

JOHN L. TAVERAS

```
=CONCATENATE("Please", " ", "Enjoy")  
paste("Please", "Enjoy", sep=" ")
```



FOR EXCEL USERS

**AN INTRODUCTION TO R
FOR EXCEL ANALYSTS**

R for Excel Users

An Introduction to R for Excel Analysts

John L Taveras

This book is for sale at <http://leanpub.com/r-for-excelusers>

This version was published on 2016-08-10



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2016 John L Taveras

For Keren and Julian, the biggest loves of my life.

Contents

| | |
|------------------------------|---|
| Enjoy this sample! | i |
|------------------------------|---|

| | |
|-------------------|----|
| Preface | ii |
|-------------------|----|

| | |
|---|----------|
| Part 1 - Introduction & Set Up | 1 |
|---|----------|

| | |
|------------------------------------|----------|
| 1. Getting Set Up | 2 |
|------------------------------------|----------|

| | |
|--|---|
| 1.1 Getting mentally ready to write code | 2 |
|--|---|

| | |
|--|---|
| 1.2 Downloading the software | 3 |
|--|---|

| | |
|---------------------------------------|---|
| 1.3 Navigating the software | 3 |
|---------------------------------------|---|

| | |
|------------------------------------|---|
| 1.4 Using other Software | 6 |
|------------------------------------|---|

| | |
|-------------------------|---|
| 1.5 Libraries | 7 |
|-------------------------|---|

| | |
|----------------------------|---|
| 1.6 Console tips | 9 |
|----------------------------|---|

| | |
|----------------------------|----|
| 1.7 Getting help | 10 |
|----------------------------|----|

| | |
|--|-----------|
| 2. Programming Basics | 12 |
|--|-----------|

| | |
|-----------------------------------|----|
| 2.1 Assigning variables | 12 |
|-----------------------------------|----|

| | |
|----------------------------------|----|
| 2.2 Testing conditions | 13 |
|----------------------------------|----|

| | |
|-------------------------------|----|
| 2.3 Data structures | 13 |
|-------------------------------|----|

| | |
|------------------------------|----|
| 2.4 Special values | 14 |
|------------------------------|----|

| | |
|------------------------------------|----|
| 2.5 Commenting your code | 16 |
|------------------------------------|----|

| | |
|----------------------------------|----|
| 2.6 Control Structures | 16 |
|----------------------------------|----|

| | |
|---|-----------|
| 3. Quick Start - Analysis Examples | 17 |
|---|-----------|

| | |
|--|----|
| 3.1 Example 1: General data analysis | 17 |
|--|----|

| | |
|---|----|
| 3.2 Example 2: Using SQL in R | 22 |
|---|----|

| | |
|--|----|
| 3.3 Example 3: Multiple regression model | 25 |
|--|----|

CONTENTS

| | | |
|-----|-------------------|----|
| 3.4 | Summary | 28 |
|-----|-------------------|----|

Part 2 - Building Blocks: Cells and Formulas 29

| | | |
|-----|--------------------------------------|-----------|
| 4. | Cells are Vectors | 30 |
| 4.1 | Individual cells | 30 |
| 4.2 | Cell ranges | 32 |
| 4.3 | They are all vectors | 33 |
| 4.4 | Why vectors matter | 34 |
| 4.5 | Working with vectors | 35 |
| 4.6 | How vector operations work | 39 |
| 4.7 | Summary | 43 |
| 5. | Like the sample? | 44 |

Enjoy this sample!

Thanks for downloading this sample. Included here are some early chapters to get you up and running and excited about learning R.

Once you're ready to get the full version (digital or print), go to www.rforexcelusers.com/book.

And remember:

- Get a 30-day refund if you don't like it
- Enjoy early access to new learning tools and materials

Preface

If you are an analyst in the business world, you might be feeling pressure to upgrade your skills. *Data science* and *big data* are now standard terminology in many circles: business intelligence, marketing, operations, sales, strategy, finance, etc.

So it makes sense why you've picked up this book. But I want to be clear about one thing: do not think of R as a *replacement* for Excel! Instead think of R as just another tool you can pull out when the time is right. R will enable you work on bigger, more complex projects. You can be more productive at work, and depending on your function, you might even look like a genius to your colleagues.

These are three ways Excel comes up short, and where R can kick in:

1. **Excel does not handle big data sets well.** In theory, Excel can hold 1M+ rows and 16K+ columns of data, but in practice it slows down tremendously when using just a fraction of all those cells.
2. **Excel does not do well with complex or sloppy data.** Sometimes your raw data is not well structured. For example, you might have many values stuffed in each cell that need to be parsed out. Or perhaps you are working with several scattered datasets that need to be merged into a master sheet. Or your data is shaped in a way that prevents you from using PivotTables effectively.
3. **Excel does not offer robust statistical functionality.** A simple statistical task like creating a histogram is surprisingly labor-intensive in Excel. More advanced procedures like multiple regression usually require paid add-on packages.

If you are facing any of these situations, then you might be ready to make the leap into a data programming language like R.

What is R and why should I learn it?

R is a programming language designed for processing, analyzing and visualizing data. In recent years, R has become more popular in business. It can now be found across industries (financial services, technology, retail, etc.) and job functions (marketing, operations, finance, etc.).

Goal of this book

This book is for beginners, and the goal is to *get you started*. R is known to have a steep learning curve, but I really think it has three separate curves:

1. **Data manipulation.** Importing, creating, cleaning, summarizing, transforming, reshaping and combining data sets.
2. **Analysis.** Statistics, hypothesis testing, predictive modeling, machine learning, etc.
3. **Visualization.** Charts, graphics, from simple to elegant.

Many books and courses combine those three things, forming one long, painful learning curve that discourages many. And many eager learners try to “run before they crawl” by jumping to statistical modeling without understanding how to work with data sets, leading to frustration.

This book will be focused on data manipulation: how to import, manipulate, transform, combine and summarize data. Once you are able to work with data sets, performing statistical analysis will mostly be a matter of applying your own domain knowledge and finding the appropriate functions to use. And learning graphics also won't be too hard; you can learn from just looking at chart galleries.

I will make little effort to teach you statistics or graphics.

I will keep technical lingo to a minimum, trying to keep this as practical as possible. I will also draw connections to Excel where I see fit, hoping that will help concepts sink in.

How the book is organized

In Chapters 1 and 2 you will set up the software and get your bearings. In Chapter 3 you will see three sample analyses from beginning to end, with minimal commentary. This is to both show you the big picture of what a basic analysis might look like, as well as to satisfy the over-eager learners who need just a little to get started.

The remaining chapters go in depth into concepts like vectors and functions (Chapters 4-5), then walk through the different data management tasks like importing, manipulating, shaping, and summarizing data sets (Chapters 6-12).

And then starting in Chapter 13 we start getting into more advanced topics that don't have a clear Excel analogy. Lists and loops are at the core of those chapters.

Data used in this book

We will use data sets found freely on the internet or built into R. You can download the free data from the website at www.rforexcelusers.com/book-files.

Part 1 - Introduction & Set Up

1. Getting Set Up

In this chapter you will learn to install the software, navigate the screens and find help when you need it.

1.1 Getting mentally ready to write code

If you have been using Excel for some time, I have news for you: you are *already a programmer*. If you have written a *vlookup* formula, rather than finding and copying values manually, then you have programmed the machine to do work for you. If, in the course of an analysis, something has looked wrong and you worked backwards to find and fix the root cause (usually a formula error), then you have essentially debugged a program. At the risk of trivializing the job of a professional programmer, many things you do in Excel are things a programmer would do with code.

In other words, you have what it takes to write code; you just need to make some mental adjustments. Your canvas is no longer an intuitive point-and-click interface, but a simple command line.

It will take some time to get used to it, but I assure you two big benefits of writing code will far outweigh the fear:

- *Code is self-documenting.* Since code is linear, read from top to bottom, anyone can pick up your code and follow the logic through the end. We have all seen large Excel workbooks with multiple analyses scattered across different tabs and no clear thought process leading from point A to point B. And we have most certainly tried deciphering someone else's analysis containing hard-coded values with no reference to where numbers came from. These problems tend to not happen with programming, because we are forced to give the computer specific instructions.

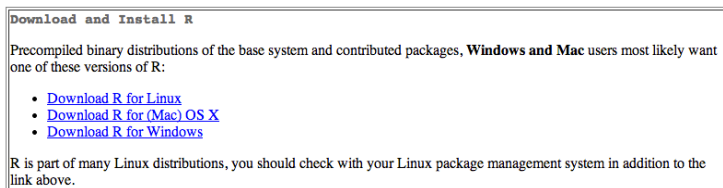
- *You can do almost anything.* You are not limited to what Microsoft Corporation thought about when creating Excel. With R, you can write your own functions or leverage the thousands of functions other people have shared through *free* libraries.



Learning a programming language is a lot like learning a spoken language. You can read all the books and attend all the classes, but you really won't "get it" until you immerse yourself. So go ahead, turn on your computer and actively follow the examples. And experiment.

1.2 Downloading the software

To get started, you only need the R application. You can download it at <http://cran.r-project.org>. Look for the download section, which looks like this:



R download screen

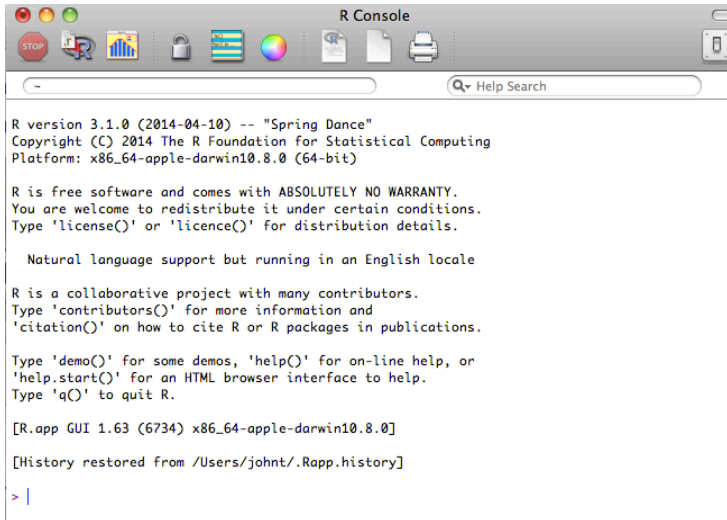
Click on the appropriate link based on your operating system.

In the next page, Mac users click one of the *.pkg* files based on your version of Mac OS X. Instructions for Windows users will be straightforward.

After that, you will go through a typical installation with all the preset default settings.

1.3 Navigating the software

When you open the R (or R64) software, you will see a screen that looks like this:



```
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.63 (6734) x86_64-apple-darwin10.8.0]

[History restored from /Users/johnt/.Rapp.history]

> |
```

R Console

This is called the *console* and is where the command line action happens. You enter code right after the `>` on the command line, then press enter to run that line of code. For example, try a few operations like this:

```
> #type 5+5 after the ">" then press enter
> 5 + 5
[1] 10

> 100 / 3
[1] 33.33333

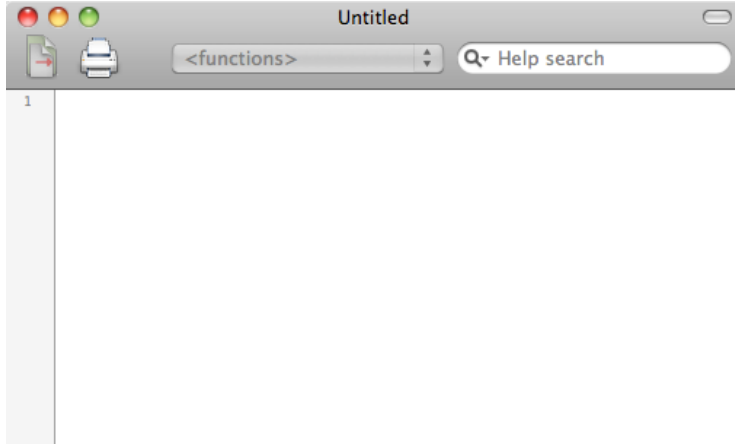
> nchar("john")
[1] 4

> toupper("taveras")
[1] "TAVERAS"
```

In the first line, we are just doing simple arithmetic, namely $5+5$. In the third example, we use the `nchar()` function to get the number of characters in my first name (like Excel's `LEN()`). In the last example, we use the `toupper()` function to capitalize the

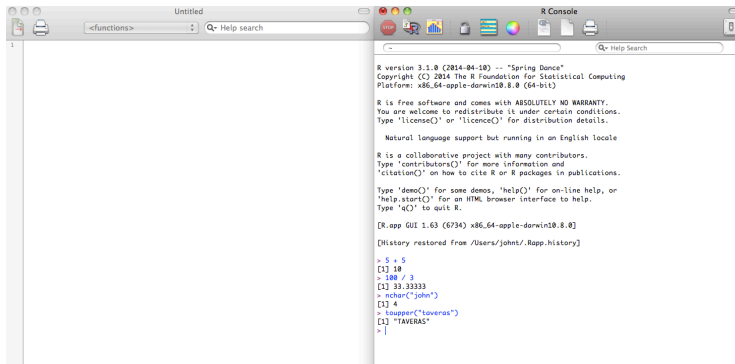
letters in my last name (like Excel's UPPER()). You will learn more about functions in Chapter 5.

You can write code in the console as we did above, but that would not allow you to save it or share it with others. For that, we can open R's built-in text editor by going `File > New Document`. That will open a new window like this:



R's built-in text editor

Here you can write code and save it as you would an ordinary file. You can also execute your code from the editor, either line by line as you type, or in chunks. I recommend setting up the editor and console windows in a way that you can see both at once. For example, this is how I have mine:



My usual window setup. Sometimes I do two horizontal windows instead.

My process will be to write code on the left, and stuff will run in the console on the right. Let's run the same code we did before, but write it in the text editor instead. You can either run each line after you type it, or type a few lines, then run them together. To run them from the text editor, you put the cursor anywhere on the line you wish to run, or select the lines you wish to run, then press [Command]+[Enter] on Mac or [Ctrl]+[r] on Windows.

```
# type each line in the editor, then Run each line
5 + 5
100 / 3
nchar("john")
toupper("taveras")
```

Note the first line of text above is called a comment. Comments begin with a pound sign (#) and do not execute anything; they are simply for the coder to write useful notes.

1.4 Using other Software

There are a number of text editor options. My favorite is RStudio, which is what I personally use for all my R work. RStudio is an IDE (integrated development environment) that is free to download at www.rstudio.com. It offers a slick interface with nice features and shortcuts.

1.5 Libraries

One of R's biggest assets is the large community of developers who create add-on functionality *for free*. They create libraries (aka packages) that add sets of functions related to a specific theme or analysis. Think of it like Excel's popular add-ons, *Analysis Toolpak* (for regression) and *Solver* (for optimization); except *free* and with vibrant communities (both online and offline).

R already comes with a lot of functionality built in and for many things you will not need additional libraries. But for added functionality we resort to libraries. The libraries you need will depend on your use cases, but here are some that I commonly use for basic data manipulation and visualization:

Data management libraries

- **dplyr**. Allows speedy merging, sorting, subsetting and manipulating data frames with intuitive syntax. You will see this in Chapters 10-12.
- **reshape2**. Allows transposing and reshaping of data sets, like turning rows into columns and vice versa. You will learn about this in Chapter 12.
- **sqldf**. If you know SQL, you can write SELECT statements on your data sets. This can significantly ease your R learning curve. You will see this in Chapter 10-12.
- **readxl**. Allows to import Excel (.xlsx) files.
- **xlsx**. Allows to import, create and format Excel files.

Visualization libraries

- **ggplot2**. Popular visualization package that creates beautiful graphs.
- **googleVis**. Interactive web-based visuals. Easy to use but more limited in capability compared to ggplot2.

To use a library, you first install it, and any time you want to use its functionality, you load the library. You only install the library once. For example, let's install the `dplyr` library. Run this command (either directly in your console, or from the text editor):


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

You will be prompted to select a mirror site (you can pick any) and, generally speaking, the installation should be a breeze. Any time we start an analysis that requires ggplot2 visuals, we load the library as follows:

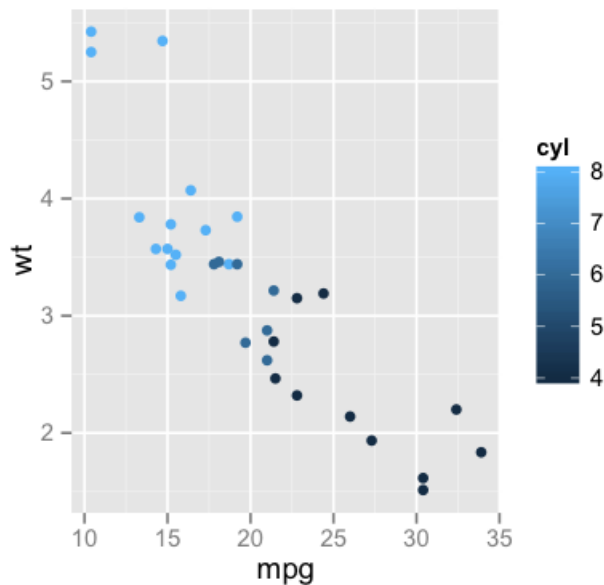
```
library(ggplot2)
```

It's good practice to put all your library calls together, at the top of your script. You only need to run it once within a session.

Now that the library is loaded, we can run specific functions that are part of the library, like `qplot()`, which is part of the ggplot2 library:

```
qplot(x = mpg, y = wt, data = mtcars, colour = cyl)
```

That short piece of code generates the following scatterplot using a sample dataset (*mtcars*) that comes with the ggplot2 library.



ggplot2's `qplot()` function

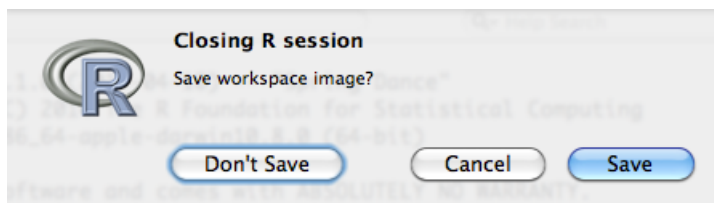
Here we are plotting miles per gallon vs weight, and a third variable, number of cylinders, represented by color.



What else does the library offer? There are a few good ways to see what other functions a library offers. Running `help(package = dplyr)` provides the list of resources (functions, data sets, etc) available in the library. Additionally, every library has a PDF reference manual on the R Cran website with the same information.

1.6 Console tips

- **Run past commands.** Put your cursor at the command prompt in the console and press the up arrow. This will recall prior commands you have run, which can be a time-saver.
- **Help text.** As with Excel, R shows you the structure of a function (Excel formula) as you type it. When you type a function (e.g., `toupper()`) in the console or editor, you will see help text at the bottom of the window showing the structure of the function.
- **Save your work.** Aside from saving your text file with the code, you can also save an “image” of your workspace, which includes all the objects (data sets, vectors, etc) you have created. When you close out of R, you will be prompted with the image below. If you click save and reopen R later, all your data sets, vectors, and so forth, will be available for use.



Option to save workspace

1.7 Getting help

You will run into problems and have questions that this book does not cover. Here are some of the general types of issues you will run into:

You want to do a certain kind of analysis but don't know which functions to use

Google search all the way. Someone most likely has done this analysis and there likely are functions that address this kind of problem.

You know which function to use, but don't know how to use it

You can load help docs for a function by typing one of the following:

```
?functionname
```

- or -

```
help(functionname)
```

For example, `?qplot` will describe the graphing function we ran above. Before doing that, you will need to have loaded the `ggplot2` library, since `qplot()` is part of that library. If that fails, try Google.

Your code is generating an error

Sometimes the error messages are somewhat helpful, but often they will be cryptic and not give you a clear indication of what went wrong. Over time you will get better at figuring them out or avoiding them altogether. Here are some common causes for formula errors:

- Variable type. Doing math on non-numeric variables will give you an error. Same with dates, text strings, etc.
- Syntax. Check for missing commas, proper case (R is case-sensitive), missing parentheses, etc.
- Many more...

You loaded a library but don't know which functions are now available

You can use the `help()` function to open help docs for the entire library, like this:

```
help(package=reshape2)
```

This will open a window with a list of functions within the library. Another good resource is the Cran website; every library comes with a pdf reference manual.

2. Programming Basics

In this chapter we will review some programming basics to get you up and running.

2.1 Assigning variables

When writing a formula in Excel, we can simply point-and-click on a cell to reference, and Excel will write the cell reference for us (e.g., B3, B:B). In R, we don't have that luxury; we need to name everything we plan to reference. There are two ways to assign variables:

Code: Two ways to assign variables

```
> a = 5  
> a <- 5
```

Both versions have the same effect of assigning the variable *a* the value 5. This is similar to setting the value of a cell in Excel, say, B4, the value 5. We can then perform operations on the variable *a*.

The choice of = or <- is personal preference. I prefer the equal sign because it is instantly intuitive, easier to type and nearly every other programming language uses it. However, R purists scoff at this, so you will usually see the <- notation in books and websites. In this book, I will use the equal sign exclusively because that is what I use in real life.

2.2 Testing conditions

Below is the notation used when comparing values, say, in an *if* statement:

- Equality: `==`
- Not equal: `!=`
- Greater / Less than: `>` or `<`
- Greater / Less than or equal to: `>=` or `<=`

Only the first two are different from Excel. The first one will be confusing if you have not seen this before. Remember, single equal sign *assigns* values, while double equal sign *tests* values for equality. Here are some examples:

Code: Testing conditions

```
> # Does 5 equal 4?
> 5 == 4
[1] FALSE

> # Is 10 less than or equal to 11?
> 10 <= 11
[1] TRUE

> # Does TRUE not equal TRUE?
> TRUE != TRUE
[1] FALSE
```

2.3 Data structures

There are five basic data structures in R: (1) Vector, (2) Matrix, (3) List, (4) Data Frame and (5) Array. In this book we will primarily work with **vectors** and **data frames**, as they are the most analogous to the basic data structures in Excel. Lists will be covered in the advanced topics section.

Vectors are roughly like rows and columns of a table, and a vector of length one can be thought of as a single cell in Excel. You will learn more about vectors in Chapter 4.

Data frames are like data tables; they contain rows and columns, each column has a name and the values within each column are of a single type (e.g., a column of numbers, characters, dates, etc). Data frames are discussed in Chapters 6 and onward.

Lists are arbitrary collections of items, with essentially no rules. You could think of lists as entire workbooks – a workbook has many sheets, and some sheets might have big tables, small lookup tables, a few cells with text, etc.

And if you need to work with matrices and arrays, you are probably reading the wrong book.

2.4 Special values

There are a few special values in R: NA, NaN, NULL, Inf.

NA or <NA> (“not applicable”)

Represents missing values. For example, in a data set containing names and ages, you might have some missing values for people who did not report their age. NA is used for missing numeric values and <NA> is used for missing string values. Similar to a blank cell in Excel, or sometimes the #N/A value.

In Excel you can test if a value is missing with the ISNA() formula. In R, you can use the `is.na()` function. You will learn about functions in Chapter 5.

Code: Examples of NA / missing values

```
> # create two variables
> num1 = NA
> num2 = 5

> # check if they are NA
> is.na(num1)
[1] TRUE
> is.na(num2)
[1] FALSE

> # try adding the two numbers
> num1 + num2
[1] NA
```

NaN (“not a number”)

You might see this one occasionally when writing formulas. If you divide 0 by 0, the result is NaN, not a number. Or subtracting Infinite from Infinite. Similar to Excel’s #NUM!.

Code: Examples of NaN / Not a Number

```
> # divide 0 by 0
> num1 = 0
> num2 = 0
> num1 / num2
[1] NaN
```

Inf and -Inf (plus and minus “infinity”)

You will see this, among other instances, when dividing a number by 0 (like Excel’s #DIV/0!).

Code: Examples of NaN / Not a Number

```
> # divide 5 by 0
> 5 / 0
[1] Inf

> # compute infinite + 1
> Inf + 1
[1] Inf
```

2.5 Commenting your code

You have already seen comments in some of the code snippets. A comment is code that is not executable; it is simply meant for note-taking. Commenting code is absolutely a best practice. Helpful comments include information about data sources, describe a custom function, explain hacks or workarounds that may not be intuitive, and so forth. It's sometimes helpful to use comments to visually break up code into chunks; for example, the data manipulation chunk, the summarization chunk, the visualization, etc.

The R text editor in both R and R Studio comes with a handy shortcut for commenting. You may select multiple lines and press the shortcut keys to prepend a pound sign at the start of each line. See the *Edit* menu in R and *Code* menu in R Studio.

2.6 Control Structures

I consider control constructs like *for* loops, *while* loops and *if* advanced topics. You can read about them in Chapter 14.

Note – *if* is different from *ifelse*. *ifelse* is the equivalent of Excel's IF() formula that returns something if true, and something else if false. *ifelse()* is discussed in Chapter 8.

3. Quick Start - Analysis Examples

WARNING: The purpose of this chapter is to showcase analysis examples at a high level, fairly quickly and with minimal explanation. So if things go over your head, don't worry. Most of it will be covered in later chapters.



Data used in examples

We will import .csv files for examples in this chapter. Download them at www.rforexcelusers.com/book-files.

3.1 Example 1: General data analysis

In this example we will see some basic data operations: import, merge and filter data sets. We will then calculate a new column and visualize with a histogram.

Question: What is the distribution of players' age?

We have two data sets we can import for this analysis: *Batting.csv*, which has annual batting statistics by player, and *Master.csv*, which has some demographic data on each player. We can calculate players' age by subtracting their birth year (in *Master.csv*) from the batting year (from *Batting.csv*).

Let's first import these two data sets and take a peek at them:

Code: Import and take quick peek

```
> # import the two data sets
> players = read.csv("/path/Master.csv", stringsAsFactors=FALSE)
> batting = read.csv("/path/Batting.csv", stringsAsFactors=FALSE)

> # quick peek - first five rows, first 6 columns
> players[1:5, 1:6]
  playerID birthYear birthMonth birthDay birthCountry birthState
1 aardsda01      1981         12      27          USA          CO
2 aaronha01      1934          2        5          USA          AL
3 aaronto01      1939          8        5          USA          AL
4 aasedo01       1954          9        8          USA          CA
5 abadan01       1972          8       25          USA          FL

> # quick peek - first five rows, first 10 columns
> batting[1:5, 1:10]
  playerID yearID stint teamID lgID  G G_batting AB R H
1 aardsda01  2004     1   SFN   NL 11         11  0 0 0
2 aardsda01  2006     1   CHN   NL 45         43  2 0 0
3 aardsda01  2007     1   CHA   AL 25          2  0 0 0
4 aardsda01  2008     1   BOS   AL 47          5  1 0 0
5 aardsda01  2009     1   SEA   AL 73          3  0 0 0
```

We used the `read.csv()` function to import the two csv files, and then took a peek using the standard bracket syntax for filtering and referencing data frames. From that quick peek, we can see the *players* data frame has one row per `playerID`, and the *batting* data frame has a row per `playerID`, by year (and also by stint, which you cannot tell from this. If a player played for multiple teams in the same year, those stints are segmented).

Ultimately we want to calculate each player's age every year they played. If this were Excel, we would add a `VLOOKUP` to the *batting* table to bring in each player's `birthYear`. In R, that can be done with the `merge()` function:

Code: Merge batting and players

```
> ## merge batting and two columns of players table
> batting_extra = merge(batting, players[,c("playerID", "birthYear")])

> # quick peek at new data frame
> batting_extra[1:10, c("playerID", "yearID", "teamID", "G", "birthYear")]
  playerID yearID teamID   G birthYear
1 aardsda01  2004   SFN  11    1981
2 aardsda01  2006   CHN  45    1981
3 aardsda01  2007   CHA  25    1981
4 aardsda01  2008   BOS  47    1981
5 aardsda01  2009   SEA  73    1981
6 aardsda01  2010   SEA  53    1981
7 aardsda01  2012   NYA   1    1981
8 aardsda01  2013   NYN  43    1981
9 aaronha01  1955   ML1 153    1934
10 aaronha01  1956   ML1 153    1934
```

The `merge()` function combines two data frames based on overlapping columns; in this case the full batting table and just two columns of the players table. Chapter 11 discusses `merge()` further. We then took another quick peek, this time using column names instead of column numbers to specify which columns to see. We can now see that `birthYear` is alongside `yearID` and other columns from the *batting* table.

Often times, very young players go to the major leagues, and get sent back to the minor leagues if they do not perform well. If we do not want those players to skew our distribution, we can exclude them from the analysis. Since we do not have a variable for that, we can use number of times at bat as a proxy. Let's, arbitrarily, keep only players with more than 200 at bats per season. Let's also keep seasons post-1950.

Below we apply that filter, and also add the age column to the table:

Code: Apply filter and create new column

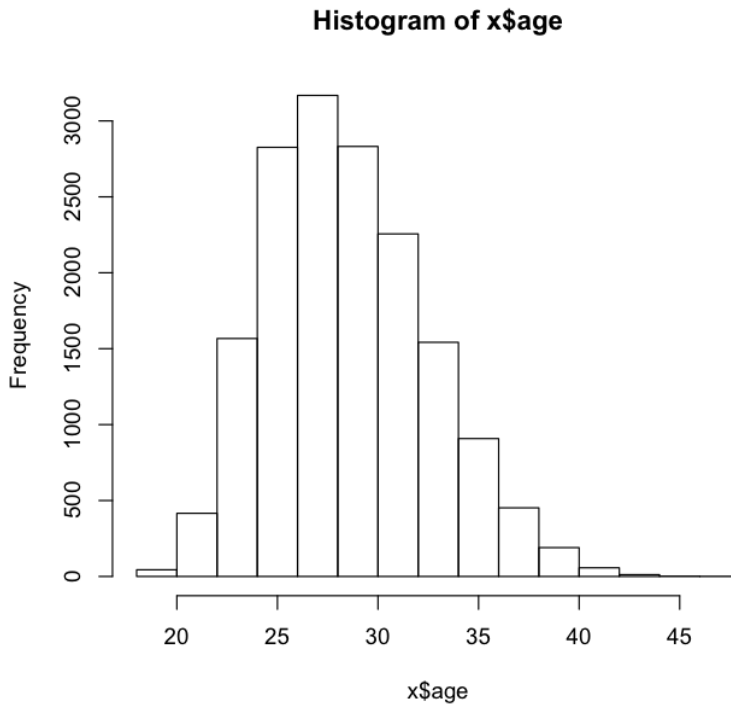
```
> # filter batting_extra: players with > 200 at bats, post 1950
> x = batting_extra[which(batting_extra$AB > 200 &
                          batting_extra$yearID >= 1950), ]

> # calculate players' age as yearID - birthYear
> x$age = x$yearID - x$birthYear
```

We saved the filtered data set to *x* so that we do not change the underlying data set. Note how individual columns are referenced with the *\$* operator.

If that looks ugly to you, you're not alone. There are better, cleaner and more intuitive ways to filter data sets. The *dplyr* package offers the `filter()` function, which is covered in Chapter 10. But it's important that you still learn these “base” ways to do things.

Now we can plot a histogram by running `hist(x$age)`:



Plot: Histogram of age column

We could also see a statistical summary of the age column:

Code: Apply filter and create new column

```
> summary(x$age)
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-------|---------|--------|-------|---------|-------|
| 19.00 | 26.00 | 29.00 | 28.99 | 32.00 | 47.00 |

In this example we saw how to import and merge data sets, apply filters, create columns and plot histograms using built-in “Base R” functions. We also saw one way to reference specific columns using the `df$column` syntax. It is awkward and annoying at first, but you simply get used to it.

3.2 Example 2: Using SQL in R

SQL is a fairly common skill among Excel analysts. If you know SQL, then good news: you can write SELECT statements in R. SQL can serve as your gateway into R, as you can do a lot of basic data manipulation with it. If you don't know SQL, you can skip this section, although you should probably try to learn SQL too.

Question: how have baseball salaries trended over the years in the American and National League?

Since we will be using the sqldf library, we first need to install and load the library. You only install it once and then load it any time you plan to use it.

Code: Install and load the sqldf library

```
# install the sqldf library (one-time setup)
install.packages("sqldf")

# load the sqldf library.
library(sqldf)
```

Then we import the raw data we will analyze. There are two data sets: one that contains each player's salary every year (Salaries.csv), and the other contains information about each team (Teams.csv), including which league it belongs to.

Code: Import the two data sets

```
> # import the salaries.csv file and name it "salaries"
> salaries = read.csv("Baseball Data/Salaries.csv")

> # import the teams.csv file and name it teams
> teams = read.csv("Baseball Data/Teams.csv")

> # first 6 rows gives you a sense of the data frame
> # each player's salary by year
> head(salaries)
  yearID teamID lgID  playerID  salary
1  1985     BAL   AL murraed02 1472819
```

| | | | | | |
|---|------|-----|----|-----------|---------|
| 2 | 1985 | BAL | AL | lynnfr01 | 1090000 |
| 3 | 1985 | BAL | AL | ripkeca01 | 800000 |
| 4 | 1985 | BAL | AL | lacyle01 | 725000 |
| 5 | 1985 | BAL | AL | flanami01 | 641667 |
| 6 | 1985 | BAL | AL | boddimi01 | 625000 |

We first used the `read.csv()` function to import each data set, then used the `head()` function to get a quick look at the salaries data frame. That function gives us the first six rows of a data frame by default.

Next we merge the two data sets to bring the team name column into the salaries table (not needed for analysis; just demonstration). This is similar to a `VLOOKUP()` in Excel as well as the `merge()` function in the last example, but using SQL instead:

Code: SQL JOIN the two tables

```
> # write a SQL JOIN to merge salaries and teams data frames
> salaries_teams = sqldf("SELECT a.*, b.name
                           FROM salaries a
                           JOIN teams b ON(a.yearID = b.yearID
                           AND a.teamID = b.teamID)")
> head(salaries_teams)
  yearID teamID lgID  playerID  salary      name
1  1985    BAL   AL  murraed02 1472819 Baltimore Orioles
2  1985    BAL   AL  lynnfr01 1090000 Baltimore Orioles
3  1985    BAL   AL  ripkeca01  800000 Baltimore Orioles
4  1985    BAL   AL  lacyle01  725000 Baltimore Orioles
5  1985    BAL   AL  flanami01  641667 Baltimore Orioles
6  1985    BAL   AL  boddimi01  625000 Baltimore Orioles
```

Once again we used the `head()` function to quickly inspect the output. We see the `lgID` column is now alongside the player salary data. Now we can summarize average salary by league, by year using a SQL CASE statement:

Code: SQL summary by year

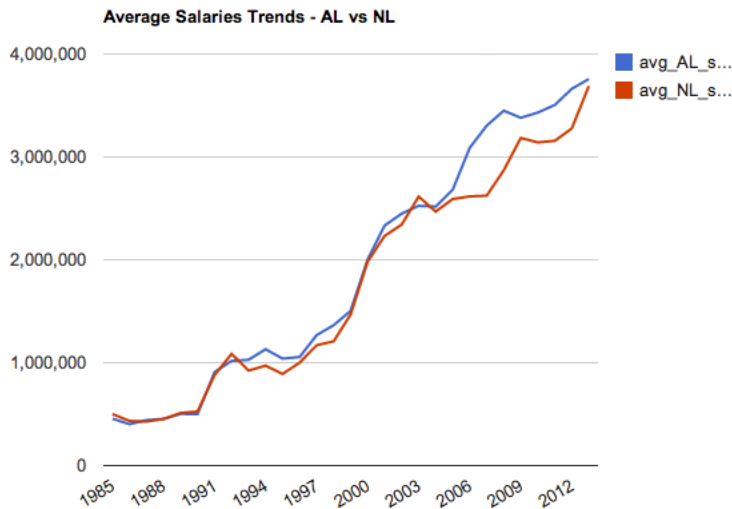
```
> al_nl_salary =  
  sqldf("SELECT yearID,  
          AVG(CASE WHEN lgID = 'AL' THEN salary END)  
            as avg_AL_salary,  
          AVG(CASE WHEN lgID = 'NL' THEN salary END)  
            as avg_NL_salary  
        FROM salaries_teams GROUP BY 1")
```

Instead of printing the summary data, let's instead visualize it with the googleVis package, which provides interactive, web-friendly graphics:

Code: SQL summary by year

```
> # one-time package install  
> install.packages("googleVis")  
  
> # load googleVis  
> library(googleVis)  
  
> # graph avg_AL_salary and avg_NL_salary by year  
> g = gvisLineChart(al_nl_salary, xvar = "yearID",  
                    yvar = c("avg_AL_salary", "avg_NL_salary"),  
                    options = list(title = "Average Salaries Trends -  
                                AL vs NL"))  
>  
> plot(g)
```

Sit back and enjoy the beautiful output. Salaries are clearly trending up, and in the mid-2000s, the American League started breaking away with higher average salaries.



Plot: Average salary by league, over time

3.3 Example 3: Multiple regression model

This book will not dive into regression and other statistical methods in detail. But let's work on one example to demonstrate some of the functionality.

Regression can be used to explain or predict numeric outcomes. Let's say we want to build a formula for used car prices as a function of characteristics like mileage, vehicle type, etc. We can use the *Car Values* data set to come up with that formula. That data set has information on 805 used vehicles less than one year old at the time and considered to be in excellent condition. (Download .csv file from website)

We can run a regression model for Price as follows:

Code: Import data and run regression

```
> # import data
> car_prices = read.csv("car_values_kuiper.csv",
                        stringsAsFactors = FALSE)

> # build regression model and save model to object named "m"
> m = lm(Price ~ Mileage + Type + as.factor(Cylinder),
        data = car_prices)
```

First we imported the data needed for the analysis and saved it to the object *car_prices*. Then we run the `lm()` function, which runs a linear regression model. We specify the formula we are modeling, which is Price as a function of Mileage, Car Type and Cylinders. Note that Cylinder takes on three values, 4, 6, 8. Rather than entering the model as a numeric variable and getting just a single point estimate, we want to treat those three values as discrete, so we denote as `.factor()` to accomplish that.

Something important to note is that we are saving the result of `lm()` to a new variable named *m*. This represents a powerful aspect of R, but also a tough one to grasp initially. We are simply creating an object named *m*, which holds information pertaining to the model. That object is a list, which is a collection of objects. Lists will be covered in Chapter 13.

To understand the makeup of this result list, we can use the `names()` function, which returns a vector of object names within that list:

Code: Get names of lm object

```
> names(m)
[1] "coefficients" "residuals"    "effects"      "rank"
[5] "fitted.values" "assign"        "qr"           "df.residual"
[9] "contrasts"     "xlevels"       "call"         "terms"
[13] "model"
```

If you are familiar with regression, some of those object names should be familiar. For example, *coefficients* stores the model estimates and can be accessed by `m$coefficients`.

For the purpose of this quick start, let's look at the regression summary:

Code: Get summary of lm object

```
> summary(m)
```

```
Call:
```

```
lm(formula = Price ~ Mileage + Type + as.factor(Cylinder),
    data = car_prices)
```

```
Residuals:
```

```
      Min       1Q   Median       3Q      Max
-14071  -3380   -822    1976   18846
```

```
Coefficients:
```

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------------|------------|------------|---------|--------------|
| (Intercept) | 3.669e+04 | 9.484e+02 | 38.686 | < 2e-16 *** |
| Mileage | -1.794e-01 | 2.349e-02 | -7.636 | 6.41e-14 *** |
| TypeCoupe | -1.944e+04 | 9.143e+02 | -21.263 | < 2e-16 *** |
| TypeHatchback | -2.125e+04 | 1.081e+03 | -19.659 | < 2e-16 *** |
| TypeSedan | -1.660e+04 | 8.423e+02 | -19.712 | < 2e-16 *** |
| TypeWagon | -1.014e+04 | 1.059e+03 | -9.575 | < 2e-16 *** |
| as.factor(Cylinder)6 | 4.322e+03 | 4.434e+02 | 9.746 | < 2e-16 *** |
| as.factor(Cylinder)8 | 1.964e+04 | 6.318e+02 | 31.086 | < 2e-16 *** |

```
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 5447 on 796 degrees of freedom
```

```
Multiple R-squared:  0.6989,    Adjusted R-squared:  0.6963
```

```
F-statistic: 264 on 7 and 796 DF,  p-value: < 2.2e-16
```

This summary has four sections, the main ones being the Coefficients table and the section that follows it. Looking at the Coefficients table, we see all the p-values ($\text{Pr}(>|t|)$) are less than 0.05, meaning all the estimates are significant at the 95% confidence level. Mileage is a numeric variable, and for every unit increase in Mileage, Price drops by 0.17. Sedans are $\sim 16\text{K}$ less expensive than the base class, which in this case happens to be a Convertible. And 8-Cylinder is $\sim 19.6\text{K}$ more expensive than a 4-Cylinder (base class).

I am setting aside validity concerns, multicollinearity, and so forth. The purpose of this exercise was to demonstrate the ease of building a regression model once the data is set up.

3.4 Summary

Hopefully these quick examples have grabbed your interest enough to continue learning R. If you count the number of lines of code needed for each of these exercises, you realize how much can be done with so little. The rest of the book will dive deeper into the data manipulation functionalities and more.

Part 2 - Building Blocks: Cells and Formulas

At its most basic level, Excel is made up of cells and formulas. Likewise, R's foundation is based on applying functions to vectors.

In this section, you will learn about vectors, which are like individual cells, rows and columns in Excel. Then you will learn about functions, which are like formulas in Excel. Vectors may be a bit abstract in the beginning, but should make more sense when you start working with data frames (tables).

4. Cells are Vectors

Vectors might be the only technical jargon in this book. But as you will soon learn, they are actually quite simple and important. In this chapter we will build vectors from the ground up, starting with individual cells and building up to cell ranges. Understanding vectors will help you more effectively work with data frames, which will be the main focus of the remaining chapters.

4.1 Individual cells

In Excel you might have something that looks like this:

| | A | B | C |
|---|---|----|--------|
| 1 | 5 | 12 | =A1+B1 |

Operating on single cells in Excel

This is a trivial example, but you have *programmed* Excel to add two numbers based on user input.

Here is an R equivalent:

Code: Operating on single 'cells' in R

```
> a = 5
> b = 12
> a + b
[1] 17
```

Here we have defined two objects, *a* and *b*, and assigned them a value in the first two lines. In line 3, we output the sum of *a* and *b*. Note in the third line that when

we simply enter the operation, without assigning it to a variable, the result is output on the screen.

Just like Excel, R could also handle character strings. Here is a little Excel blurb with string functions:

| | A | B | C |
|---|---------|-------------|-------|
| 1 | Stephen | =LEN(A1) | 7 |
| 2 | Joseph | =LEFT(A2,3) | Jos |
| 3 | James | =UPPER(A3) | JAMES |

Excel text formulas

And here is one way to replicate that in R:

Code: Text function examples

```
> person1 = "Stephen"
> nchar(person1)
[1] 7

> person2 = "Joseph"
> substr(person2, 1, 3)
[1] "Jos"

> person3 = "James"
> toupper(person3)
[1] "JAMES"
```

In the first chunk, we set the variable *person1* to “Stephen” and then used the `nchar()` function to count the number of characters, which works like Excel’s `LEN()` function. In the next chunk we used the `substr()` function to extract a substring from the name “Joseph”. `substr()` can do what Excel’s `MID()`, `LEFT()` and `RIGHT()` functions do for pulling parts of a string.



Reminder: character strings are wrapped within quotes; numeric variables are not. The names in the example above are surrounded by quotes.

4.2 Cell ranges

In Excel you have also see operations on entire ranges of cells, like this:

| | A | B | C | D | E |
|---|----|----|----|-------------|----|
| 1 | 22 | 15 | 52 | =MAX(A1:C1) | 52 |

Operating on a range in Excel

The same can be done in R:

Code: Operating on a 'range' in R

```
> numbers = c(22, 15, 52)
> max(numbers)
[1] 52
```

`c()` is a function for creating vectors. The *c* stands for combine, in that you are combining different items into a vector. In this example, we named this vector “numbers”, and used the `max()` function to get the maximum value in that vector.

And of course, you have seen this before – applying the same formula to every cell in a range:

| | A | B | C |
|---|---------|----------|---|
| 1 | Name | Length | |
| 2 | Stephen | =LEN(A2) | 7 |
| 3 | Joseph | =LEN(A3) | 6 |
| 4 | James | =LEN(A4) | 5 |

Formula on a range in Excel

It's a little different in R. You can put the three names in a vector, and run the function once:

Code: Function on a 'range' in R

```
> people = c(person1, person2, person3)
> nchar(people)
[1] 7 6 5
```

The result is also a vector with three values: 7, 6, 5, each corresponding to the elements of the *people* vector. Note that in this example, I manually created the *people* vector, but often when working with data tables, vectors will simply come from the tables themselves, so you would not “create” them manually.

One more example based on the Excel snippet above: we can think of the column headers also as a vector. In fact, a vector can be any collection of cells in a row or column in a table. Let’s convert the column headers to lowercase using `tolower()`, which is like Excel’s LOWER() formula:

Code: Vector function on column headers

```
> headers = c("Name", "Length")
> tolower(headers)
[1] "name" "length"
```

Just like the `nchar()` example, we input a vector of n (2) values and got a vector of n (2) values in return. The importance of this may not be obvious at first. Throughout the next few chapters, as you learn how to work with data frames, it will become more clear. This here is the basis of manipulating and working with data frame columns.

4.3 They are all vectors

All the R objects you saw in these examples are called *vectors*. Vectors can be thought of as one-dimensional collections of values. Here are more examples of vectors:

Code: Vector examples

```
> # A vector of numbers
> c(5, 92, 12)
[1] 5 92 12

> # A vector of logical values. T and F are shorthand
> c(TRUE, FALSE, T, F, F, F)
[1] TRUE FALSE TRUE FALSE FALSE FALSE

> # A vector of character strings
> c("Jim", "Jack", "Jill", "Jeff")
[1] "Jim" "Jack" "Jill" "Jeff"

> # A vector with one value
> 15
[1] 15

> # Same vector with one value
> c(15)
[1] 15
```

All of the above are vectors because they are one-dimensional. In contrast, multi-dimensional collections in R are matrices, data frames and lists. We will spend very little time with matrices and lists and a lot on data frames because data frames are analogous to data tables in Excel.

The values of a vector can be character, numeric, logical (true/false), and all values of a vector must be of the same type. That is, you cannot have numeric variables mixed with character or logical; R will coerce them to a type that accommodates all of them. For example, try `c(5, 9, "blah")` and you will see the 5 and 9 end up with quotes, indicating that they are encoded as character values.

4.4 Why vectors matter

Everything you do in Excel revolves around data stored in cells. Ditto for vectors in R. A lot revolves around vectors, so a solid grasp of the topic will be crucial for your

understanding of R.

4.5 Working with vectors

Here we will see how to (a) create a vector, (b) access values of a vector and (c) run functions on a vector.

a. Create a vector

You already saw in the above examples how to create vectors from scratch using the `c()` function. There are also shortcuts when dealing with special types of vectors. Here are some common ones:

Code: Vector shortcuts

```
# create a sequence from 12 to 20
> 12:20
[1] 12 13 14 15 16 17 18 19 20

> seq(12, 20, 1)
[1] 12 13 14 15 16 17 18 19 20

# combine vectors to create a larger vector
> c(9, 8, 7, c(1, 2, 3))
[1] 9 8 7 1 2 3

# there are some preset vectors in R
> letters
[1] "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"

> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug"
    "Sep" "Oct" "Nov" "Dec"
```

When dealing with structured data (like data frames), you will rarely have to create vectors from scratch. Instead, vectors will come from the tables themselves. For example, to get the vector of column names, you will use the `names()` function. You will learn more on that in later Chapters (7-9).

b. Access values of a vector

Once your vector is defined, you will want to access values from it. For example, get the second value of the column headers vector. The syntax for accessing values is `vectorname[index]` where `vectorname` is the name of the vector, and *index* is the index number of the element(s) you are looking for. Using the *people* vector we created earlier, here are some examples:

Code: Access vector values

```
# 1. print the people vector just for reference
> people
[1] "Stephen" "Joseph"  "James"

# 2. get the first value from the vector
> people[1]
[1] "Stephen"

# 3. evaluate math within the brackets
> people[1+2]
[1] "James"
> people[length(people)]
[1] "James"

# 4. print the 1st and 3rd values, in different order
> people[c(1,3)]
[1] "Stephen" "James"
> people[c(3,1)]
[1] "James"  "Stephen"
```

The in-line comments should make the three first chunks self-explanatory.

The last chunk is worth discussing. There we see how to use a vector of index numbers to pull multiple values from the *people* vector. Yes, a vector within a vector! This will become important once we learn to *apply filters* to data frames (like Excel Auto Filter). Notice also how order of the index numbers determines the order of the output. This will be important when you learn to *sort* data frames; i.e., put rows in a certain order.

Alternatively, instead of using index numbers to specify the items to keep, we can use a vector of TRUE/FALSE values. It is best if that TRUE/FALSE vector is the same length as the actual vector we are filtering. Let's see some examples:

Code: Access vector values with TRUE/FALSE

```
# 1. a vector of three TRUE/FALSE values
> people[c(TRUE, TRUE, FALSE)]
[1] "Stephen" "Joseph"

# 2. a vector of 1 TRUE/FALSE value
> people[c(TRUE)]
[1] "Stephen" "Joseph" "James"

# 3. tricky - a vector of 2 TRUE/FALSE values
> people[c(TRUE, FALSE)]
[1] "Stephen" "James"
```

The first chunk should be self-explanatory. The *people* vector has three items, so specifying a vector of `c(TRUE, TRUE, FALSE)` will result in the first two items of the *people* vector.

The next two examples are not so clear because the TRUE/FALSE vectors do not have three values. At the end of this chapter you will learn about vector recycling. After reading about recycling come back to this code so you can make sense of what's happening. Hint: in the second example, the vector expands to `c(TRUE, TRUE, TRUE)` and in the third example it expands to `c(TRUE, FALSE, TRUE)`.

Filtering programmatically

You might be wondering: is there a formula-driven way to do this instead? Manually specifying index numbers and TRUE/FALSE vectors seems odd. Here is where the power of vectors starts kicking in.

Instead of specifying a TRUE/FALSE vector manually, we can use functions to generate this vector programmatically. One way is with the `%in%` functionality. Let's continue with our *people* vector.

```
people %in% c("Jim", "John", "Maria", "Stephen")
```

This will give us a TRUE/FALSE vector indicating which items in *people* are found in the other vector. The result is: `c(TRUE, FALSE, FALSE)` because only the first item in *people* is found in the other vector. This means we can plug the above command right into the brackets:

```
people[people %in% c("Jim", "John", "Maria", "Stephen")]
```

This might still seem a bit odd to you, but will make more sense when you start working with data frames. Just imagine *people* is a column in a small data table, and the other vector is a column in another table. A command like this can be used to get the list of values in the *people* table that are found in the other table.

A more advanced application is *regular expressions*, a technique for pattern-matching text. For example, let's pull the items in the *people* vector starting with the letter J. The `grepl()` function is useful here:

```
grepl("^J", x = people)
```

The first argument specifies the regular expression, and the second argument specifies the vector to pattern on. Entire books and websites are dedicated to regular expressions, so we will not go into detail. (Regular expression, aka regex, are not specific to R).

The above command results in a vector `c(FALSE, TRUE, TRUE)` because the second and third items start with J.

So we can plug it into brackets: `~ people[grepl("^J", x = people)] ~`

And that will result in a vector of `c("Joseph", "James")`.

Finally, we can use the `which()` function to convert a TRUE/FALSE vector to index numbers. This function is like Excel's MATCH() formula:

```
which(grepl("^J", x = people))
```

This can be read as “which items in the *people* vector start with J?”. And the result is a vector of `c(2, 3)`, indicating the second and third items in the *people* vector fit the criteria.

c. Run functions on a vector

At this point you have already seen enough examples applying functions to vectors; for example, `length()`, `nchar()`, `toupper()`, `max()` and others. In the next chapter we will review functions in more detail, and share a list of commonly-used Excel formulas and their R counterparts.

4.6 How vector operations work

You can think of vector operations as being performed pairwise, item-by-item across vectors. The following example shows the result of adding two vectors of equal length:

Code: Adding vectors of equal length

```
> c(15, 13, 5, 2) + c(5, 8, 17, 21)
[1] 20 21 22 23
```

The first element of the two vectors are added (15+5) and becomes the first element in the result (20). Next 13+8 results in 21, and so forth. As an Excel user, you can think of this as adding two cell ranges to result in a range of equal length.

This logic does not only apply to arithmetic. It applies to any kind of operation on a vector. Here are some more:

Code: More vector operations

```
> # define two vectors
> texts = c("first", "second", "third", "fourth", "fifth")
> texts_2 = c("1st", "2nd", "3rd", "4th", "5th")
> booleans = c(TRUE, TRUE, TRUE, FALSE, FALSE)

> # 1. concatenate the two vectors with paste()
> paste(texts, texts_2)
[1] "first 1st" "second 2nd" "third 3rd" "fourth 4th" "fifth 5th"

> # 2. use booleans in ifelse()
> ifelse(booleans == TRUE, paste(texts, texts_2), "IGNORE")
[1] "first 1st" "second 2nd" "third 3rd" "IGNORE" "IGNORE"

> # 3. use boolean again
> ifelse(booleans == TRUE, nchar(texts), -999)
[1] 5 6 5 -999 -999
```

We first created a few vectors. In the first example we concatenate two of the vectors with the `paste()` function. You see the same parallel operation here: concatenate the first string in each vector, then the second string in each vector, and so on, and the result is a vector with five strings.

In the second and third examples we use `ifelse()` which is like Excel's IF(). The function looks at each value of *booleans*, and when TRUE, concatenates the

corresponding values from *texts* and *texts_2*; when `FALSE`, it returns a dummy value. In both cases it's all pairwise operation again. That is, the first values of each vector are paired up, then the second value, and so on.

Recycling

But how about operating on vectors of different lengths? Let's see with a simple example:

Code: Vector recycling

```
> # define two vectors
> texts_2 = c("1st", "2nd", "3rd", "4th", "5th")
> one_word = "string"

> # concatenate the two vectors with paste()
> paste(texts_2, one_word)
[1] "1st string" "2nd string" "3rd string" "4th string" "5th string"
```

To start we defined two vectors, but one of them only has one value. Remember, this is a vector even if it does not look like one. When we `paste()` the two vectors, we see the word “string” is appended to each value of text. Why is that?

In the background, R is *recycling* the shorter vector to make it the same length as the longer one. It does so by duplicating the shorter one as many times as needed to make it the same length as the longer one.

Here are some more examples of that:

Code: Even recycling

```
> # 4-cell vector plus 1-cell vector
> c(15, 13, 5, 2) + 5
[1] 20 18 10 7

> # Comparing 4-cell vs 2-cell vector
> c(15, 13, 5, 2) > c(10, 1)
[1] TRUE TRUE FALSE TRUE
```

The first example is a lot like the `paste()` example above: we are joining a 4-item vector with a single-item vector. This results in 15+5, 13+5, 5+5 and 2+5 because the one-item vector is converted to a four-element with the same value. In the second example, R essentially compares `c(15,13,5,2)` to `c(10,1,10,1)` by duplicating that second vector, resulting in a four-element vector of TRUE and FALSE values.

These examples are “even” because the length of the big vector is a multiple of the length of the small vector. If that is not the case, R will issue a warning. Here is an example:

Code: Uneven recycling

```
> c(15, 13, 5, 2) + c(5, 8, 17)
[1] 20 21 22 7
Warning message:
In c(15, 13, 5, 2) + c(5, 8, 17) :
  longer object length is not a multiple of shorter object length
```

By recycling the three-element vector, R effectively adds `c(15,13,5,2)` to `c(5,8,17,5)` where that last 5 is the first value of the shorter vector. R issues a warning that is quite intuitive.

Generally speaking there is no reason to operate on uneven vectors, and if you ever come across this warning, chances are you might have done something wrong.

4.7 Summary

One of those moments when R really started to make sense was when I realized how flexible vectors are and how important they are to effectively working with data frames.

This stuff might seem a bit abstract and not practical; but hang in there, because it will all make more sense once you start working with data frames. In fact, you might want to return here after reading Chapters 7-9.

Next you will learn about functions, the other building block of R.

5. Like the sample?

You just read 4 of the 17 chapters. I hope you liked it!

When you're ready to get the full version (print or digital), go to www.rforexcelusers.com/book.

- Get a 30-day refund if you don't like it
- Early access to new learning tools