

UMass PBS Instructors R Manual

Andrea Cataldo & Will Hopper

Jun 6, 2019

Contents

Preface	5
0.1 Getting Started	5
0.2 Formatting Conventions	6
0.3 Additional Materials	7
1 Introducing R and RStudio	9
1.1 Pane #1: The R Console	9
1.2 Pane #2: The Editor	10
1.3 Pane #3: Your R Session	12
1.4 Pane #4: Miscellaneous	14
1.5 Conclusion	15
2 R Programming Fundamentals	17
2.1 Classes of Data	17
2.2 Data Structures	21
2.3 Operators	24
2.4 Functions	27
2.5 Data Manipulation	31

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 recommended 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!

0.1 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.

0.1.1 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 computers 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)

0.1.2 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](#)

0.2 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) to 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.

0.3 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 Golemund & 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:

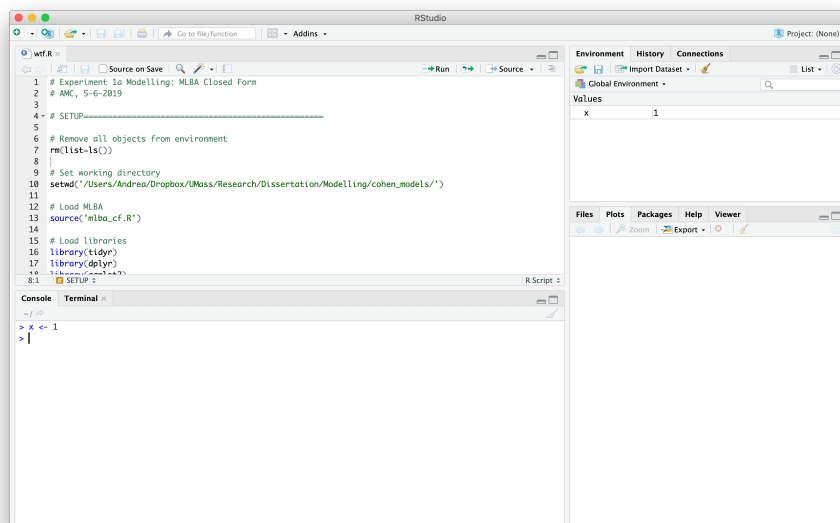
```
sessionInfo()
```

```
## 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 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
- `^` for exponentiation

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

R respects the order of operations (i.e. PEMDAS) by default, so if you want to for a specific operation to be executed first, you need to surround it with parenthesis. To see how grouping with parenthesis 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 you need to repeat the same step again in the future (i.e. re-use 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 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 like 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. And they take all the over 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, new programmers frequently over apply this rule - they 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 be 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.

```
# Hi I am a comment.  
# The R interpreter 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 `<-`, using 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 being:

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

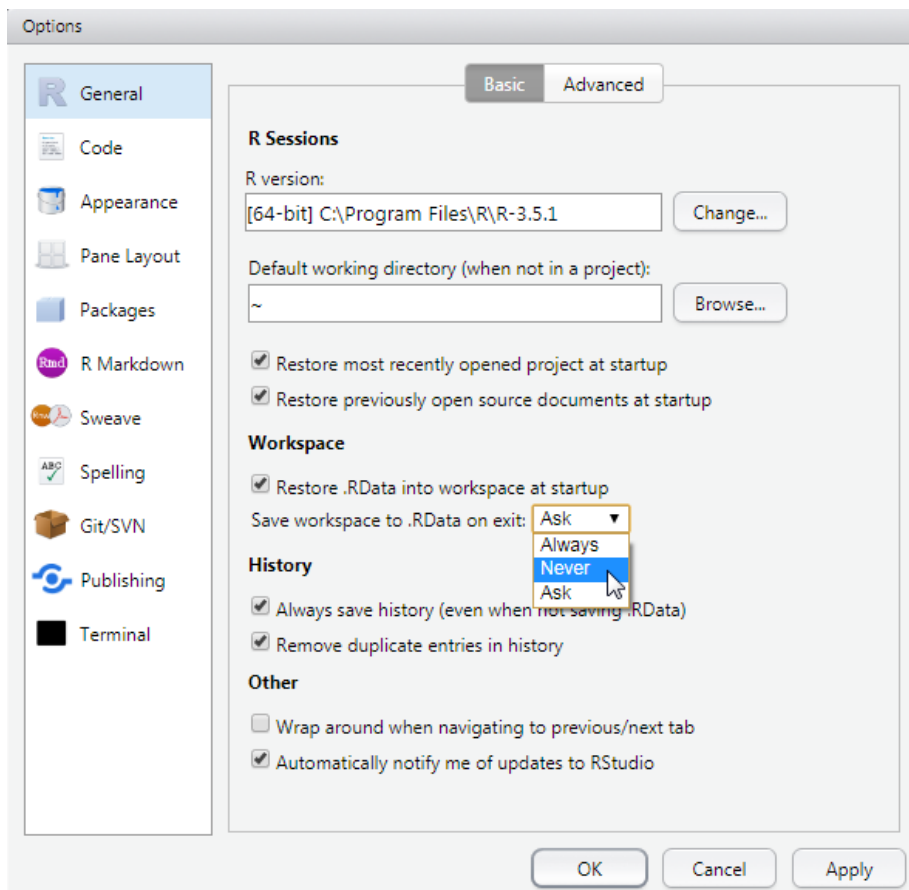
When you exit RStudio, you will get a pop-up asking you if you want to “Save workspace image...” to some location on your hard drive. Any person’s natural inclination is to answer “Yes” - what fool doesn’t want to save their work? However, saving your “workspace image” and “saving your work” are not the same thing.

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 they everything can be reloaded the next time you start RStudio. Which sounds great. Until you get a month into the semester, and you’ve keep saving and loading an entire month of R sessions each time. 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. And now RStudio takes 2 minutes to start and shut down because you’ve created a behemoth.

In short, saving your workspace sounds great, but encourages very bad practices. It encourages letting 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 do 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.



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 discarded code because you thought it wasn't useful or necessary, but later realize 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 up arrow key will take you through older commands, and press the down 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 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 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 Comprehensive R Archive Network. 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 provide 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 help on, you can type 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
```

press 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 next sections, we demonstrate the basic tenants 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 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.

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* be abbreviated as `T` and `F`, but it is not recommended (The names `T` and `F` can be over-written with non-logical values, But `TRUE` and `FALSE` are reserved names that can never be overwritten. Such a restrictive class of data may not seem useful, but play an important role in data manipulation, as they 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`s, but is useful in situations where you have a function which produces one type of logical, and but you’re interested in the opposite. We will look at such a case later in the logical indexing section.

Special Values

R uses the special value `NA` to denote when a value is missing. `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 (" " or ' ' are both acceptable).

```
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 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.`) which provides a `TRUE` or `FALSE` response, and a class conversion function (starting with the prefix `as.`). Consider the following examples:

```
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 encounter 4 types of data structures: Vectors, Matrices, Data Frames, and Lists.

2.2.1 Vectors

Vectors are 1 dimensional data structures which 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

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

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

```
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:

```
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 elements 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, etc.)

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

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, etc.)

```
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 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 controls how R interprets character vectors (a.k.a. strings) when forming the data frame. Here, using `stringsAsFactors = TRUE` 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 doesn'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. But carefully consider if there is another way, because 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 the same 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
- ^ 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 examining 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 it’s 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 together 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 the 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, without providing 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 function. 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* have 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 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 functions’ 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”.

But, we will focus on other cryptic parts of the “Usage” section. But the usage section is also cryptic! What is `x`? What is `trim`? What do they do? For clarification, we must go the arguments sections.

2.4.3 Arguments

The detailed descriptions in the arguments section 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., `x`, `na.rm`, `trim`). 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 **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 is beyond the scope of this guide (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 parameters inputs that aren’t otherwise explicitly declared. 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 know what 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 manipulations 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 manipulations 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 a CSV file looks like before it is imported into R:

```
commas <- read.csv("data/commas.csv")
write.csv(commas, file="", quote=FALSE, row.names=FALSE)
## 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, and . 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, 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 criteria. 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
```

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
## [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 things that are the opposite of missing
## [1] 10 1 4 2 99
```

Converting a Logical to a Positional Index

A useful function to know is `which()`. When used on a logical vector, it will return to you 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
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.

```
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.

```
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 our 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 column 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 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$spray
## [1] A A A A A A A A A A A A B B B B B B B B B B B C C C C C C
## [reached getOption("max.print") -- omitted 42 entries ]
## Levels: A B C D E F
InsectSprays$spray=="A"
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE 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[, 'neeeeeeeighhhhh']
```

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

It's 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
```