UMass PBS Instructors R Manual

Andrea Cataldo & Will Hopper

Jun 13, 2019

Contents

Preface

This book serves as a guide for instructors teaching statistics and research methods with R in the UMass Amherst Psychological & Brain Sciences Department. The book is organized into three main sections:

1. An introduction to the R language and the RStudio development environment.
2. A reference for performing and interpreting common statistical tests in R.
3. Recommended best practices for instructors.

Section 1 is primarily intended for instructors who are unfamiliar with the R language and its ecosystem. However, we recommend even experienced R users read it, as it is useful to remind yourself of the perspective the majority of your students will be approaching R from, and to get "on the same page" as your fellow instructors.

Much of the information in this guide can be found in any "Introduction to R" book or resource. So, why write another one? There are two main reasons:

1. To provide you with the information you need specifically for Psych 240 and 241, and none of the information you don't.
2. To provide opinionated guidance and reference material for the instructor specifically, not just the "average" R user.

We hope that you find this guide useful as you embark on teaching a new class!

Getting Started

To follow along with the examples in this book, it is recommended that you install the latest version of R and RStudio to your personal computer.

The R Language

R is a complete programming language and computing environment designed to enable statistical modeling, data manipulation, and reporting. Despite its focus on statistics, it is still very feature rich - we can use it to perform mathematical calculations, create data visualizations, connect to remote databases, save results as files on your computer's hard drive, and more. In fact, this entire manual was created in R!

To get started, download and install the most current version of R for your operating system:

- Windows: https://cran.r-project.org/bin/windows/base/
- Mac OS X: https://cran.r-project.org/bin/macosx/
- Linux: https://cran.r-project.org/bin/Linux/ (as always, see instructions for your specific Linux distribution)

RStudio

RStudio is a graphical program that simplifies common R-related tasks, organizes information about your current R session, and generally makes it easier to use R. This type of program is called an **I**ntegrated **D**evelopment **E**nvironment, or IDE for short.

We **strongly** recommend using R within RStudio. However, it is important to understand that they are separate and distinct programs, and R can freely be used outside of RStudio.

To help your students understand their relationship, you could offer the following analogy: RStudio is like a workbench and toolbox, where R is like a hammer. You do not *need* a workbench and toolbox to build something with your hammer - but having a workbench to rest your project on while you work, and a toolbox where all your nails are organized is very convenient, and will probably help you do your work faster and more accurately.

The company which produces the RStudio software has several different products with "RStudio" in the name. You and your students will want to install the "RStudio Desktop - Open Source License" version (i.e., the free version):

- RStudio Desktop for Windows: http://rstudio.org/download/latest/stable/desktop/ windows/RStudio-latest.exe
- RStudio Desktop for Mac OS X: http://rstudio.org/download/latest/stable/desktop/ mac/RStudio-latest.dmg
- RStudio Desktop for Linux: Get .deb and .rpm packages here

Formatting Conventions

R source code (i.e., code run in the R console or from an R script) and output are presented in monospace font in regions with a light gray background. We **do not** include command line prompts (> and +, like you would see in a real R console) in the R source code. This is to allow you to conveniently copy and run the code without having to delete the leading prompt symbols. All text output from executing R commands is denoted with two preceeding hashes (##). So, any time you see ## following a block of R code, you are looking at the output of the preeceeding command.

Additional Materials

As alluded to earlier, this is far from the only "Intro to R" material in existence, though this guide has been developed with PSYCH 240 and 241 specifically in mind. But we do encourage you to supplement your knowlege with outside sources. We recommend the following resources to learning about R programming:

- R in a Nutshell by Joseph Adler
- The Art of R Programming by Norman Matloff
- R for Data Science by Garrett Grolemund & Hadley Wickham
- Advanced R by Hadley Wickham

On campus, there are two great consulting resources for getting help with statistical analysis in R:

- The Institute for Social Science Research
- The Center for Research on Famlies

Reproducibility

The R session information when compiling this book is shown below:

```r
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.5.1  magrittr_1.5    bookdown_0.11   htmltools_0.3.6
##  [5] tools_3.5.1     yaml_2.2.0      Rcpp_1.0.1      codetools_0.2-15
##  [9] stringi_1.4.3   rmarkdown_1.13  knitr_1.22      stringr_1.3.1
## [13] xfun_0.6        digest_0.6.19   packrat_0.4.9-3 evaluate_0.14
```

Chapter 1

Introducing R and RStudio
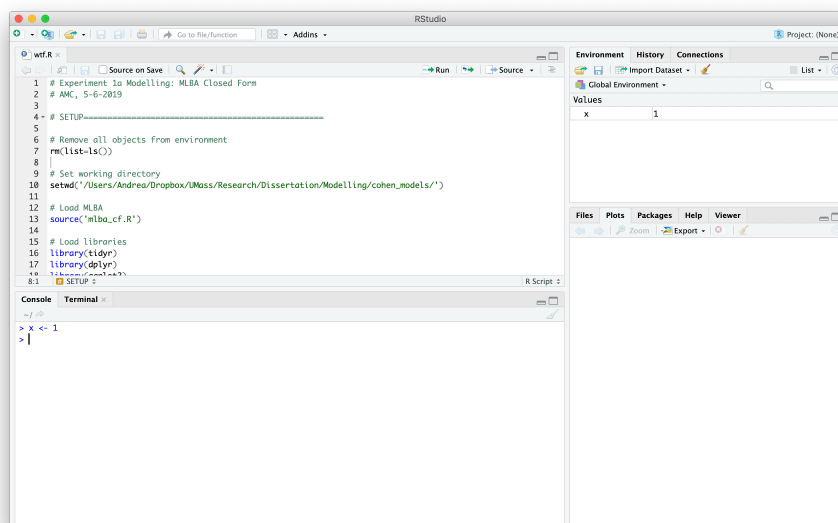
1.1   Pane #1: The R Console

The R console allows you to execute R code (often called expressions, or commands) and see the results printed out. To execute R commands, place your cursor at the prompt (the > symbol), type in your code, and press Enter.

A simple and easy way to demonstrate using the R console is to perform some basic arithmetic, just like a calculator. R has 5 basic arithmetic operators:

- + for addition
- − for subtraction
- ∗ for multiplication
- / for division
- ˆ for exponentiation

```
2 + 2
## [1] 4
```

R respects the order of operations (i.e. PEMDAS) by default, so if you want for a specific operation to be executed fist, you need to surround it with parentheses. To see how grouping with parentheses affects arithmetic operations, compare the following two examples:

```
2*10 + 3/10
## [1] 20.3
2*(10 + 3)/10
## [1] 2.6
```

The R console is the best place to start immersing yourself in the language by experimentation. It allows you to "code as you go"; run one command, see what you get, adjust it, and test it again. However, the interactive nature of using the R console makes it a poor choice for saving your work to use again later, and for complicated, multi-step operations.

When you know that you will need to repeat the same step again in the future (i.e., reuse code), or you have a task that requires intermediate steps, you should organize all your code into an R script. RStudio allows you to easily write and interactively test out your R scripts in the **Editor Pane**, which is introduced below.

1.2 Pane #2: The Editor

The editor pane is where you can create, modify, and save plain text documents, and it is designed to help you create and execute R scripts. An R script is just a text file that contains valid R code (the same kind of commands you would enter into the console), and commonly carries the .R file extension. You can think of an R script as a stand-alone computer program, but instead of double-clicking to run it, you run it via the R interpreter.

You can create a new, blank R script by going to the File Menu ➡ New File ➡ R Script.

RStudio provides several methods for executing the code you've written. You can:

- Run a single line of code by placing your cursor on that line and pressing Ctrl + Enter (⌘ Command + Enter on Mac).
- Run multiple lines of code by selecting all the lines you want to run with your cursor and pressing Ctrl + Enter (⌘ Command + Enter on Mac).

- Run the entire R script by pressing the "Source" button in the top-right corner of the editor pane, or using the Ctrl + Shift + S keyboard shortcut (⌘ Command + Shift + S on Mac)

No matter which method you use, the code you choose to run will be executed in the R console below, and you will see any results printed out there as well. Keeping your code in an R script allows you to quickly and easily run many commands in a row, without having to type each one in every time you need to do it.

Another reason to write your code in an R script is to keep a short explanation of what you are doing (and why!) together with your code. These short explanations are called *comments*. In R, you write a comment by prefixing the comment's text with the pound sign #. Comments can go on their own line, or at the end of an R expression.

A Word Of Advice

When new programmers keen to improve their skills hear the advice "keep your code in an R script, not the console", they often take it quite literally. They proceed to take all the R code they have run in the console, copy it to an R script, save it, and run it all again later. Best practices achieved, right?

However, this scenario is often a dicey over-extension of the suggested rule. New programmers can be prone to keep *literally everything* they've run, even commands that are syntactically incorrect and give errors or incorrect results. An R script should not be a historical record - rather, it should evolve through editing and careful consideration, much the same way an essay does.

So, when saving your code in an R script, try keeping the following rule of thumb: Your R script should have all the commands you need, and none of the ones you don't. This can be challenging for new programmers - the ability to identify what is necessary and what isn't develops with time and expertise. So, don't expect to be perfect at this in the beginning, but do expect to edit and re-edit your R scripts, and don't treat your "Chapter 2 HW.R" file as your diary of working on your Chapter 2 homework.

```r
# Hi I am a comment.
# The R interprer ignores me!
2*10 + 3/10 # comments can go here too!
## [1] 20.3
```

1.3   Pane #3: Your R Session

### 1.3.1   Variables and the Environment

As your R scripts grow beyond adding and subtracting a few numbers, you will often want to save the results of your computations (like datasets, or the results of statistical

models) to use multiple times, or to increase the clarity of your code. In R, you can save a value for later use by assigning it to a *variable* name.

You can create a new variable using the assignment operator `<-`, with the syntax `name <- value` where `name` is a syntactically valid name and `value` is the value you wish to assign. To demonstrate creating a variable, consider the code below, where I create a variable named `x`, assign it a value of `1`, and then print out its value by typing `x` into the console and pressing Enter.

```
x <- 1
x
## [1] 1
```

If you execute the first command `x <- 1` in your own console, you should see it appear in the "Environment" pane of the RStudio window. The environment pane is helpful for keeping track of all the variables you've created. If you ever want to clear your

Environment (i.e., delete all the variables you've created), you can press the 🧹 button.

Variable names are mostly arbitrary (we could have used `elephant` or `jumpluff` instead of `x`), but there are some restrictions on variable names. The most important restrictions are:

1. Spaces are not allowed.
2. The name cannot begin with a number.

See the "Details" section of the make.names help page, and the list of reserved keywords for a complete set of naming rules.

A Word of Advice

When you exit RStudio, you will get a pop-up asking you if you want to "Save workspace image…" to some location on your hard drive. Any person's natural intuition is to answer "Yes" - what fool doesn't want to save their work? However, saving your "workspace image" is not the same thing as properly saving your work.

Saving your **workspace** means saving the state of your R session (i.e., all the variables you've created, packages you've loaded, etc.) so that everything can be reloaded the next time you start RStudio. Which sounds great, until you get a month into the semester and find that you've been saving and loading an entire month's worth of R sessions. You've run out of variable names and have resorted to naming things "xxxxxx" to avoid naming collisions. Or you forget to avoid naming collisions and obliterate that one variable you actually *do* care about. Regardless, RStudio now takes 2 minutes to start and shut down because you've created a behemoth.

In short, saving your workspace sounds great, but can lead to some very bad practices. It encourages you to let RStudio keep track of your past work instead of writing R scripts, and you end up storing 10 pieces of junk for every one thing you actually need.

We **STRONGLY** advise you to disable this feature, and teach your students to disable it

as well. To disable this feature, navigate to Tools ➡ Menu ➡ Global ➡ Options ➡ Basic.
On this tab, set the "Save workspace to .RData on exit" option to "Never" using the drop
down menu, as shown below. Your future self thanks you.
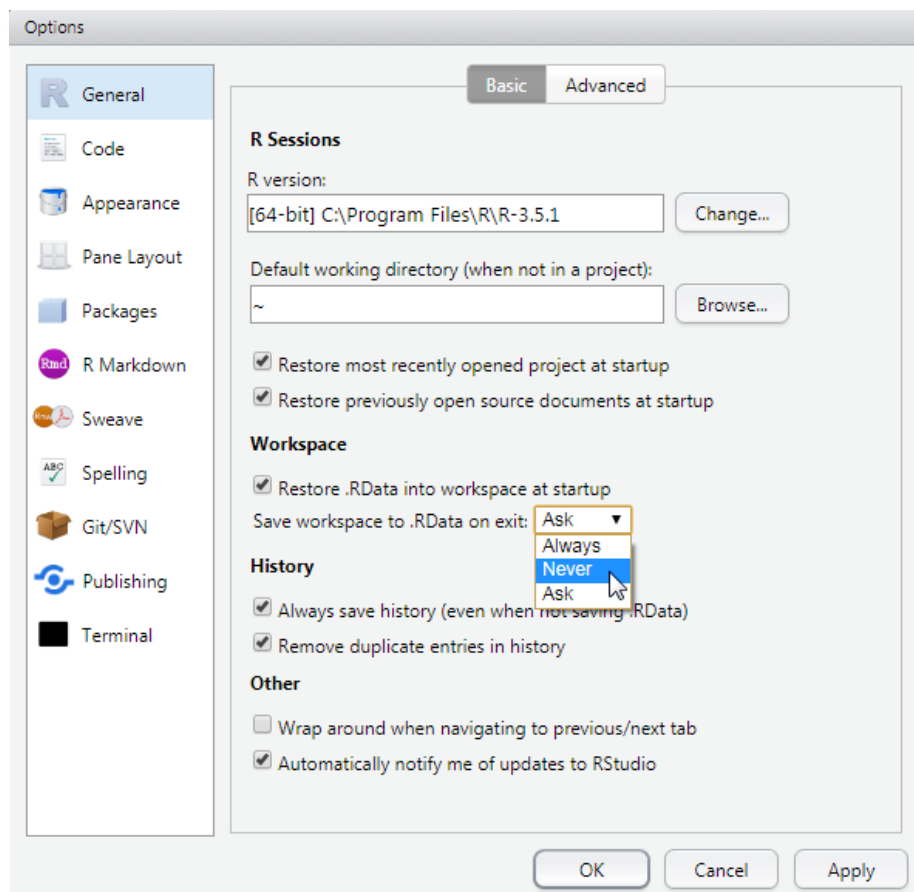
1.3.2 History

RStudio also keeps track of the R commands that you've run in the past. This is useful
during experimentation and testing, because it allows you to quickly copy a previous
command, modify it, and try it again. It can also be helpful when you find that you've
discarded code because you thought it wasn't useful or necessary, but later realize that
it was, and need to remind yourself what you did.

The best way to access the history of your most recent commands is place your cursor in
the console, and use the up and down arrow keys. Pressing the ⬆ arrow key will take you
through older commands, and press the ⬇ arrow key will take you back to more recent
commands. So, next time you need to edit the command you just gave, use the arrow
keys to retrieve it instead of typing it out again!

To access commands more than say, 10, commands in the past, you can use the "History" tab in the top-right pane of the RStudio window. This allows you to examine your history like scrolling through a file, or find commands matching key words, like using a search engine. Remember, the **best** way to remember what you've done is to record it in an R script to use again later, but the "History" tab provides a useful backstop for those moments when you "slip up".

1.4 Pane #4: Miscellaneous

Don't let the name of this section mislead you - this pane is just as important and useful as the others. It provides a window into three of the most important things you need when using R: plots, packages, and help!

1.4.1 Plots

This tab might be the killer feature of R Studio. As you might imagine, this is where figures and plots that you create are displayed. R has many simple yet powerful visualization tools, but historically, actually getting those plots to show up somewhere has involved annoying, platform-specific boilerplate code. With the R Studio plot tab, you can forget about these details 95% of the time, and your plots just show up automatically.

1.4.2 Packages

The so-called "base" installation of R that you downloaded from cran.r-project.org has tons of useful tools for statistical modeling, data manipulation, and visualization built-in, but it is far from exhaustive, and new things are being invented all the time! Thankfully, R has a fantastic extension mechanism called "packages". R packages provide sets of additional commands you can easily install, load into R, and then use for yourself.

Where can you find these packages, and how can you install them yourself? The gold-standard for R packages is the [CRAN repository](#), which stands for the **C**omprehensive **R A**rchive **N**etwork. You can install packages from this repository using the `install.packages()` command. For example, if I wanted to install the `dplyr` package (a popular packages which provides opinionated tools for data manipulation), I would execute the following command in the console:

```
install.packages("dplyr")
```

If this package has any dependencies (i.e., other packages it needs installed in order to function), R will download and install these dependencies automatically. Once the one-time installation process finishes, you can load it into your R session with the `library` command, like so:

```
library("dplyr")
```

Remember, *installing* a package is a one-time operation, but you do need to *load* the package with `library` each time you start R.

The "Packages" pane provides a way to see what packages you have installed, and which ones are loaded (loaded packages have a check-mark in front of them). You can also use this pane to install new packages (instead of using the `install.packages` command), bulk-update all your installed packages, and remove an installed package you no longer want to keep.

1.4.3 Help

The help pane provides a way for you to access documentation explaining how to use different R functions. You can search all the help pages from the search bar on the "Help" pane. If you know the name of the function you want to view the help documentation for, you can type the name of the function in the console preceded by a question mark, and hit Enter. For example, if I wanted to see more information on how to use the `install.packages` function, I could type:

```
?install.packages
```

...then press kbd>Enter, and the help page would automatically open.

1.5 Conclusion

Here, we have given an introduction to some of the R language's basic functionality (the interactive console, arithmetic, variables, plots and packages) by using the layout of the RStudio IDE as a guide. In the following sections, we demonstrate the basic tenets of the R language and basic R programming tasks, such as calling functions, creating and manipulating data structures, and importing data from files on your hard drive into R.

Reproducibility

The R session information when compiling this book is shown below:

```
sessionInfo()
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
```

```
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.5.1  magrittr_1.5    bookdown_0.11    htmltools_0.3.6
##  [5] tools_3.5.1     Rcpp_1.0.1      codetools_0.2-15 stringi_1.4.3
##  [9] rmarkdown_1.13  knitr_1.22      stringr_1.3.1    xfun_0.6
## [13] digest_0.6.19   packrat_0.4.9-3 evaluate_0.14
```

Chapter 2

R Programming Fundamentals

In this section, we outline the basic concepts of the R language (data classes, data structures, operators and functions), and demonstrate how to perform some common tasks.

2.1   Classes of Data

The data you work with in R belong to one of several possible classes. The distinction between classes of data is crucial, because the class you instantiate your data with determines the set of possible operations that you can perform on that data.

R has 6 "atomic" classes of data, but you will only encounter 4 of these in PSYCH 240 and 241. These classes are: Numeric, Integer, Logical, and Character. They are dubbed "atomic" classes because they are the fundamental units of the language (i.e., you cannot access a more basic type of object than these 6 classes) and they provide the building blocks for more complex classes and data structures. The 4 frequently used classes are described in more detail below.

2.1.1   The Numeric Class

The numeric class is used to represent real numbers (i.e., numbers with a fractional component), for example, 6.23. The numeric class is also known as the "double" class, because internally it utilizes double floating point precision. The numeric class is the default class used to represent numbers in R. So, if you were to execute the command 2 + 2 in the R console, R would represent those 2's using the numeric class.

17

Special Values

R has several special numeric values. `Inf` and `-Inf` are used to represent positive and negative Infinity, respectively. `NaN` is used when the results of a numeric computation produce "Not A Number", such as when multiplying a number produces a value too small for the computer to represent. `NaN` is often abused to represent missing values, but the special value NA should be preferred.

2.1.2   The Integer Class

Just as in mathematics, the integer class can only represent whole numbers, e.g., 6. However, if you type 6 into the R console, R will not use the integer class to represent the 6. As mentioned just above, R will use the numeric data type to represent the 6. If you want to *force* R to use the integer class, you must terminate the number with an upper-case L, e.g., 2L. In practice, it usually makes little difference whether you use the integer class or the numeric class to represent whole numbers, so sticking with the default class of numeric is safe to do.

2.1.3   The Logical Class

Logical data can only take on 2 possible values: TRUE (1) or FALSE (0). This type of datum is used to represent whether some state exists (is true) or does not exist (is false). TRUE and FALSE **must be uppercase**. They *can* can be abbreviated as T and F, but it is not recommended because while the names T and F can be over-written with non-logical values, `TRUE` and `FALSE` are reserved names that can never be overwritten. Such a restrictive class of data may not seem useful, but it plays an important role in data manipulation, as it can be used to indicate the presence (or absence) of other data values you are interested in.

You can change a `TRUE` to a `FALSE` (or a `FALSE` to a `TRUE`) by prefixing them with the `!` negation operator.

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

The `!` operator isn't useful when you are typing out literal `TRUE` and `FALSE` values, but is useful in situations where you have a function which produces one type of logical, and but you're actually interested in its opposite. We will look at such a case later in the logical indexing section.

Special Values

R uses the special value $\texttt{NA}$ to denote when a value is missing. $\texttt{NA}$ is *technically* a logical value, but can be used alongside other atomic classes of data without issue.

2.1.4 The Character Class

The character data type (sometimes referred to as the "string" data type) is used for representing textual data.

To encode a string of text as character data in R, it must be wrapped in quotes. Both " " and ' ' are acceptable, as long as the same character is used to open and close the string (i.e., you can't open with a single quote and close with a double quote).

```
a <- "foobar"
a
## [1] "foobar"
```

Without the quotes, R will interpret the text as the name of an R variable, and attempt to locate that variable. As demonstrated below, missing quotes is a common source of "object not found" errors.

```
a <- foobar
```

```
## Error in eval(expr, envir, enclos): object 'foobar' not found
```

Sometimes, a value can be represented using *either* a Character or Numeric class. For example, 4.2 could be represented as a numeric by typing 4.2 or a character by typing "4.2". While these look similar, and print similarly, they behave *very* differently. Consider the following two operations.

```
4.2 + 1
## [1] 5.2
"4.2" + 1
```

```
## Error in "4.2" + 1: non-numeric argument to binary operator
```

This illustrates the earlier point that different classes of data support different operations. Here, we can see that you cannot perform mathematical operations on Character data, even if the data **could** be interpreted numerically.

2.1.5 Factors

Factors are not an atomic data class, but it is nearly impossible to use the R language and avoid encountering data with the Factor class. Factors are a hybrid class of data, in which values are printed out with character labels, but are represented internally with integers. Importantly, when a Factor variable is created, the range of possible new values it can take on is restricted - only values that belong to the set of initial values can be inserted

or appended. Factors exist in order to have a data class that can represent categorical variables in statistical models such as ANOVA.

The reason Factors are unavoidable is because many functions in R *automatically* convert Character data into Factors. Two notorious culprits are the `data.frame` function, and the `read.csv` function (both of which we will encounter later). New programmers often use these functions unaware of the implicit conversations they carry out behind the scenes, and are surprised later on when they find that they cannot perform some operation (like appending new Character values to ones they already have). Luckily, the conversion-happy behavior of these functions can be stopped by changing one of R's global options, `stringsAsFactors`. To turn off this conversion, execute the command `options(stringsAsFactors = FALSE)` in your R Script or console. We **STRONGLY** recommend disabling this conversion.

2.1.6   Class detection and conversion

Speaking of conversion, programmers commonly face the need to convert data of one class into data of another class. To determine which class of data you have, you can use the `class` function. For each atomic class, R provide a class detection function (starting with the prefix `is.`) that provides a TRUE or FALSE response, and an associated class conversion function (starting with the prefix `as.`). Consider the following examples:

```r
x <- 1
class(x)
## [1] "numeric"
is.numeric(x)
## [1] TRUE
is.integer(x)
## [1] FALSE
x <- as.integer(x)
class(x)
## [1] "integer"
is.integer(x)
## [1] TRUE
x <- as.logical(x) # 1 converts to TRUE
x
## [1] TRUE
x <- as.character(x) # TRUE converts to "TRUE"
x
## [1] "TRUE"
is.character(x)
## [1] TRUE
```

Class conversion is not always possible (e.g., "Z" cannot be converted into a number) or without loss of information (e.g., a numeric like 6.23 cannot be converted to an integer without losing the .23).

2.2 Data Structures

In order to introduce variables and classes, we have restricted our examples to using single, scalar values, like 6 and `"foobar"`. Of course, when analyzing data, it is necessary to group multiple values together. R provides several different types of **data structures** for this purpose. Think of data structures in R as big containers used for grouping together many values. After storing your data in these containers, you can reuse it multiple places (e.g. create an R object to store it) or access different subsets of it by position or name.

Each type of data structure focuses on representing different classes of data, and different relationships between the values in the data structure. In PSYCH 240 and 241, you will encountered 4 types of data structures: Vectors, Matrices, Data Frames, and Lists.

2.2.1 Vectors

Vectors are one-dimensional data structures that can hold values of a single atomic class. Classes of data *cannot* be mixed within a vector. In other words, a vector can hold either Integer, Numeric, Character, or Logical values, but not any combination of those values.

Vectors, no matter what type of data they hold, can be created by using the `c()` function in R, short for concatenate. Just place each value you want to be included in the vector inside the parenthesis, separated with a comma.

The individual values held in the vector are referred to as elements, and vectors have a length equal to the number of elements it contains. You can check how many elements there are in a vector using the `length()` function:

```r
new_vector <- c(1, 10, 45, -1)
length(new_vector)
## [1] 4
char_vector <- c("foo", "bar", "herp", "derp")
length(char_vector)
## [1] 4
```

The `c()` function can also combine existing vectors into a single, larger vector""

```r
big_vec <- c(new_vector, c(1, 2, 3, 4, 5))
big_vec
## [1]   1 10 45 -1   1   2   3   4   5
```

If you need to create a vector that contains a long one-by-one numeric sequence, there is a shortcut for typing out all the values you need - the `:` operator. Put the first number in the sequence before the colon and the final number in the sequence after the colon:

```r
5:50
##  [1]   5   6   7   8   9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
## [24] 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

After you create a vector, you can give each element a name using the `names()` function and a character vector:

```
names(new_vector) <- c("A", "B", "C", "D")
new_vector
##  A  B  C  D
##  1 10 45 -1
```

Names can be useful when you want to select a subset of the values in a vector, which will come up in Section 2.5.3.

2.2.2 Matrices

Matrices in R are analogous to matrices from linear algebra, with the notable difference of being able to hold non-numeric types of data. They are a rectangular 2-D data structure, meaning that the number of elements in the matrix is be equal to the product of the number of rows and the number of columns.

- rows = dimension 1
- columns = dimension 2

Like vectors, data types may *not* be mixed in a matrix (e.g., you cannot have some elements be characters and other be numeric).

A matrix is created by feeding a vector into the `matrix()` function, and specifying either the number of rows, the number of columns, or both.

```
wide <- matrix(c(1:3, 99:101), ncol = 3)
wide
##      [,1] [,2] [,3]
## [1,]    1    3  100
## [2,]    2   99  101
long <- matrix(c(1:3, 99:101), nrow = 3)
long
##      [,1] [,2]
## [1,]    1   99
## [2,]    2  100
## [3,]    3  101
```

We recommend always specifying both the number of rows and the number of columns to avoid errors.

If you want a large matrix but with only a few unique values, take advantage of R's ability to **recycle** input:

```
matrix(c(4), nrow = 3, ncol = 10)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    4    4    4    4    4    4    4    4    4     4
```

```
## [2,]    4    4    4    4    4    4    4    4    4    4
## [3,]    4    4    4    4    4    4    4    4    4    4
```

Note the matrix is filled up by column (i.e. first element goes to row 1 column 1, second element goes to row 2, column 1, and so on):

```
matrix(c(4,0), nrow = 2, ncol = 10)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    4    4    4    4    4    4    4    4    4    4
## [2,]    0    0    0    0    0    0    0    0    0    0
```

2.2.3 Data Frames

Like matrices, Data Frames are are 2D, rectangular data structures, but are more flexible because they allow for different data types to be stored in each column. A Data Frame is usually the best way to store and work with a dataset that mixes qualitative and quantitative variables.

Data frames can be created by passing `name = value` pairs to the `data.frame()` function. The values should be vectors (of any type) and the names should be unquoted strings of text, which will be used to label each column. Importantly, all the vectors stored in the data frame must be of *identical length*.

Essentially, a data frame is a container that imposes a relational structure on a set of vectors.

```
df <- data.frame(x = c(1,4,4,2),
                 y = c(3,3,1,4),
                 month = c("Sep","Oct","Nov","Jan"),
                 stringsAsFactors = FALSE)
df
##   x y month
## 1 1 3   Sep
## 2 4 3   Oct
## 3 4 1   Nov
## 4 2 4   Jan
```

This example also shows an extra argument, `stringsAsFactors = FALSE`, that was **not** a variable to store in the data frame. This argument is another way to control how R interprets character vectors (a.k.a. strings) when forming the data frame. Here, using `stringsAsFactors = FALSE` forces R to leave your character vectors as they are when creating the data frame.

2.2.4 Lists

Lists are the most abstract and flexible data structure in R. Lists can hold any type of R object, but don't impose any relationship between them. You can have a list holding matrices, data frames, vectors, and even lists holding other lists!

Think of lists like a folder on your hard drive. You can stuff any kind of file you like in there and give it a name, but there is no relationship between them inside that folder, other than the order they are sorted in. Use a list when you need to group data structures of different sizes and types together. However, you should be aware that the lack of structured relationships between the data in different list elements can make them tricky to work with.

As you might have guessed, you can create lists of your own with the `list()` function. Like the `data.frame()` function, you pass in `name=value` pairs. But now, the values can be any R object of any size, not just vectors with equal lengths.

```
biglist <- list(first = -10:-15, second = data.frame(x=c("A","B"), y = 1:2))
biglist
## $first
## [1] -10 -11 -12 -13 -14 -15
##
## $second
##   x y
## 1 A 1
## 2 B 2
```

2.3 Operators

Creating data structures is great, but creating them is rarely ever the goal we have in mind - we always want to manipulate and perform computations using our data. Nearly every analysis you perform will make use of R's inline **operators**. We've seen these operators already when we did arithmetic in the R console. As a reminder, R has 5 basic arithmetic operators:

- `+` for addition
- `−` for subtraction
- `∗` for multiplication
- `/` for division
- `ˆ` for exponentiation

R also has two other types of operators: **Relational** operators and **Logical** operators.

2.3.1 Relational Operators

As the name implies, relational operators are used to examine the relationship between values. R has 6 relational operators:

- <: The "less than" operator
- <=: The "less than or equal to" operator
- >: The "greater than" operator
- >=: The "greater than or equal to" operator
- == The "equal to" operator
- != The "not equal to" operator

All 6 of these operators can be used sensibly on Integer and Numeric data, while only the last two can be used sensibly on Logical and Character data.

Each of these operators returns its answer in the form of a Logical value. Consider the following examples that demonstrate each operator:

```
1 < 2
## [1] TRUE
1 > 2
## [1] FALSE
2 > 2
## [1] FALSE
2 >= 2
## [1] TRUE
2 == 2
## [1] TRUE
2 != 2
## [1] FALSE
1 != 2
## [1] TRUE

"hi" == "hello"
## [1] FALSE
"bye" == "bye"
## [1] TRUE
"bye" != "bye"
## [1] FALSE
```

These relational operators can be applied to matrices and vectors as well. When applied to matrices and vectors, they operate **element-wise**, meaning they operate on each element of the vector or matrix one at a time, and give you an answer for each individual element. So, if you apply the == operator to a vector with 10 values, you will get 10 TRUE/FALSE answers.

```
c(10, 5, -10) > 0
## [1]  TRUE  TRUE FALSE
c(10, 5, -10) == 0
```

```
## [1] FALSE FALSE FALSE
c("foo", "bar", "herp", "derp") == "bar"
## [1] FALSE  TRUE FALSE FALSE
```

When you have a vector or matrix on *both* sides of the operator, it still operates element-wise. In this situation, it will match up the elements on each data structure by position (first with first, second with second, etc.).

```
c(10, 5, -10) > c(20, -5, 0)
## [1] FALSE  TRUE FALSE
c(10, 5, -10) == c(20, -5, 0)
## [1] FALSE FALSE FALSE
```

When using relational operators with two vectors/matrices, make sure they both have the same number of elements. If not, R will recycle values from the beginning of the shorter vector in order to "pad" its length. This is probably not what you want to happen, but R will **only** warn you if the length of the longer vector is not a multiple of the shorter vector. If it is a multiple, then R will recycle silently. Be vigilant!

```
c(10, 5) == c(20, -5, 0, 50) # No warning, 4 is a multiple of 2 !!!
## [1] FALSE FALSE FALSE FALSE
```

```
c(10, 5) == c(20, -5, 0) # Recycling warning, 3 is not a multiple of 2
```

```
## Warning in c(10, 5) == c(20, -5, 0): longer object length is not a multiple
## of shorter object length
## [1] FALSE FALSE FALSE
```

2.3.2 Logical Operators

R's logical operators are used for combining multiple Logical values into a single Logical value. The 5 main logical operators are:

- !: The "negation" operator (mentioned in the Logical Data section)
- &: The "element-wise and" operator
- &&: The "scalar and" operator
- ||: The "element-wise and" operator
- |: The "scalar and" operator

We will not cover the use of these operators in this guide, as they go beyond the programming scope expected of PSYCH 240 and 241 students. Interested instructors are advised to look at examples here:

- Manipulating Data
- Logical Operators in R

2.4 Functions

We have used functions in several previous examples, but we have not yet provided an explanation of what an R function is, or how they are used generally. But understanding the basics of functions in R is important, because all of R's analysis and modeling tools are accessed by using functions. In this section, we will describe what a function is generally, and explain how to learn what a function does by reading documentation.

So, what is a function? A function is fixed piece of code that accepts input values, performs some operations or calculations on these values, and returns some results. The purpose of having functions in a programming language is to allow you to repeat an operation *without* having to repeat all of the code that defines the operation - the code is "bundled" into a function, and can be re-used infinitely without copying and pasting the code each time. A good way to start thinking about functions by analogy to equations. For instance, if we have the equation $y = \sqrt{x}$:

- x is the input
- $\sqrt{}$ is the operation performed on the input
- y is the output, in this case the result of taking the square root of x

When you use a function, you are performing a 'function call' (in computer science-ish terms). This leads to colloquialisms like "Call `mean` on that matrix" or "This code calls `diag` to extract the diagonal elements". 'Call' does not mean anything special to us. All using the word 'call' means in this context is 'use a function'.

The inputs to functions go by several names, but most often they are called "**arguments**" or "**parameters**". Calling a function with some specific input is often called "passing an argument". Don't confuse function parameters with population parameters from the statistics side of class.

Perhaps the best way to understand the properties of R functions, and how they can be used, is to look the documentation of an R function. Here, we'll examine the documentation for a function that we've used previously, the `matrix` function. We'll access it by executing the command `?matrix` in the console.

```
?matrix
```

We'll focus on the Description, Usage, and Arguments section, shown below:

```
## Matrices
##
## Description:
##
##      'matrix' creates a matrix from the given set of values.
##
##      'as.matrix' attempts to turn its argument into a matrix.
##
##      'is.matrix' tests if its argument is a (strict) matrix.
##
## Usage:
```

```
##
##       matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
##              dimnames = NULL)
##
##       as.matrix(x, ...)
##       ## S3 method for class 'data.frame'
##       as.matrix(x, rownames.force = NA, ...)
##
##       is.matrix(x)
##
## Arguments:
##
##    data: an optional data vector (including a list or 'expression'
##          vector).  Non-atomic classed R objects are coerced by
##          'as.vector' and all attributes discarded.
##
##     nrow: the desired number of rows.
##
##     ncol: the desired number of columns.
##
##   byrow: logical. If 'FALSE' (the default) the matrix is filled by
##          columns, otherwise the matrix is filled by rows.
##
## dimnames: A 'dimnames' attribute for the matrix: 'NULL' or a 'list' of
##          length 2 giving the row and column names respectively.  An
##          empty list is treated as 'NULL', and a list of length one as
##          row names.  The list can be named, and the list names will be
##          used as names for the dimensions.
##
##        x: an R object.
##
##      ...: additional arguments to be passed to or from methods.
##
## rownames.force: logical indicating if the resulting matrix should have
##          character (rather than 'NULL') 'rownames'.  The default,
##          'NA', uses 'NULL' rownames if the data frame has 'automatic'
##          row.names or for a zero-row data frame.
```

2.4.1 Description

As you might expect, the Description section describes what the function is used for, and lists the functions that are documented in this page. Here, the matrix, as.matrix and is.matrix functions are documented.

2.4.2 Usage

The usage section tells you:

- The syntax for invoking the function.
- The names of the accepted arguments.
- The order of the arguments.
- Which arguments are **required** and which are **optional**:
    - Arguments with an = are optional.
    - All others are required.

We can see that the `matrix` function has 5 arguments, `data`, `nrow`, `ncol`, `byrow` and `dimnames`, each with a default argument. Because all the arguments have defaults, we know that we can call the `matrix` function with no arguments and still get a result!

Lines saying "S3 Method for class …" tell you about the function's behavior when called on objects of a specific class. For example, this help page tells us that when the `as.matrix` function is called on a data.frame, there is an optional argument called `rownames.force` that isn't used when the input is some other data structure (like a vector). We can safely ignore the cryptic term "S3 Method".

We will, however, focus on other cryptic parts of the "Usage" section. What is `data`? What is `byrow`? What do they do? For clarification, we must go the argument-specific subsections.

2.4.3 Arguments

The detailed descriptions in the argument subsection tell us what types of values each argument is permitted to take on. It also tells us what aspect of the function's behavior each argument controls. For example, the `byrow` argument must be a logical value (i.e., `TRUE` or `FALSE`) and it controls whether the matrix is row-by-row, or column-by-column.

2.4.4 Named vs. Unnamed Arguments

As we see in the "Usage" and "Arguments" sections, every function argument has a name (e.g., `data`, `nrow`, `byrow`). When you call a function, those names can be used in a `keyname = value` style of syntax, or they may be omitted in favor of just specifying the value.

If you wish to omit the names of the arguments when calling a function, **then you must order your inputs in the exact same order as they appear in the** `Usage` **section!!!**. If you specify arguments as `keyname=value` pairs, they may be passed in any order. If you mix and match named and unnamed, unnamed inputs that R encounters will be paired up with the unmatched arguments following their order in the Usage section.

```
x <- c(4,10,3,33,2,NA,43,22,31,95)
matrix(data=x, byrow=TRUE, nrow=2, ncol=5) # named key/value style
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4   10    3   33    2
## [2,]   NA   43   22   31   95
matrix(x, 2, 5, TRUE)  # unnamed style, byrow goes last
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4   10    3   33    2
## [2,]   NA   43   22   31   95

matrix(x, TRUE, 2, 5) #5 gets matched up with byrow, not ncol!
##      [,1] [,2]
## [1,]    4   10
```

2.4.5 A Word of Advice

So, which style should you prefer when writing code in an R script: named or unnamed arguments? We recommend using named arguments for all arguments past the first. This strikes a balance between verbosity and clarity - it is often easy to remember what the first argument to a function does and what kind of values it should take on, but it is often difficult to remember the role and order of arguments beyond that.

For example, you have just read the documentation for the `matrix` function - can you remember whether the `nrow` or `ncol` argument goes first? Are you confident enough to just write some code without looking it up. And are you confident that you'll still remember whether you're making a 10 by 50 or a 50 by 10 matrix when you re-read your code tomorrow?

We would venture to guess the answer to these questions is "No", which makes a strong case for naming your arguments when you write your code. Trust us, if you ever venture into a programming language without support for named arguments (I'm looking at you, MATLAB), you'll yearn for named arguments.

In summary, name your arguments.

Special Arguments

You may have noticed that in the "Usage" section, the `as.matrix` and the `is.matrix` functions have an argument called `...`. In fact, many R functions have such an argument. A full discussion of the `...` construct (or the ellipsis, if you're trying to Google it) is beyond the scope of this guide. For our purposes, we can understand it as a special "catch all" device for any inputs whose names aren't explicitly declared ahead of time. The `...` is used to enable argument passing between functions: it allows one function to capture arguments intended for another function, and send them directly to the other function, without ever knowing the names of the arguments for the other function. Neat!

Examples

The last section of the `matrix` help page we will look at is "Examples" sections

```
## Matrices
##
## Examples:
##
##      is.matrix(as.matrix(1:10))
##      !is.matrix(warpbreaks)  # data.frame, NOT matrix!
##      warpbreaks[1:10,]
##    as.matrix(warpbreaks[1:10,]) # using as.matrix.data.frame(.) method
##
##      ## Example of setting row and column names
##    mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
##                    dimnames = list(c("row1", "row2"),
##                                     c("C.1", "C.2", "C.3")))
##      mdat
```

The examples sections demonstrate a simple application of the function. When using a function for the first time, or you find yourself confused by a part of the documentation, running and tweaking the examples you find here is a great way to get a concrete understanding of how the function behaves.

The Return Value

The `matrix` function lacks one field in the help file that most R function have - the "Value" field. This section describes what the function outputs, i.e., what it "returns" to the caller. The `matrix` function can get away with omitting this section, because its return value is fairly obvious - a matrix! But functions with more complicated outputs need to describe what they return in more detail, so the user can understand how to process the output in their own code.

2.5 Data Manipulation

Data manipulation describes the processes of editing and re-organizing a set of observations in order to facilitate a subsequent analysis. In this guide, we will cover three main data manipulation processes: indexing, subsetting, and replacement. Since data manipulation typically begins by importing data from a file on your hard drive into R, we will begin this section describing how to import or "read in"

2.5.1 Tidy Data

It's worth beginning with an outline of a well-formatted data set.

- The data is represented in a rectangular structure (table with rows and columns)
- Each column represents a specific variable, with a header signifying the name of this variable
- Each row is represents an observation
- Avoids names or values with blank spaces
- Avoids using names that contain symbols such as :, ;, ?, $, %, ^, &, *, (,), -, #, ?, < , >, /, |, [,], { and }
- Any missing values in your data set are indicated with NA

Adhering to these principles when you save new data, or manipulate data you have, will greatly simplify analysis performed in R.

2.5.2   Importing Data

CSV files

A CSV file is a type of plain-text document, and is indicated by the .csv file extension. Plain text files consist only of sequences of characters codes, including spaces, tabs, new lines and delimiters. They have no styling associated with them (e.g. no italics or bolding, no images). Files with extensions such as .txt, .R and .html are plain-text files, while files such as .doc, .docx (Word documents) and .xlsx (Excel documents) are **not** plain text files. We recommend you use plain-text formats for sharing data, because they have the greatest deal of interoperability between computer operating systems and analysis programs.

In a CSV file, the content is arranged in a tabular format. Each new line in the file represents a row, and distinct values within each row are separated by commas to form the different columns. Below is an example of what the text of a CSV file looks like before it is imported into R:

```
condition,trial,rating
a,1,3
b,2,1
c,3,11
```

We can see that the file has 3 columns, a header row, and 3 observation rows. Before we can import this file into R, we must know how to instruct R where to find the file on our computer. To do this, we must understand about file paths on our hard drive, and how R looks for files.

File Paths and Working Directories

All the files stored on your computer's hard drive are associated with a named location in the file system's hierarchy. For example, Windows users are likely familiar storing files inside the "My Documents" folder (also known as a "directory").

Much like a file, the R session you have open is also associated with a directory on your hard drive. But, unlike a file, your R session can easily change its current location without copying the session. The directory your R session currently inhabits is called the "Current Working Directory". You can see what this directory is by issuing the command `getwd()`.

```
getwd()
## [1] "C:/Users/will/source/PBS R Manual"
```

As you can see, the current working directory of my R session is the PBS R Manual folder. I can change this location using the `setwd()` command, and providing the name of another directory to move the R session to. The new directory I move to needs to be specified as a Character value (i.e., surrounded with quotation marks). However, I have to be very clear and explicit when describing the location of this directory. Specifically, I have describe this directories location using either a **relative** or an **absolute** path.

An absolute file path describes the location in relationship to the beginning of the entire file system, while a relative path describes the location in relationship to R's current working directory. This is important, because not every location on your hard drive is visible from R's current directory - R can only see files *below* its current working directory in the file system hierarchy.

If you need to access a file that *is not* below your current working directory, the best way to do this is with an absolute file path. On Windows, the start of each file system is given a letter prefix; the prefix of the file system holding the Window's installation is `C:\`. Directories are separated with **backward** slashes (e.g. `C:\Users\will` is an absolute path). On Mac OSX and Linux, the start of the file system is / (read as "root"). Here, directories are separated with **forward** slashes (e.g., `/Users/will` is an absolute path). But in R, you don't have to worry about forward slashes vs. backward slashes. **You can use forward slashes in your code, and it will work on either Mac or Windows**

If you need to access a file that *is* not below your current working directory, the best way to do this is with a relative file path. A relative file path doesn't need to begin with `C:\` or `"/"`, it can just begin with the name of the file or directory. Let's do this now with the CSV file we saw in the previous section.

read.csv

The R function used for importing CSV files is called `read.csv`. It has one required argument, the file path describing the name and location of the CSV file to import. In this case, the CSV file is named `commas.csv` and it is stored in a directory named `data` that is in my current working directory. Let's import it now:

```
commas <- read.csv("data/commas.csv")
class(commas)
## [1] "data.frame"
commas
##   condition trial rating
```

```
## 1          a      1      3
## 2          b      2      1
## 3          c      3      11
```

Note that I assigned the output of the `read.csv` function to variable named `commas`, and that the function imported the CSV file as a `data.frame` object. Also note that the values in the first row of the CSV file were used as names for each columns, rather than a row of data.

Importing Excel Files

Excel files are ubiquitous, but because of their history as a proprietary format, R does not have native support for importing them. However, all is not lost: you can install the `readxl` package and use its `read_excel` function to import .xls and .xlsx files into R as data frames.

2.5.3   Subsetting

Subsetting describes the processes of "extracting" or "slicing out" a subset of the values from one data structure into another. In R, the processes of subsetting **does not** remove the values you subset from the original data structure. Rather, it creates a copy of the subset you ask for, and puts that copy into your new data structure. So, subsetting is a safe operation that will not result in any data loss.

Subsetting a data structure is performed in R using the `[]` operator, which are called square brackets. All R data structures can be subsetted using the `[]` operators. To subset a data structure, put the `[]` operator immediately after the data structure. Between the two square brackets, you place what is known as an **index vector**. An index vector is a vector that describes which values in the data structure you want included in the subset. There are 3 types of index vectors:

- Numeric Index Vectors, which describe the *position* (e.g. first, third, or 19th) of the elements you want included in the subset.
- Character Index Vectors, which describe the *names* of the elements you want included in the subset (only useful when the elements have names).
- Logical Index Vectors, which specify for each element in the data structure whether it should be included, or excluded, from the subset.

We'll start with subsetting vectors with a numeric index vectors to get a feel for the general procedure.

Subsetting with a Numeric Index

```
alphabet <- c("a","b","c","d","e","f","g","h","i","j","k","l","m","n","o",
              "p","q","r","s","t","u","v","w","x","y","z")
```

```
alphabet[c(1,26)] # Extract First and 26th element
## [1] "a" "z"
alphabet[10:20] # Extract tenth through 20th
##  [1] "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
```

One of the most common mistake is including a value in your indexing vector which is greater than the length of the vector you are subsetting

```
alphabet[100] # there are not 100 letters in the alphabet
## [1] NA
```

The NA means the value is missing. This is commonly referred to as an "index out of bounds" error, although R does not explicitly give you an error.

Another common mistake is forgetting to concatenate the values you want to use for the indexing vector (i.e. forgetting the c() function).

```
alphabet[1,5]
```

```
## Error in alphabet[1, 5]: incorrect number of dimensions
```

This time, R does give us an error, letting us know that we've attempted to index a vector like a matrix.

2.5.3.0.1 "Negative" Subsetting

Instead of creating a vector of values you *do* want to pick out, it may be easier to come up with a vector of ones you *don't* want. We can use negative number's to specify which vector elements we don't want.

```
alphabet[c(-1,-26)] # Same as alphabet[2:24]
##  [1] "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
## [18] "s" "t" "u" "v" "w" "x" "y"
alphabet[-1:-10]
##  [1] "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Indexing with positive vectors is usually preferred, as the intent of the code is more clear, but sometimes this form can be clearer when constructing the "anti-set" is easier (e.g. when dropping the first value).

Subsetting with a Character Index

If the elements of our vector have names, we can use those names instead of their positions.

```
x <- 1:5
names(x) <- c("A","B","C","D","F")
x
## A B C D F
```

```
## 1 2 3 4 5
x[c("B","F")]
## B F
## 2 5
```

Subsetting with a Logical Index

When subsetting with a logical index vector, you supply a vector specifying whether to extract a specific element (with a TRUE) or to *not* extract a specific element (with a FALSE).

Let's revisit the example of selecting the first and last elements of the alphabet vector: We make a vector of logicals and stick it in the square brackets after your vector.

```
alphabet[c(1,26)]
## [1] "a" "z"
alphabet[c(TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
          FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
          FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE)]
## [1] "a" "z"
```

But this specific example is not a good use case for logical vectors. Why?

1. Longer Code: length of the logical vector must match the length of the object its subsetting.
2. Duplicating work: If you already know the position of the elements you want, just put them into a vector and you're done!

The logical vector's utility comes into play when you *don't* know the numeric positions of the elements you are interested in. But, how can you determine which values you want to keep without knowing their position or their name? In these cases, we must *search* for values meeting a specific criterion. Searching for values within a data structure is a processes called **Indexing**.

2.5.4   Indexing

Indexing a data structure in a search for specific values is a job for R's relational operators. Remember, relational operator are applied to all the elements of a data structure individually (i.e., "element-wise"). Thus, we can apply them to search for specific values, and use the Logical TRUE/FALSE values that result from this search as an index vector.

```
x <- 2:11
print(x)
##  [1]  2  3  4  5  6  7  8  9 10 11
x <= 5 # Apply the less than or equals test
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

As you can see, values that meet the criteria (<= 5) return as TRUE.

```
x[x <= 5] # Index vector x with the results of the test.
## [1] 2 3 4 5
```

When this logical vector is used to index the vector x, only the elements where the logical vector has value TRUE are returned.

We index character vectors using the == and != operators, but not the greater/less than operators. Quantity makes no sense for characters!

```
months <- c("January", "February", "March", "April", "May", "June", "July",
            "August", "September", "October", "November", "December")
months == "June" # The sixth element is TRUE
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
months[months == "June"]
## [1] "June"
months[months != "July"]
##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "August"    "September" "October"   "November"
## [11] "December"
```

Other Useful Tests: `is.na()`

Unfortunately, we often have to deal with missing observations in real world data sets. R codes missing data as NA (or sometimes NaN). We can use the is.na() function to find any missing values in a vector.

```
missingno <- c(10,NA,1,4,2,NA,NA,99,NaN, NA)
is.na(missingno)
##  [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
missingno[!is.na(missingno)] # Select the *not* missing observations
## [1] 10  1  4  2 99
```

Removing missing values in a common step in data manipuation before an anlysis. For examples of removing missing values with other data structures, and more realistic data sets, the examples in the Data Sets section.

Converting a Logical to a Positional Index

A useful function to know is `which()`. When used on a logical vector, it will return to your the position indices of the vector's TRUE element. It is useful when you want to know **where** in the vector your matches occur.

```
is.na(missingno)
##  [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
```

```r
which(is.na(missingno))
## [1]  2  6  7  9 10
```

2.5.5 Replacement

To replace the values in a vector (e.g., to replace empty characters with NA values), move the indexing and subsetting operation to the right-hand side of the assignment operator, and put the replacement value(s) on the left-hand side.

```r
song <- c("Happy", "Birthday", "", "You")
song[song == ""] <- NA
song
## [1] "Happy"    "Birthday" NA         "You"
```

A Word of Advice

Unlike subsetting, replacement *does* present the risk of data loss, because once a value is replaced, it can't be undone. So, when you are experimenting, we recommend you make a "backup" copy of your data structure before editing it. In the previous example, with the "Happy Birthday" lyrics, you might do the following.

```r
song <- c("Happy", "Birthday", "", "You")
song_original <- song
song[song == ""] <- NA
song
## [1] "Happy"    "Birthday" NA         "You"
song_original
## [1] "Happy"    "Birthday" ""         "You"
```

This way, you have a copy of the original data, in case there was a bug in your code, or you need the raw data later on for another operation.

2.5.6 Matrices & Data Frames

Now we'll learn how to index data structures with more than one dimension, like matrices and data frames. Recall that matrices and data frames have both rows **and** columns,meaning that when we subset or index them, we must specify which rows and/or columns we would like out subset or search to apply to.

Matrices

To index a matrix, all that is required is to have two vectors inside our square brackets, separated from each other by a comma. The template is: `OurBigMatrix[rowIndex, columnIndex]`

Like with vectors, the index vectors can be either:

- Numeric vectors specifying the position of the rows/columns we want to access
- Character vectors specifying the names of the rows/columns we want to access (if they have names)
- Logical vectors specifying for each column and row whether we want to access it (TRUE) or ignore it (FALSE)

```
dummy <- matrix(6:1, nrow = 2)
dummy
##      [,1] [,2] [,3]
## [1,]    6    4    2
## [2,]    5    3    1
dummy[1,2:3] # Row 1, Column 2 and 3. Output is a vector!
## [1] 4 2
dummy[1:2,2:3] # Row 1 and 2, Column 2 and 3. Output is a matrix.
##      [,1] [,2]
## [1,]    4    2
## [2,]    3    1
```

If you want to select **all** of one dimension, (e.g., keep all rows or all columns) but index the other dimension, provide the separating comma as usual, but don't give any indexing vector for the dimension you want to stay 100% intact.

```
dummy[1,] # First Row, all columns
## [1] 6 4 2
dummy[1,1:3] # Same as previous
## [1] 6 4 2
dummy[,2] # All rows, second columns
## [1] 4 3
dummy[1:2,2] # Same as previous
## [1] 4 3
```

We can apply our relational operators to entire matrices in the same manner as vectors. The resulting logical matrix has the same dimensions as the one we apply the test to.

```
dummy < 4 # 2 x 3
##       [,1]  [,2] [,3]
## [1,] FALSE FALSE TRUE
## [2,] FALSE  TRUE TRUE
```

We can also apply logical testing and logical indexing to specific dimensions of a matrix. This example here keeps all the columns of the matrix with a sum less than 8.

```
colSums(dummy) # colSums() adds up each column
## [1] 11  7  3
colSums(dummy) < 8 # Does each column sum to less than 8?
## [1] FALSE  TRUE  TRUE
dummy[,colSums(dummy) < 8] # Select columns with a sum less than 8
```

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

Data Frames

To learn about data frames, we're going to use several data frames that come
built-in with R as part of the `datasets` package. Try typing `InsectSprays`, `iris`,
`airquality` and `mtcars` into the console to be sure they are loaded and available to
you. Since they are included as part of a package, you will *not* see them listed in your
environment pane.

The `[row, column]` indexing style used with matrices also applies to data frames.
However, data frames also support a different subsetting technique based on a special
syntax that applies to its column names. Subsetting or indexing a data frame using col-
umn names should be preferred to using column numbers, because that name is unlikely
to change, while the row or column number is **very** likely to get changed throughout the
course of an analysis. It's also much easier to remember the name of something than
remember its position in a data frame!

2.5.6.0.1   Subsetting with $ syntax

To subset a single column from a data frame, we can use that column's name, and
the $ operator. In this case, quotes around the column's name are not required. To
demonstrate, we will subset the `mpg` column from the `mtcars` dataset.

```
mtcars
##                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant            18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360         14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D          24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230           22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
##   [ reached getOption("max.print") -- omitted 23 rows ]
mtcars$mpg
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

This $ syntax *can not* be applied to rows.

2.5.6.0.2 Subsetting with [] syntax

To subset *multiple* columns, or to subset specific rows, we need to use [row,column] style indexing (not the $).

But we're not forced to use numeric vectors just because we're using the [operator. We can select multiple columns by their names using a character vector that has the names of our desired columns as its elements.

```
mtcars[,c("mpg","disp","gear")] # need c() function as well as the quotes here
##                     mpg  disp gear
## Mazda RX4          21.0 160.0    4
## Mazda RX4 Wag      21.0 160.0    4
## Datsun 710         22.8 108.0    4
## Hornet 4 Drive     21.4 258.0    3
## Hornet Sportabout  18.7 360.0    3
##  [ reached getOption("max.print") -- omitted 27 rows ]
```

One of the most common subsetting tasks with a data frame (or matrix) is the need to select values in one column where the values in another column meet a certain criteria. For example, you might want to select all the values in the column holding reaction times where participants were incorrect. There are 2 syntactic approaches to this, both of which use relational operators and logical indexing.

2.5.6.0.3 Method 1: Index the data frame itself

We will use the [row,column] method to pick out the values of the count column in InsectSprays where spray A was used. First, we will build up a logical vector to index the correct rows by testing where the spray column has value 'A'

```
InsectSprays$spray
##  [1] A A A A A A A A A A A A B B B B B B B B B B B B C C C C C C
##  [ reached getOption("max.print") -- omitted 42 entries ]
## Levels: A B C D E F
InsectSprays$spray=="A"
##  [1]   TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]   TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [ reached getOption("max.print") -- omitted 42 entries ]
```

Next, we combine this with a character vector of the column names we're interested in, and put it inside our [] brackets

```
InsectSprays[InsectSprays$spray=="A",'count']
##  [1] 10  7 20 14 14 12 10 23 17 20 14 13
```

If we leave the column vector out, this statement will return a data frame. Can you guess how many unique values will be in the spray column in this case?

Method 2: Index a vector *from* the data frame

Here, we will use the $ operator to subset the `count` column from the `InsectSprays` data frame. Then, index this vector with the logical vector resulting from a relational test

```
InsectSprays$count[InsectSprays$spray=="A"] # Same result as before
##  [1] 10  7 20 14 14 12 10 23 17 20 14 13
```

2.5.6.0.4   Errors when indexing by name

If you try to subset a column of a data frame using the $ operator, but the name of the column doesn't exist, R will return `NULL`

```
InsectSprays$neeeeeeighhhhh
## NULL
```

But, if you use the `[row,column]` style of indexing and ask for a column that doesn't exist, you get a formal error.

```
InsectSprays[, 'neeeeeeighhhhh']
```

```
## Error in `[.data.frame`(InsectSprays, , "neeeeeeighhhhh"): undefined columns selected
```

Its also a common mistake to forget the quotes around names inside the `[]` brackets, which will throw an "object not found" error (unless the an object with the same name just happens to exist by coincidence, in which case you will probably get another type of error).

```
InsectSprays[, spray]
```

```
## Error in `[.data.frame`(InsectSprays, , spray): object 'spray' not found
```

Reproducibility

The R session information when compiling this book is shown below:

```
library(printr)
## Warning: package 'printr' was built under R version 3.5.3
sessionInfo()
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
```

```
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] printr_0.1
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.5.1   magrittr_1.5     bookdown_0.11     htmltools_0.3.6
##  [5] tools_3.5.1      Rcpp_1.0.1       codetools_0.2-15 stringi_1.4.3
##  [9] rmarkdown_1.13   knitr_1.22       stringr_1.3.1     xfun_0.6
## [13] digest_0.6.19    packrat_0.4.9-3  evaluate_0.14
```

Chapter 3

Statistics

3.1 The Data

We rely on the `sat.act` data set from the `psych` package for all demonstrations in this chapter.

To make the data set available on your own computer, run the following code:

```r
install.packages('psych')
library(psych)
```

You should now be able to access the `sat.act` data set:

```r
summary(sat.act)
```

```
##     gender         education          age             ACT
##  Min.   :1.000   Min.   :0.000   Min.   :13.00   Min.   : 3.00
##  1st Qu.:1.000   1st Qu.:3.000   1st Qu.:19.00   1st Qu.:25.00
##  Median :2.000   Median :3.000   Median :22.00   Median :29.00
##  Mean   :1.647   Mean   :3.164   Mean   :25.59   Mean   :28.55
##  3rd Qu.:2.000   3rd Qu.:4.000   3rd Qu.:29.00   3rd Qu.:32.00
##  Max.   :2.000   Max.   :5.000   Max.   :65.00   Max.   :36.00
##
##       SATV            SATQ
##  Min.   :200.0   Min.   :200.0
##  1st Qu.:550.0   1st Qu.:530.0
##  Median :620.0   Median :620.0
##  Mean   :612.2   Mean   :610.2
##  3rd Qu.:700.0   3rd Qu.:700.0
##  Max.   :800.0   Max.   :800.0
##                  NA's   :13
```

As you can see, this data set includes self-reported SAT and ACT scores along with demographic information from 700 respondents. It therefore contains both categorical and continuous variables that are appropriate for both within- and between-subjects analyses. For more information, refer to the help documentation by running the command `?sat.act`.

3.2 Formula Objects

Formulas are special objects in R that are used in a range of functions discussed in this chapter, notably `plot()`, `aov()`, and `lm()`. The use of formulas in these specific functions will be discussed in more detail below, but we first introduce their general properties here.

The basic structure is as follows:

```
y ~ x
```

The y variable is the dependent variable, the x variable is the independent variable, and the ~ operator can be taken to mean "as a function of". Thus, the above example is a formula for "y as a function of x".

For a more concrete example, consider the effect of the independent variable age on the dependent variable ACT scores:

```
sat.act$ACT ~ sat.act$age
```

We can have multiple independent variables in the formula. For instance, if we wanted to define ACT scores as a function of age and education, we would do so as follows:

```
sat.act$ACT ~ sat.act$age + sat.act$education
```

The above formula is an example of an "additive" model, in which ACT scores are a function of the additive effects of age and education. To define the interactive model, we would separate age and education by an asterisk instead of a plus sign:

```
sat.act$ACT ~ sat.act$age*sat.act$education
```

Note that the above formula defines the *full* interactive model, so ACT scores are defined as a function of (1) the main effect of age, (2) the main effect of education, and (3) the interaction of age and education. This same model can be implemented by defining each component separately in the formula object, like this:

```
sat.act$ACT ~ sat.act$age + sat.act$education + sat.act$age:sat.act$education
```

By defining the components separately, you can pick and choose which to keep. For instance, you could specify only the main effect of age and the interaction of age and education, as follows:

```
sat.act$ACT ~ sat.act$age + sat.act$age:sat.act$education
```

Formulas can get much more complex than the examples here, which only extend as far necessary for the functions and tests that you are likely to use as an instructor in undergraduate psychology program. A comprehensive tutorial can be found here.

3.3 Descriptive Statistics

3.3.1 Mean

To calculate the mean of a vector by hand, use the `sum()` and `length()` functions. For instance, we can calculate the mean verbal SAT score in the `sat.act` data set as follows:

```
sum(sat.act$SATV)/length(sat.act$SATV)
```

[1] 612.2343

We can also get the same answer by using the `mean()` function:

```
mean(sat.act$SATV)
```

[1] 612.2343

Note that `NA` values can be removed from the vector when using the `mean()` function by setting the `na.rm` argument to `TRUE`. For instance, when we try to take the mean of the quantitative SAT scores without removing `NA` values, we get `NA` as the output:

```
mean(sat.act$SATQ)
```

[1] NA

But by setting `na.rm=TRUE`, we get a sensible answer:

```
mean(sat.act$SATQ, na.rm=TRUE)
```

[1] 610.2169

The `NA` values cannot be removed this easily when calculating the mean by hand because while the `sum()` function will take the `na.rm` argument, the `length()` function will not. Thus, if you plan to ask students to calculate the mean by hand in R, be sure to choose a method of removing NA values manually. You can either remove the values yourself beforehand, give the students code to do it, or teach them to look out for and handle `NA` values themselves.

3.3.2 Median

You can ask students to calculate the median of a vector by hand by having them index the output of the `sort()` function. To demonstrate, imagine that we want to know the median age of respondents in the `sat.act` data set. Calling `sort()` function on the age variable will take all of the respondents' ages and put them in ascending order:

```r
sort(sat.act$age)
```

```
##   [1] 13 14 15 15 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17
##  [24] 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
##  [47] 17 17 17 17 17 17 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
##  [70] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
##  [93] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
## [116] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 19 19 19
## [139] 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
## [162] 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
## [185] 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
## [208] 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 20 20 20 20 20
## [231] 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
## [254] 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
## [277] 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 21 21 21 21 21 21 21
## [300] 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21
## [323] 21 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22
## [346] 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 23 23 23 23
## [369] 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23
## [392] 23 23 23 23 23 23 23 23 23 23 23 23 24 24 24 24 24 24 24 24 24 24 24
## [415] 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 25 25 25 25 25 25
## [438] 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 26
## [461] 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 27 27
## [484] 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 28 28 28 28 28 28
## [507] 28 28 28 28 28 28 28 28 28 28 28 28 28 29 29 29 29 29 29 29 29 29 29
## [530] 29 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 31 31 31 31
## [553] 31 31 31 31 31 32 32 32 32 32 33 33 33 33 33 33 33 33 33 33 34 34 34
## [576] 34 34 34 34 34 34 34 34 34 34 35 35 35 35 35 35 35 35 35 35 35
## [599] 35 35 36 36 36 36 36 36 36 36 36 36 37 37 37 37 37 37 37 37 37 37 37
## [622] 38 38 38 38 38 38 38 39 39 39 40 40 40 40 40 40 40 40 40 41 41 41 41
## [645] 42 42 42 42 43 43 43 43 43 43 43 44 44 44 44 44 44 45 45 46 46 46 46
## [668] 47 48 48 49 50 50 50 50 50 52 52 52 53 53 54 54 55 55 55 55 56 57 57
## [691] 57 57 57 58 58 58 61 61 62 65
```

We can then find the median by indexing the middle value of the sorted vector. If the vector has an odd number of values, this is simply half of the length of the vector plus .5:

```r
sort(sat.act$age)[(length(sat.act$age)/2)+.5]
```

Of course, there are 700 respondents in the sat.act data set, so the above command is not appropriate. When the vector has an even number of values like ours, the median is calculated as the average of the two middle values:

```r
# Find the two middle values
middle_vals <- sort(sat.act$age)[(length(sat.act$age)/2):((length(sat.act$age)/2)+1)]
```

```
# Take the mean of the two middle values
mean(middle_vals)
```

[1] 22

Note that you might also allow students to simply print the output of `sort()` and find the median visually.

We can also get the same answer by using the `median()` function:

```
median(sat.act$age)
```

[1] 22

Note that `NA` values can be removed from the vector when using the `median()` function by setting the `na.rm` argument to `TRUE`. For instance, when we try to take the median of the quantitative SAT scores without removing `NA` values, we get `NA` as the output:

```
median(sat.act$SATQ)
```

[1] NA

But by setting `na.rm=TRUE`, we get a sensible answer:

```
median(sat.act$SATQ, na.rm=TRUE)
```

[1] 620

The `NA` values are automatically removed when calculating the median by hand using the `sort()` function, however you can also choose to have missing values placed last or first in the output by setting the `na.last` argument to 'TRUE' or 'FALSE', respectively.

3.3.3 Variance

Calculating the variance by hand requires more distinct sequencing than for the mean and the median. To demonstrate, imagine that we want to find the variance of the verbal SAT scores reported in the `sat.act` data set.

Recall from your knowledge of statistics that the formula for the variance of a sample is:

$$s^2 = \frac{\sum(x_i - \bar{x})^2}{n - 1}$$

You might first have students find the squared difference between each value and the mean:

```
SATV_sq_diff <- (sat.act$SATV-mean(sat.act$SATV))^2
```

You can then have students sum up these squared deviations and divide by the total number, minus one:

```r
sum(SATV_sq_diff)/(length(SATV_sq_diff)-1)
```

[1] 12746.99

We can also get the same answer by using the `var()` function:

```r
var(sat.act$SATV)
```

[1] 12746.99

Note that `NA` values can be removed from the vector when using the `var()` function by setting the `na.rm` argument to TRUE. For instance, when we try to take the variance of the quantitative SAT scores without removing `NA` values, we get `NA` as the output:

```r
var(sat.act$SATQ)
```

[1] NA

But by setting na.rm=TRUE, we get a sensible answer:

```r
var(sat.act$SATQ, na.rm=TRUE)
```

[1] 13372.45

The `NA` values cannot be removed this easily when calculating the variance by hand because while the `sum()` function will take the `na.rm` argument, the `length()` function will not. Thus, if you plan to ask students to calculate the variance by hand in R, be sure to choose a method of removing NA values manually. You can either remove the values yourself beforehand, give the students code to do it, or teach them to look out for and handle `NA` values themselves.

3.3.4 Standard Deviation

To calculate the standard deviation by hand, simply extend the previous commands for calculating the variance one step further by calling `sqrt()` on the previous result:

```r
# Find the squared deviations
SATV_sq_diff <- (sat.act$SATV-mean(sat.act$SATV))^2

# Sum squared deviations and divide by n-1
SATV_var <- sum(SATV_sq_diff)/(length(SATV_sq_diff)-1)

# Take the square root
sqrt(SATV_var)
```

[1] 112.9026

More simply, students could take the square root of the output of `var()`:

```r
sqrt(var(sat.act$SATV))
```

[1] 112.9026

Note that the square root of a value can also be found by raising it to .5:

```r
var(sat.act$SATV)^.5
```

```
## [1] 112.9026
```

We can also get the same answer by using the `sd()` function:

```r
sd(sat.act$SATV)
```

```
## [1] 112.9026
```

Note that NA values can be removed from the vector when using the `sd()` function by setting the `na.rm` argument to TRUE. For instance, when we try to take the standard deviation of the quantitative SAT scores without removing NA values, we get NA as the output:

```r
sd(sat.act$SATQ)
```

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

But by setting `na.rm=TRUE`, we get a sensible answer:

```r
sd(sat.act$SATQ, na.rm=TRUE)
```

```
## [1] 115.6393
```

The NA values cannot be removed this easily when calculating the standard deviation entirely by hand (that is, without using the `var()` function) because while the `sum()` function will take the `na.rm` argument, the `length()` function will not. Thus, if you plan to ask students to calculate the standard deviation by hand in R, be sure to choose a method of removing NA values manually. You can either remove the values yourself beforehand, give the students code to do it, or teach them to look out for and handle NA values themselves.

3.3.5 Standard Error of the Mean

To calculate the standard error of the mean, simply extend the previous commands for calculating the standard deviation one step further by dividing the previous result by the number of observations minus one:

```r
# Find the squared deviations
SATV_sq_diff <- (sat.act$SATV-mean(sat.act$SATV))^2

# Sum squared deviations and divide by n-1
SATV_var <- sum(SATV_sq_diff)/(length(SATV_sq_diff)-1)

# Take the square root
SATV_sd <- sqrt(SATV_var)
```

```r
# Divide by n-1
SATV_sd/(length(SATV_sq_diff)-1)
```

```
## [1] 0.1615201
```

More simply, students could take divide the output of `sd()` by the number of observations minus one:

```r
sd(sat.act$SATV)/(length(sat.act$SATV)-1)
```

```
## [1] 0.1615201
```

In either case, because `length()` is used in the final step, the `NA` values cannot be easily removed when calculating the standard error of the mean by hand. You should therefore be sure to choose a method of removing NA values manually. You can either remove the values yourself beforehand, give the students code to do it, or teach them to look out for and handle `NA` values themselves.

Because there are different methods for calculating the standard error depending on the nature of the sample statistic, there is no central function for calculating the standard error in R. The `std.error()` function is available in the package `plotrix` and handily includes the `na.rm` argument. However, if `NA` values aren't an issue, you may find that calculating the standard error is a simple enough extension of the standard deviation to have students do it by hand.

3.3.6 Correlation & Proportion of Variance Explained

We do not recommend having students calculate Pearson's r by hand. Instead, the correlation coefficient can be found using the `cor()` function.

To demonstrate, imagine that we are curious as to whether there is a linear relationship between respondents' verbal and quantitative SAT scores:

```r
cor(sat.act$SATV, sat.act$SATQ)
```

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

We immediately note that the output is `NA` because there are missing values in the quantitative SAT scores. To remedy this, set the `use` argument to `complete.obs`:

```r
cor(sat.act$SATV, sat.act$SATQ, use='complete.obs')
```

```
## [1] 0.6442999
```

With this method, `cor()` will remove any pairs containing at least one missing value before calculating the correlation coefficient.

To calculate the proportion of variance explained, R^2, simply square the correlation coefficient:

```r
cor(sat.act$SATV, sat.act$SATQ, use='complete.obs')^2
```

```
## [1] 0.4151224
```

This is a handy way of demonstrating that R^2 is very simply r, squared.

3.3.7 Summary Functions

The standard method for summarizing data with multiple descriptive statistics at once is to use the `summary()` function. This function can be used to simultaneously summarize each variable in a data set, such as those in the `iris` data frame:

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##   Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##   1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##   Median :5.800   Median :3.000   Median :4.350   Median :1.300
##   Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##   3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##   Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##         Species
##   setosa    :50
##   versicolor:50
##   virginica :50
##
##
##
```

Numeric variables are summarized with the minimum and maximum values, the .25, .5, and .75 quantile cutoff values, and the mean value. Factors are summarized with a table of counts for each factor level. Character strings are summarized in terms of their total length.

A related function that is sometimes helpful is the `str()` function, which summarizes an object in terms of its class and the class of each of its elements:

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

While it can be inferred from the output of `summary()` that `Species` is a factor and the remaining variables are numeric, `str()` makes this explicit.

Note that additional techniques exist for summarizing data with multiple descriptive statistics. The `describe()` function in the `psych` package provides a range of useful statistics for numeric variables. Tools provided in the `dplyr` package, particularly the

combination of `group_by()` and `summarize()`, allow the user to specify their own descriptive statistics.

3.4 Plotting

One of the most compelling reason to learn and use R is because it includes highly flexible and powerful tools for visualizing many kinds of data. After all, if a single tool can perform your analyses and create your figures, why not use the "one ring to rule them all"?

Many visualizations can be created using the `plot()` function, so we'll begin our introduction using that function. As you'll soon see, the `plot()` function accepts many arguments, as there are many ways to customize a plot. Many of the other visualization tools in R accept similar arguments. So, if `plot()` accepts an argument, there is a fair chance another function will accept it as well.

3.4.1 Scatter & Line Plots

If you have two vectors representing pairs of observations, you can plot them against one another in a scatter plot using the `plot()` function. Let's demonstrate this by plotting the SAT Verbal and SAT Quantitative scores from the `sat.act` dataset against one another.

```r
plot(sat.act$SATV, sat.act$SATQ) # Access the columns as vectors using $ syntax
```
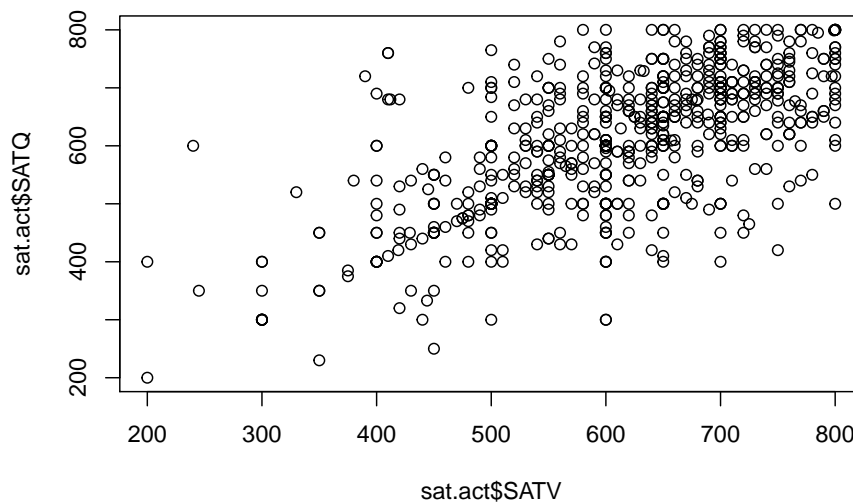
The points in the plot above represent pairs SAT Verbal and SAT Quantitative scores. The scores in each vector are paired up by position within each vector - the first value in SATV is paired with the first value in SATQ, the second value in SATV is paired with the second value in SATQ, etc. The vector given as the first argument determines the X axis position, and the vector given as the second argument determines the Y axis position.

The axis titles are determined by the variable names of the vectors supplied to the `plot` function. However, variable names aren't always the most informative names for other humans to read. We can customize the axes and plot with informative titles using the `xlab`, `ylab` and `main` arguments for the `plot` function:

```
plot(sat.act$SATV, sat.act$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores")
```

Relationship between SAT Scores
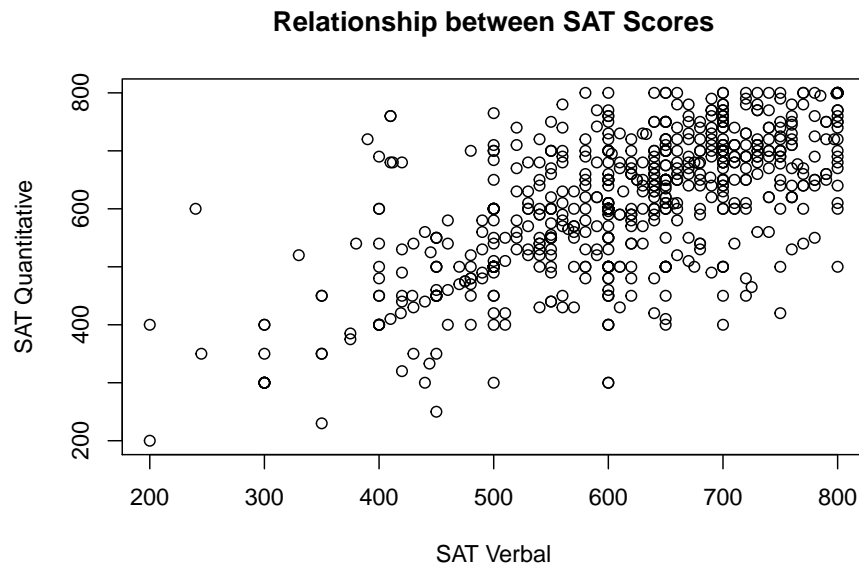
3.4.1.1 Linetypes, Symbols, and Colors

You can customize the scatterplot to draw lines between the points, or just draw lines without the point symbols with the `type` argument. **This is not a sensible thing to do for these data**, but we demonstrate it anyway for completeness.

```
# Lines and points
plot(sat.act$SATV, sat.act$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores",
     type = "o")
```

```
# Just lines
plot(sat.act$SATV, sat.act$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores",
     type = "l")
```

You can also customize the types of symbols that are used for each point using the `pch`, the style of lines connecting these points using the `lty` argument, and the `col` argument to adjust the color of lines and symbol borders

With the `pch` argument, you specify the symbol used for each point using a numeric code. For example, the value 0 is the code for an unfilled square, while the value 15 is the value for a filled square with no border. Unfortunately, it has been scientifically
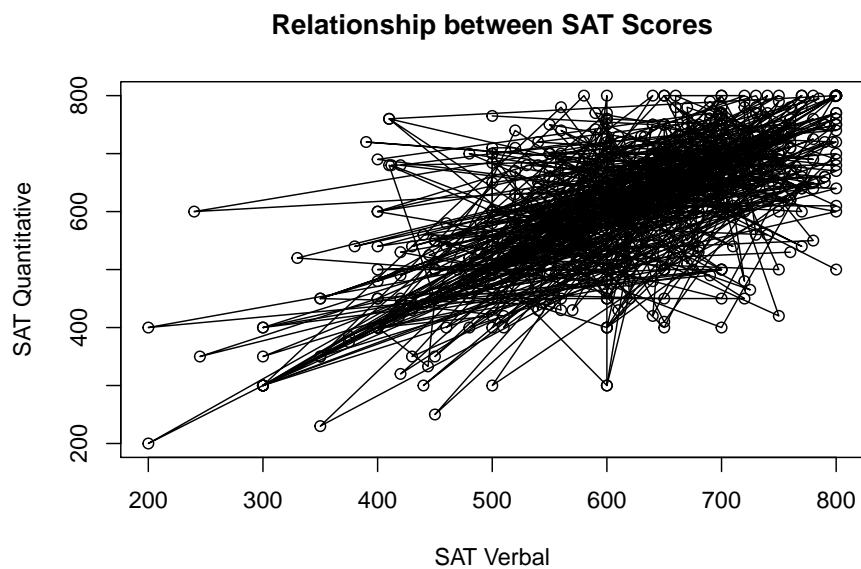
Relationship between SAT Scores

Figure 3.1: Points and lines with type='o'
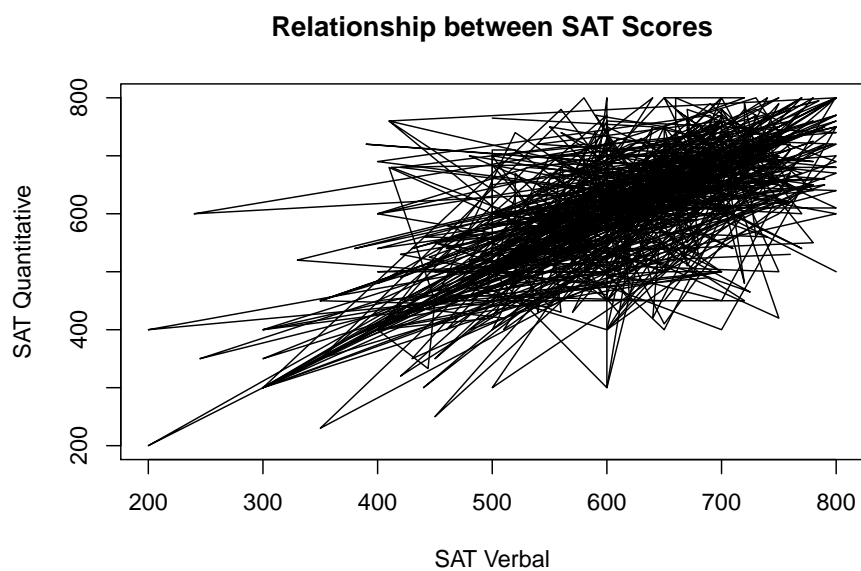
Relationship between SAT Scores

Figure 3.2: Just using lines with type='l'

proven that remembering the code for all the possible symbols is impossible. Luckily, a helpful mnemonic device called a "chart" has been invented to help R programmers customize their plots. This chart is shown below:

Note that if you supply a character instead of a numeric values between 0 and 25, R will use that character as the plotting symbol. So, you could make a plot full of pound signs if you are so inclined!

The `lty` argument has a bit more flexibility. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash".

The `col` argument accepts character strings describing either a "built in" color name (see `?colors` for a list) or an RGB triplet encoded in hexadecimal notation. Unless you are intimately familiar with RGB color space, it's a good idea to use a tool like the W3 Organizations Online Color Picker to help you figure out the RGB hex code for the color you want.

To demonstrate, lets take our original plot (with no lines) and customize it to use unfilled, magenta triangles:

```
# Lines and points
plot(sat.act$SATV, sat.act$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores",
     col="magenta", pch=2)
```

Relationship between SAT Scores
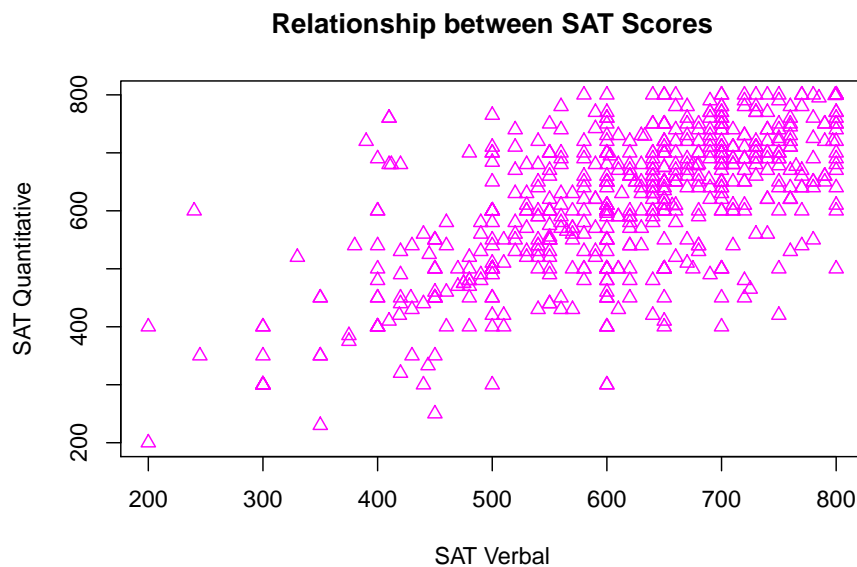
3.4.1.2 Axis limits

R usually does a good job of determining good values for the range of each axis, but if you wish to override the defaults, you can do do using the `xlim` and `ylim` arguments. These arguments accepts 2-element numeric vectors specifying the minimum and maximum values along each axis (in that order). For example, if we wanted to include 0 in the range of SAT scores shown on the plot, we could use `c(0, 800)` for both our axis limits:

```r
# Lines and points
plot(sat.act$SATV, sat.act$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores",
     xlim = c(0, 800),
     ylim=c(0, 800))
```

Relationship between SAT Scores
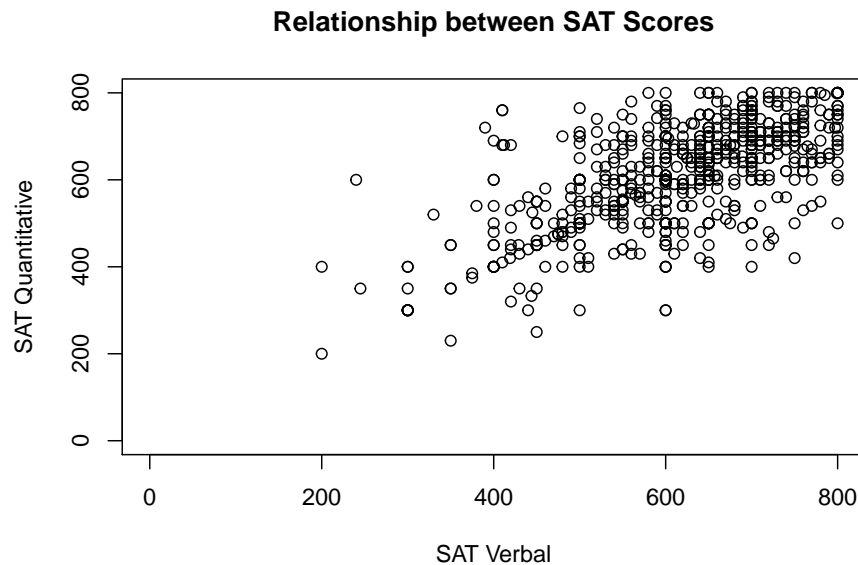
This isn't particularly sensible, because 0 is not a possible score on the SAT, so consider this just a demonstration of R's capabilities.

3.4.1.3 Adding to an existing plot

In many cases, it is useful to have *different* symbols, colors, and linetypes on the same plot - for example, to denote observations from different groups. There is more than one way to achieve this effect, but when you want different symbols, colors, and linetypes on the same plot, often the easiest way to do it is plot each group one at a time.

To do this, you begin with the familiar plot command, but use it to plot only a subset of the data. Then, you can use the `points` or `lines` function to add additional points (or lines) to the plot. We'll demonstrate how you can plot the SAT scores for males and females in different colors by subsetting the dataset, and then using the `plot` and `points` functions:

```r
males <- sat.act[sat.act$gender == 1, ]
females <- sat.act[sat.act$gender == 2, ]
plot(males$SATV, males$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores",
     col="red")
points(females$SATV, females$SATQ, col="blue")
```

**Relationship between SAT Scores**
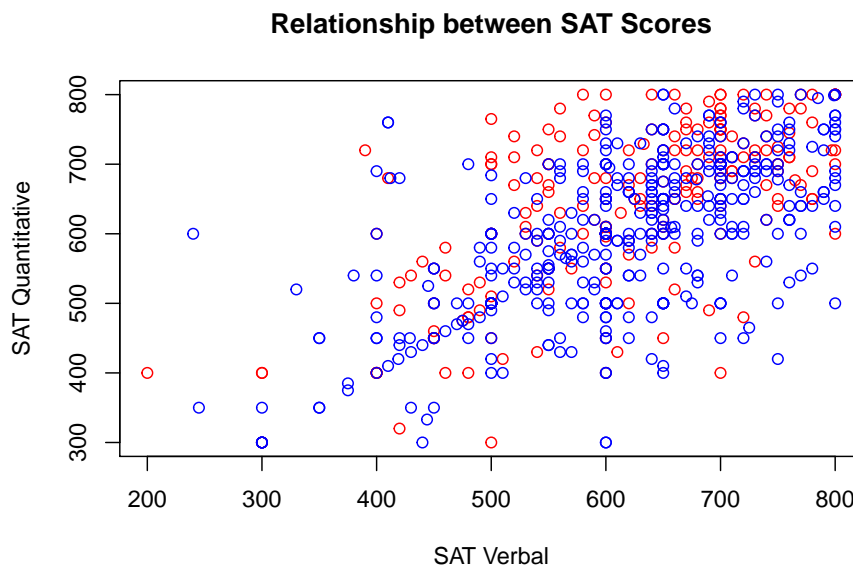


### 3.4.1.4 Adding a legend

If you use multiple symbols, colors, and linetypes on the same plot, you'll also need a legend telling the viewer how to interpret the different symbols, colors, and linetypes. You can add legends to an existing plot using the legend function. Unfortunately, the legend function doesn't know what symbols, colors, and linetypes you've used, or what the different groups in your data are. So, you'll have to re-capitulate this information to the legend function. Make *absolutely* sure that you specify the labels, colors, symbol codes, and linetypes in the exact same order you added them to the plot! You also must specify a pch or linetype value, even if you used the default value in your plot!

Perhaps the trickiest part about using the legend function is figuring out where to place the legend. The location may also be specified by setting the first argument to the legend function to a single keyword from the set "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" or "center". Alternatively, you can get fine-grained control over the placement by providing two values that represent an x,y coordinate for the top-right corner of the legend:

```
plot(males$SATV, males$SATQ,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores",
     col="red")
points(females$SATV, females$SATQ, col="blue")
legend(x=200, y=800,
       legend = c("Males", "Females"),
```

```
        col = c("red", "blue"),
        pch=1)
```

Relationship between SAT Scores
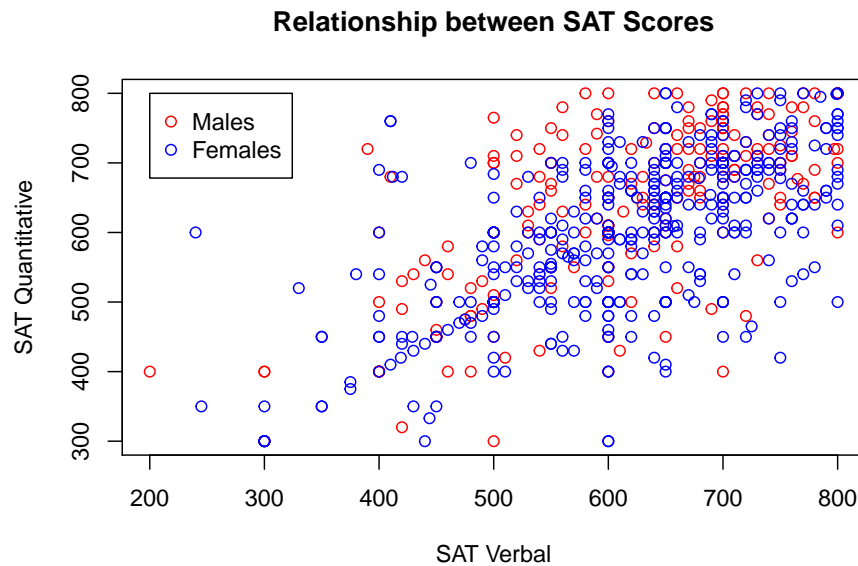
3.4.1.5   Adding a regression line

It is often useful to summarize the linear relationship between variables shown in a scatterplot. It is easy to add the "line of best fit" from a linear regression to your scatter plot using the `lm` function (short for "linear model) and the `abline` function (used for drawing straight lines).

The `lm` function performs regression analysis based on the formula you specify. If you intend on adding the regression line to your scatterplot, we recommend you use the formula interface to the `plot()` function. Instead of using an x vector and a y vector, you can provide a formula of the form `y ~ x`, and the name of the data frame containing your vaiable, just like when using the `lm` function. In this case, you would use the same formula to specify your plot that you use to specify your regression model. Remember of course, that the response variable (i.e., the y variable) goes first in the formula, not second as it would if you were providing two separate vectors.

To summarise, the basic steps are:

1. Perform your linear regression with `lm`, and save the output to a variable
2. Create your scatterplot using the same formula specification as in your linear model from Step 1.
3. Provide the `lm` object you saved in Step #1 to the `abline` function

4. Marvel at your wonderful new plot

```r
regression_results <- lm(SATQ ~ SATV, data=sat.act)
plot(SATQ ~ SATV, data=sat.act,
     xlab = "SAT Verbal", ylab = "SAT Quantitative",
     main = "Relationship between SAT Scores")
abline(regression_results, lwd=2) # lwd = "line width",
```

Relationship between SAT Scores
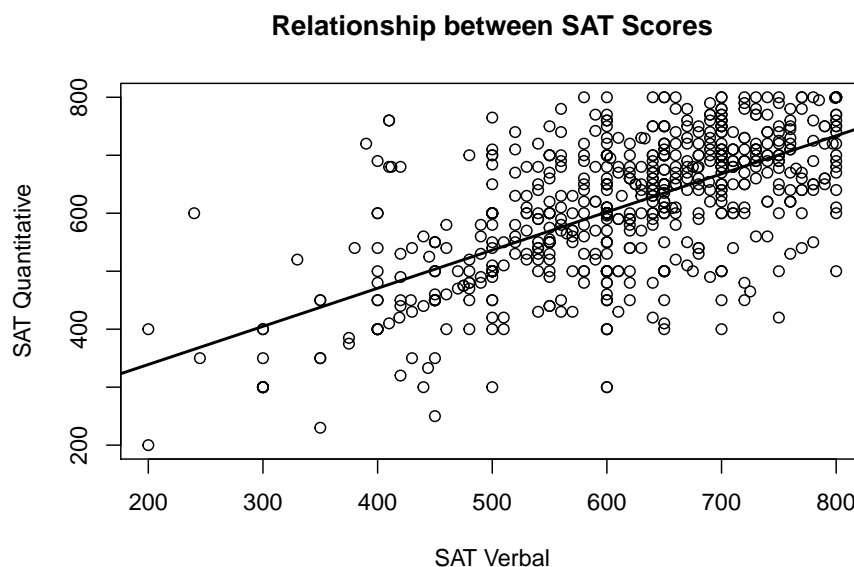
In the call to `abline` above, we have specified the `lwd` argument, short for "line width", a.k.a. thickness. The default value is 1, and we have double the thickness here so the regression line stands out from the points more.

3.4.2 Histograms

You can create a histogram showing the counts or proportion of values from a single variable using the `hist` function.

The `hist` function works with a vector of numeric values, and breaks this vector up into a number of discrete bins. By default, the `hist` function decides what the boundary values of the bins should be, and how many bins should be used, by applying the Sturges algorithm. However, you can manually control where the boundaries are placed by providing a vector of boundary values, or by providing a single value specifying the number of bins that should be used. We demonstrate both techniques below in plotting a histogram of ACT scores:

```
hist(sat.act$ACT, breaks = 30)
```

Histogram of sat.act$ACT
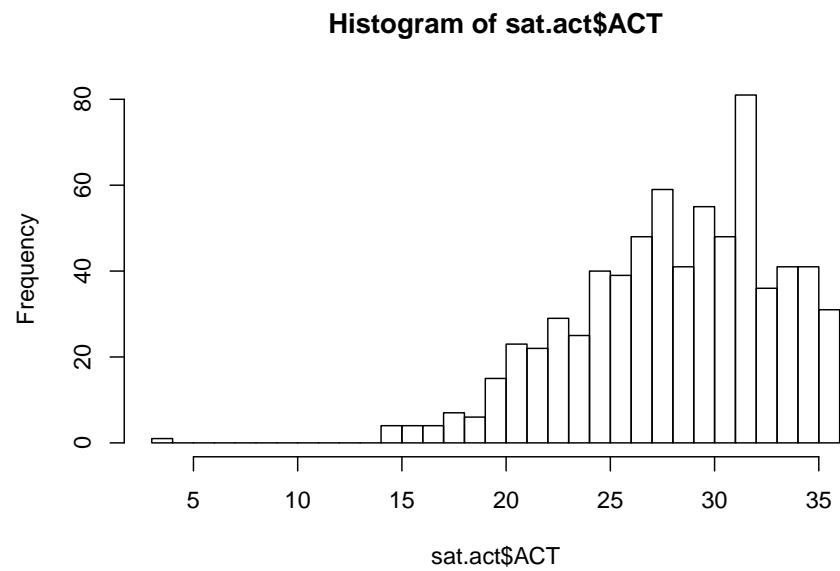

```
bin_boundaries <- seq(0, max(sat.act$ACT), by=2) # from 0 to 36 by 2's
hist(sat.act$ACT, breaks = bin_boundaries)
```

Histogram of sat.act$ACT
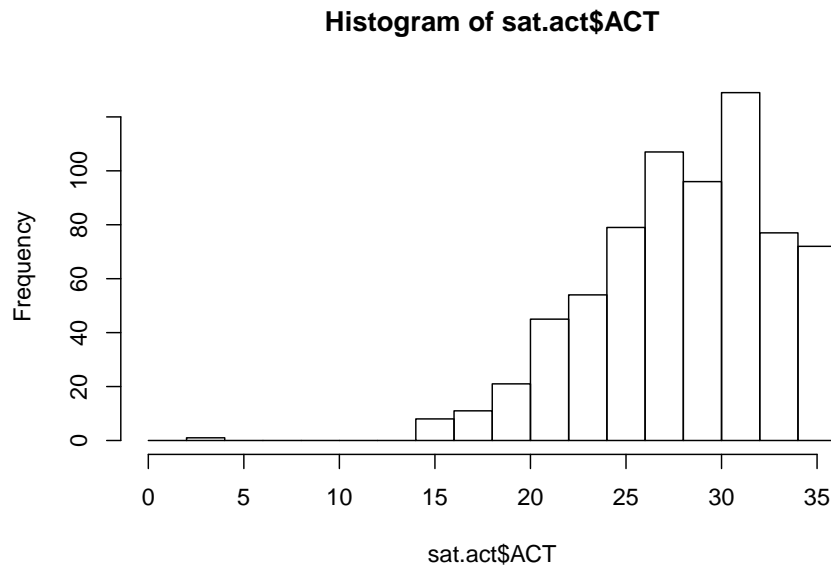
If you want proportions instead of counts, set the `freq` argument to false. You can also use the same `xlab`, `ylab`, and `main` arguments we used with `plot` to get more informative titles/

```
hist(sat.act$ACT, breaks = 30, freq=FALSE,
     main = "Distribution of ACT Scores",
     xlab = "ACT Score")
```

Distribution of ACT Scores
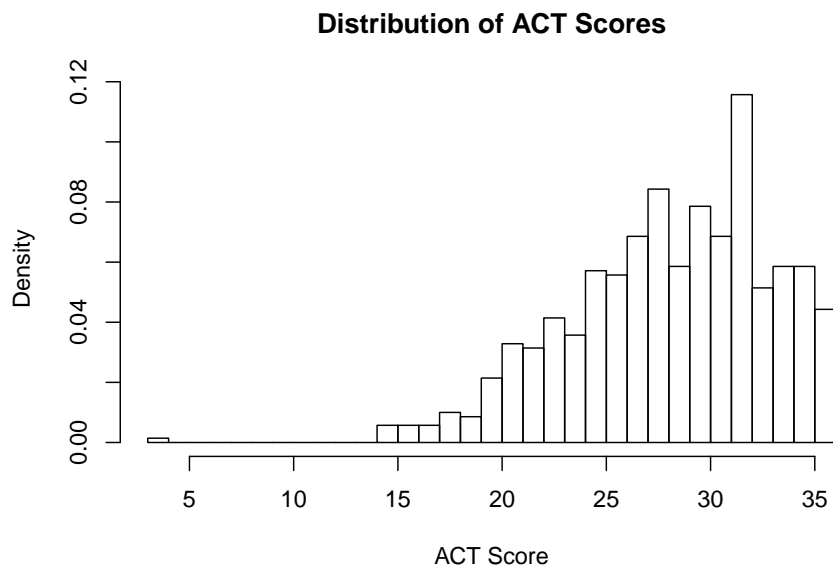

3.4.3   Boxplots

Box plots are a good way to present the shape of distribution of samples from several groups, along with some summary statistics of each group. We can create these plots in R using the `boxplot` function.

If your to-be-plotted data is in a data frame, the best way to instruct the `boxplot` function what variables should be plotted is using a formula. If they are not in a data frame, you can provide each to-be-plotted variable as a separate vector. Here, we'll demonstrate the formula interface by plotting the SAT Verbal scores for respondents with education levels. Remember, when using the formula interface, you also have to tell the `boxplot` function what data frame to look for the variables in!

```
boxplot(SATV ~ education, data=sat.act,
        names=c("< High School", "High School", "Some College",
                "2 Yr. College", "4 Yr. College", "Grad. Work"),
        ylab="SAT Verbal Score", xlab="Education")
```
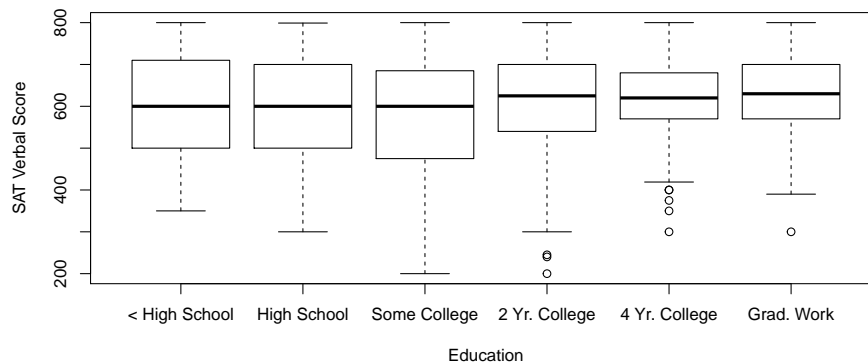
Note the use of the `names` argument to override the labels on each X axis point. De-
pending on how long the names of each variable are, some variable names might be
ommitted from the axis labels, in order to prevent having overlapping characters ob-
scuring the names. This can usually be be remedied by re-sizing the plot window to be
larger, so that all the names can fit in.

3.4.4 Bar plots

Bar plots are used to visualize single summary statistics, like the mean or median across
groups. We can create a bar plot in R using the `barplot` function. We'll demonstrate
this by plotting the mean SAT Verbal score for males and females. The to-be-plotted
value should be specified in a vector or matrix, with one element for each bar you want
plotted

```r
SATV_means <- c(mean(sat.act$SATV[sat.act$gender==1]),
                mean(sat.act$SATV[sat.act$gender==2])
                )
names(SATV_means) <- c("Males","Females")
barplot(SATV_means,
        ylim=c(0,800), ylab="Average SAT Verbal Score",
        main="SAT Verbal Scores for Males & Females"
        )
```

SAT Verbal Scores for Males & Females
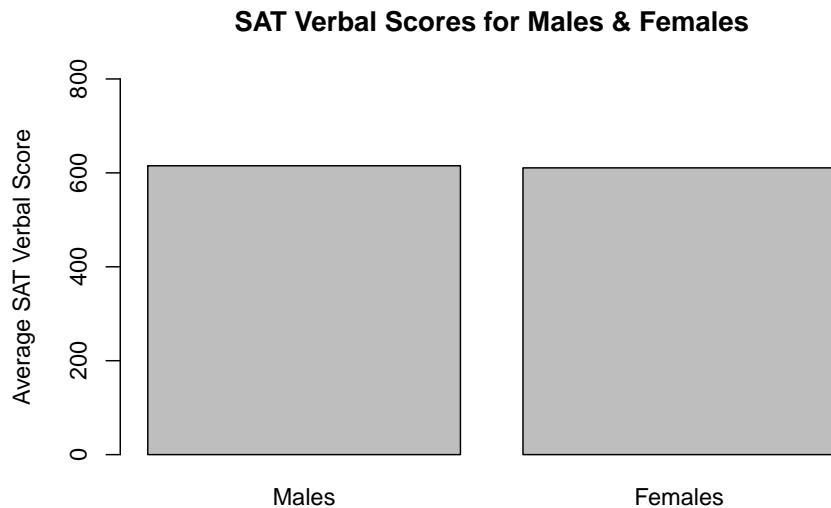
### 3.4.5 Multiple plots in single figure

In many situations, it is useful to have several different plots in a single figure to facilitate comparison. But in the examples we have encountered so far, each time we create a new plot, it has removed the existing one from out figure window. We can change this behavior to allow multiple figures within the same window by using the par function, which changes R's global graphics parameters.

We can set the number of plots that go into a figure window much in the same way we set the dimensions of a matrix - by specifying how many rows of plots there can be, and how many columns of plots there can be. The product of these two values determines the number of plots that are put on a single figure window.

There are two global graphics parameters that allow us to set these dimensions - the mfrow parameter, and the mfcol parameter. The mf stands for "multiple figures", and the row or col part means the figures are filled in by rows, or by columns. Here, we demonstrate how to use the mfrow option to create a 2 by 2 grid of plots, which will allow us to compare the distribution of ACT scores in histograms with different numbers of breaks:

```r
par(mfrow = c(2,2))
hist(sat.act$ACT, breaks = 10, main = "10 breaks")
hist(sat.act$ACT, breaks = 30, main = "30 breaks")
hist(sat.act$ACT, breaks = 50, main = "50 breaks")
```

```r
hist(sat.act$ACT, breaks = 100, main = "100 breaks")
```

10 breaks

30 breaks

50 breaks

100 breaks
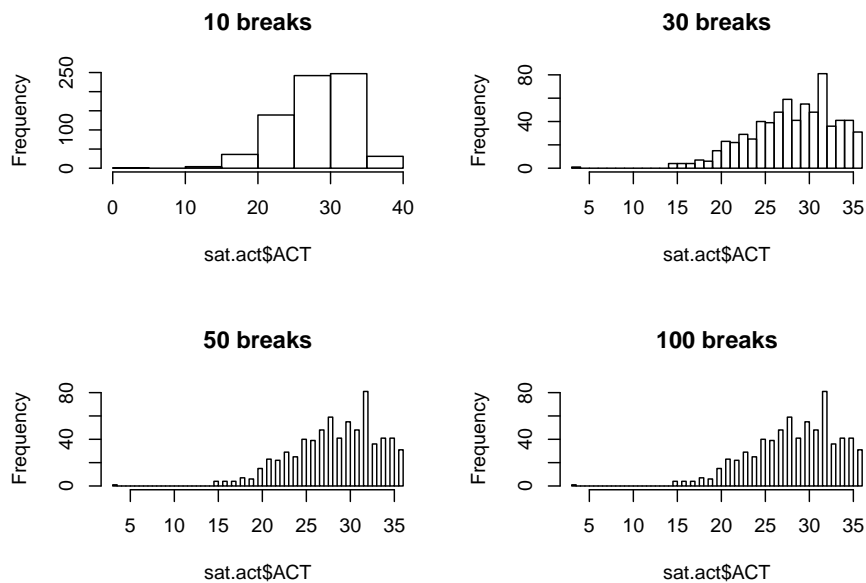
```r
par(mfrow = c(1,1)) # Reset back to default
```

Always make sure to set your `mfrow` or `mfcol` values back to `c(1,1)` afterwards, or *all* subsequent plots will use the same layout!

3.4.6 Saving Plots

Usually the purpose of creating a visualization in R is to include it in some kind of homework, report, or paper. It is easy to save a plot made in R to an image file on your hard drive using the "Export" option in the R Studio plot pane. Simply click the "Export" drop down menu, and select either the "Save to Image", "Save to PDF" or "Copy to Clipboard" options.
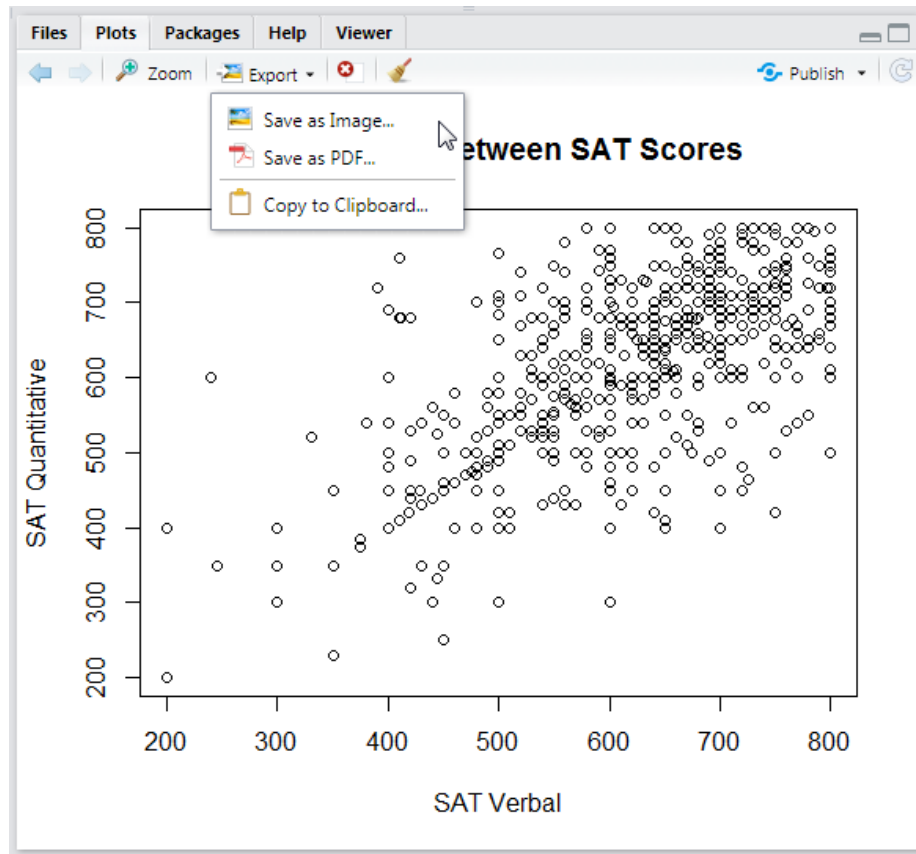
If you choose the "Save as Image" option, you'll get to choose which image format you want to save it as, and what the width and height (in pixels) the image should be saved as.

3.4.7   What about ggplot?

The ggplot2 package is an extremely popular visualization tool, which provides an alternative to the so-called "base graphics" functions described here. The popularity ggplot has achieved is well deserved - its visualizations are attractive, and its data-driven, modular coding style is very powerful. The programming style of ggplot is *very* different from the base graphics style. Perhaps the best way to summarize the ggplot approach is that it is builds on top of the organization of the observations inside a data frame and allows you to "map" the variables in your data frame to the "aesthetics" in a plot (shape, position, color, size, etc.).

We will not cover ggplot here, because it is not a necessity for 240 and 241 (though, in our opinion, a proficient R programmer should know how both ggplot *and* base graphics). However, we will re-iterate the advice the ggplot package authors give for those wishing

to learn how to use ggplot:

> If you are new to ggplot2 you are better off starting with a systematic intro-
> duction, rather than trying to learn from reading individual documentation
> pages. Currently, there are three good places to start:
>
> 1. The data visualisation and graphics for communication chapters in R
>   for data science.
>
> 2. If you'd like to take an interactive online course, try Data visualisation
>   with ggplot2 by Rick Scavetta on DataCamp.
>
> 3. If you want to dive into making common graphics as quickly as possi-
>   ble, I recommend The R Graphics Cookbook by Winston Chang.

3.5 Inferential Statistics

3.5.1 Distribution Functions

The sections below introduce functions for executing common statistical tests. How-
ever, R's family of distribution functions also make it possible to explore the underlying
probability distributions for each test, and to carry out each test by hand. We therefore
begin our demonstration of inferential statistics by giving an overview of these powerful
instructional tools.

The full list of available distributions can be found here. The following table provides an
overview the distributions that are likely to be relevant to you as an instructor, including
the suffix that each distribution is identified by in the functions below:

| Distribution	Suffix	Parameters
Normal	`norm`	mean, SD
Student's T	`t`	df, non-centrality
Chi-Square	`chisq`	df, non-centrality (non-negative)
F	`f`	df_1, df_2, non-centrality
Binomial	`binom`	size (number of observations), probability
Uniform	`unif`	minimum, maximum

View the help documentation for default parameter values and additional details.

Each distribution can be accessed by a set of four functions, each labelled with the
distribution's suffix and a task-specific prefix:

Prefix	Task	Example
d	Get the probability density associated with a point on the x-axis.	`dnorm()`

Prefix	Task	Example
p	Get the probability of a value above or below a point on the x-axis.	`pnorm()`
q	Get the x-axis value marking a cumulative probability.	`qnorm()`
r	Get a random sample from the distribution.	`rnorm()`

These functions are each demonstrated in the following sub-sections using the normal distribution. In these examples, consider that IQ scores are meant to be normally distributed with a mean of 100 and a standard deviation of 15:
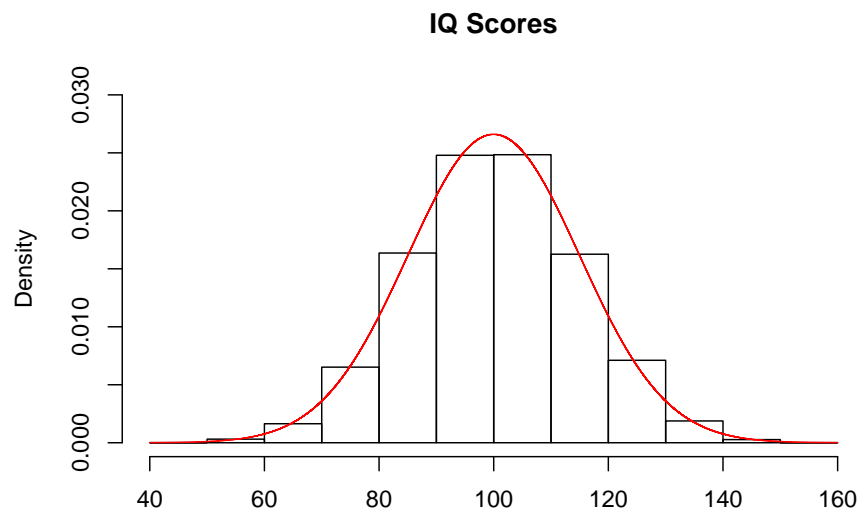
3.5.1.1 Getting Densities

The functions for getting the probability density associated with a point on the x-axis, such as `dnorm()` in the case of the normal distribution, are unlikely to come up in class. However, they may be useful for generating figures such as the one above, which was created with the following code:

```
hist(rnorm(10000, mean=100, sd=15), freq=FALSE,
     ylim=c(0, .03), xlab='', main='IQ Scores')
points(x=seq(40, 160, .001),
       y=dnorm(seq(40, 160, .001), mean=100, sd=15),
       type='l', col='red')
```

Note that the input to `dnorm()` was a sequence of values from 40 to 160, by steps of .001. The output was the probability density for each value in the sequence. These values were then used to create a continuous curve that was added to the histogram of the data, in which the argument `freq=FALSE` ensured that the y-axis denoted probability densities instead of counts. A more detailed tutorial is provided here.

3.5.1.2 Getting Probabilities

You can use `pnorm()` to demonstrate that there is a 9.12% chance of getting an IQ score that is less than or equal to 80:

```r
pnorm(80, mean=100, sd=15)
```

```
## [1] 0.09121122
```

To find the probability of getting a value *greater than* or equal to a cutoff, such as 80, you can either add the argument `lower.tail=FALSE`:

```r
pnorm(80, mean=100, sd=15, lower.tail=FALSE)
```

```
## [1] 0.9087888
```

...or, since the area under the curve sums to 1, you can subtract the output of the previous command from 1:

```r
1-pnorm(80, mean=100, sd=15)
```

```
## [1] 0.9087888
```

The latter method may be better for instructional purposes.

3.5.1.3 Getting Quantiles

You can use `qnorm()` to demonstrate that an IQ score of 124.67 marks the 95th percentile of scores:

```r
qnorm(.95, mean=100, sd=15)
```

```
## [1] 124.6728
```

Similarly, you can add the argument `lower.tail=FALSE` to see the point on the x-axis that marks the *top* 95% of scores:

```r
qnorm(.95, mean=100, sd=15, lower.tail=FALSE)
```

```
## [1] 75.3272
```

Note that subtracting from 1 does not work here as it did with `pnorm()` because the output of `qnorm()` is a quantile, which does not need to sum to 1 as probabilities do.

3.5.1.4 Getting Random Samples

You can use `rnorm()` to randomly sample a class of 50 students from the population of IQ scores:

```
rnorm(50, mean=100, sd=15)
```

```
## [1]  79.97722  94.64574  88.38941 110.46310 122.43748 109.51011  99.32890
## [8] 118.99534  93.90164 122.71929  96.81165  97.69226 133.44205 106.67826
## [15] 129.71835 104.25703 116.49437 112.07160 118.26661 101.65404 117.92000
## [22]  88.69193 103.65274  76.82224  92.64148 106.76429  92.65695 103.94104
## [29]  94.33039 111.14742  92.60759 102.11480  90.63179 123.82827  87.21271
## [36] 118.16661  80.41064 118.34311 108.36824 104.30713  94.55646  95.63206
## [43] 110.99418  90.23955  83.22521 121.29637 115.69330  92.97865  84.05580
## [50]  89.48882
```

3.5.1.5 Running Statistical Tests by Hand

These functions can be used to conduct statistical analyses by hand. Consider a demonstration of a z-test using the functions for the normal distribution. To test whether an IQ score of 135 is significantly different from the mean IQ score, first find the z-score:

```
IQ135_z <- (135-100)/15
IQ135_z
```

```
## [1] 2.333333
```

You could then compare that z-score to the critical z-scores associated with an alpha of .05, here assuming a two-tailed hypothesis:

```
qnorm(.025)
```

```
## [1] -1.959964
```

```
qnorm(.975)
```

```
## [1] 1.959964
```

The z-score of an IQ of 135 is greater than the critical z-score for the upper tail, so we can reject the null hypothesis. Note that for a one-tailed hypothesis, simply use one call to `qnorm()` with .05 or .95 for the lower or upper tail, respectively.

Alternatively, you could find the probability of getting a z-score that is equal to the test score or more extreme (in this case, more positive), and compare that to the alpha value instead of the critical z-score:

```
IQ135_z_prob <- 1-pnorm(IQ135_z)
```

The probability of getting an IQ of 135 is less than alpha (either .025 for a two-tailed hypothesis or .05 for a one-tailed hypothesis), so we reject the null hypothesis.

Note that you could also apply this method by finding the probability of an IQ of 135 given a mean of 100 and standard deviation of 15, and compare that to the alpha value instead:

```
IQ135_prob <- 1-pnorm(135, mean=100, sd=15)
```

This provides the same probability as for the z-score, so we make the same conclusion. However, though it uses one line of code instead of two, you may find that this method is more confusing for some students, particularly if they are used to seeing distributions centered on 0 elsewhere in the course.

This same process can be extended to a range of statistical tests, including t tests, F tests, Chi-Square tests, and binomial tests.

3.5.2 Confidence Intervals

Confidence intervals can be calculated by hand in R using the distribution functions. Recall that the formula for a confidence interval is:

$$CI = \bar{x} \pm (critical\ statistic)\frac{s}{\sqrt{n}}$$

The following demonstration will be conducted using the normal distribution, but the same process can be generalized to other statistics. We will continue with the theoretical distribution of IQ scores used above in which the mean is 100 and the standard deviation is 15.

Imagine that we tested the IQ of 100 students, simulated here with a call to `rnorm()`:

We know that the sample size, n, is 100. To find the 95% confidence interval, we would need to also find the sample mean, the sample standard deviation, and the critical z score that marks the .975 percentile:

```
# Find the mean
mean_IQ <- mean(sample_IQ)

# Find the standard deviation
sd_IQ <- sd(sample_IQ)

# Find the critical z-score (upper tail)
z_crit <- qnorm(.975)
```

Note that we only need to find the z-score that marks the .975 percentile because the normal distribution is symmetric, so the critical z-score for the .025 percentile is simply the negative of this value. You can, however, choose to ask your students to find both separately if you think it would help from an instructional standpoint.

Then, we combine all of these ingredients to find the upper and lower limits of the confidence interval:

```r
# Find the lower limit
CI_lower <- mean_IQ - z_crit*(sd_IQ/sqrt(100))
CI_lower
```

```
## [1] 96.68746
```

```r
# Find the upper limit
CI_upper <- mean_IQ + z_crit*(sd_IQ/sqrt(100))
CI_upper
```

```
## [1] 102.1662
```

The confidence interval ranges from 98.41 to 104.88, suggesting that if this experiment had been repeated over and over an infinite number of times, the true population mean would be within that range 95% of the time.

Note that while calculating confidence intervals by hand is a useful instructional tool, some functions for statistical tests do give the confidence interval "for free", such as `t.test()` and `binom.test()`. Make sure to be clear with students when it is acceptable to get confidence intervals from these functions to complete assignments.

3.5.3 `t.test()`

The `t.test()` function can be used to execute one-sample, paired, or two-sample t-tests. Each case is demonstrated below, but first note three general arguments that can be used to augment the test:

Argument	Values
`alternative`	Set to `two.sided` for a two-sided test, `less` for the lower tail, and `greater` for the upper tail; the default is `two.sided`.
`var.equal`	Set to TRUE to assume equal variance and FALSE otherwise; the default is FALSE.
`conf.level`	Provide a number between 0 and 1 to specify the range for the confidence interval; the default is .95.

3.5.3.1 One-sample t-tests

To demonstrate a one-sample t-test, imagine that we are curious whether the average verbal SAT score in the `sat.act` data set is different from 0. We could test that question with the following call:

```r
t.test(sat.act$SATV)
```

```
##
##  One Sample t-test
##
```

```
## data:  sat.act$SATV
## t = 143.47, df = 699, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##   603.8560 620.6126
## sample estimates:
## mean of x
##   612.2343
```

The output of `t.test()` gives us a nice summary of the results, including the t-statistic, degrees of freedom, p-value, and 95% confidence interval. The p-value is essentially 0, so we can reject the null hypothesis that the average verbal SAT score is 0.

If we wanted to instead see if the average verbal SAT score was different from 650, we could change the null comparison value by setting the `mu` parameter:

```
t.test(sat.act$SATV, mu=600)
```

```
##
##  One Sample t-test
##
## data:  sat.act$SATV
## t = 2.867, df = 699, p-value = 0.004269
## alternative hypothesis: true mean is not equal to 600
## 95 percent confidence interval:
##   603.8560 620.6126
## sample estimates:
## mean of x
##   612.2343
```

The p-value is 0.004, so we can reject the null hypothesis that the average verbal SAT score is 600. Instead, it appears to be higher.

3.5.3.2   Paired t-tests

To demonstrate a paired t-test, imagine that we are curious as to whether verbal SAT scores are different from quantitative SAT scores. We can conduct this in one of two ways.

First, we could calculate the difference score for each pair, then run a one-sample t-test comparing the average difference score to 0:

```
# Calculate difference scores
sat.act$SAT_diff <- sat.act$SATV-sat.act$SATQ

# Run a one-sample t-test
t.test(sat.act$SAT_diff)
```

```
##
```

```
##  One Sample t-test
##
## data:  sat.act$SAT_diff
## t = 0.57483, df = 686, p-value = 0.5656
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -5.116109  9.351917
## sample estimates:
## mean of x
##  2.117904
```

The p-value is .566, so we fail to reject the null hypothesis that the difference between verbal and quantitative SAT scores are approximately zero.

In the second method, we could have `t.test()` compute the difference scores for us by specifying both samples and setting the `paired` argument to TRUE:

```
t.test(sat.act$SATV, sat.act$SATQ, paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  sat.act$SATV and sat.act$SATQ
## t = 0.57483, df = 686, p-value = 0.5656
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.116109  9.351917
## sample estimates:
## mean of the differences
##                2.117904
```

We get exactly the same result. However, note that though this method uses fewer lines of code it risks obscuring the role of difference scores for the student.

3.5.3.3 Two-sample t-tests

To demonstrate a two-sample t-test, imagine that we are curious as to whether male and female students reported different quantitative SAT scores.

The function call is very similar to the second method for conducting a paired t-test, but we set the `paired` argument to FALSE instead:

```
t.test(sat.act$SATQ[sat.act$gender==1], sat.act$SATQ[sat.act$gender==2], paired=FALSE)
```

```
##
##  Welch Two Sample t-test
##
## data: sat.act$SATQ[sat.act$gender == 1] and sat.act$SATQ[sat.act$gender == 2]
## t = 4.3544, df = 492.94, p-value = 1.624e-05
```

```
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   21.88433 57.87165
## sample estimates:
## mean of x mean of y
##   635.8735  595.9955
```

The p-value is approximately .00002, so we can reject the null hypothesis that male and female students report essentially the same quantitative SAT scores. Instead, the mean score for males appears to be higher.

Note that the default value for the `paired` argument is FALSE, so running the above call with the `paired` argument omitted would produce the same result. However, you might want to encourage students to always specify the argument for a t-test with two samples so that they can be sure they are running the intended test.

Further, note that the above example indexes the male and female groups within the call to `t.test()`, but you could also ask students to do this separately beforehand:

```
# Subset the groups
male_SATQ <- sat.act$SATQ[sat.act$gender==1]
female_SATQ <- sat.act$SATQ[sat.act$gender==2]

# Run a two-sample t-test
t.test(male_SATQ, female_SATQ, paired=FALSE)
```

```
##
##  Welch Two Sample t-test
##
## data:  male_SATQ and female_SATQ
## t = 4.3544, df = 492.94, p-value = 1.624e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   21.88433 57.87165
## sample estimates:
## mean of x mean of y
##   635.8735  595.9955
```

3.5.4 `aov()`

The `aov()` function can be used to execute an ANOVA. The basic call includes one argument, which is a formula object. Whether `aov()` is used to run a one-way, two-way, or multi-way ANOVA, with or without interaction terms, is based entirely on the formula.

3.5.4.1 One-way ANOVA

To demonstrate a one-way ANOVA, imagine that we are curious as to whether verbal SAT
scores vary across education level:

```r
aov(sat.act$SATV ~ sat.act$education)
```

```
## Call:
##    aov(formula = sat.act$SATV ~ sat.act$education)
##
## Terms:
##                   sat.act$education Residuals
## Sum of Squares                19247   8890899
## Deg. of Freedom                   1       698
##
## Residual standard error: 112.8613
## Estimated effects may be unbalanced
```

The standard output of aov does not contain readily interpretable results. To get the
ANOVA table, it is better to save the output of aov(), then call summary() on that
saved object:

```r
educ_aov <- aov(sat.act$SATV ~ sat.act$education)
summary(educ_aov)
```

```
##                     Df  Sum Sq Mean Sq F value Pr(>F)
## sat.act$education    1   19247   19247   1.511  0.219
## Residuals          698 8890899   12738
```

Now we can see that the p-value is greater than an alpha of .05, so we fail to reject the
null hypothesis that verbal SAT scores are equal across levels of education.

We might neaten up our code a bit by using the data argument to specify the data frame
that our variables belong to. This allows us to just use the column names in the formula
object:

```r
educ_aov <- aov(SATV ~ education, data=sat.act)
summary(educ_aov)
```

```
##               Df  Sum Sq Mean Sq F value Pr(>F)
## education      1   19247   19247   1.511  0.219
## Residuals    698 8890899   12738
```

3.5.4.2 Two-way ANOVA

To demonstrate a two-way ANOVA, imagine that we are curious as to whether quantita-
tive SAT scores vary across levels of education and gender:

```r
educ_gender_aov <- aov(SATQ ~ education + gender, data=sat.act)
summary(educ_gender_aov)
```

```
##                  Df  Sum Sq Mean Sq F value   Pr(>F)
## education     1   10998   10998   0.845    0.358
## gender        1  261978  261978  20.133 8.47e-06 ***
## Residuals   684 8900522   13012
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 13 observations deleted due to missingness
```

The p-value for education is .358, so we fail to reject the null hypothesis that quantitative SAT scores are equal across levels of education. The p-value for gender, however, is approximately zero, so we can reject the null hypothesis that quantitative SAT scores are equal across genders. To determine which gender has the highest scores, you might have your students find the means or create a barplot.

The two-way ANOVA that we just ran assumed an additive model, but perhaps we think that the effect of gender on quantitative SAT scores is different across different levels of education. To test this hypothesis with the interactive model, change the + symbol between the independent variables to an *:

```
educ_gender_aov_int <- aov(SATQ ~ education * gender, data=sat.act)
summary(educ_gender_aov_int)
```

```
##                     Df  Sum Sq Mean Sq F value   Pr(>F)
## education        1   10998   10998   0.844    0.358
## gender           1  261978  261978  20.111 8.57e-06 ***
## education:gender  1    3396    3396   0.261    0.610
## Residuals      683 8897126   13027
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 13 observations deleted due to missingness
```

The p-value for the interaction term is .610, so we fail to reject the null hypothesis that the effect of gender on quantitative SAT scores is the same across levels of education.

3.5.5 `cor.test()`

The `cor.test()` function can be used to test the significance of a correlation. The correlation is measured by Pearsons's r by default, but note that it can also be set to Kendall's τ or Spearman's ρ with the `method` argument.

To demonstrate, imagine that we are curious as to whether verbal and quantitative SAT scores are linearly related:

```
cor.test(sat.act$SATV, sat.act$SATQ)
```

```
##
##   Pearson's product-moment correlation
##
## data:  sat.act$SATV and sat.act$SATQ
```

```
## t = 22.05, df = 685, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.5983352 0.6860379
## sample estimates:
##       cor
## 0.6442999
```

The p-value is essentially zero, so we can reject the null hypothesis that the correlation between verbal and quantitative SAT scores is zero. The estimate instead is .644, indicating a positive linear relationship.

This test was run with default parameter values for the alternative hypothesis and confidence level, which can be modified as follows:

Argument	Values
alternative	Set to `two.sided` for a two-sided test, `less` for the lower tail, and `greater` for the upper tail; the default is `two.sided`.
conf.level	Provide a number between 0 and 1 to specify the range for the confidence interval; the default is .95.

3.5.6 `binom.test()`

The `binom.test()` function can be used to test the significance of a proportion calculated from a binary variable. The test assumes a null proportion of .5, but this can be modified with the `p` argument.

To demonstrate, imagine that we are curious as to whether there are more males than females reporting their test scores.

We first need to find the number of males and females in the data set, which we can do by creating a table of the gender variable:

```
gender_table <- table(sat.act$gender)
```

We can then call `binom.test()` on the table we created to test whether the proportion of males (the first category in the table) is significantly different from .5:

```
binom.test(gender_table)
```

```
##
##  Exact binomial test
##
## data:  gender_table
## number of successes = 247, number of trials = 700, p-value =
## 5.974e-15
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
```

```
##  0.3174274 0.3895333
## sample estimates:
## probability of success
##               0.3528571
```

The p-value is essentially zero, so we can reject the null hypothesis that the proportion of males is .5; instead, it seems there are more females than males.

Note that the same test can be run by specifying only the number of males in the first argument, then specifying the total number of observations (i.e., the number of rows in sat.act) using the n argument:

```r
# Find the number of males
n_males <- length(sat.act$gender[sat.act$gender==1])

# Run the binomial test
binom.test(n_males, n=nrow(sat.act))
```

```
##
##  Exact binomial test
##
## data:  n_males and nrow(sat.act)
## number of successes = 247, number of trials = 700, p-value =
## 5.974e-15
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
##  0.3174274 0.3895333
## sample estimates:
## probability of success
##               0.3528571
```

The first method is likely simpler if you are sticking to examples or assignments with real data, but the second method is more amenable to hypothetical scenarios, such as asking students to test whether a coin is fair if it produced 60 heads out of 100 flips.

This test was run with default parameter values for the alternative hypothesis and confidence level, which can be modified as follows:

Argument	Values
alternative	Set to two.sided for a two-sided test, less for the lower tail, and greater for the upper tail; the default is two.sided.
conf.level	Provide a number between 0 and 1 to specify the range for the confidence interval; the default is .95.

3.5.7 `chisq.test()`

The `chisq.test()` function can be used to execute either a Chi-Square test for goodness-of-fit or a Chi-Square test of independence, determined by whether a table with one or two groups is supplied, respectively.

3.5.7.1 Chi-Square test of Goodness-of-Fit

To demonstrate the Chi-Square goodness-of-fit test, imagine that we are curious as to whether there are equal numbers of people in each education level measured in the `sat.act` data set.

We first need to find the number of respondents in each education level, which we can do by creating a table of the `education` variable:

```r
educ_table <- table(sat.act$education)
```

We can then call `chisq.test()` on the table we created to test whether there are equal proportions of respondents in each education level:

```r
chisq.test(educ_table)
```

```
##
##  Chi-squared test for given probabilities
##
## data:  educ_table
## X-squared = 343.66, df = 5, p-value < 2.2e-16
```

The p-value is approximately zero, so we can reject the null hypothesis that the proportions are equal. From the table, we can see that respondent are more likely than not to have recieved at least some higher education.

By default, the Chi-Square test of independence tests the null hypothesis that the proportions are equal across groups, but this can be changed by providing a vector of proportions to the p argument. For instance, imagine that we wanted to test whether the proportions were equal to those from respondents in a neighboring state, which were much less likely to have been educated past high school:

```r
chisq.test(educ_table, p=c(.25, .25, .125, .125, .125, .125))
```

```
##
##  Chi-squared test for given probabilities
##
## data:  educ_table
## X-squared = 661.41, df = 5, p-value < 2.2e-16
```

The p-value is approximately zero, so we can reject the null hypothesis that the proportions are the same as for the neighboring state.

3.5.7.2   Chi-Square test of Independence

To demonstrate the Chi-Square test of independence, imagine that we are curious as to whether the proportions of males and females reporting their test scores are equal across levels of education.

We first need to find the number of males and females in each level of education, which we can do by creating a table of the gender and education variables:

```r
educ_gender_table <- table(sat.act$education, sat.act$gender)
```

We can then call chisq.test() on the table we created to test whether there are equal proportions of males and females in each education level:

```r
chisq.test(educ_gender_table)
```

```
## 
##  Pearson's Chi-squared test
## 
## data:  educ_gender_table
## X-squared = 16.085, df = 5, p-value = 0.006605
```

The p-value is .007, so we can reject the null hypothesis that level of education is independent of gender. Instead, gender seems to be more balanced in the lower education levels, whereas there are more females than males in the higher education levels.

Reproducibility

The R session information when compiling this book is shown below:

```r
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
## 
## Matrix products: default
## 
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
## 
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
## 
## other attached packages:
```

```
## [1] psych_1.8.12
##
## loaded via a namespace (and not attached):
## [1] Rcpp_1.0.1     bookdown_0.11  lattice_0.20-35 packrat_0.4.9-3
## [5] digest_0.6.19  grid_3.5.1      nlme_3.1-137    magrittr_1.5
## [9] evaluate_0.14  stringi_1.4.3   rmarkdown_1.13  tools_3.5.1
## [13] foreign_0.8-70 stringr_1.3.1   xfun_0.6        parallel_3.5.1
## [17] compiler_3.5.1 mnormt_1.5-5    htmltools_0.3.6 knitr_1.22
```

Chapter 4

Tools & Advice for Instructors

4.1 Best Practices

4.1.1 Commands vs. Buttons

Though we strongly encourage you to use R through R Studio, we also encourage you and your students to rely on commands instead of button presses where possible. For instance, it is possible to save plots by exporting them from the Plots pane in R Studio, but it is also possible to save them with commands such as `pdf()` or `png()`. There are several benefits to prioritizing commands over button presses. First, it has the practical effect of reinforcing a command-line approach to programming your analyses, which may be new for you and is very likely new for your students. Second, and perhaps more importantly, it encourages reproducibility. If your students document all of the actions that they took in completing an assignment by having a command for each one, it will be easier for you to make sure that they did everything on their own, correct their habits where needed, and find the source of an error. It will also help you to reproduce your own work from semester to semester if your assignments are coded in R.

4.1.2 Naming Objects

There are relatively few constraints to naming objects in R compared to other statistical software packages. We make several recommendations here for good naming practices.

The "naming conventions" of a particular programming language refer to the formatting guidelines for object names. There are no agreed-upon naming conventions in R, but a good overview can be found here. In general, the best advice is to pick a convention that appeals to you and use it consistently. We recommend the "underscore_separated" scheme. Because period separation can sometimes refer to a different process in R, we do not recommend using the "period.separated" scheme.

Regardless of the convention, it is also important to choose object names that are informative. That is, someone with little knowledge of the script, including some future version of yourself, should be able to get a general idea of what data an object contains based on the name. For instance, simply naming all versions of a data frame `data`, `data_2`, `data_3`, etc., is not informative because it is unclear what changes have been made between versions.

4.1.3   Scripting

More than just a collection of commands, the script is an excellent tool for both both reproducibility and documentation of an analysis. It is therefore worthwhile to teach students good habits for writing scripts.

First and foremost, scripts should include *all of the commands that you need to conduct an analysis, and none of the commands that you don't*. That is, you should be able to run all of the lines in a script in order and get the same result each time (with the obvious exception of things like drawing random samples). To that end, all the commands necessary for an analysis should be in the proper sequence, and deny commands that are not necessary for the analysis but kept for posterity should be "commented out" by adding the # symbol at the start of the line.

Second, scripts should be well-documented. Recall that you can add comments to any line by preceding the comment with the # symbol. Comments are an efficient way to explain to someone unfamiliar with the script what the intention is for each section of code. Ample documentation is not only a good programming habit to teach your students, it will also likely be very useful in helping you decipher their code.

4.1.4   Formatting Requirements for Assignments

We encourage you to give your students clear formatting guidelines for assignments that require a significant amount of R programming to complete. The specific requirements that you decide on should be based on (1) the extent to which you want them to leave the class with generalizeable programming skills, and (2) the amount of detail that you would like from your students in terms of their code.

Asking students to print the contents of their console will allow you to see all of the commands that they ran, the order in which they ran them, as well as the output and any error messages, but it will be quite a lot of information for you to sort through yourself. At the other extreme, asking for only the formatted results in a Word document makes it difficult to judge the quality of their code, which may be a primary goal of the assignment. A nice compromise is to ask for an electronic version of their script with all the necessary commands so that you can reproduce their results. This can either act as the main document or as a supplement to a formatted report.

It has also become increasingly popular for instructors to provide students with prepared R Markdown (.Rmd) files with fill-in-the-blank style prompts for students to enter their

code. This produces a nicely formatted document with their code and output, which may be easier for you administratively. However, this can also obscure the coding process for students and limit their independence. For this reason, we recommend against using prepared R Markdown files, and encourage you to teach students how to create and maintain their own scripts instead.

4.2   Error Messages

All messages printed to the console in red text are meant to get your attention, but they differ in urgency. A message with `Error` printed at the beginning means that there was a problem that prevented the command from being evaluated, and you will not be able to run that command until you fix the problem. A message with `Warning` printed at the beginning means that the command *was* evaluated, but it might not have worked the way you intended, or R might have made some decision for you in order to execute the command. All other messages can generally be taken as updates on the status of your command.

If you receive an error message that is unclear, the best way to determine the cause is to Google it. A list of the most common errors is provided here for convenience.

There is no output from a command, and there is a + in the console instead of a >.

The user has not completed the command, most likely by forgetting to close a quotation (or being inconsistent with the use of single vs. double quotations) or a set of parentheses. You can either add the end quotation or parenthesis if it is the final character, or press the escape button and start again.

`Error: unexpected ')' in '___'`

The user has included one too many end parentheses.

`Error: Object '___' not found`

The user has tried to call an object that is not in the environment. This may be because there is a typo in the name of the object, or because the object was never assigned to a label with the `<-` operator.

`Error in ___ : non-numeric argument to binary operator`

The user is most likely trying to conduct a mathematical operation on data that is of a non-numeric class, such as character or factor. If the data should be numeric, try

overwriting it with `as.numeric()`.

Error in ___(x) : could not find function '___'

The user is attempting to use a function that is not currently available in R, either because there is a typo in the name of the function or because the function's package has not been loaded in this session.

Error in ___ : cannot open the connection

The user is attempting to read or write a file at a location on the hard drive that does not exist, likely due to a typo in the file path or file name.

Error in ___ : undefined columns selected** or **Error in ___: subscript out of bounds

The user is attempting to index an area of an object that does not exist, for instance, the ninth column of a seven-column data frame.

Error: unexpected symbol in '___'

R cannot interpret part of the command, likely because the user has forgotten a comma between function arguments.

4.3 Data Sets

This section includes an overview of some selected data sets that may be useful for in-class demonstrations or assignments. The first section includes data sets that are "built-in" as part of the base R `datasets` package, whereas the remaining sections review data sets from other selected packages that need to be installed. Of course, this list is non-exhaustive. The packages used here have more data sets than those we have chosen to highlight, and there are many more interesting data sets in other packages we have not highlighted. So, feel free to explore more options.

To see all of the data sets available from all *loaded* packages, including those are built-in with base R o, enter the following command:

```r
data()
```

You should see a list of all the available data sets with a brief summary of each, separated by the package they belong to. To see a list of datsets in *all* installed packages (loaded or not), run the command `data(package = .packages(all.available = TRUE))`.

4.3.1 How to Handle $\mathtt{NA}$ Values

Most of the data sets below include at least some missing data, marked as NA, in at least one variable. Because, as you know, missing data is a problem that is sure to appear in the wild for students who continue on a research path, you may choose to teach students how to handle NA values themselves. Alternatively, you may choose to remove NA values on your own for simplicity, either by removing them from the data set beforehand (in which case the data would need to be saved as a .CSV file for the students to read in later), or providing students with code to run before beginning the assignment. Regardless, this section reviews how to handle NA values.

As mentioned previously, NA values can be identified with the is.na() function. For instance, imagine that we want to identify the missing quantitative SAT scores in the sat.act data set. We can first call is.na() on the variable to create a logical vector marking each NA as TRUE:

```r
is.na(sat.act$SATQ)
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [177] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [188] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [199] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [210] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [221] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [232] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [243] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [254] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [265] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [276] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [287] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [298] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [309] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
## [320] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [331] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [342] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [353] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [364]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [375] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [386] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [397] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [408] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [419]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [430] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [441] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [452] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [463] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [474] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [485] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [496] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [507] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [518] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [529] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [540] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [551] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [562] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [573] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [584] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [595] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [606] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [617] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [628] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [639]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [650] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [661] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [672] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [683] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [694] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

This logical vector can then be used to index the variable in question:

```
sat.act[is.na(sat.act$SATQ), 'SATQ']
```

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

The previous command returns the NA values because we have asked R to show us all of the values in sat.act$SATQ variable that are coded as NA. However, we are often more interested in the values that are NOT missing than those that are. To find the *non-missing* values, we can reverse the logical vector by adding the negation operator, !, to the beginning of the call to is.na():

```
sat.act[!is.na(sat.act$SATQ), 'SATQ']
```

```
##   [1] 500 500 470 520 550 640 500 560 600 800 710 600 600 725 630 590 650
##  [18] 620 580 680 450 500 700 600 540 650 700 385 450 500 500 710 710 570
##  [35] 580 600 600 570 540 690 610 610 700 610 740 540 650 600 440 690 780
##  [52] 600 590 760 740 700 650 640 450 760 700 750 650 760 550 680 530 610
##  [69] 600 550 450 740 530 750 700 565 590 590 375 550 500 500 760 580 590
##  [86] 700 530 550 700 250 720 690 700 760 600 720 600 729 600 660 650 750
## [103] 660 680 700 610 650 550 710 480 740 720 630 560 700 700 500 660 500
## [120] 500 550 650 800 400 600 730 560 700 500 500 500 570 400 790 600 600
## [137] 620 700 720 680 610 690 770 770 550 530 720 320 800 800 600 800 700
## [154] 700 745 500 630 465 500 800 350 600 650 760 720 700 750 695 680 670
## [171] 540 500 500 800 680 640 600 680 600 650 580 400 750 678 530 600 675
## [188] 630 560 575 650 500 350 700 610 600 650 600 678 720 660 650 660 450
## [205] 670 620 480 500 555 800 710 630 770 710 600 560 550 350 450 460 700
## [222] 540 680 620 650 620 580 660 600 430 580 670 480 600 650 500 490 640
## [239] 570 680 720 500 660 600 570 650 550 200 690 460 525 675 700 700 650
## [256] 500 570 600 430 540 300 500 700 760 770 600 500 500 700 700 475 400
## [273] 750 600 650 480 520 720 300 780 520 400 550 450 680 800 725 540 690
## [290] 730 640 700 540 540 610 640 650 420 600 500 480 640 770 450 740 510
## [307] 650 600 700 700 750 450 450 600 750 650 300 700 750 540 300 600 480
## [324] 650 420 730 790 620 350 500 650 300 700 660 540 520 742 400 500 800
## [341] 570 690 680 500 640 700 600 650 520 540 609 800 690 780 640 580 750
## [358] 770 680 500 450 680 600 660 450 230 570 580 600 500 530 710 500 730
## [375] 450 550 760 450 740 770 690 540 650 300 440 420 640 690 400 710 500
## [392] 650 475 730 490 630 635 550 780 450 600 350 520 670 450 500 550 620
## [409] 550 600 740 620 670 560 710 700 480 800 760 650 550 720 650 700 660
## [426] 700 710 650 400 420 770 600 490 400 510 550 480 670 500 750 780 670
## [443] 640 710 750 650 300 700 720 600 600 590 650 500 650 680 400 480 400
## [460] 750 700 590 654 630 600 600 600 680 450 620 590 440 450 720 470 600
## [477] 700 640 500 530 640 799 800 600 800 600 550 600 450 750 560 590 500
## [494] 730 600 430 480 400 500 400 333 450 460 450 700 750 765 500 610 450
## [511] 450 500 600 600 600 670 620 660 700 600 620 680 677 640 596 660 530
## [528] 710 780 710 650 590 680 700 600 500 420 410 650 550 440 600 500 300
## [545] 530 600 400 730 610 600 720 780 490 700 680 400 800 740 760 720 720
## [562] 770 670 620 720 730 690 720 710 700 680 600 740 640 670 800 740 660
## [579] 800 400 640 710 680 680 700 640 690 720 510 690 770 640 500 590 640
## [596] 670 650 600 660 670 500 690 350 600 650 750 300 795 700 600 690 600
## [613] 650 600 710 690 500 700 740 430 684 610 750 430 780 770 600 550 760
## [630] 680 400 800 720 700 800 800 750 670 530 750 620 500 500 530 740 300
## [647] 670 700 690 800 750 500 580 700 600 540 690 600 520 500 720 600 700
## [664] 620 790 700 550 600 410 710 600 760 490 600 500 680 600 690 610 700
## [681] 660 680 500 630 630 780 600
```

Now we've asked R to show us all of the values in the `sat.act$SATQ` variable that are NOT coded as `NA`.

To index the entire `sat.act` data frame so that *all* columns are returned, not just the quantitative SAT scores, use bracket notation without specifying any columns:

```
sat.act[!is.na(sat.act$SATQ), ]
```

```
##       gender education age ACT SATV SATQ
## 29442      2         3  19  24  500  500
## 29457      2         3  23  35  600  500
## 29498      2         3  20  21  480  470
## 29503      1         4  27  26  550  520
## 29504      1         2  33  31  600  550
## 29518      1         5  26  28  640  640
```

Now we have the full `sat.act` data set, but only those rows where the quantitative SAT score is not `NA`. To be complete, we recommend saving this subset as a new object:

```
sat.act_noNA <- sat.act[!is.na(sat.act$SATQ), ]
```

Note that you can also overwrite `NA` values, in cases where it is methodologically appropriate. For instance, perhaps you know that the students with missing quantitative SAT scores are treated as having a score of 0. You could replace the `NA` values with 0 as follows:

```
# First save a new version of the data
sat.act_NA0 <- sat.act

# Recode NA values as 0 in new version
sat.act_NA0[is.na(sat.act$SATQ), 'SATQ'] <- 0
```

Now all missing quantitative SAT scores will be treated as a 0 in future mathematical operations, using this new version of the data. Importantly, the original data frame is also preserved.

4.3.2 Built-In

There are several small data sets included in base R that are useful for quick demos. Since they are part of base R, it is not necessary to load these packages; just type and enter their name in the console to view them.

Note that the descriptions and summary statistics shown for each dataset were computed using the `skim()` function from the `skimr` pacakge.

4.3.2.1 iris

The `iris` data set gives the lengths and widths of the sepals and petals of 50 irises from each of three different species:

```
## Skim summary statistics
##  n obs: 150
```

```
##  n variables: 5
##
## -- Variable type:factor -------------
##  variable missing   n n_unique                        top_counts
##   Species       0 150        3 set: 50, ver: 50, vir: 50, NA: 0
##
## -- Variable type:numeric ------------
##      variable missing   n mean   sd     hist
##  Petal.Length       0 150 3.76 1.77
##   Petal.Width       0 150 1.2  0.76
##  Sepal.Length       0 150 5.84 0.83
##   Sepal.Width       0 150 3.06 0.44
```

4.3.2.2 warpbreaks

The warpbreaks data set gives the number of breaks in each of 54 looms of yarn, varied across two types of wool and three levels of tension:

```
## Skim summary statistics
##  n obs: 54
##  n variables: 3
##
## -- Variable type:factor -------------
##  variable missing  n n_unique             top_counts
##   tension       0 54        3 L: 18, M: 18, H: 18, NA: 0
##      wool       0 54        2       A: 27, B: 27, NA: 0
##
## -- Variable type:numeric ------------
##  variable missing  n   mean   sd     hist
##    breaks       0 54 28.15 13.2
```

4.3.2.3 mtcars

The mtcars data set gives a range of numeric and categorical measurements of 32 cars, including MPG, number of cylinders, horsepower, weight, and transmission type:

```
## Skim summary statistics
##  n obs: 32
##  n variables: 11
##
## -- Variable type:numeric ------------
##  variable missing  n   mean     sd     hist
##        am       0 32   0.41    0.5
##      carb       0 32   2.81   1.62
##       cyl       0 32   6.19   1.79
##      disp       0 32 230.72 123.94
```

```
##        drat       0 32    3.6    0.53
##        gear       0 32   3.69    0.74
##          hp       0 32 146.69  68.56
##         mpg       0 32  20.09    6.03
##        qsec       0 32  17.85    1.79
##          vs       0 32   0.44     0.5
##          wt       0 32   3.22    0.98
```

4.3.2.4 ToothGrowth

The `ToothGrowth` data set gives the tooth lengths of 60 guinea pigs, varied across three vitamin C dosage levels and two dosage delivery methods:

```
## Skim summary statistics
##  n obs: 60
##  n variables: 3
##
## -- Variable type:factor --------------
##  variable missing  n n_unique           top_counts
##      supp       0 60        2 OJ: 30, VC: 30, NA: 0
##
## -- Variable type:numeric -------------
##  variable missing  n   mean    sd      hist
##      dose       0 60   1.17  0.63
##       len       0 60  18.81  7.65
```

4.3.3 FiveThirtyEight

The `fivethirtyeight` package includes data sets used for articles written on Nate Silver's website for data geeks, FiveThirtyEight. From an instructional standpoint, these data sets are useful not only for being particularly engaging but because there is a published article associated with each one.

To make the data sets available, install and load the `fivethirtyeight` package:

```
install.packages('fivethirtyeight')
library(fivethirtyeight)
```

4.3.3.1 bechdel

The `bechdel` data set is a great all-purpose data set with both within- and between-subject continuous and categorical variables. Based on the Bechdel Test for female representation in film, this data set includes 1,794 movies from 1970-2013 along with their test outcome, budget, and gross profits:

```
## Skim summary statistics
##  n obs: 1794
##  n variables: 15
##
## -- Variable type:character -----------
##  variable missing     n min max empty n_unique
##    binary         0 1794   4   4     0        2
##      code         0 1794   8   8     0       85
##      imdb         0 1794   8  10     0     1794
##      test         0 1794   2  16     0       10
##     title         0 1794   1  83     0     1768
##
## -- Variable type:factor --------------
##    variable missing    n n_unique                       top_counts
## clean_test        0 1794        5 ok: 803, not: 514, men: 194, dub: 142
##
## -- Variable type:integer -------------
##      variable missing    n       mean       sd     hist
##        budget        0 1794    4.5e+07 4.8e+07
##  budget_2013         0 1794    5.5e+07 5.5e+07
##  decade_code       179 1794       1.94    0.69
##  period_code       179 1794       2.42    1.19
##         year        0 1794    2002.55    8.98
##
## -- Variable type:numeric -------------
##       variable missing    n       mean         sd     hist
##       domgross       17 1794     6.9e+07  8e+07
##  domgross_2013        18 1794     9.5e+07    1.3e+08
##       intgross       11 1794     1.5e+08    2.1e+08
##  intgross_2013        11 1794 2e+08          2.8e+08
```

4.3.3.2 fandango

The fandango data set shows ratings from 146 movies across the Fandango, IMDB,
Rotten Tomatoes, and Metacritic websites. It is particularly useful for demonstrating
correlation:

```
## Skim summary statistics
##  n obs: 146
##  n variables: 23
##
## -- Variable type:character -----------
##  variable missing    n min max empty n_unique
##      film        0 146   3  63     0      146
##
## -- Variable type:integer -------------
```

```
##                    variable missing   n     mean      sd    hist
##            fandango_votes        0 146  3848.79  6357.78
##        imdb_user_vote_count      0 146 42846.21 67406.51
##               metacritic        0 146    58.81    19.52
## metacritic_user_vote_count      0 146   185.71   316.61
##            rottentomatoes        0 146    60.85    30.17
##          rottentomatoes_user    0 146    63.88    20.02
##
## -- Variable type:numeric -------------
##                   variable missing   n    mean   sd      hist
##          fandango_difference      0 146    0.24 0.15
##        fandango_ratingvalue      0 146    3.85 0.5
##             fandango_stars        0 146    4.09 0.54
##                       imdb        0 146    6.74 0.96
##                  imdb_norm        0 146    3.37 0.48
##            imdb_norm_round        0 146    3.38 0.5
##             metacritic_norm      0 146    2.94 0.98
##       metacritic_norm_round      0 146    2.97 0.99
##             metacritic_user      0 146    6.52 1.51
##         metacritic_user_nom      0 146    3.26 0.76
## metacritic_user_norm_round        0 146    3.27 0.79
##                    rt_norm        0 146    3.04 1.51
##              rt_norm_round        0 146    3.07 1.51
##               rt_user_norm        0 146    3.19 1
##          rt_user_norm_round      0 146    3.23 1.01
##                       year        0 146 2014.88 0.32
```

4.3.3.3 comma_survey

The comma_survey data set has Likert ratings as well as demographic information collected from 1,129 respondents to a survey on the Oxford comma:

```
## Skim summary statistics
##  n obs: 1129
##  n variables: 13
##
## -- Variable type:character -----------
##               variable missing    n min max empty n_unique
##                 gender       92 1129   4   6     0        2
##               location      102 1129   7  18     0        9
##   more_grammar_correct        0 1129  57  58     0        2
##         write_following     36 1129  76  77     0        2
##
## -- Variable type:factor --------------
##               variable missing    n n_unique
##                    age       92 1129        4
```

```
##            care_data        38 1129       4
##     care_oxford_comma       30 1129       4
##   care_proper_grammar       70 1129       5
##             education      103 1129       5
##      household_income      293 1129       5
##                              top_counts
##   45-: 290, > 6: 272, 30-: 254, 18-: 221
##   Not: 403, Som: 352, Not: 203, A l: 133
##   Som: 414, A l: 291, Not: 268, Not: 126
##      Ver: 688, Som: 333, NA: 70, Nei: 26
##    Bac: 344, Som: 295, Gra: 276, NA: 103
##    NA: 293, $50: 290, $10: 164, $25: 158
##
## -- Variable type:logical -------------
##              variable missing     n mean                          count
##  data_singular_plural       38 1129  0.5 FAL: 547, TRU: 544, NA: 38
##    heard_oxford_comma       30 1129  0.6 TRU: 655, FAL: 444, NA: 30
##
## -- Variable type:numeric -------------
##        variable missing     n    mean          sd     hist
##   respondent_id        0 1129 3.3e+09 1072966.47
```

4.3.3.4 avengers

The avengers data set includes the death and revival status for 173 Marvel comic book characters. It largely includes count and categorical data:

```
## Skim summary statistics
##  n obs: 173
##  n variables: 21
##
## -- Variable type:character -----------
##                     variable missing    n min max empty n_unique
##  full_reserve_avengers_intro       14 173   5   6     0       93
##                       gender        0 173   4   6     0        2
##                     honorary        0 173   4  12     0        4
##                   name_alias       10 173   4  35     0      162
##                        notes       98 173  21 255     0       71
##           probationary_intro      158 173   6   6     0       12
##                          url        0 173  36  67     0      173
##
## -- Variable type:integer -------------
##             variable missing   n    mean     sd     hist
##          appearances        0 173  414.05 677.99
##                 year        0 173 1988.45  30.37
##  years_since_joining        0 173   26.55  30.37
```

```
##
## -- Variable type:logical -------------
##  variable missing   n mean                      count
##   current        0 173 0.47    FAL: 91, TRU: 82, NA: 0
##    death1        0 173 0.4   FAL: 104, TRU: 69, NA: 0
##    death2      156 173 0.94  NA: 156, TRU: 16, FAL: 1
##    death3      171 173 1             NA: 171, TRU: 2
##    death4      172 173 1             NA: 172, TRU: 1
##    death5      172 173 1             NA: 172, TRU: 1
##   return1      104 173 0.67 NA: 104, TRU: 46, FAL: 23
##   return2      157 173 0.5    NA: 157, FAL: 8, TRU: 8
##   return3      171 173 0.5    NA: 171, FAL: 1, TRU: 1
##   return4      172 173 1             NA: 172, TRU: 1
##   return5      172 173 1             NA: 172, TRU: 1
```

4.3.3.5 bob_ross

The bob_ross data set is a great tool for demonstrating count data and tables. For each of 403 episodes, the data set marks the presence or absence of a series of possible elements in Bob's painting. Useful combinations include tables of trees, lakes, and snowy mountains:

```
## Skim summary statistics
##  n obs: 403
##  n variables: 71
##
## -- Variable type:character -----------
##  variable missing   n min max empty n_unique
##   episode        0 403   6   6     0      403
##     title        0 403   8  27     0      401
##
## -- Variable type:integer -------------
##            variable missing   n   mean    sd      hist
##         apple_frame        0 403 0.0025 0.05
##     aurora_borealis        0 403 0.005  0.07
##                barn        0 403 0.042  0.2
##               beach        0 403 0.067  0.25
##                boat        0 403 0.005  0.07
##              bridge        0 403 0.017  0.13
##            building        0 403 0.0025 0.05
##              bushes        0 403 0.3    0.46
##               cabin        0 403 0.17   0.38
##              cactus        0 403 0.0099 0.099
##        circle_frame        0 403 0.005  0.07
##              cirrus        0 403 0.069  0.25
##               cliff        0 403 0.02   0.14
```

```
##                 clouds    0 403 0.44    0.5
##                 conifer   0 403 0.53    0.5
##                 cumulus   0 403 0.21    0.41
##                 deciduous 0 403 0.56    0.5
##           diane_andre     0 403 0.0025 0.05
##                 dock      0 403 0.0025 0.05
##     double_oval_frame     0 403 0.0025 0.05
##                 farm      0 403 0.0025 0.05
##                 fence     0 403 0.06    0.24
##                 fire      0 403 0.0025 0.05
##         florida_frame     0 403 0.0025 0.05
##                 flowers   0 403 0.03    0.17
##                 fog       0 403 0.057   0.23
##                 framed    0 403 0.13    0.34
##                 grass     0 403 0.35    0.48
##                 guest     0 403 0.055   0.23
##     half_circle_frame     0 403 0.0025 0.05
##       half_oval_frame     0 403 0.0025 0.05
##                 hills     0 403 0.045   0.21
##                 lake      0 403 0.35    0.48
##                 lakes     0 403 0        0
##           lighthouse      0 403 0.0025 0.05
##                 mill      0 403 0.005   0.07
##                 moon      0 403 0.0074 0.086
##             mountain      0 403 0.4     0.49
##             mountains     0 403 0.25    0.43
##                 night     0 403 0.027   0.16
##                 ocean     0 403 0.089   0.29
##           oval_frame      0 403 0.094   0.29
##           palm_trees      0 403 0.022   0.15
##                 path      0 403 0.12    0.33
##                 person    0 403 0.0025 0.05
##                 portrait  0 403 0.0074 0.086
##   rectangle_3d_frame      0 403 0.0025 0.05
##     rectangular_frame     0 403 0.0025 0.05
##                 river     0 403 0.31    0.46
##                 rocks     0 403 0.19    0.39
##         seashell_frame     0 403 0.0025 0.05
##                 snow      0 403 0.19    0.39
##         snowy_mountain    0 403 0.27    0.44
##           split_frame     0 403 0.0025 0.05
##             steve_ross    0 403 0.027   0.16
##             structure     0 403 0.21    0.41
##                 sun       0 403 0.099   0.3
##           tomb_frame      0 403 0.0025 0.05
##                 tree      0 403 0.9     0.31
```

```
##              trees       0 403 0.84    0.37
##        triple_frame      0 403 0.0025 0.05
##          waterfall       0 403 0.097  0.3
##              waves       0 403 0.084  0.28
##           windmill       0 403 0.0025 0.05
##        window_frame      0 403 0.0025 0.05
##             winter       0 403 0.17    0.38
##         wood_framed      0 403 0.0025 0.05
##
## -- Variable type:numeric -------------
##      variable missing   n mean   sd      hist
##  episode_num        0 403     7 3.75
##       season        0 403    16 8.96
```

4.3.4 psych

In addition to a range of new functions, the psych package includes data sets intended to demonstrate common statistical analyses for psychological research.

To make the data sets available, install and load the psych package:

```r
install.packages('psych')
library(psych)
```

4.3.4.1 Tal.Or

The Tal.Or data set is from Study 2 of Tol-Or, Cohen, Tasfati, & Gunther (2010), which used an experimental manipulation to test the effects of media on the attitudes of 123 respondents, measured with Likert scales:

```
## Skim summary statistics
##  n obs: 123
##  n variables: 6
##
## -- Variable type:numeric -------------
##  variable missing   n  mean   sd      hist
##       age        0 123 24.63 5.8
##      cond        0 123  0.47 0.5
##    gender        0 123  1.65 0.48
##    import        0 123  4.2  1.74
##       pmi        0 123  5.6  1.32
##  reaction        0 123  3.48 1.55
```

4.3.4.2 affect

The affect data set consists of responses from 330 participants to state and trait mood scales given before and after viewing (1) a Frontline documentary about the liberation of a concentration camp, (2) the horror movie Halloween, (3) a National Geographic nature film about the Serengeti, or (4) the comedy movie Parenthood:

```
## Skim summary statistics
##  n obs: 330
##  n variables: 20
##
## -- Variable type:factor --------------
##  variable missing   n n_unique              top_counts
##     Study       0 330        2 fla: 170, map: 160, NA: 0
##
## -- Variable type:integer -------------
##  variable missing   n  mean    sd     hist
##      Film       0 330  2.52  1.13
##       MEQ     160 330 39.36 10.28
##
## -- Variable type:numeric -------------
##   variable missing   n  mean    sd     hist
##        BDI     170 330  0.31  0.28
##        EA1       0 330  9.19  7.11
##        EA2       0 330 11.01  6.85
##        ext       0 330 13.16  4.47
##        imp       0 330  4.29  1.97
##        lie       0 330   2.3  1.48
##        NA1       0 330  3.69  4.31
##        NA2       0 330  4.65  5.19
##       neur       0 330 10.22  5.04
##        PA1       0 330  8.89  6.79
##        PA2       0 330  8.98  6.43
##        soc       0 330  7.51  2.86
##     state1       0 330 40.83 10.76
##     state2     160 330 42.45 10.76
##        TA1       0 330 12.91  4.42
##        TA2       0 330 15.65  4.88
##   traitanx       0 330 39.52  9.77
```

4.3.4.3 epi

The epi data set includes responses to the Eysenck Personality Inventory from 3570 subjects:

```
## Skim summary statistics
```

```
##  n obs: 3570
##  n variables: 57
##
## -- Variable type:integer -------------
##  variable missing    n mean   sd     hist
##        V1     296 3570 1.28 0.45
##       V10      64 3570 1.86 0.35
##       V11      77 3570 1.3  0.46
##       V12      66 3570 1.2  0.4
##       V13      75 3570 1.44 0.5
##       V14      72 3570 1.25 0.43
##       V15      75 3570 1.79 0.41
##       V16      80 3570 1.51 0.5
##       V17      75 3570 1.22 0.42
##       V18      71 3570 1.17 0.37
##       V19      70 3570 1.21 0.41
##        V2      85 3570 1.47 0.5
##       V20      90 3570 1.23 0.42
##       V21      80 3570 1.35 0.48
##       V22     100 3570 1.43 0.49
##       V23      75 3570 1.62 0.49
##       V24      78 3570 1.74 0.44
##       V25      72 3570 1.31 0.46
##       V26      89 3570 1.7  0.46
##       V27     108 3570 1.38 0.49
##       V28      74 3570 1.35 0.48
##       V29      87 3570 1.67 0.47
##        V3      73 3570 1.45 0.5
##       V30      71 3570 1.18 0.39
##       V31      98 3570 1.35 0.48
##       V32      99 3570 1.72 0.45
##       V33      78 3570 1.63 0.48
##       V34      96 3570 1.4  0.49
##       V35      69 3570 1.86 0.35
##       V36     109 3570 1.72 0.45
##       V37      86 3570 1.63 0.48
##       V38      83 3570 1.76 0.43
##       V39      89 3570 1.46 0.5
##        V4      65 3570 1.57 0.5
##       V40      82 3570 1.47 0.5
##       V41      79 3570 1.71 0.45
##       V42      68 3570 1.2  0.4
##       V43      70 3570 1.81 0.39
##       V44      76 3570 1.9  0.3
##       V45      85 3570 1.79 0.41
##       V46      81 3570 1.54 0.5
```

```
##        V47       78 3570 1.76 0.43
##        V48       75 3570 1.18 0.38
##        V49       80 3570 1.17 0.38
##         V5       73 3570 1.2  0.4
##        V50       80 3570 1.46 0.5
##        V51       78 3570 1.76 0.43
##        V52       89 3570 1.65 0.48
##        V53       94 3570 1.58 0.49
##        V54       79 3570 1.42 0.49
##        V55       74 3570 1.52 0.5
##        V56       76 3570 1.57 0.49
##        V57       77 3570 1.8  0.4
##         V6       68 3570 1.43 0.5
##         V7       68 3570 1.41 0.49
##         V8       70 3570 1.65 0.48
##         V9       71 3570 1.46 0.5
```

4.3.4.4 sat.act

The sat.act data set includes self-reported SAT and ACT scores along with demographic information from 700 respondents, making it useful for within- or between-subject comparisons:

```
## Skim summary statistics
##  n obs: 700
##  n variables: 6
##
## -- Variable type:integer -------------
##   variable missing   n    mean     sd    hist
##        ACT       0 700   28.55   4.82
##        age       0 700   25.59   9.5
##  education       0 700    3.16   1.43
##     gender       0 700    1.65   0.48
##       SATQ      13 700  610.22 115.64
##       SATV       0 700  612.23 112.9
```

4.3.5 Simulating your own Data Sets

If none of the available data sets satisfy your needs, or if you simply want to get creative, it is possible to simulate your own data set by drawing random samples from a distribution. This can be a great method of designing engaging data sets. It also ensures that you will have a nice "clean" data set, with no unwanted NA values, balanced or unbalanced designs as needed, and null and significant effects as desired.

To demonstrate how to simulate a numeric variable, the following code simulates income for 1,000 families from a population in which income is normally distributed around

$50,000 with a standard deviation of $5,000:

```
sim_income <- rnorm(n=1000, mean=50000, sd=5000)
hist(sim_income)
```

Histogram of sim_income
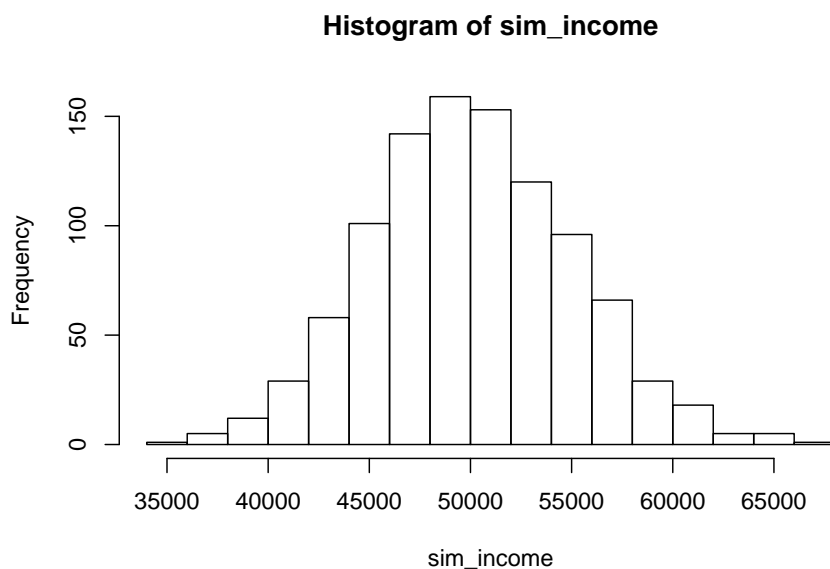
To demonstrate how to simulate a categorical variable, the following code simulates grades for 50 students in a pass/fail course in which there is a 50% chance of passing in the population:

```
sim_grades <- rbinom(n=50, size=1, prob=.5)
sim_grades <- factor(sim_grades, levels=0:1, labels=c('FAIL', 'PASS'))
barplot(table(sim_grades))
```
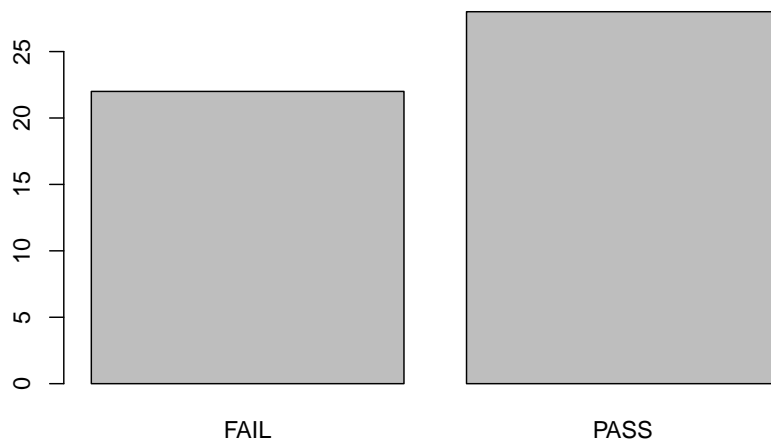
Other potentially useful "population" distributions include `rbeta()`, `rbinom()`, `rchisq()`, `rexp()`, `rf()`, `rgamma()`, `rlogis()`, `rlnorm()`, `rpois()`, `rt()`, and `runif()`. A full list can be found here. See the help documentation for each to see which parameter values can be specified.

Reproducibility

The R session information when compiling this book is shown below:

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
```

```
## [1] stats     graphics grDevices utils     datasets methods   base
##
## other attached packages:
## [1] skimr_1.0.6         psych_1.8.12         fivethirtyeight_0.4.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.1      knitr_1.22      magrittr_1.5    tidyselect_0.2.5
##  [5] mnormt_1.5-5    lattice_0.20-35 R6_2.4.0         rlang_0.3.4
##  [9] stringr_1.3.1   dplyr_0.8.1     tools_3.5.1      parallel_3.5.1
## [13] grid_3.5.1       packrat_0.4.9-3  nlme_3.1-137      xfun_0.6
## [17] cli_1.1.0       htmltools_0.3.6 digest_0.6.19   assertthat_0.2.1
## [21] tibble_2.1.1    crayon_1.3.4    bookdown_0.11   tidyr_0.8.3
## [25] purrr_0.3.2     glue_1.3.1      evaluate_0.14   rmarkdown_1.13
## [29] stringi_1.4.3   pillar_1.4.1    compiler_3.5.1  foreign_0.8-70
## [33] pkgconfig_2.0.2
```