.3     20.6         190      3650 MALE   2007
## 6 Adelie Penguin (Py~ Torge~     38.9     17.8         181      3625 FEMA~  2007
```

6.6.4 Analysis

Now let's create our final analysis data frame object called `output_analysis`. We will use the data set called penguins (it's a cleaner version of the data set we just created from penguins_raw). The steps to perform include:

1. Remove rows with `NA` values in the `body_mass` and `sex` columns.
2. Group the data by species and sex.
3. Summarize each group by calculating:

- Number of samples (n)
- Mean body mass (mean_bm)
- Median body mass (median_bm)
- Standard deviation of body mass (SD_bm)

4. Add confidence intervals

- LCL: Lower confidence limit (mean - 2 × SD)
- UCL: Upper confidence limit (mean + 2 × SD)

```
output_analysis <- penguins %>%
  # Filter out rows where body_mass is NA (filter using !is.na)
  dplyr::filter(!is.na(body_mass_g)& !is.na(sex)) %>%
  # Group data by species and sex (group_by)
  group_by(species, sex) %>%
  # Summarize data (n, mean, median, sd)
  summarize(n = n(), # number of samples (n)
            mean_bm = mean(body_mass_g), # calculate mean body mass
            median_bm = median(body_mass_g), # calculate median body mass
            SD_bm = sd(body_mass_g)) %>% # caluclate the standard deviation of body mass
  # Add confidence intervals using mutate
  mutate(LCL = mean_bm - 2 * SD_bm, # calculate the lower confidence interval level
         UCL = mean_bm + 2 * SD_bm) # calculate the upper confidence interval level
```

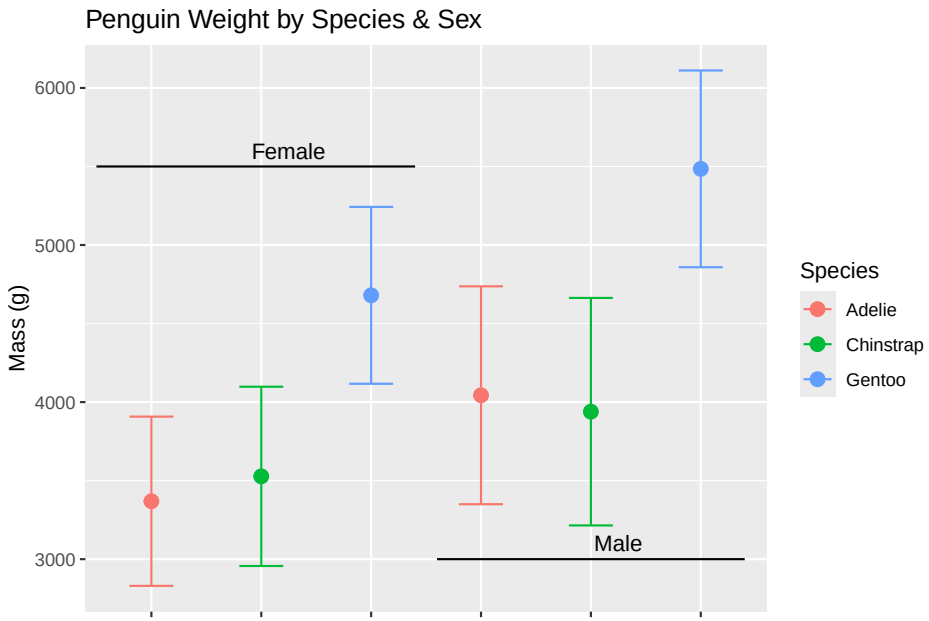
```
## `summarise()` has grouped output by 'species'. You can
## override using the `.groups` argument.
```

6.6.5 Plot results using ggplot2

Finally, we will generate a ggplot from our analysis using the data frame above.

```
ggplot(data = output_analysis, aes(x = paste0(sex, species), y = mean_bm, ymin = LCL, ymax = UCL,
  geom_point(size = 3) +  # Add points
  geom_errorbar(width = 0.4) + #  add error bars
  # angle text on x axis for readability
  theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1)) +
  # Hide x an y axis tick mark labels
  theme(axis.text.x = element_blank()) +
  # add annotations to graph
```

```
  annotate("text", x = 2.25 , y = 5600, label = "Female")+
  annotate("segment", x = 0.5, xend = 3.4, y = 5500,  colour = "black")+
  annotate("text", x = 5.25 , y = 3100, label = "Male")+
  annotate("segment", x = 3.6, xend = 6.4, y = 3000,  colour = "black")  +
  # label axes and graph
  labs(
    title = "Penguin Weight by Species & Sex",
    x = "",
    y = "Mass (g)",
    color = "Species"
  )
```



Chapter 7

Importing & Exporting Data

7.1 csv files

```
# Load csv file into R
lion_crosstab <- read.csv("path/to/your/file.csv")

# Save it as a csv file
write.csv(lion_crosstab, "path/to/save/your_new_file.csv", row.names = FALSE)
# row.names = FALSE makes sure a column of row numbers isn't created
```

Exercise

1. Load the lion crosstab table
2. Use head() to examine the first 3 rows of the dataset.

7.2 xlsx files

To read an xlsx file into R, we first need to install the `readxl` package. If the Excel file contains multiple sheets, you can specify the sheet you want to load by name or by index.

```
install.packages("readxl")
library(readxl)

# read data
```

```
data <- read_excel("path/to/your/file.xlsx", sheet = "Sheet1")

# save data
write.xlsx(data, "path/to/save/your_new_file.xlsx", sheetName = "Sheet1")
```

Exercise 1. Load the lion crosstab xlsx table 2. Use head() to examine the first 3 rows of the dataset.

7.2.1 rds files

An RDS file is a file format in R used to store a single R object, such as a data frame, vector, or model. It's a convenient way to save your R work so you don't lose data or have to recreate complex objects. You can use saveRDS() to save an object and readRDS() to load it back into your R session. RDS files preserve the exact data structure, including data types (e.g., factors, dates, lists, and complex objects), and metadata.

```
# Load RDS file into R
my_data <- readRDS("path/to/your/file.RDS")

# Save it as an RDS file
saveRDS(my_data, "path/to/save/your_new_file.RDS")

# View the loaded data
print(loaded_data)
```

Exercise

1. Load the lion RDS capture table
2. Use head() to examine the first 3 rows of the dataset.
3. Export capture table as an RDS file using a different name
4. Export capture table as a csv file using a different name

7.3 Connect to a database on Windows OS

To connect to a Microsoft Access database in R on a Windows computer, you can use the `RODBC package`. You will need the Access ODBC driver installed, and then use `odbcDriverConnect()` with the correct file path to the `.mdb` or `.accdb` file. Once connected, you can use `sqlFetch()` to load tables or `sqlQuery()` to run custom SQL queries.

```
library(RODBC)

# 1. Connect to the bighorn database
conAV <- RODBC::odbcDriverConnect("Driver={Microsoft Access Driver (*.mdb, *.accdb)};
                                  DBQ=C:/R/MigrationEnergeticsModel/20181109_bighorn.mdb")

# 2. Fetch desired database tables: grab database tables from bighorn database
collar.data <- RODBC::sqlFetch(conAV,"AllCollarLocations", colnames=F, rownames=F)
capt.data <- RODBC::sqlFetch(conAV, "BighornCapt", colnames=F, rownames=F)

# 2. SQL Query for Custom Data Extraction: Write query using SQL syntax. This code can be grabbed
sql.qry_GPS_Parm <- "SELECT GPSParameters.ProgramName_Descrip, BigHornCapt.AnimalID, BigHornCapt.
  LU_GPS_Programs.FixesPerWeek, BigHornCapt.Herd_Rels, BigHornCapt.Herd_Capt,BigHornCapt.Sex
                FROM LU_GPS_Programs INNER JOIN (GPSParameters INNER JOIN BigHornCapt ON
  GPSParameters.CollarSerialNo_Date = BigHornCapt.GPSCollarSerialNo_Date_FK) ON
  LU_GPS_Programs.ProgramName_Descrip = GPSParameters.ProgramName_Descrip
                WHERE ((LU_GPS_Programs.FixesPerWeek)>27 And (LU_GPS_Programs.FixesPerWeek)<250))

# Execute query
high.fix <- RODBC::sqlQuery(conAV, sql.qry_GPS_Parm)

# Remove query after run
rm(sql.qry_GPS_Parm)

# Close database connection
close(conAV)
```

7.4 Load database files on a Macbook

You can't use Access databases on a Mac in the same way you can on a Windows machine. Microsoft Access `.mdb` and `.accdb` files aren't natively supported on macOS because the required ODBC drivers are Windows-only. On my MacBook, I typically load individual tables using R and then join them within R, rather than querying directly through Access. (Note: we won't cover joins in this class.) Another option is to convert the Access database into a SQL-compatible format (such as `SQLite` or `PostgreSQL`) and then write SQL queries against that database.

```
# export Capture Table
system("mdb-export /Users/a02399564/Documents/Research/PhD-SNBS/data/snbs/databases/bighorn.mdb E
```

EXERCISE

1. Import the bighorn mortality table and store it in a data frame named

`bighorn_mort`.

2. Import the bighorn capture table and store it in a data frame named `bighorn_capt`.
3. Combine the two tables using a tidyverse join command (sytax below)

- `capt_mort <- left_join(x = bighorn_capt, y = bighorn_mort, by = join_by(AnimalID == Animal_ID))`
  - All bighorn are in `bighorn_capt` data frame, only some bighorn are in the `bighorn_mort` data frame (i.e. this will result in bighorn have NA's if they are still alive)

Chapter 8

Data Organization

Let's start by downloading some data and exploring it in a data frame. In this example, we'll load a CSV file.

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

8.1 Exploring Data Frames

Let's take a look at the data. Exploring your dataset is one of the most important first steps when working in R. It helps you understand the structure, types of variables, and potential issues before doing any analysis. A good place to start is by examining the structure of the data to see what you're working with.

```
# str() shows us the structure of the data, including the data mode,
# the dimensions of the dataframe including the data mode, 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" "Hor
##  $ GPS_CollarSerialNo_Date_FK: chr  "" "" "" "" ...
##  $ CollarSN                    : chr  "" "" "" "" ...
##  $ VHF_freq                    : chr  "" "" "" "" ...
##  $ Location                    : chr  "On Shepherd Pass trail, 1 mi up from TH, ~6800
##  $ InvestReason                : chr  "Opportunistic" "Opportunistic" "Lion Cluster" "
##  $ 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
```

```
names(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 of our data 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.

We can use the functions `head()` and `tail()` to also look at the first or last 6 rows of the data frame. We can adjust the number of rows that are printed using the `n` argument.

```
head(mort, n = 3) # n = refers to how many rows to print
```

```
##   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
```

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" ...
```

We can index a data frame column using the `$` operator and `[`. Below, the `$` operator accesses the `Animal_ID` columns and `[1]` grabs the first value in that column.

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

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

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

```
mort[["AnimalYear"]][500] # extract the 500th row of the AnimalYear column
```

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

We can also use square brackets to extract specific elements from a data frame. Remember that data frame indexing follows the format [rows, columns], so the first number refers to the row, and the second to the column.

```
# [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], n = 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
```

8.2 Vectorized Operations

R is a natively vectorized language, which means it can automatically perform operations on entire vectors (columns of data) without the need for explicit loops. This makes data manipulation in R efficient and concise.

For example, let's say all the ages in our bighorn mortality data set are off by one year. To correct this, we can simply add 1 to the entire Age column. This works because R applies the operation to each element of the vector. But first, we need to make sure the `Age` column is numeric. If it's stored as character or factor, we need to coerce it to numeric before performing the math. Because R vectorizes math operations, the `+ 1` is automatically applied to every value in the `Age` column, and the result is stored in the new `AgeNew` column — no loop required.

```
class(mort$Age)
```

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

```
mort$AgeNew <- as.numeric(mort$Age)
```

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

```
mort$AgeNew <- mort$AgeNew + 1
```

We see a warning about NAs being introduced when coercing values to a numeric data type. This happens because some entries in the Age column aren't purely numeric—for example, values like "4+" or ">7" contain extra characters (+ and >). Since R can't interpret these as numbers, it replaces them with `NA`.

In practice, we could clean this column so it contains only numeric values. One way to do that is by using the `gsub()` function along with regular expression to remove non-numeric characters. I'll show you the code below for reference, but we won't be covering regular expressions in detail during this workshop.

```
mort$AgeNew <- gsub("[^0-9.]", "", mort$Age)  # Remove all non-numeric characters
mort$AgeNew <- as.numeric(mort$AgeNew) # Coerce from character to numeric data mode
head(mort$AgeNew, n = 10)
```

```
##  [1]  6 11  0  5  4  9  0  3 10  2
```

```
mort$AgeNew <- mort$AgeNew + 1
head(mort$AgeNew, n = 10)
```

```
##  [1]  7 12  1  6  5 10  1  4 11  3
```

8.3 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 created a simple vector of TRUE/FALSE values. To add it to our data frame, we can use the `cbind()` function, which binds columns together. In this case, it combines our existing data frame with the new logical vector as an additional column.

```
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 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 AgeNew below_five below_five
## 1                   N                    7      FALSE      FALSE
## 2                   N                   12       TRUE       TRUE
## 3                   Y  267_20241202      1       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
## 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 AgeNew below_five
## 1                   N                    7      FALSE
## 2                   N                   12       TRUE
## 3                   Y  267_20241202       1       TRUE
```

If we try to add the below_five vector to our data frame R will return an error. This is because the below_average vector has a different number of elements than rows in the mort data frame. The number of entries in the below_five vector must match the number of rows in the mort data fram to align correctly.

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

Now let's look at how we might add rows to a data frame.

```
# 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 type
mort_new <- rbind(mort_short, new_row)
tail(mort_new) # lets you look at the last 6 lines of data
```

EXERCISE

1. Load the bighorn mort table from the bighorn database and assign it the variable name of "mort"
2. Print the names of the mort dataframe using `names()` and `dput(names(mort))`. Why might using dput(names(mort)) be helpful?
3. Select these columns: DeadDate, Animal_ID, COM, Sex, Age, Herd, Recorder
4. Subset to just females using three different methods of subsetting: `subset()`, which indexing, and `filter()`. Ensure that each method provides the same results.
5. Subset to herds that are not Cathedral or Big Arroyo. You will need to use `!` and the `%in%`. Any subsetting method is ok to use.
6. Subset to herds in the Southern Recovery Unit: Mt. Baxter, Mt. Langley, Taboose Creek, Sawmill Canyon, Bubbs Creek, Mt. Williamson. You will

need to use `%in%`. Any subsetting method is ok to use

7. Subset to only lion caused mortalities. Use the `unique()` function on the column `COM` to print the category names in the `COM` column. This will provide information on the correct syntax to use for subsetting. For example, is it "Lion", "lion", or "mountain lion". Syntax matters in R.
8. How many rows and columns are in the mort data frame.
9. Use `View()` to view you data frame

Chapter 9

Control Structures

Control structures are important tools for writing clear and efficient code by using conditional statements and repeated actions. Common control structures in R include `if`, `else`, and `ifelse()` for making decisions, and loops `for`, `while`, and `repeat` for running code multiple times. For example, if statements run code only when a condition is true, while for loops repeat actions for each item.

9.1 Loops

There are three main types of loops: `for` (most common), `while`, and `repeat`. For loops are not always the most efficient way to perform repetitive tasks. In many cases, vectorized operations (e.g. apply functions) are faster and more efficient. Vectorized operations apply to entire vectors (or arrays) at once, whereas for loops iterate through individual elements sequentially. We will not be covering vectorization operations in this course, but it is good to be aware that they exist.

```
for(i in 1:n) { # n = number of times you run the loop
  # execute a task
} # end loop
```

It's good practice to indent the code inside your loop and add a comment at the end to mark that the loop is finished. This becomes especially helpful when you have nested loops, as it makes your code easier to read and understand.

```
for(i in 1:10) {
  print(i + 3) # indented code chunk
} # comment that this is the end of the ith loop
```

```
## [1] 4
## [1] 5
```

```
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
```

When using a for loop, you often need to store the results from each iteration. Start by creating an empty object before the loop begins. Inside the loop, assign each result to a specific index of that object. To aid in debugging, use the cat() function to print the current iteration number. This allows you to track the loop's progress and easily identify issues by running it step by step.

```
# create empty vector of desired dimensions
my_vector <- rep(NA, length = 10)

for(i in 1:10) {
  cat(i) # print current iteration of loop
  my_vector[i] <- i + 3 * (i-1) # save output in the ith position of my_vector
} # end i loop
```

```
## 12345678910
```

```
my_vector
```

```
##  [1]  1  5  9 13 17 21 25 29 33 37
```

9.2 Troubleshooting For-loops

Troubleshooting for-loops can be tricky. Below are a few hints for troubleshooting code in a for-loop.

1. Check loop syntax
2. Print iteration progress
3. Check object dimensions
4. Check variable types
5. Avoid indexing errors
6. Check loop boundaries
7. Validate conditional statements
8. Use try-catch for errors
9. Ensure proper object initialization
10. Check for missing values

Example: Converting Sheep Weight from Kilograms to Pounds

Let's start by creating a sample capture data frame that includes sheep IDs and their weight in kilograms.

```
capt <- data.frame(animal_id = c("S431", "S488", "S490", "S511"),
                   weight_kg = c(43, 55, 48, "50kg"))
```

Next, we'll run a for loop to convert the weight from kilograms to pounds. Before doing so, we need to create a new column in the data frame to store the converted weight in pounds.

```
capt$weight_lb <- NA # R automatically assigns each of the 4 rows as NA
```

Now, here's a for loop that attempts to transform the weight from kilograms to pounds. This loop has two issues that we need to address:

```
for (i in 1:nrow(capt)) {
  pounds <- capt$weight_kg[i] * 2.20462
  capt$weight_lb <- pounds
}
```

```
## Error in capt$weight_kg[i] * 2.20462: non-numeric argument to binary operator
```

Issue 1: Error in the First Iteration

It looks like the loop threw an error. To find where the error occurred, we can print the current iteration value (i), which will help identify the problem:

```
i
```

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

The error occurs in the first iteration. To debug further, we can execute the loop's code step by step. Let's start with the first line:

```
pounds <- capt$weight_kg[i] * 2.20462
```

```
## Error in capt$weight_kg[i] * 2.20462: non-numeric argument to binary operator
```

Issue 2: Non-Numeric Data in weight_kg

The error occurs because the weight_kg column contains a non-numeric entry: "50kg". Let's check the data type of capt$weight_kg:

```
class(capt$weight_kg)
```

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

Because the `weight_kg` column is of character type, we cannot perform an arithmetic operation, and the code throws an error. To fix this, we need to convert the weight_kg column to numeric.

```
capt$weight_kg_new <- as.numeric(capt$weight_kg)
```

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

Looks like an NA was thrown. Let's check the original weight_kg column to understand why:

```
capt$weight_kg
```

```
## [1] "43"   "55"   "48"   "50kg"
```

The fifth row contains "50kg", which causes the entire column to be treated as character data. To fix this, we can remove any alphabetic characters from the weight_kg column using regular expressions (regex). Although we won't cover regex in detail here, it's a useful tool for cleaning data.

```
capt$weight_kg <- gsub("[A-Za-z]", "", capt$weight_kg)
capt$weight_kg <- as.numeric(capt$weight_kg)
```

Issue 3: Incorrect Weight Conversion in the Loop

Now that the first issue is resolved, let's try running the for loop again:

```
for (i in 1:nrow(capt)) {
  pounds <- capt$weight_kg[i] * 2.20462
  capt$weight_lb <- pounds
}
```

```
capt$weight_lb
```

```
## [1] 110.231 110.231 110.231 110.231
```

This time, the loop runs, but it appears that the same weight value was applied to all rows. Let's debug this by running through the loop step by step.

First, we'll reset the weight_lb column to NA and set i = 1 to check the output for the first row:

```
capt$weight_lb <- NA # R automatically assigns each of the 4 rows as NA

for (i in 1:nrow(capt)) {
  pounds <- capt$weight_kg[i] * 2.20462
  capt$weight_lb <- pounds
}
```

Issue 4: Forgetting to Index the weight_lb Column Correctly

The problem is that we forgot to properly index the `capt$weight_lb` column with `i` to ensure the calculated pounds value is assigned to the correct row. Let's fix the loop by updating the assignment to `capt$weight_lb[i]`:

```
capt$weight_lb <- NA  # Reset the weight_lb column to NA

# Correct the loop
for (i in 1:nrow(capt)) {
  pounds <- capt$weight_kg[i] * 2.20462
  capt$weight_lb[i] <- pounds  # Use indexing to assign pounds to the correct row
}
```

Additional Debugging Tip: Experiment with Iteration Values

A helpful debugging trick is to manually set the value of `i` to different numbers to observe how the behavior of the loop changes with different data. For example:

```
i <- 2  # Test with a specific iteration
capt$weight_kg[i] * 2.20462  # Check the result for the second iteration
```

```
## [1] 121.2541
```

This allows you to understand how the loop behaves with specific input values.

EXERCISE

1. Write a for loop that runs for 10 iterations, adding 3 to the loop index during each iteration

---

9.3 if, else, and ifelse statements

If, else, and ifelse statements allow you to control the flow of your code by making decisions based on specific conditions. This can be particularly helpful in for loops, where you might want to execute different actions depending on the iteration.

For instance, conditional statements allow you to alternate between different operations or calculations during each iteration of a loop. This provides flexibility, enabling you to adjust the behavior of your code based on dynamic factors like the loop index or other variables. Below is an example where a for loop and ifelse statement are used to store a logical value in a vector if a randomly generated number is less than or equal to 50.

```
# create vector of length 10 with random numbers
my_vect <- sample(x = 1:100, size = 15)
output_vect <- NULL
for (i in 1:length(my_vect)) {
  if(my_vect[i] <= 50) {
    output_vect[i] <- TRUE
  }
  else{
    output_vect[i] <- FALSE
  }
}

my_vect
```

```
##  [1] 88 83 92  1 62 72 99 90 80 40 87 64 58 93  6
```

```
output_vect
```

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

You can also use the ifelse() statement for a conditional statement.

```
species <- c("snbs", "lion", "deer", "deer")

# Use ifelse to label values as "small" or "large"
result <- ifelse(species %in% c("snbs", "deer"), "prey", "predator")

result
```

```
## [1] "prey"     "predator" "prey"     "prey"
```

Why use ifelse() instead of if-else?

While both ifelse() and traditional if-else statements perform conditional logic, ifelse() is often preferred when you need to apply conditions element-wise (e.g., to vectors or columns in a data frame) and so is better for vectorized operations.

For example, ifelse() is faster and simpler when you want to apply a condition across an entire vector or column at once, without having to explicitly loop over each element.

In contrast, a traditional if-else statement is typically used when you need more complex or multi-step conditions and is better suited for situations that involve single-value conditional logic within control structures like loops.

9.4 next and break

In a for loop, the `next` and `break` commands provide control over the loop's behavior:

- next: Skips the current iteration and moves to the next one. It's useful when you want to bypass certain iterations based on a condition without exiting the entire loop.

```
for (i in 1:5) {
  if (i == 3) next  # Skip iteration when i is 3
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 4
## [1] 5
```

- break: Immediately exits the loop, stopping further iterations. This is useful when you want to terminate the loop early based on a specific condition.

```
for (i in 1:5) {
  if (i == 3) break  # Exit the loop when i reaches 3
  print(i)
}
```

```
## [1] 1
## [1] 2
```

Both `next` and `break` give you fine-grained control over loop behavior, allowing you to skip unnecessary steps or stop the loop early when certain conditions are met.

9.5 Keeping track of loop

You can have R print something out at each loop iteration. This is particularly useful with nested loops and determining where you have data problems, i.e. unexpected results in the middle of a loop because of a data anomaly.

For this you can insert a `cat()` function (which prints to the console, so you can see what is happening in real time)

```
# Example
for (x in 1:10) {
  cat("\n Iteration n =", x, "is being executed now") # \n prints on new line
  x_sqrt <- sqrt(x)
  cat("\n The square root of", x, "=", x_sqrt, "\n")
}
```

```
##
##  Iteration n = 1 is being executed now
##  The square root of 1 = 1
##
##  Iteration n = 2 is being executed now
##  The square root of 2 = 1.414214
##
##  Iteration n = 3 is being executed now
##  The square root of 3 = 1.732051
##
##  Iteration n = 4 is being executed now
##  The square root of 4 = 2
##
##  Iteration n = 5 is being executed now
##  The square root of 5 = 2.236068
##
```

```
##  Iteration n = 6 is being executed now
##  The square root of 6 = 2.44949
##
##  Iteration n = 7 is being executed now
##  The square root of 7 = 2.645751
##
##  Iteration n = 8 is being executed now
##  The square root of 8 = 2.828427
##
##  Iteration n = 9 is being executed now
##  The square root of 9 = 3
##
##  Iteration n = 10 is being executed now
##  The square root of 10 = 3.162278
```

`EXERCISE`

1. **Create a data frame** Create a data frame that includes the following columns:

- deer_id: a unique identifier for each deer (as a character)
- month: the month the data was recorded (as a number from 1 to 12)
- year: the calendar year of the observation (as a number)

2. **Add an animal year column using a for loop and ifelse** Using a for loop, add a new column called animal_year to the data frame.

- If the month is between May (5) and December (12), add 1 to the year
- If the month is between January (1) and April (4), keep the year unchanged
- Use an `ifelse()` statement inside the loop to apply this logic.
- BONUS: Use the `cat()` command to print what iteration is being executed in the for loop, the calendar year, and the new animal year is.

Chapter 10

Statistical Summaries

This is a work in progress and will be updated.

The following exercise can be done on your own or in groups. You'll need to use what we've covered so far and make use of the help function in R. Don't hesitate to ask questions—if anyone gets stuck, we'll work through it together as a class.

1. Load the bighorn capture table from the bighorn database
2. Select the following collumns: Animal_ID, Sex, Herd_Rels, BCS score, and weight
3. Rename the Herd_Rels column to Herd: use `rename()` in tidyverse or names(df) <- c(in baseR)
4. Are there any NA values? Use sum(is.na()). Remove NA values with `na.omit()` or `is.na()`
5. Use the baseR `table()` function to tabulate the number of captures by herd across time
6. Group by herd using `group_by()`
7. Use the summarize function to calculate the sample size, mean, and median of BCS by herd
8. Create a scatterplot in baseR or ggplot2 of bighorn by BCS and weight
9. Create a boxplot of weight in each herd
10. Compare if bighorn weights differ between Gibbs and Baxter using a T-test.

Chapter 11

Debugging Code

Artwork by Allison Horst

Even when you know R well, you will inevitably find yourself in situations where you don't understand how a package works, encounter an unexpected error, aren't sure which function to use, or need ideas for how to write the correct code. This is a normal part of coding and problem-solving. Experienced R users frequently search documentation, check forums like Stack Overflow, read vignettes, and experiment with small pieces of code to troubleshoot issues or explore new approaches. The key is knowing how to identify the problematic code, ask good questions, and know where to look for help.

11.1 Locating Problematic Code

If your code throws an error, begin by finding the line where it occurs. Run your code one line at a time until the error shows up. If the issue isn't clear on that line, break it into smaller parts and test each section individually. This step-by-step process helps you pinpoint the exact source of the problem.

EXERCISE 1

1. Trouble shoot the following code. `x <- c(1, 2, 3) y <- c(4, 5, "six") result <- x + y`

11.2 Syntax errors

Check Your Parenthesis!

A common error in R occurs when parentheses are not properly balanced. Every opening parenthesis `(` must have a corresponding closing parenthesis `)`. If they don't match, R will throw an error or hang while waiting for the rest of the expression to be executed. For example, mean(c(1, 3, 5, 6)) works correctly because the parentheses are balanced, while mean(c(1, 3, 5, 6) will result in an error due to the missing closing parenthesis. R will highlight the matching parenthesis when you place your cursor just before an opening `(` or just after a closing `)`, helping you check that your parentheses are properly balanced.

Check your commas!

In R, syntax errors related to commas often occur when they are either missing or misplaced. Commas are used to separate function arguments or elements in vectors, lists, and other data structures. A missing comma can cause R to misinterpret the code, leading to errors.

`EXERCISE 2`

1. Trouble shoot the following code. Start by running each line of code to understand where the error occurs. `x <- c(1, 2, 3 y <- c(4, 5 "six")`

11.3 Missing and Undefined Data

There are three common types of missing or undefined data in R: `Null`, `NA`, and `NaN`. Knowing the differences in these data types can help with interpreting error messages.

Type	Description	Use Case	Example
NA	Represents a missing or unavailable value in data.	Used in data frames or vectors to indicate missing data.	`x <- c(1, 2, NA, 4)`
NaN	Stands for "Not a Number". Represents undefined numerical results (e.g., division by zero).	Typically appears when performing invalid mathematical operations.	`x <- 0 / 0` (Results in `NaN`)
NULL	Represents the absence of a value or object.	Used to indicate an empty object or no value assigned.	`x <- NULL`

11.4 Coercian Problems

In R, coercion occurs when elements of different types are combined, and R automatically converts them to a common type to maintain consistency. This can lead to unintended results if not handled carefully. Always check your data types with `str()`, `typeof()`, or `classs()` to avoid coercion issues.

A common coercion error in R occurs when you convert a factor to numeric without first converting it to a character. Doing as.numeric() directly on a factor returns the internal integer codes of the factor levels, not the actual numeric values you might expect.

```
f <- factor(c("10", "20", "30"))

# incorrect coercian
as.numeric(f)
```

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

```
# correct coercian
as.numeric(as.character(f))
```

```
## [1] 10 20 30
```

11.5 Labeling function arguments

Labeling function arguments in R improves clarity and reduces errors. While you can pass arguments by position (e.g., `seq(1, 10, 2)`), naming them explicitly (e.g., `seq(from = 1, to = 10, by = 2)`) makes your code easier to read and avoids mistakes—especially when argument order is unclear or changes.

11.6 Conflicting Functions

Duplicate function names in different libraries can create confusion and lead to errors in R because when multiple packages define functions with the same name, R will prioritize one function over another. This can cause unintended behavior and errors if a different function from a different package to be executed. For example, if two packages define a function called `filter()`, and one is from `dplyr` and the other from `stats`, R might use one by default, potentially breaking the code or leading to incorrect results if the wrong version is applied. Managing these conflicts is essential for ensuring that the correct function is used in the desired context.

Option 1: Using `::` Syntax

To specify which package's function to use, you can use the `::` syntax. For example, to use the `select()` function from `dplyr`, write `dplyr::select()`. This ensures you are explicitly calling the correct function from the right package.

Option 2: Using the Conflict Package

The conflict package in R is designed to help manage situations where two or more loaded packages have functions with the same name. When you load multiple packages that contain functions with identical names, R will prioritize one function over another, which can lead to confusion or errors in your analysis. For example, both `amt` and `dplyr` have a `select()`. The conflict package provides tools to manage these conflicts explicitly and prevent unexpected behavior.

```
library(conflicted)
conflict_prefer("select", "dplyr")  # This will prefer the select() function
```

```
## [conflicted] Will prefer dplyr::select over any other
## package.
```

```
# from the dplyr package.
```

11.7 Use the Help ? Function!

In R, you can use the ? function to access R documentation for functions, packages, or datasets. For example, `?select` will bring up the documentation for the `select()` function. If you want information about a specific package, you can use `?package_name`, such as `?dplyr`, to view its documentation. This is a quick way to learn about function arguments, usage, and examples directly within R.

11.8 Google

A quick Google search often leads to answers, and most R packages include vignettes or documentation to explain how functions work. For example, if you're unsure how to use the seq function in R to generate a sequence with a step size of 0.5, you could search for "how do I use the seq function in R to generate a sequence of numbers by 0.5?" It's important to specify that you're working with R and clearly describe your problem to get better results.

Another invaluable resource is Stack Overflow, a community-driven Q&A website where developers and data scientists share knowledge. On Stack Overflow, you can search for similar issues, or if your question hasn't been asked yet, you can post your own.

11.9 AI

AI (e.g. ChatGPT) can be a valuable tool for debuggin code, but it's important to remember that it's not always correct. Relying too heavily on AI can also hinder your ability to develop and strengthen your coding skills.

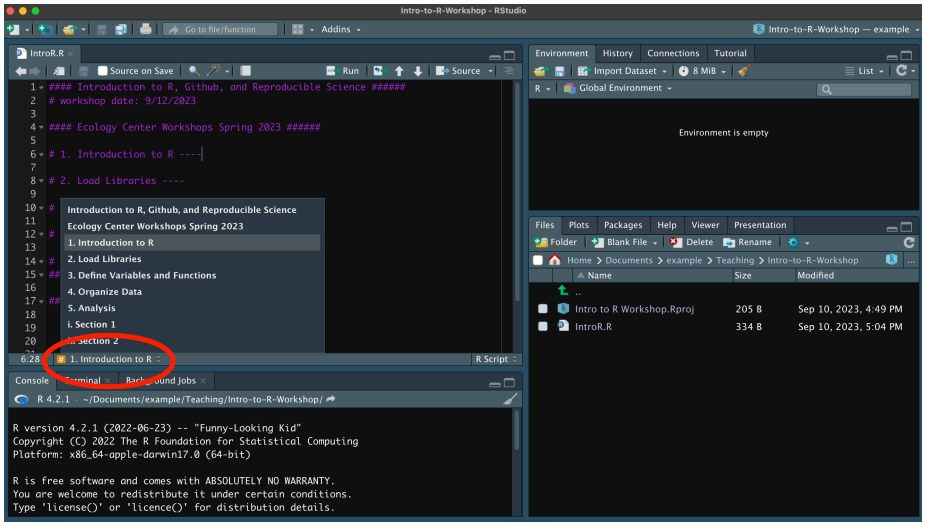
Chapter 12

Making Readable R Code

12.1 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 ----
```



12.2 Pseudocoding

Pseudocoding is basically a way to map out your code by writing down the logic in plain English. It helps you figure out what you're trying to do, organize your ideas, and spot potential issues before you actually start coding. It's like sketching a rough draft before creating the final version. Pseudo coding can also be used to identify problematic code.

To write pseudocode:

1. Define the problem: Understand what you want to solve.
2. Outline the steps: Break down the problem into logical, sequential steps.
3. Use simple language: Write instructions in a clear and understandable format, avoiding technical jargon.
4. Focus on logic: Focus on the flow of data, control structures (loops, conditionals), and key operations.

12.3 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). to track changes and collaborate
11) Indent your code chunks
12) Don't write past the page margin

Chapter 13

Class Wrap Up

Artwork by Allison Horst

We hope that you have picked up some useful skills in this IntroR course and are motivated to keep learning R. If you run into questions down the road, feel free to reach out to either one of us.

We're also planning a second workshop emphasizing spatial data manipulation and analysis, which will include:

1. The apply family
2. Joining data
3. Wrangling Spatial Data using the simple feature package 4 Using dates and times
4. Mapping spatial data (e.g. leaflet, baseR plotting, etc.)
5. Implementing and exploring key differences between Kaplan Meier and Cox Proportional Hazard survival analyses.