Introduction to R for Natural Resource Scientists

Ben Staton

with contributions from Henry Hershey

Contents

Overview

This is intended to be a first course in R programming. It is by no means comprehensive, but instead attempts to introduce the main topics needed to get a beginner up and running with applying R to their own work. There will be no prior knowledge assumed on the part of the student regarding R or programming. In the later chapters, e.g., Chapters 3 and 4, an understanding of statistics at the introductory undergraduate level would be helpful.

Chapter 1

Introduction to the R Environment

Chapter Overview

In this first chapter, you will get familiar with the basics of using R. You will learn:

- the use of R as a basic calculator
- some basic object types
- some basic data classes
- some basic data structures
- how to read in data
- how to write out data
- how to write your own functions

Before You Begin: Install R and RStudio

First off, you will need to get R and RStudio[1] onto your computer. Please see **Appendix A** for details on installing these programs on your operating system.

1.1 The R Studio Interface

Once you open up R Studio for the first time, you will see three panes: the left hand side is the **console** where results from executed commands are printed, and the two panes on the right are for additional information to help you code more efficiently - don't worry too much about what these are at the moment. For now, focus your attention on the console.

[1] While it is possible to run R on it's own, it rather clunky and you are strongly advised to use RStudio given its compactness, neat features, code tools (like syntax and parentheses highlighting). This workshop will assume you are using RStudio

1.1.1 Write Some Simple Code

To start off, you will just use R as a calculator. Type these commands (not the lines with **##**, those are output[2]) one at a time and hit **CTRL + ENTER** to run it. The spaces don't matter at all, they are used here for clairity and for styling.[3]

```
3 + 3
```

```
## [1] 6
```

```
12/4
```

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

Notice that when you run each line, it prints the command and the output to the console.

R is an **object oriented language**, which means that you fill objects with data do things with them. Make an object called `x` that stores the result of the calculation `3 + 3` (type this and run using **CTRL + ENTER**):

```
x = 3 + 3
```

Notice that running this line did not return a value as before. This is because in that line you are **assigning** a value to the object `x`. You can view the contents of `x` by typing its name alone and running just that:

```
x
```

```
## [1] 6
```

When used this way, the `=` sign denotes assignment of the value on the right-hand side to an object with the name on the left-hand side. The `<-` serves this same purpose so in this context the two are interchangable:

```
y <- 2 + 5
```

You can highlight smaller sections of a line to run as well. For example after creating `y` above, press the **up arrow** to see the line you just ran, highlight just the `y`, and press **CTRL + ENTER**. From this point forward, the verb "run" means execute some code using **CTRL + ENTER**.

You can use your objects together to make a new object:

```
z = y - x
```

1.2 Saving Your Code: Scripts

If you closed R at this moment, your work would be lost. Running code in the console like you have just done **does not save a record of your work**. To save R code, you must use what is called a **script**, which is a plain-text file with the extension `.R`. To create a new script file, go to *File > New File > R Script*, or use the keyboard shortcut **CTRL + SHIFT + N**. A new pane will open called the **source** pane - this is where you will edit your code and save your progress. R Scripts are a key feature of reproducible research with R, given that if they are well-written they can present a complete roadmap of your statistical analysis and workflow.

[2]The formatting used here includes **##** on output to denote code and output separately. You won't see the **##** show up in your console.

[3]To learn more about standard R code styling, check out Hadley Wickham's great chapter about it.

1.3 The Working Directory

You will want to save your hard work. A key part of doing saving your work is thinking about **where** you save it. In R, a key concept is the **working directory**. This is the location (i.e., folder) on your computer where your current R session will "talk to" by default. The working directory is where R will read files from and write files to by default, and is where all of your data should be stored for your analysis in R. Because you'll likely be visiting it often, it should probably be somewhere that is easy to remember and not too deeply buried in your computer's file system.

Save your script somewhere like `C:/Users/YOU/Documents/R-Workshop/Chapter1` on your computer. To set the working directory to this location, you have three options:

1. **Go to Session > Set Working Directory > Source File Location**. This will set the working directory to the location of the file that is currently open in your source pane.
2. **Go to Session > Set Working Directory > Choose Directory**. This will open an interactive file selection window to allow you to navigate to the desired directory.
3. **Use code**. In the console, you can type `setwd("C:/Users/YOU/Documents/R-Workshop/Chapter1")`. If at any point you want to know where your current working directory is set to, you can either look at the top of the console pane, which shows the full path or by running `getwd()` in the console.

The main benefits of using a working directory are:

- Files are read from and written to a consistent and predictable place everytime
- Everything for your analysis is organized into one place
- You don't have to continously type file paths to your work. If `file.txt` is a file in your current working directory, you can reference it your R session using `"file.txt"` rather than with `"C:/Users/YOU/Documents/R-Workshop/Chapter1/file.txt"` each time.

1.4 R Object Types

R has a variety of object types that you will need to become familiar with.

1.4.1 Functions

Much of your work in R will involve functions. A function is called using the syntax:

```
fun(arg1 = value1, arg2 = value2)
```

Here, `fun` is the **function name** and `arg1` and `arg2` are called **arguments**. Functions take input in the form of the arguments, do some task with them, then return some output. The parentheses are a sure sign that `fun` is a function.

We have passed the function two arguments by name: all functions have arguments, all arguments have names, and there is always a default order to the arguments. If you memorize the argument order of functions you use frequently, you don't have to specify the argument name:

```
fun(value1, value2)
```

would give the same result as the command above in which the argument names were specified.

Here's a real example:

```
print(x = z)
```

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

The function is `print`, the argument is `x`, and the value we have supplied the argument is the object `z`. The task that `print` does is to print the value of `z` to the console.

R has lots of built-in information to help you learn how to use a function. Take a look at the help file for the mean function. Run `?mean` in the console: a window on the right-hand side of the R Studio interface should open. The help file tells you what goes into a function and what comes out. For more complex functions it also tells you what all of the options (i.e., arguments) can do. Help files can be a bit intimidating to interpret at first, but they are all organized the same and once you learn their layout you will know where to go to find the information you're looking for.

1.4.2 Vectors

Vectors are one of the most common data structures. A vector is a set of numbers going in only one dimension. Each position in a vector is termed an **element**, and the number of elements is termed the **length** of the vector. Here are some ways to make some vectors with different elements, all of length five:

```
# this is a comment. R will ignore all text on a line after a #
# the ; means run everything after it on a new line

# count up by 1
month = 2:6; month
```

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

```
# count up by 2
day = seq(from = 1, to = 9, by = 2); day
```

```
## [1] 1 3 5 7 9
```

```
# repeat the same number (repeat 2018 5 times)
year = rep(2018, 5); year
```

```
## [1] 2018 2018 2018 2018 2018
```

The `[1]` that shows up is a element position, more on this later (see Section 1.7). If you wish to know how many elements are in a vector, use `length`:

```
length(year)
```

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

You can also create a vector "by-hand" using the `c` function[4]:

```
# a numeric vector
number = c(4, 7, 8, 10, 15); number
```

```
## [1]  4  7  8 10 15
```

```
# a character vector
pond = c("F11", "S28", "S30", "S8", 'S11'); pond
```

```
## [1] "F11" "S28" "S30" "S8"  "S11"
```

Note the difference between the numeric and character vectors. The the terms "numeric" and "character"" represent **data classes**, which specify the type of data the vector is holding:

- A **numeric vector** stores numbers. You can do math with numeric vectors

[4]The `c` stands for **concatenate**, which basically means combine many smaller objects into one larger object

- A **character vector** stores what are essentially letters. You can't do math with letters. A character vector is easy to spot because the elements will be wrapped with quotes[5].

A vector can only hold one data class at a time:

```
v = c(1,2,3,"a"); v
```

```
## [1] "1" "2" "3" "a"
```

Notice how all the elements now have quotes around them. The numbers have been **coerced** to characters[6]. If we attempted to calculate the sum of our vector:

```
sum(v)
```

```
## Error in sum(v): invalid 'type' (character) of argument
```

we would find that it is impossible in its current form.

1.4.3 Matrices

Matrices act just like vectors, but they are in two dimensions, i.e., they have both rows and columns. One easy way to make a matrix is by combining vectors you have already made:

```
# combine vectors by column (each vector will become a column)
m1 = cbind(month, day, year, number); m1
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     4   5 2018      8
## [4,]     5   7 2018     10
## [5,]     6   9 2018     15
```

```
# combine vectors by row (each vector will become a row)
m2 = rbind(month, day, year, number); m2
```

```
##        [,1] [,2] [,3] [,4] [,5]
## month     2    3    4    5    6
## day       1    3    5    7    9
## year   2018 2018 2018 2018 2018
## number    4    7    8   10   15
```

Just like vectors, matrices can hold only one data class (note the coersion of numbers to characters):

```
cbind(m1, pond)
```

```
##      month day year   number pond
## [1,] "2"   "1" "2018" "4"    "F11"
## [2,] "3"   "3" "2018" "7"    "S28"
## [3,] "4"   "5" "2018" "8"    "S30"
## [4,] "5"   "7" "2018" "10"   "S8"
## [5,] "6"   "9" "2018" "15"   "S11"
```

[5] " " or ' ' both work as long as you use the same on the front and end of the element

[6] The coersion works this way because numbers can be expressed as characters, but a letter cannot be unambiguously be expressed as a number.

1.4.4 Data Frames

Many data sets you will work with require storing different data classes in different columns, which would rule out the use of a matrix. This is where **data frames** come in:

```
df1 = data.frame(month, day, year, number, pond); df1
```

```
##   month day year number pond
## 1     2   1 2018      4  F11
## 2     3   3 2018      7  S28
## 3     4   5 2018      8  S30
## 4     5   7 2018     10   S8
## 5     6   9 2018     15  S11
```

Notice the lack of quotation marks which indicates that all variables (i.e., columns) are stored as their original data class.

It is important to know what kind of object type you are using, since R treats them differently. For example, some functions can only use a certain object type. The same holds true for data classes (numeric vs. character).You can quickly determine what kind of object you are dealing with by using the `class` function. Simply run `class(object.name)`:

```
class(day); class(pond); class(m1); class(df1)
```

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

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

```
## [1] "matrix"
```

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

1.5 Factors

At this point, it is worthwhile to introduce an additional data class: factors. Notice the class of the `pond` variable in `df1`:

```
class(df1$pond)
```

```
## [1] "factor"
```

The character vector `pond` was coerced to a factor when you placed it in the data frame. A vector with a **factor** class is like a character vector in that you see letters and that you can't do math on it. However, a factor has additional properties: in particular, it is a grouping variable. See what happens when you print the `pond` variable:

```
df1$pond
```

```
## [1] F11 S28 S30 S8  S11
## Levels: F11 S11 S28 S30 S8
```

That looks weird, huh? A factor has levels, with each level being a subcategory of the factor. You can see the unique levels of your factor by running:

```
levels(df1$pond)
```

```
## [1] "F11" "S11" "S28" "S30" "S8"
```

Additionally, factor levels have an assigned order (even if the levels are totally nominal), which will become important in Chapter 3 when you learn how to fit linear models to groups of data, in which one level is the "reference" group that all other groups are compared to.

If you're running into errors about R expecting character vectors, it may be because they are actually stored as factors. When you make a data frame, you'll often have the option to turn off the automatic factor coersion. For example:

```
data.frame(month, day, year, number, pond, stringsAsFactors = F)
read.csv("streams.csv", stringsAsFactors = F)  # see below for details on read.csv
```

will result in character vectors remaining that way as opposed to being coerced to factors. This can be preferable if you are doing many string manipulations, as character vectors are often easier to work with than factors.

1.6 Vector Math

R does vectorized calculations. This means that if supplied with two numeric vectors of equal length and a mathematical operator, R will perform the calculation on each pair of elements. For example, if you wanted to add the two vectors vector `day` and `month`, then you would just run:

```
dm = day + month; dm
```

```
## [1]  3  6  9 12 15
```

You typically should ensure that the vectors you are doing math with are of equal lengths.

You could do the same calculation to each element (e.g., divide each element by 2) with:

```
dm/2
```

```
## [1] 1.5 3.0 4.5 6.0 7.5
```

1.7 Data Queries/Subsets/Retrievals

This perhaps the most important and versatile skills to know in R. So you have an object with data in it and you want to use it for analysis. But you don't want the whole dataset. You want to use just a few rows or just a few columns, or perhaps you need just a single element from a vector. This section is devoted to ways you can extract certain parts of a data object (the terms **query** and **subset** are often used interchangeably to describe this task). There are three main methods:

1. **By Index** – This method allows you to pull out specific rows/columns by their location in an object. However, you must know exactly where in the object the desired data are. An **index** is a location of a element in a data object, like the element position or the position of a specific row or column. To subset by index, you specify the object, then what rows, then what columns. The syntax for subsetting a vector by index is `vector[element]` and for a matrix it is `matrix[row,column]`. Here are some examples

```
# show all of day, then subset the third element
day; day[3]
```

```
## [1] 1 3 5 7 9
```

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

```
# show all of m1, then subset the cell in row 1 col 3
m1; m1[1,3]
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     4   5 2018      8
## [4,]     5   7 2018     10
## [5,]     6   9 2018     15
```

```
## year
## 2018
```

```
# show all of df1, then subset the entire first column
df1; df1[,1]
```

```
##   month day year number pond
## 1     2   1 2018      4  F11
## 2     3   3 2018      7  S28
## 3     4   5 2018      8  S30
## 4     5   7 2018     10   S8
## 5     6   9 2018     15  S11
```

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

Note this last line: the `[,1]` says "keep all the rows, but take only the first column".

Here is another example:

```
# show m1, then subset the 1st, 2nd, and 4th rows and every column
m1; m1[c(1,2,4),]
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     4   5 2018      8
## [4,]     5   7 2018     10
## [5,]     6   9 2018     15
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     5   7 2018     10
```

Notice how you can pass a vector of row indices here to exclude the 3rd and 5th rows.

2. By name – This method allows you to pull out a specific column of data based on what the column name is. Of course, the column must have a name first. The name method uses the `$` operator:

```
df1$month
```

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

You can combine these two methods:

```
df1$month[3]
```

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

```
# or
df1[,"year"]
```

```
## [1] 2018 2018 2018 2018 2018
```

The by name (`$`) method is useful because it can be used to add columns to a data frame:

```
df1$dm = df1$day + df1$month; df1
```

```
##   month day year number pond dm
## 1     2   1 2018      4  F11  3
## 2     3   3 2018      7  S28  6
## 3     4   5 2018      8  S30  9
## 4     5   7 2018     10   S8 12
## 5     6   9 2018     15  S11 15
```

3. **Logical Subsetting** – This is perhaps the most flexible method, and is described in Section 1.11.

EXERCISE 1A

Take a break to apply what you've learned so far. In this exercise, you'll be using what you learned about creating objects and the differences between different data classes. You will be entering this data frame into R by hand and doing some basic data subsets.

Lake	Area	Time	Fish
Big	100	1000	643
Small	25	1200	203
Square	45	1400	109
Circle	30	1600	15

1. Create a new file called `Ex_1A.R`. Set the working directory to the folder containing `Ex_1A.R`.
2. Enter these data into vectors. Call the vectors whatever you would like. Should you enter the data as vectors by rows, or by columns? (*Hint: remember the properties of vectors*).
3. Combine your vectors into a data frame. Why should you use a data frame instead of a matrix?
4. Subset all of the data from Small Lake.
5. Subset the area for all of the lakes.
6. Subset the number of fish for Big and Square lakes.
7. You realize that you sampled 209 fish at Square Lake, not 109. Fix the mistake. There are two ways to do this, can you think of them both? Which do you think is better?
8. Save your script. Close R and re-open your script to see that it was saved.

1.8 Read External Data Files

It is rare that you will enter data by hand as you did in Exercise 1A. Often, you have a dataset that you wish to analyze or manipulate. R has several ways to read information from data files and in this workshop, we will be using a common and simple method: reading in `.csv` files. `.csv` files are data files that separate columns with commas[7]. If your data are in a Microsoft Excel spreadsheet, you can save your spreadsheet file as a CSV file (*File > Save As > Save as Type> CSV (Comma Delimited)*). Several dialog boxes will open asking if you are sure you want to save it as a `.csv` file.

The syntax for reading in a `.csv` file is:

[7]Note that if your computer is configured for a Spanish-speaking country, Microsoft Excel might convert decimals to commas. This can really mess with reading in data - I would suggest changing the language of Excel if you find this to be the case.

```
dat = read.csv("FileName.csv")
```

The data files for this workshop are found in the GitHub repository: https://github.com/bstaton1/au-r-workshop-data/tree/master. Navigate to this repository, find the file called `streams.csv`, download that file[8], and place it in your working directory for this session. This document assumes your working directory is `C:/Users/YOU/Documents/R-Workshop/Chapter1`, though the one you are actually using may be slightly different. We can read the contents of `streams.csv` into R using this code (ensure your working directory is set first):

```
dat = read.csv("streams.csv")
```

If the data set is in your working directory, all you need to provide is the file name. If you do not get an error, congratulations! However, if you get an error that looks like this:

```
## Warning in file(file, "rt"): cannot open file 'streams.csv': No such file
## or directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

then fear not. This has must be among the most common errors encoutered by R users world-wide. It simply means the file you told R to look for doesn't exist where you told R to find it. Here is a trouble-shooting guide to this error:

1. The exact case and spelling matters, as well as do the quotes and `.csv` at the end. Ensure the file name is typed correctly.
2. Check what files are in your working directory: run `dir()`. This will return a vector with the names of the files located in your working directory. Is the file you told R was there truly in there? Is your working directory set to where you thought it was?
3. If the file is not in your working directory, and you wish to keep it that way, you must point R to where that file is. Pretend for a moment that we put the data file in `/R-Workshop/DataHere`. Here are two examples that would both read in the file in this case:

```
dat = read.csv("C:/Users/YOU/Documents/R-Workshop/Data/streams.csv")
dat = read.csv("../Data/streams")
```

The first line is the full file path to the file in question. The second line uses a **relative path**: the `../` says "look one folder up from the working directory for a folder called `Data` then find a file called `streams.csv` in it."

If you did not get any errors, then the data are in the object you named (`dat` here) and that object is a data frame. Do not proceed until you are able to `read.csv` to run successfully.

A few things to note about reading in `.csv` files:

- If even a single character is found in a numeric column in `FileName.csv`, the *entire column* will be coerced to a character/factor data class after it is read in (i.e., no more math with data on that column until you remove the character). A common error is to have a `#VALUE!` record left over from an invalid Excel function result. You must remove all of these occurences in order to use that column as numeric. Characters include anything other than a number ([0-9]) and a period when used as a decimal. None of these characters: `!@#$%^&*()_-+=[a-z];[A-Z]` should never be found in a column you wish to do math with (e.g., take the mean of that column).
- If a record (i.e., cell) is truly missing and you wish R to treat it that way (i.e., as an `NA`), you have three options:
  - Hard code an `NA` into that cell in Excel

[8] To download a single file from GitHub, click on the file, then click "Raw" in the toolbar on the topright, then right-click anywhere in the document and click "Save As...". You can also download a zip folder with all of the `.csv` files, go to "Clone or Download"

  - Leave that cell completely empty
  - Enter in some other character (e.g., `"."`) alone in all cells that are meant to be coded as `NA` in R and use the `na.strings = "."` argument of `read.csv`.

- If a cell in the first row with text contains a space between two words, R will insert a `"."` between the words.
- If your data have column names (i.e., characters) in the first row, R will bring those in as column names by default.
- R brings in CSV files in as data frames by default.
- If at some point you did "Clear Contents" in Microsoft Excel to delete rows or columns from your `.csv` file, these deleted rows/columns will be read in as all `NA`s, which can be annoying. To remove this problem, open the `.csv` file in Excel, then highlight and **delete** the rows/columns and save the file. Read it back into R again using `read.csv`.

1.9 Explore the Data Set

Have a look at the data. You could just run `dat` to view the contents of the object, but it will show the whole thing, which may be undesirable if the data set is large. To view the first handful of rows, run `head(dat)` or the last handful of rows with `tail(dat)`.

You will now use some basic functions to explore the streams data before any analysis. The `summary` function is very useful for getting a coarse look at how R has interpretted the data frame:

```
summary(dat)
```

```
##        state    stream_width        flow
##  Alabama  :5   Min.   :17.65   Min.   : 28.75
##  Florida  :5   1st Qu.:46.09   1st Qu.: 65.50
##  Georgia  :5   Median :61.80   Median : 95.64
##  Tennessee:5   Mean   :60.88   Mean   : 91.49
##                3rd Qu.:79.34   3rd Qu.:120.55
##                Max.   :94.65   Max.   :149.54
##                                NA's   :1
```

You can see the spread of the numeric data and see the different levels of the factor (`state`) as well as how many data points belong to each category. Note that there is one `NA` in the variable called `flow`.

To count the number of elements in a variable (or any vector), remember the `length` function:

```
length(dat$stream_width)
```

```
## [1] 20
```

Note that R counts missing values as elements as well:

```
length(dat$flow)
```

```
## [1] 20
```

To get the dimensions of an object with more than one dimension (i.e., a data frame or matrix) we can use the `dim` function. This returns a vector with two elements: the first number is the number of rows and the second is the number of columns. If you only want one of these, use the `nrow` or `ncol` functions (but remember, only for objects with more than one dimension; vectors don't have rows or columns!).

```
dim(dat); nrow(dat); ncol(dat)
```

```
## [1] 20  3
```

```
## [1] 20
```

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

You can extract the names of the variables (i.e., columns) in the data frame using `colnames`:

```
colnames(dat)
```

```
## [1] "state"        "stream_width" "flow"
```

Calculate the mean of all of the flow records:

```
mean(dat$flow)
```

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

It returned an `NA` because there is an `NA` in the data for this variable. There is a way to tell R to ignore this `NA`. Include the argument `na.rm = TRUE` in the `mean` function (separate arguments are always separated by commas). This is a **logical** argument, meaning that it asks a question. It says "do you want to remove `NA`s before calculating the mean?" `TRUE` means "yes" and `FALSE` means "no." These can be abbreviated as `T` or `F`. Many of R's functions have the `na.rm` argument (e.g. `mean`, `sd`, `var`, `min`, `max`, `sum`, etc. - most anything that collapses a vector into one number).

```
mean(dat$flow, na.rm = T)
```

```
## [1] 91.49053
```

which is the same as (i.e., the definition of the mean with the `NA` removed):

```
sum(dat$flow, na.rm = T)/(nrow(dat) - 1)
```

```
## [1] 91.49053
```

What if you need to do something to more than one variable at a time? One of the easiest ways to do this (though as with most things in R, there are many) is by using the `apply` function. This function applies the same summary function to individual subsets of a data object at a time then returns the individual summaries all at once:

```
apply(dat[,2:3], 2, FUN = var, na.rm = T)
```

```
## stream_width         flow
##     581.1693    1337.3853
```

The first argument is the data object you want to apply the function to. The second argument (the number `2`) specifies that you want to apply the function to columns, `1` would tell R to apply it to rows. The `FUN` argument specifies what function you wish to apply to each of the columns; here we are calculating the variance which takes the `na.rm = T` argument. This use of `apply` alone is very powerful and can help you get around having to write the dreaded `for` loop (introduced in Chapter 4).

There is a whole family of `apply` functions, the base `apply` is the most basic but a more sophisticated one is `tapply`, which applies a function based on some grouping variable (a factor). Calculate the mean stream width **separated by state**:

```
tapply(dat$stream_width, dat$state, mean)
```

```
##   Alabama   Florida   Georgia Tennessee
##    53.664    63.588    54.996    71.290
```

The first argument is the variable you want to apply the `mean` function to, the second is the grouping variable, and the third is what function you wish to apply. Try to commit this command to memory given this is a pretty common task.

1.10 Logical/Boolean Operators

To be an efficient and capable programmer in any language, you will need to become familiar with how to implement numerical logic, i.e., the boolean operators. These are very useful because they always return a `TRUE` or a `FALSE`, off of which program-based decisions can be made (e.g., whether to operate a given subroutine, whether to keep certain rows, whether to print the output, etc.).

Define a simple object: `x = 5` . Note that this will write over what was previously stored in the object `x`. We wish to ask some questions of the new `x`, and the answer will be printed to the console as a `TRUE` for "yes" and a `FALSE` for "no". Below are the common Boolean operators.

Equality

To ask if `x` is exactly equal to 5, you run:

```
x == 5
```

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

Note the use of the double equals-sign to denote equality as opposed to the single `=` as used in assignment when you ran `x = 5` a minute ago.

Inequalities

To ask if `x` is not equal to 5, you run:

```
x != 5
```

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

To ask if `x` is less than 5, you run:

```
x < 5
```

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

To ask if `x` is less than *or equal to* 5, you run:

```
x <= 5
```

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

Greater than works the same way, though with the `>` symbol replaced.

And

Suppose you have two conditions, and you want to know if **both are met**. For this you would use **and** by running:

```
x > 4 & x < 6
```

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

which asks if `x` is between 4 and 6.

Or

Suppose you have two conditions, and you want to know if **either are met**. For this you would use **or** by running:

```
x <= 5 | x > 5
```

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

which asks if `x` is less than or equal to 5 **or** greater than 5 - you would be hard-pressed to find a real number that did not meet these conditions!

1.11 Logical Subsetting

A critical use of logical/Boolean operators is in the subsetting of data objects. You can use a logical vector (i.e., one made of only `TRUE` and `FALSE` elements) to tell R to extract only those elements corresponding to the `TRUE` records. For example:

```
# here's logical vector: TRUE everywhere condition met
dat$stream_width > 60
```

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

```
# insert it to see only the flows for the TRUE elmements
dat$flow[dat$stream_width > 60]
```

```
##  [1] 120.48 123.78  95.64  95.82 120.06 135.63 120.61 111.34 149.54 131.22
```

gives all of the `flow` values for which `stream_width` is greater than 60.

To see all of the data from Alabama, you would run:

```
dat[dat$state == "Alabama",]
```

```
##     state stream_width   flow
## 1 Alabama        81.68 120.48
## 2 Alabama        57.76  85.90
## 3 Alabama        48.32  73.38
## 4 Alabama        31.63  46.55
## 5 Alabama        48.93  80.91
```

You will be frequently revisiting this skill throughout the workshop.

1.12 if, else, and ifelse

You can tell R to do something if the result of a question is `TRUE`. This is a typical if-then statement:

```
if (x == 5) print("x is equal to 5")
```

```
## [1] "x is equal to 5"
```

This says "if `x` equals 5, then print the phrase 'x is equal to 5' to the console". If the logical returns a `FALSE`, then this command does nothing:

```
if (x != 5) print("x is equal to 5")
```

We can tell R to do multiple things if the logical is `TRUE` by using curly braces:

```
if (x == 5) {
  print("x is equal to 5")
  print("you dummy, x is supposed to be 6")
}
```

```
## [1] "x is equal to 5"
## [1] "you dummy, x is supposed to be 6"
```

You can always use curly braces to extend code across multiple lines whereas it may have been intended to go on one line.

What if you want R to do something if the logical is `FALSE`? Then you would use the `else` command:

```
if (x > 5) print("x is greater than 5") else print("x is not greater than 5")
```

```
## [1] "x is not greater than 5"
```

Or extend this same thing to multiple lines:

```
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

```
## [1] "x is not greater than 5"
```

The `if` function is useful, but it can only ask one question at a time. If you supply it with a vector of length greater than 1, it will give a warning:

```
# vector from -5 to 5, excluding zero
xs = c(-5:-1, 1:5)

# attempt a logical decision
if (xs < 0) print("negative")
```

```
## Warning in if (xs < 0) print("negative"): the condition has length > 1 and
## only the first element will be used
```

```
## [1] "negative"
```

Warnings are different than errors in that something still happens, but it tells you that it might not be what you wanted, whereas an error stops R altogether. In short, this warning is telling you that you passed `if` a logical vector with more than 1 element, and that it can only use one element so it's picking the first one. Because the first element of `xs` is -5, `xs < 0` evaluated to `TRUE`, and we got a `"negative"` printed along with our warning.

To ask multiple questions at once, we must use `ifelse`. This function is similar, but it combines the `if` and `else` syntax into one useful function function:

```
ifelse(xs > 0, "positive", "negative")
```

```
##  [1] "negative" "negative" "negative" "negative" "negative" "positive"
##  [7] "positive" "positive" "positive" "positive"
```

The syntax is `ifelse(condition, do_if_TRUE, do_if_FALSE)`. You can cbind the output with `xs` to verify it worked:

```
cbind(
  xs,
```

```
  ifelse(xs > 0, "positive", "negative")
)
```

```
##       xs
##  [1,] "-5" "negative"
##  [2,] "-4" "negative"
##  [3,] "-3" "negative"
##  [4,] "-2" "negative"
##  [5,] "-1" "negative"
##  [6,] "1"  "positive"
##  [7,] "2"  "positive"
##  [8,] "3"  "positive"
##  [9,] "4"  "positive"
## [10,] "5"  "positive"
```

Use `ifelse` to create a new variable in `dat` that indicateds whether stream is big or small depending **whether stream with is greater or less than 50**:

```
dat$size_cat = ifelse(dat$stream_width > 50, "big", "small"); head(dat)
```

```
##     state stream_width   flow size_cat
## 1 Alabama        81.68 120.48      big
## 2 Alabama        57.76  85.90      big
## 3 Alabama        48.32  73.38    small
## 4 Alabama        31.63  46.55    small
## 5 Alabama        48.93  80.91    small
## 6 Georgia        39.42  57.63    small
```

This says "make a new variable in the data frame `dat` called `size_cat` and assign each row a 'big' if `stream_width` is greater than 50 and a 'small' if less than 50".

One neat thing about `ifelse` is that you can nest multiple statements inside another[9]. **What if we wanted three categories: 'small', 'medium', and 'large'?**

```
dat$size_cat_fine = ifelse(dat$stream_width <= 40, "small",
                           ifelse(dat$stream_width > 40 & dat$stream_width <= 70, "medium", "big")); hea
```

```
##     state stream_width   flow size_cat size_cat_fine
## 1 Alabama        81.68 120.48      big          big
## 2 Alabama        57.76  85.90      big       medium
## 3 Alabama        48.32  73.38    small       medium
## 4 Alabama        31.63  46.55    small        small
## 5 Alabama        48.93  80.91    small       medium
## 6 Georgia        39.42  57.63    small        small
```

If the first condition is `TRUE`, then it will give that row a "small". If not, it will start another `ifelse` to ask if the `stream_width` is greater than 40 *and* less than or equal to 70. If so, it will give it a "medium", if not it will get a "big". This is confusing at first. Not all nesting or embedding of functions are this complex, but this is a neat example. Without `ifelse`, you would have to use as many `if` statements as there are elements in `dat$stream_width`.

[9] You can nest **ALL** R functions, by the way.

1.13 Writing Output Files

1.13.1 .csv Files

Now that you have made some new variables in your data frame, you may want to save this work in the form of a new `.csv` file. To do this, you can use the `write.csv` function:

```
write.csv(dat, "updated_streams.csv", row.names = F)
```

The first argument is the data frame (or matrix) to write, the second is what you want to call it (don't forget the `.csv`!), and `row.names = F` tells R to not include the row names (because they are just numbers in this case). R puts the file in your working directory unless you tell it otherwise. To put it somewhere else, type in the path with the new file name at the end.

1.13.2 Saving R Objects

Another method is to save R objects directly to the working directory. You can save the new data frame using:

```
save(dat, file = "updated_streams")
```

Then try removing the `dat` object from your current session (`rm(dat)`) and loading it back in using:

```
rm(dat); head(dat)  # should give error
load(file = "updated_streams")
head(dat) # should show first 6 rows
```

1.14 User-Defined Functions

Sometimes you may want R to carry out a specific task, but there is no built-in function to do it. In this case, you can write your own functions. This one of the parts of R that makes it incredibly flexible, though you will only get a small taste of this topic here. We will be going into more detail in later Chapters, particularly in Chapter 5.

First, you must think of a name for your function (e.g., `myfun`). Then, you specify that you want the object `myfun` to be a function by using using the `function` function (I know). Then, in parentheses, you specify any arguments that you want to use within the function to carry out the specific task. Open and closed curly braces specify the start and end of you function body, i.e., the code that specifies how it uses the arguments to do its job.

Here's the general syntax for specifying your own function:

```
myfun = function(arg1) {
  # function body goes here
  # use arg1 to do something

  # return something as last step
}
```

Write a general function to take any number `x` to any power `y`:

```
power = function(x, y){
  x^y
}
```

After typing and running the function code (the function is an object that must be assigned), try using it:

```
power(x = 5, y = 3)
```

```
## [1] 125
```

Remember, you can nest or embed functions:

```
power(power(5,2),2)
```

```
## [1] 625
```

This is the equivalent of $(5^2)^2$.

EXERCISE 1B

These data are from a hypothetical pond experiment where you added nutrients to some mesocosoms and counted the densities of 4 different zooplankton taxa. In this experiment, there were two ponds, two treatments per pond, and five replicates of each treatment. There is one error in the dataset "ponds.csv" (remember, found in this repository: https://github.com/bstaton1/au-r-workshop-data/tree/master). After you download the data and place it in the appropriate directory, make sure you open this file and fix it *before* you bring it into R. Refer back to the information about reading in data (Section 1.8) to make sure you find the error.

1. Read in the data to R and assign it to an object.
2. Calculate some basic summary statistics of your data using the `summary` function.
3. Calculate the mean chlorophyll *a* for each pond (*Hint: pond is a grouping variable*)
4. Calculate the mean number of *Chaoborus* for each treatment in each pond using `tapply`. (*Hint: You can group by two variables with:* `tapply(dat$var, list(dat$grp1, dat$grp2), fun)`.
5. Use the more general `apply` function to calculate the variance for each zooplankton taxa found only in pond S-28.
6. Create a new variable called `prod` in the data frame that represents the quantity of chlorophyll *a* in each replicate. If the chlorophyll *a* in the replicate is greater than 30 give it a "high", otherwise give it a "low". (*Hint: are you asking a question of a single number or multiple numbers? How should this change the strategy you use?*)

EXERCISE 1B BONUS

1. Create a new function called `product` that multiplies two numbers that you specify.
2. Modify your function to print a message to the console and return the value `if` it meets a condition and to print another message and not return the value if it doesn't.

Chapter 2

Base R Plotting Basics

Chapter Overview

In this chapter, you will get familiar with the basics of using R for making plots and figures. You will learn:

- the different types of plotting functions
- the basics of how to make:
  - scatterplots
  - line
  - bar
  - box-and-whisker plots
  - histograms
- the basics of how to change plot features like the text displayed, the size of points, and the type of points
- the basics of multi-panel plotting
- the basics of the `par` function
- how to save your plot to an external file

R's base plotting package is incredibly versatile, and as you will see, it doesn't take much to get started making professional-looking graphs. Before we get started, I'll just mention that there are other R packages[1] for plotting (e.g. `ggplot2` and `lattice`), but they can be more complex than the base R plotting capabilities and look a bit different.

IMPORTANT NOTE: If you did not attend the session corresponding to the Chapter 1 material, you are recommended to walk through those chapters' contents before proceeding to this material. This is because you will be using some of the commands and terminology presented previously. If you are confused about a topic, you can use **CTRL + F** to find previous cases where that topic has been discussed in this document.

Before You Begin

You should create a new directory and R script for your work in this Chapter. Create a new R script called `Ch2.R` and save it in the directory `C:/Users/YOU/Documents/R-Workshop/Chapter2`. Set your working directory to that location. Revisit the Sections 1.2 and 1.3 for more details on these steps.

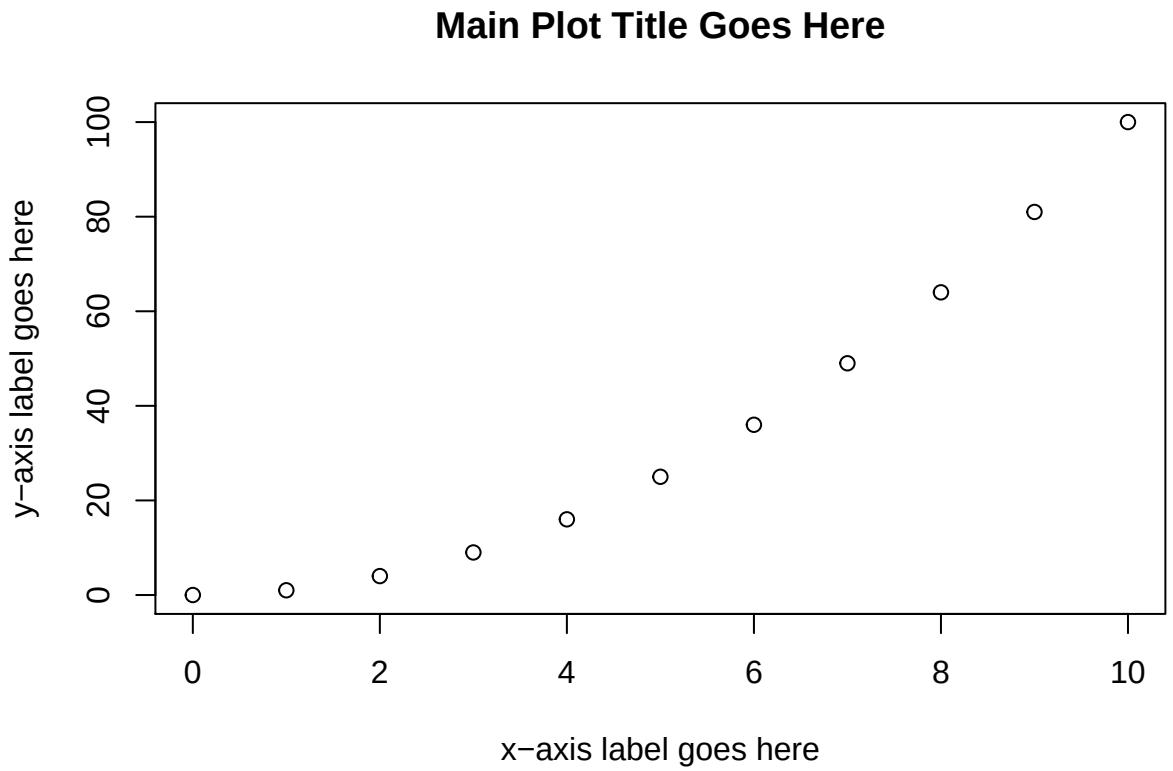
[1] An **R package** is a bunch of code that somebody has written and placed on the web for you to install, their use is first introduced in Chapter 5

2.1 R Plotting Lingo

Learning some terminology will help you get used to the base R graphics system:

- A **high-level plotting function** is one that is used to make a new graph. Examples of higher level plotting functions are the general `plot` and the `barplot` functions. When-a high level plotting function is executed, the currently displayed plot (if any) is written over.
- A **low-level plotting function** is one that is used to modify a plot that is already created. You must already have made a plot using a higher level plotting function before you use lower level plotting functions. Examples include `text`, `points`, and `lines`. When a low-level plotting function is executed, the output is simply added to an existing plot.
- The **graphics device** is the area that a plot shows up. R Studio has a built in graphics device in its interface (lower right by default in the "Plots" tab), or you can create a new device. R will plot on the active device, which is the most recently created device. There can only be one active device at a time.

Here is an example of a basic R plot:



There are a few components:

- The **plotting region**: all data information is displayed here.
- The **margin**: where axis labels, tick marks, and main plot titles are located
- The **outer margin**: by default, there is no outer margin. You can add one if you want to add text here or make more room around the edges.

You can change just about everything there is about this plot to suit your tastes. Duplicate this plot, but make the x-axis, y-axis, and main titles other than the placeholders shown here:

Table 2.1: Several of the key arguments to high- and low-level plotting functions

Arg.	Usage	Description
‘xlab‘	‘xlab = ”X-AXIS”‘	changes the x-axis label text
‘ylab‘	‘ylab = ”Y-AXIS”‘	changes the y-axis label text
‘main‘	‘main = ”TITLE”‘	changes the main title text
‘cex‘	‘cex = 1.5‘	changes the size of symbols in the plotting region^[‘cex‘ is a multiplier: ‘cex = 1.5‘ says mal
‘pch‘	‘pch = 17‘	changes the symbol type^[There are approximately 20 different ‘pch‘ settings: ‘pch = 1‘ is e
‘xlim‘	‘xlim = range(x)‘	changes the endpoints (limits) of the x-axis^[‘xlim‘ and ‘ylim‘ both require a numeric vector
‘ylim‘	‘ylim = c(0,1)‘	same as ‘xlim‘, but for the y-axis
‘type‘	‘type = ”l”‘	changes the way points are connected by lines^[The default is points only, ‘type = ”l”‘ is for
‘lty‘	‘lty = 2‘	changes the line type^[‘lty = 1‘ is solid, ‘lty = 2‘ is dashed, ‘lty = 3‘ is dotted, etc. You can
‘lwd‘	‘lwd = 2‘	changes the line width^[works just like ‘cex‘: ‘lwd = 3‘ codes for a line that is 3 times as th
‘col‘	‘col = ”blue”‘	changes the color of plotted objects^[there is a whole host of colors you can pass R by name

```
x = 0:10
y = x^2

plot(x = x, y = y,
     xlab = "x-axis label goes here",
     ylab = "y-axis label goes here",
     main = "Main Plot Title Goes Here")
```

Note that the first two arguments, `x` and `y`, specify the coordinates of the points (i.e., the first point is placed at coordinates `x[1]`,`y[1]`). `plot` has *tons* of arguments (or *graphical parameters* as the help file found using `?plot` or `?par` calls them) that change how the plot looks. Note that when you want something displayed verbatim on the plotting device, you must wrap that code in `" "`, i.e., the arguments `xlab`, `ylab`, and `main` all receive a character vector of length 1 as input.

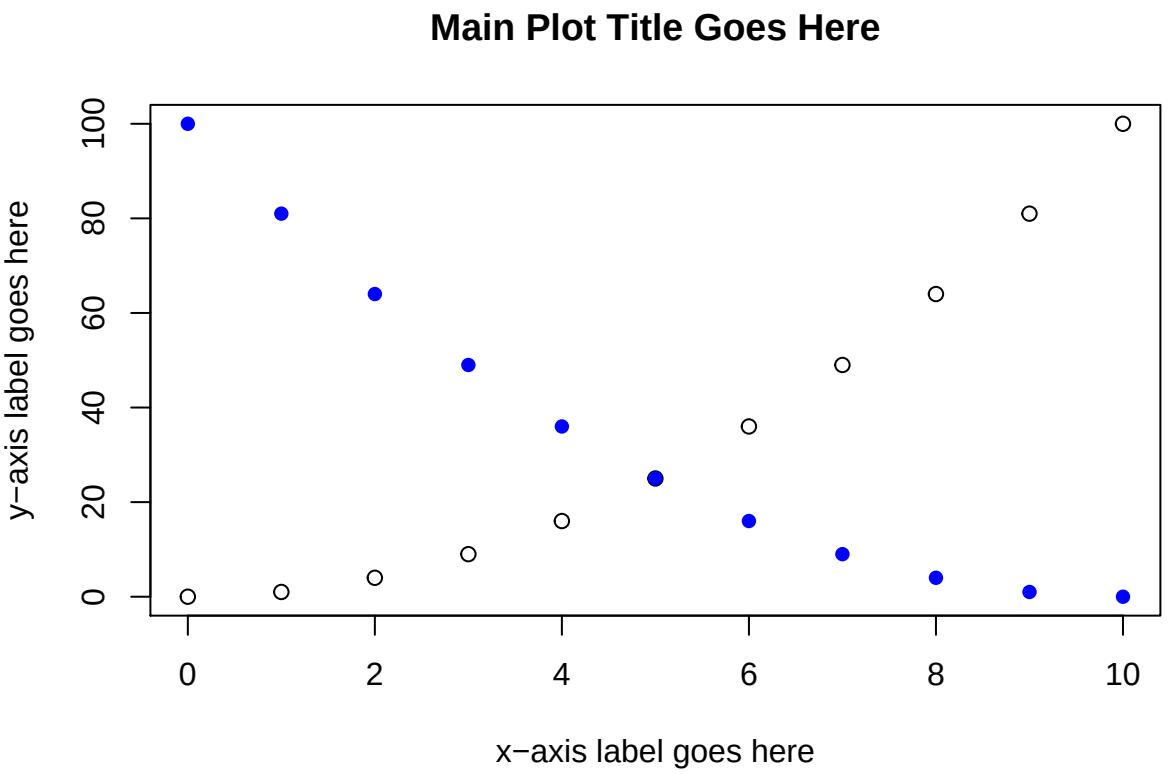
Here are just a handful of them to get you started:

You are advised to try at least some of these arguments out for yourself with your `plot(x, y)` code from above - notice how every time you run `plot`, a whole new plot is created, not just the thing you changed. There are definitely other options: check out `?plot` or `?par` for more details.

2.2 Lower Level Plotting Functions

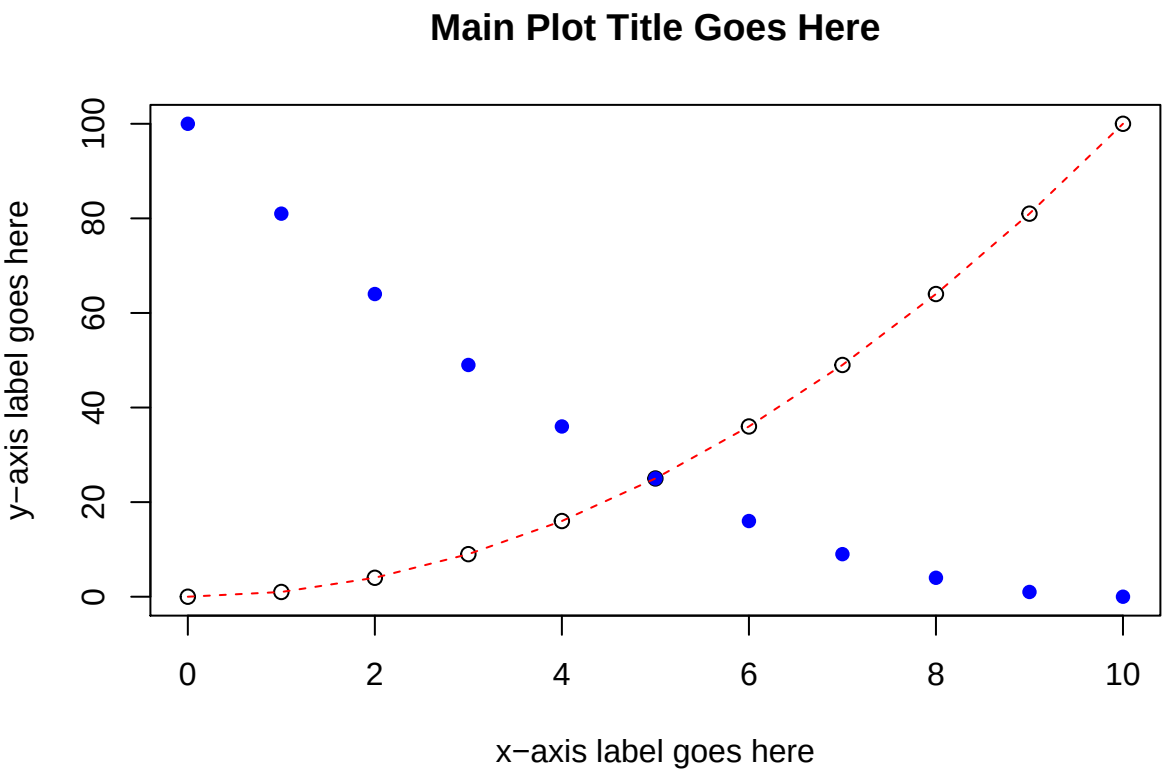
Now that you have a base plot designed to your liking, you might want to add some additional “layers” to it to represent more data or other kind of information like an additional label or text. Add some more points to your plot by putting this line right beneath your `plot(x,y)` code and run just the `points()` line (make sure your device is showing a plot first):

```
# rev() reverses a vector: so the old x[1] is x[11] now
points(x = rev(x), y = y, col = "blue", pch = 16)
```



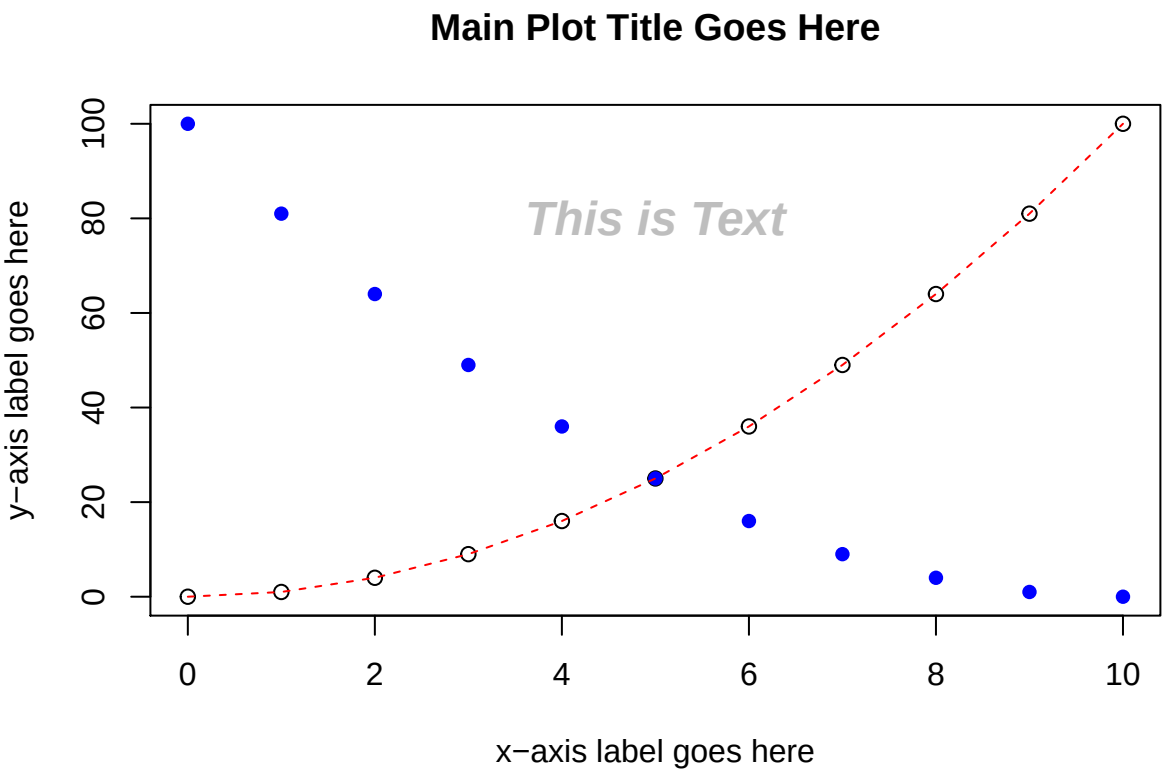
Here, `points` acted like a low-level plotting function because it added points to a plot you already made. Many of the arguments shown in Table 2.1 can be used in both high-level and low-level plotting functions (notice how `col` and `pch` were used in `points`). Just like `points`, there is also `lines`:

```
lines(x = x, y = x, lty = 2, col = "red")
```



You can also add text to the plotting region:

```
text(x = 5, y = 80, "This is Text", cex = 1.5, col = "grey", font = 4)
```

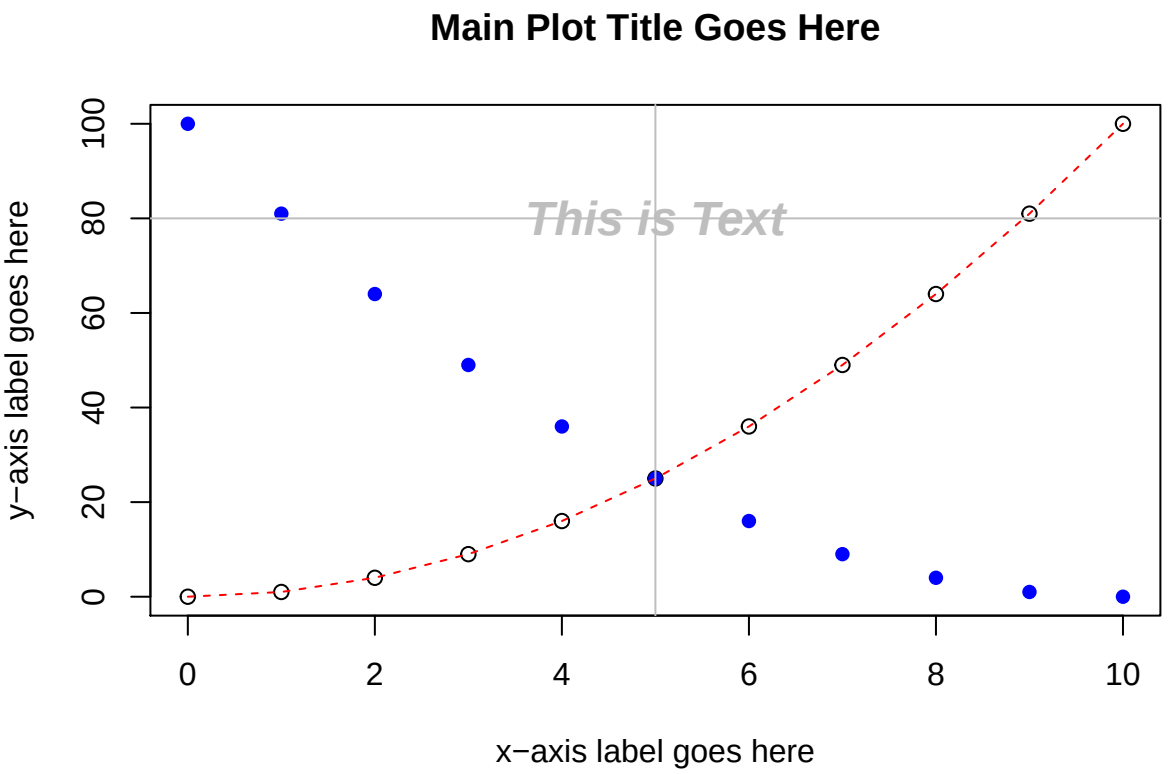


The text is centered on the coordinates you provide. You can also provide vectors of coordinates and text to write different things at once.

The easiest way to add a straight line to a plot is with `abline`. By default it takes two arguments: `a` and `b` which are the intercept and slope, respectively, e.g., `abline(c(0,1))` will draw a 1:1 line. You can also do `abline(h = 5)` to draw a horizontal line at 5 or `abline(v = 5)` to draw a vertical line at 5.

You can see that the text is centered on the coordinates `x = 5` and `y = 80` using `abline`:

```
abline(h = 80, col = "grey")
abline(v = 5, col = "grey")
```



If you accidentally add a plot element that you don't want using a low-level plotting function, the only way to remove it is by re-running the high-level plotting function to start a new plot and adding only the objects you want. Try removing the "This is Text" text and the straight lines you drew with `abline` from the plot displayed in your device.

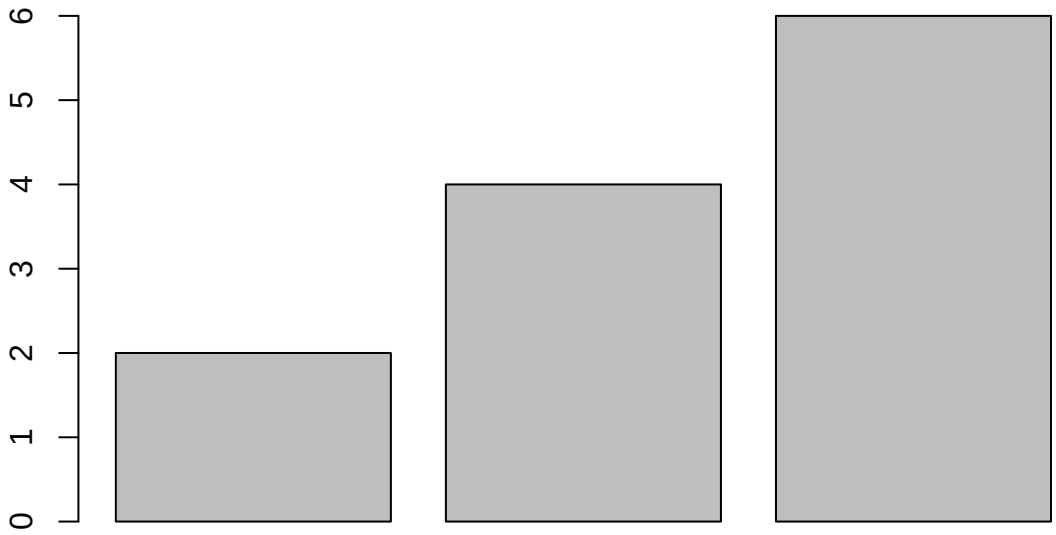
2.3 Other High-Level Plotting Functions

You have just seen the basics of making two dimensional scatter plots and line plots. You will now explore other types of graphs you can make.

2.3.1 The Bar Graph

Another very common graph is a bar graph. R has a `bargraph` function, and again, it has lots of arguments. Here you will just make two common variations: single bars per group and multiple bars per group. Let's make up some data to plot:

```
x1 = c(2,4,6)
barplot(x1)
```

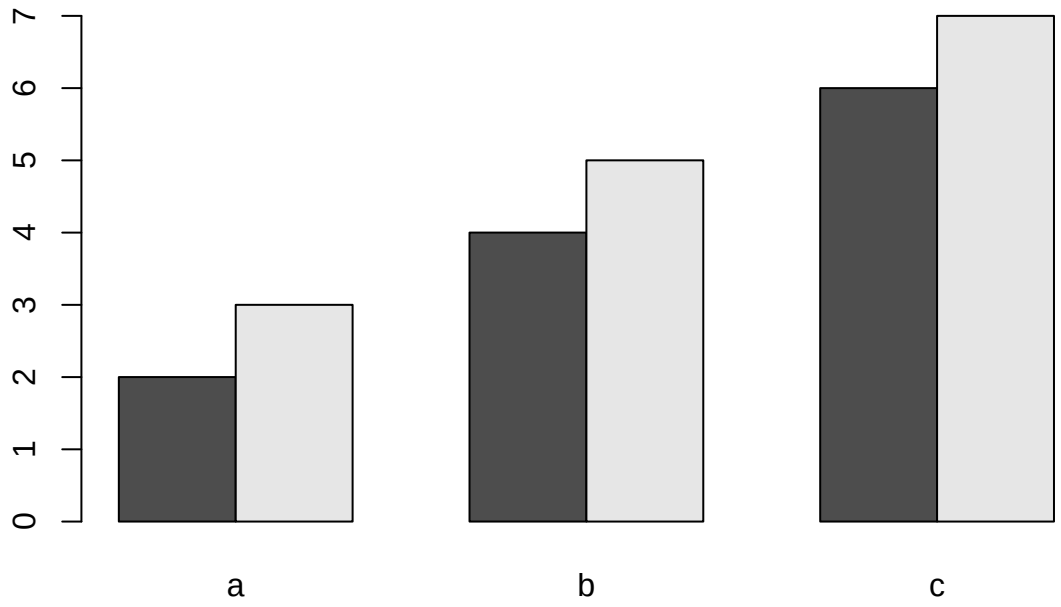


Notice that there are no group names on our bars (if our vector had names, there would be). You can add names by using the argument `names.arg`:

```
barplot(x1, names.arg = c("a", "b", "c"))
```

Add some more information by including two bars per group. Make some more data and combine it with the old data:

```
x2 = c(3,5,7)
x3 = rbind(x1, x2)

barplot(x3, names.arg = c("a", "b", "c"), beside = T)
```



To add multiple bars per group, R needs a matrix like you just made. The **columns** store the heights of the bars that will be placed together in a group. Including the `beside = T` argument tells R to plot all groups as different bars as opposed to using a stacked bar graph.

Oftentimes, we want to add error bars to a bar graph like this. To avoid digressing too much here, creating error bars is covered as a bonus topic.

2.4 Box-and-Whisker Plots

Box-and-whisker plots are a great way to visualize the spread of your data. All you need to make a box-and-whisker plot is a grouping variable (a factor, revisit Section 1.5 if you don't remember what these are) and some continuous (i.e., numeric) data for each level of the factor. Download the `creel.csv` data set from the GitHub repository https://github.com/bstaton1/au-r-workshop-data/tree/master, place it in the same directory as your R script `Ch2.R` (which is the one you should be working on right now).

Read the data in and print a summary:

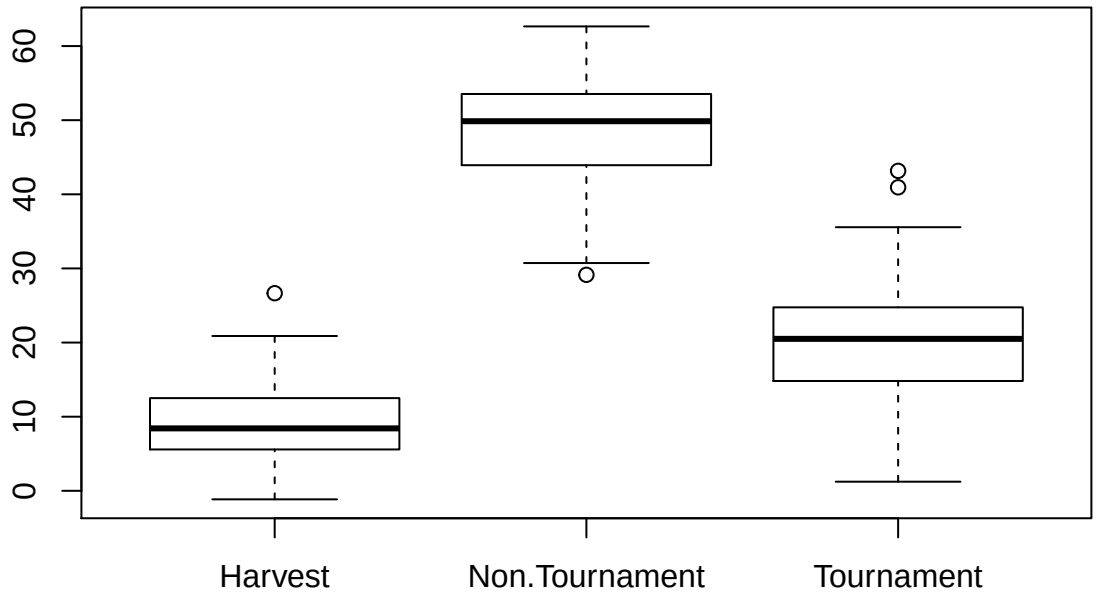
```
dat = read.csv("creel.csv")
summary(dat)
```

```
##            fishery        hours
##  Harvest       :100   Min.   :-1.150
##  Non.Tournament:100   1st Qu.: 9.936
##  Tournament    :100   Median :20.758
##                       Mean   :26.050
##                       3rd Qu.:43.896
```

```
##                   Max.   :62.649
```

This data set contains some simulated (i.e., fake) continuous and categorical data that represent 300 anglers who were creel surveyed[2]. In the data set, there are three categories (levels to the factor `fishery`) and the continuous variable is how many hours each angler fished this year. If you supply the generic `plot` function with a continuous response (`y`) variable and a categorical predictor (`x`) variable, it will automatically know that you want to make a box-and-whisker plot:

```
plot(x = dat$fishery, y = dat$hours)
```



In the box-and-whisker plot above, the heavy line is the median, the ends of the boxes are the 25th and 75th percentiles and the "whiskers" are the 2.5th and 97.5th percentiles. Any points that are outliers (i.e., fall outside of the whiskers) will be shown as points[3].

It is worth introducing a shorthand syntax of typing the same command:

```
plot(hours ~ fishery, data = dat)
```

Instead of saying `plot(x = x.var, y = y.var)`, this expression says `plot(y.var ~ x.var)`. The `~` reads "as a function of". By specifying the `data` argument, you no longer need to indicate where the variables `hours` and `fishery` are found. Many R functions have a `data` argument that works this same way. It is sometimes preferable to plot variables with this syntax because it is often less code and is also the format of R's statistical equations[4].

[2] A creel survey is a sampling program where fishers are asked questions about their fishing behavior in order to estimate effort and harvest.

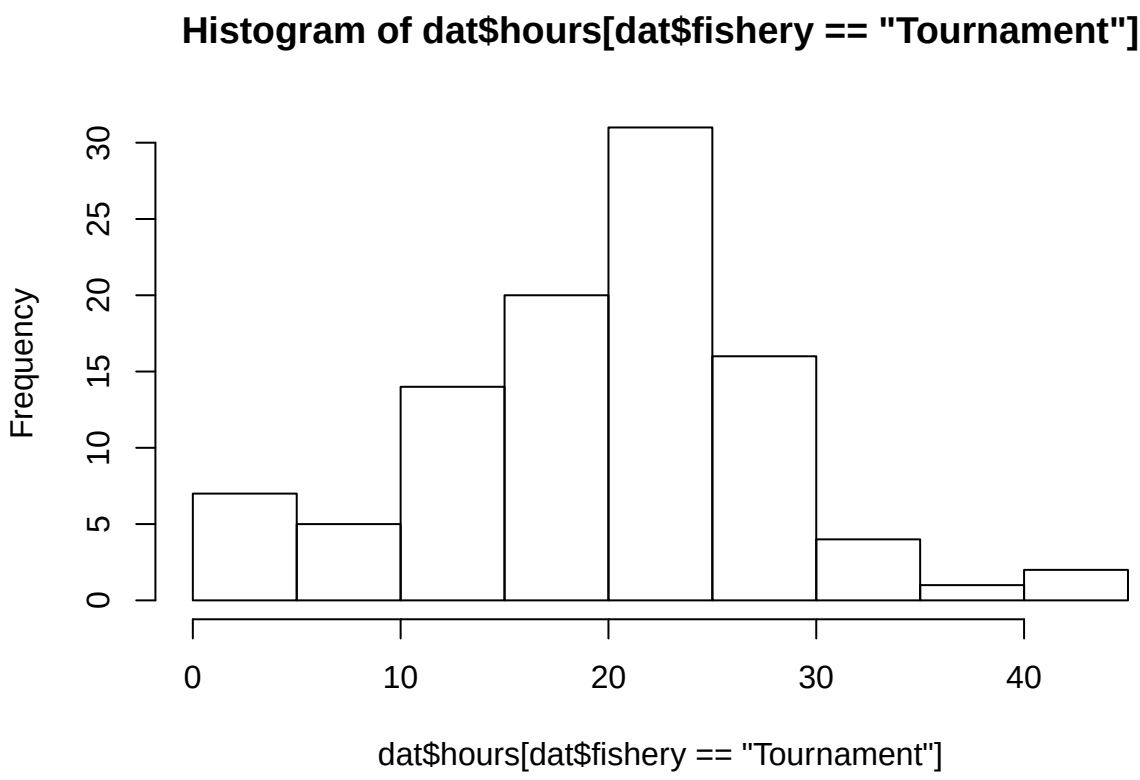
[3] Outliers can be turned off using the `outline = F` argument to the `plot` function

[4] Which allows you to easily copy and paste the code between the model and plot functions, see Chapter 3

2.5 Histograms

Another way to show the distribution of a variable is with histograms. These figures show the relative frequencies of observations in different discrete bins. Let's make a histogram of hours the surveyed tournament anglers fished this year:

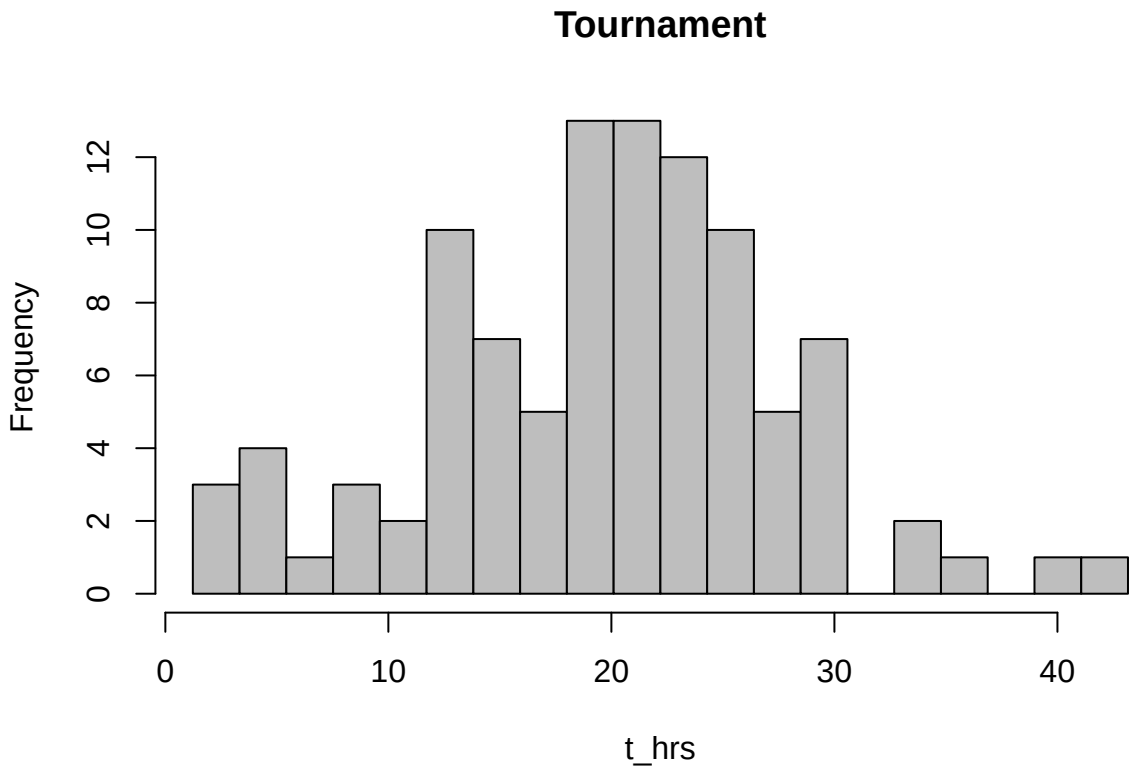
```
hist(dat$hours[dat$fishery == "Tournament"])
```



Notice the subset that extracts hours fished for tournament anglers only before plotting.

`hist` automatically selects the number of bins based on the range and resolution of the data. You can specify how many evenly-sized bins you want to plot:

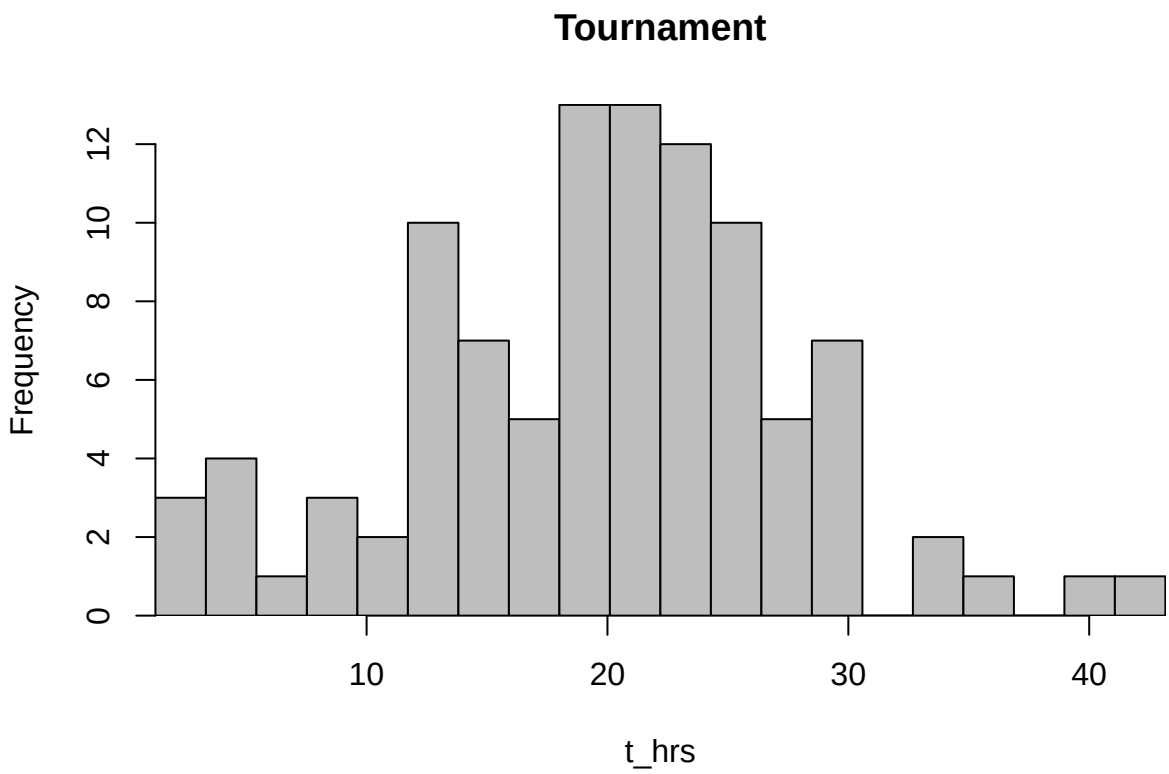
```
# extract the hours for tournament anglers
t_hrs = dat$hours[dat$fishery == "Tournament"]

# create the bin endpoints
nbins = 20
breaks = seq(from = min(t_hrs), to = max(t_hrs), length = nbins + 1)
hist(t_hrs, breaks = breaks, main = "Tournament", col = "grey")
```



2.6 The par Function

If it bothers you that the axes are "floating", you can fix this using this command:

```
par(xaxs = "i", yaxs = "i")
hist(t_hrs, breaks = breaks, main = "Tournament", col = "grey")
```



Here, you changed the graphical parameters of the graphics device by using the `par` argument. Once you change the settings in `par`, they will remain that way until you start a new device.

The `par` function is central to fine-tuning your graphics. Here, the `xaxs = "i"` and `yaxs = "i"` arguments essentially removed the buffer between the data and the axes. `par` has options to change the size of the margins, add outer margins, change colors, etc. Some of the graphical parameters that can be passed to high- and low-level plotting functions (like those in Table 2.1) can also be passed `par`. Check out the help file (`?par`) to see everything it can do. If you want to start over with fresh `par` settings, start a new device.

2.7 New Temporary Devices

If you are using RStudio, then likely all of the plots you have made thus far have shown up in the lower right-hand corner or your RStudio window. You have been using RStudio's built-in plotting device. If you wish to open a new plotting device (maybe to put it on a separate monitor), you can use the following commands, depending on your operating system:

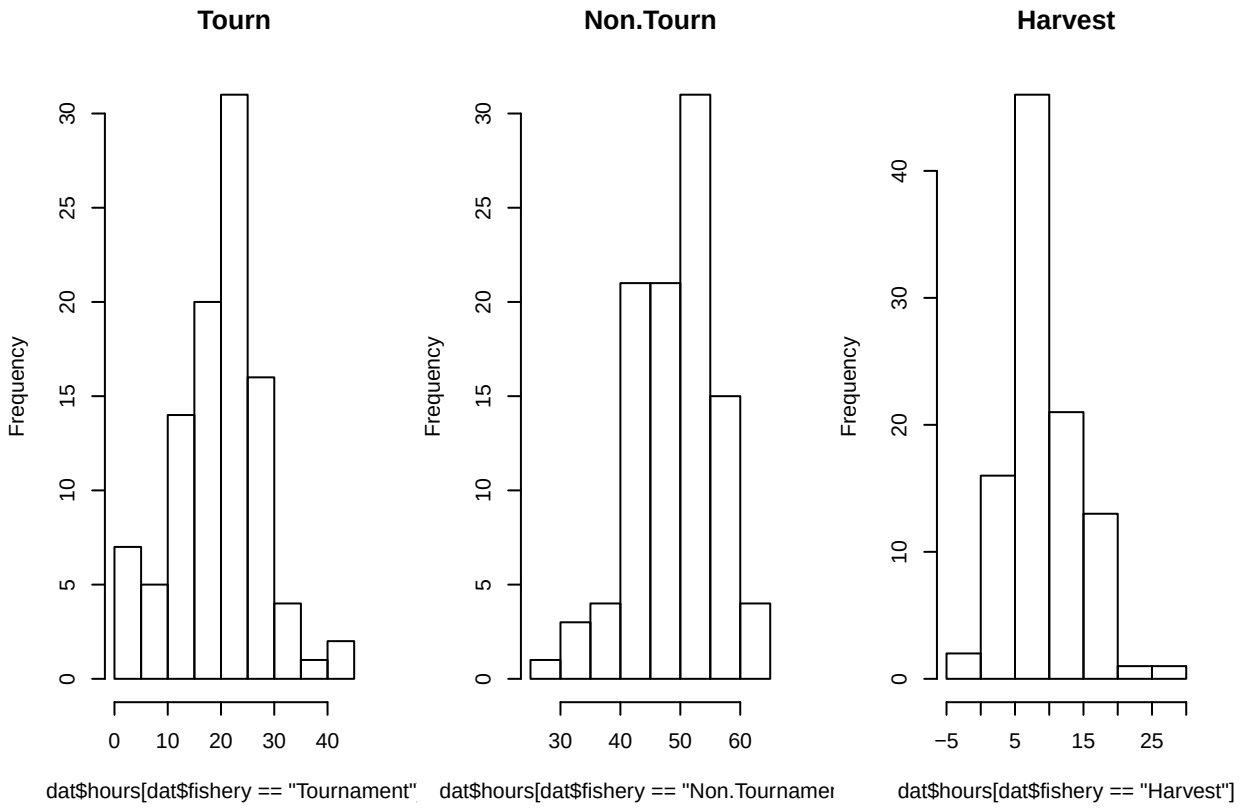
- **Windows Users** – just run `windows()` to open up a new plotting device. It will become the active device.
- **Mac Users** – similarly, you can run `quartz()` to open a new device.
- **Linux Users** – similarly, just run `x11()`.

2.8 Multi-panel Plots

Sometimes you want to display more than one plot at a time. You can make a multi-panel plot which allows for multiple plotting regions to show up simultaneously within the same plotting device. First, you need to change the layout of the plotting region. The easiest way to set up the device for multi-panel plotting is by using the `mfrow` argument in the `par` function.

Below, the code says "set up change the graphical parameters so that there is 1 row and 3 columns of plotting regions within the device". Every time you make a new plot, it will go in the next available plotting region. Make 3 histograms, each that represents a different sector of the fishery:

```
par(mfrow = c(1,3))
hist(dat$hours[dat$fishery == "Tournament"], main = "Tourn")
hist(dat$hours[dat$fishery == "Non.Tournament"], main = "Non.Tourn")
hist(dat$hours[dat$fishery == "Harvest"], main = "Harvest")
```



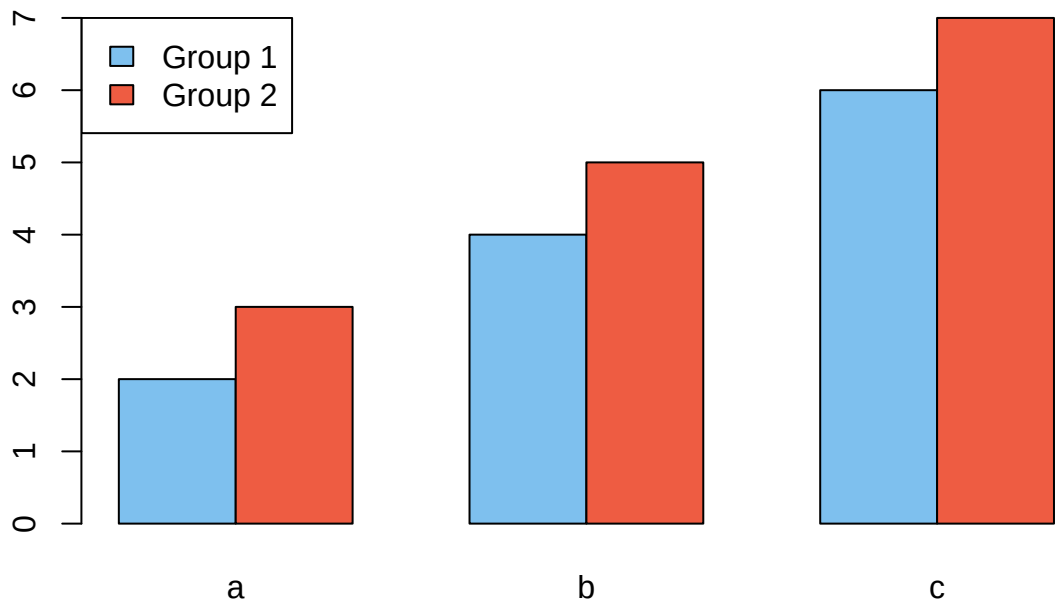
There are other ways to make multi-panel plots, however, they are beyond the scope of this beginner's workshop. See `?layout` for details. With this function you can change the size of certain plots and make them have different shapes (i.e., some squares, some rectangles, etc.), but it takes some pretty involved (though not impossible, by any means) specification of how you want the device to be split up into regions.

2.9 Legends

Oftentimes you will want to add a legend to our plots to help people interpret what it is showing. You can add legends to R plots using the low-level plotting function `legend`. Add a legend to the bar plot you made

earlier with two groups. First, re-make the plot by running the high-level `barplot` function, but change the colors of the bars to be shades of blue and red. Once you have the plot made, add the legend:

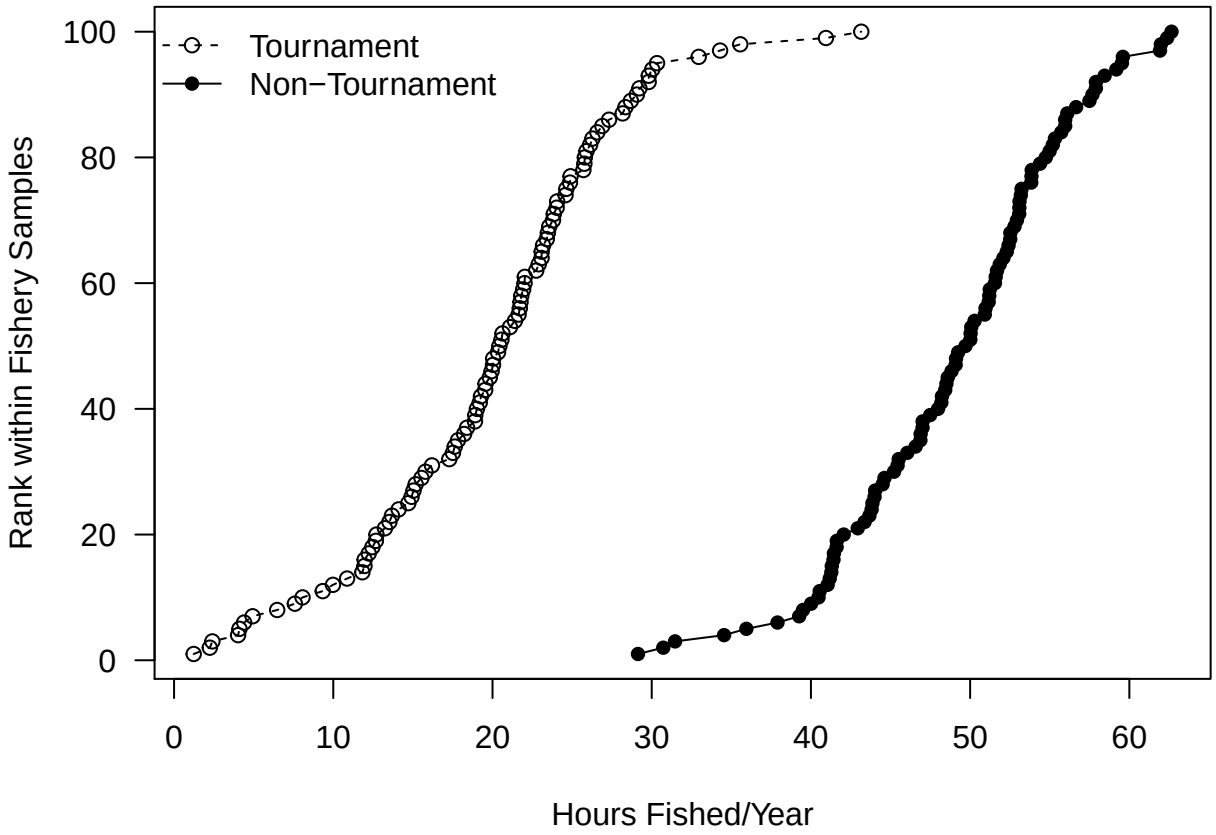
```
barplot(x3, beside = T,names.arg = c("a", "b", "c"), col = c("skyblue2", "tomato2"))
legend("topleft", legend = c("Group 1", "Group 2"), fill = c("skyblue2", "tomato2"))
```



The box can be removed using the `bty = "n"` argument and the size can be changed using `cex`. The position can be specified either with words (like above) or by using x-y coordinates.

Here is a more complex example:

```
# 1) extract and sort the hours for two fisheries from fewest hours
t_hrs = sort(dat$hours[dat$fishery == "Tournament"])
n_hrs = sort(dat$hours[dat$fishery == "Non.Tournament"])

# 2) make the plot: plot for t_hrs only, but ensure xlim covers both groups
par(mar = c(4,4,1,1))  # set the margins: 4 lines of margin space on bottom and left, 1 on top and righ
plot(x = t_hrs, y = 1:length(t_hrs),
     type = "o", lty = 2, xlim = range(c(t_hrs, n_hrs)),
     xlab = "Hours Fished/Year", ylab = "Rank within Fishery Samples",
     las = 1)  # las = 1 says "turn the y-axis tick labels to be horizontal"

# 3) add info for the other fishery
points(x = n_hrs, y = 1:length(n_hrs), type = "o", lty = 1, pch = 16)

# 4) add the legend
legend("topleft", legend = c("Tournament", "Non-Tournament"),
       lty = c(2,1), pch = c(1, 16), bty = "n")
```



Notice that you need to be careful about the order of how you specify which `lty` and `pch` settings match up with the elements of the `legend` argument. In the `plot code`, you specified that the `lty = 2` but didn't specify what `pch` should be (it defaults to `1`). So when you put the "Tournament" group first the the `legend` argument vector, you must be sure to use the corresponding plotting codes. The first element of the `lty` argument matches up with the first element of `legend`, and so on. There are a few other plotting tricks used: changing the margins using `par(mar)` and the rotation of y-axis tick mark labels using `las = 1`.

2.10 Exporting Plots

There are two main ways to save plots. First is a quick-and-dirty method that saves the plots, but they are not high-resolution. The second method produces cleaner-looking high-resolution plots.

2.11 First Method (Click Save)

- If your plot is in the RStudio built-in graphics device: Right above the plot, click *Export* > *Save as Image*. Change the name, dimensions and file type.
- If your plot is in a plotting device window (opened with `windows()` or `quartz()`: Simply go to *File* > *Save*.

All plots will be saved in the working directory by default. You can also just copy the plot to your clipboard (File > Copy to the clipboard > bitmap) and paste it where you want.

2.12 Second Method (Use a function to dump in a new file)

If you are producing plots for a final report or publication, you want the output to be as clean-looking as possible. You can save high-resolution plots through R by using the following steps:

```
# step 1: Make a pixels per inch object
ppi = 600

# step 2: Call the figure file creation function
png("TestFigure.png", h = 8 * ppi, w = 8 * ppi, res = ppi)

# step 3: Run the plot
# put all of your plotting code here (without windows())

# step 4: Close the device
dev.off()
```

A plot will be saved in your working directory containing the plot made by the code in step 3 above. The `ppi` object is pixels-per-inch. When you specify `h = 8 * ppi`, you are saying "make a plot with height equal to 8 inches". There are similar functions to make PDFs, tiff files, jpegs, etc.

2.13 Bonus Topic: Error Bars

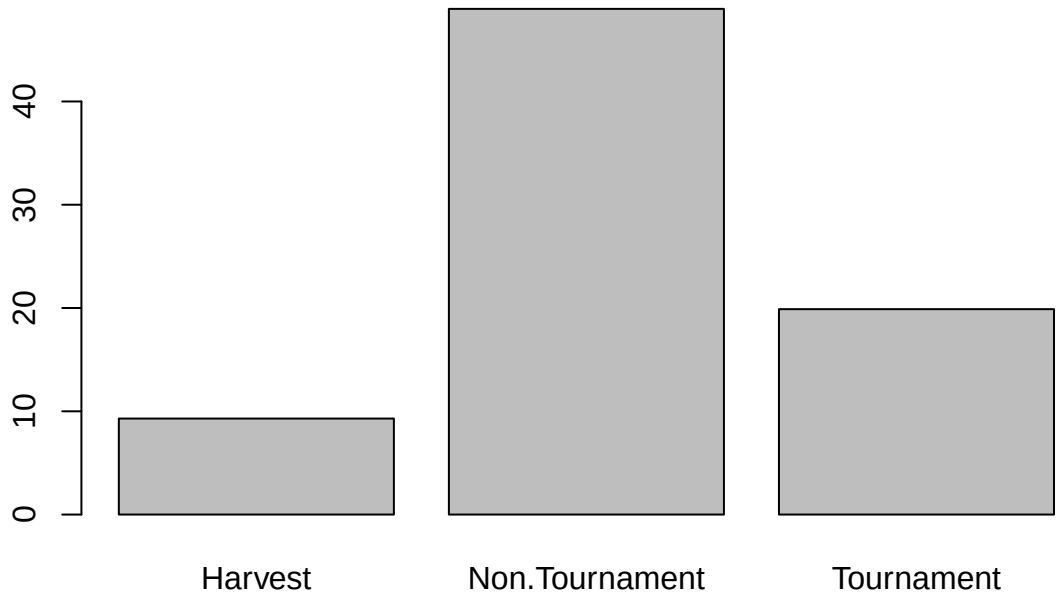
Rarely do we ever present estimates without some measure of uncertainty. The most common way for visualizing the uncertainty in an estimate is by using error bars. Error bars can be added to a plot using the `arrows` function. For `arrows`, you need:

- Vectors of the `x` and `y` coordinates of the lower bound of the error bar
- Vectors of the `x` and `y` coordinates of the upper bound of the error bar

The syntax for `arrows` is as follows: `arrows(x0, y0, x1, y1, ...)`, where `x0` and `y0` are the coordinates you are drawing "from" (e.g., lower limits) and the `x1` and `y1` are the coordinates you are drawing "to" (e.g., upper limits). The `...` represents other arguments to change how the error bars look. Let's use a simple example. Calculate the mean and standard deviations of the different fishery sectors (remember how `tapply` works?):

Create error bars on the following graph:

```
x_bar = tapply(dat$hours, dat$fishery, mean)
barplot(x_bar)
```



where the error bars represent 95% confidence intervals on the mean. You can create a 95% confidence interval using this basic formula:

$$\bar{x} \pm 1.96 * SE(\bar{x})$$

where

$$\bar{x} = \frac{1}{n}\sum_{i}^{n} x_i$$

and

$$SE(\bar{x}) = \sqrt{\frac{\sum_i^n (x_i - \bar{x})^2}{n-1}}$$

Begin by creating a function to calculate the standard error ($SE(\bar{x})$):

```
calc_se = function(x) {
  sqrt(sum((x - mean(x))^2)/(length(x) - 1))
}
```

Then calculate the standard errors for each fishery type:

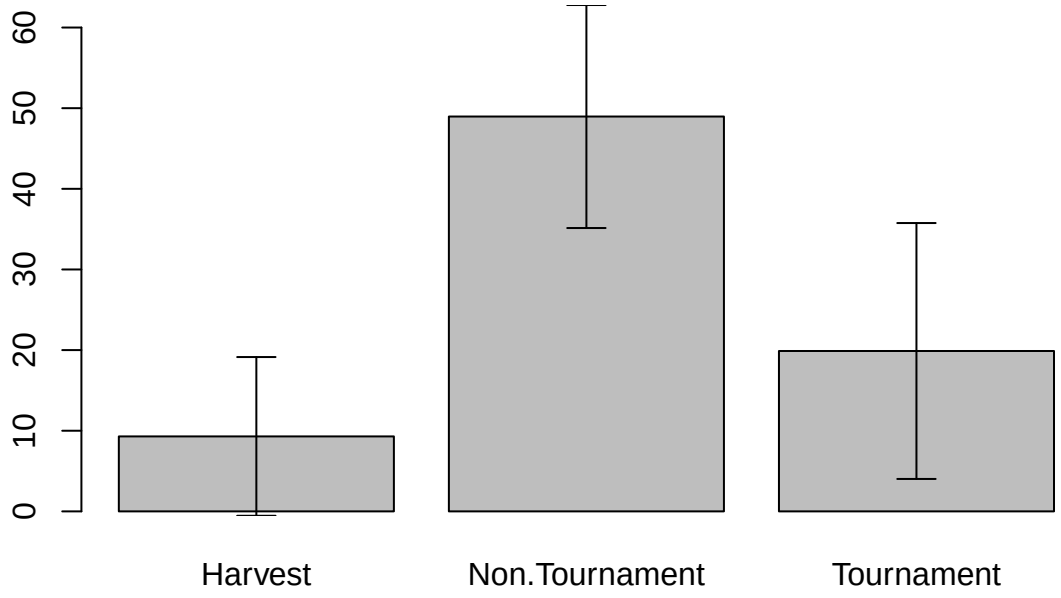
```
se = tapply(dat$hours, dat$fishery, calc_se)
```

Then calculate the lower and upper limits of your bars:

```
lwr = x_bar - 1.96 * se
upr = x_bar + 1.96 * se
```

Then draw them on using the arrows function:

```
mp = barplot(x_bar, ylim = range(c(lwr, upr)))
arrows(x0 = mp, y0 = lwr, x1 = mp, y1 = upr, length = 0.1, angle = 90, code = 3)
```



Notice four things:

- The use of `mp` to specify the `x` coordinate. If you do `mp = barplot(...)`, `mp` will contain the `x` coordinates of the midpoint of each bar.
- `x0` and `x1` are the same: you wish to have vertical bars, so these must be the same while `y1` and `y2` differ.
- The use of `ylim = range(c(lwr, upr))`: you want the y-axis to show the full range of all the error bars.
- The three arguments at the end of `arrows`:
  - `length = 0.1`: the length of the arrow heads, fiddle with this until you like it.
  - `angle = 90`: the angle of the arrow heads, you want 90 here for the error bars.
  - `code = 3`: indicates that arrow heads should be drawn on both ends of the arrow.

EXERCISE 2

For this exercise, you will be making a few of plots and changing how they look to suit your taste. You will use a real dataset this time from a sockeye salmon (*Oncorhynchus nerka*) population from the Columbia/Snake River system. This population spawns in Redfish Lake in Idaho, which feeds into the Salmon River which is a tributary of the Snake River. In order to reach the lake, the sockeye salmon must successfully pass through a total eight dams that have fish passage mechanisms in place. The Redfish Lake population is one of the most endangered sockeye populations in the U.S. and travels farther (1,448 km), higher (1,996 m), and is the southernmost population of all sockeye populations in the world (Kline and Flagg 2014). Given this uniqueness, a captive breeding program was initiated in 1991 to preserve the genes from this population. These data came from both hatchery-raised and wild fish and include average female spawner weight (g), fecundity (number of eggs), egg size (eggs/g), and % survival to the eyed-egg stage.

1. Create a new R script called `Ex2.R` and save it in the `Chapter2` directory. Download the `sockeye.csv` data set from GitHub and read it into R. Produce a basic summary of the data and take note of the data classes, missing values (`NA`), and the relative ranges for each variable.
2. Make a histogram of fish weights for only hatchery-origin fish. Set `breaks = 10` so you can see the distribution more clearly.
3. Make a scatter plot of the fecundity of females as a function of their body weight for wild fish only. Use whichever plotting character (`pch`) and color (`col`) you wish. Change the main title and axes labels to reflect what they mean. Change the x-axis limits to be 600 to 3000 and the y-axis limits to be 0 to 3500. (*Hint: The `NA`s will not cause a problem. R will only use points where there is data for both `x` and `y` and ignore otherwise*).
4. Add points that do the same thing but for hatchery fish. Use a different plotting character and a different color.
5. Add a legend to the plot to differentiate between the two types of fish.
6. Make a multi-panel plot in a new window with box-and-whisker plots that compare (1) spawner weight, (2) fecundity, and (3) egg size between hatchery and wild fish. (*Hint: each comparison will be on its own panel*). Change the titles of each plot to reflect what you are comparing.
7. Save the plot as a .png file in your working directory with a file name of your choosing.

2.13.1 EXERCISE 2 BONUS

1. Make a bar plot comparing the mean survival to eyed-egg stage for each type of fish (hatchery and wild). Add error bars that represent +/- 2SE of each mean.
2. Change the names of each bar, the main plot title, and the y-axis title.

Reference for data Kline, P.A. and T.A. Flagg. 2014. Putting the red back in Redfish Lake, 20 years of progress toward saving the Pacific Northwest's most endangered salmon population. **Fisheries**. *39(11)*: 488-500.

Chapter 3

Basic Statistics

Chapter Overview

In this chapter, you will get familiar with the basics of using R for statisitical analysis.

- the different parts of basic statistical models
- how to fit and interpret the output from various linear models:
  - simple linear regression models
  - multiple regression models
  - higher order polynomial regression models
  - T-tests (also ANOVA)
  - ANCOVA models
  - Interactions
- very basic model selection
- Bonus topic: fitting non-linear regression models using nls()
- Bonus topic: fitting custom maximum-likelihood models using `optim`

R's built-in statistical modeling framework is pretty intuitive and comprehensive.

IMPORTANT NOTE: If you did not attend the session corresponding to the Chapter 1 material or Chapter 2, you are recommended to walk through those chapters' contents before proceeding to this material. This is because you will be using some of the commands and terminology presented previously. If you are confused about a topic, you can use **CTRL + F** to find previous cases where that topic has been discussed in this document.

Before You Begin

You should create a new directory and R script for your work in this Chapter. Create a new R script called `Ch3.R` and save it in the directory `C:/Users/YOU/Documents/R-Workshop/Chapter3`. Set your working directory to that location. Revisit the Sections 1.2 and 1.3 for more details on these steps.

Chapter 4

Simulation and Randomization

Chapter 5

Large Data Manipulation