Data Wrangling with R

Bradley C. Boehmke

Data Wrangling with R

Bradley C. Boehmke

Contents

Preface 1
Who this Book is For 1
What You Need For this Book 1
Reader Feedback 2
Colophon 2

Introduction 3

The Role of Data Wrangling 5

Introduction to R 7
Open Source 7
Flexibility 8
Community 9

R Basics 10
Assignment & Evaluation 10
Vectorization 11
Getting help 13
Workspace 15
Working with packages 17
Style guide 19

Working with Different Types of Data in R 23

Dealing with Numbers 24
Integer vs. Double 24
Generating sequence of non-random numbers 25
Generating sequence of random numbers 26
Setting the seed for reproducible random numbers 30
Comparing numeric values 31
Rounding numbers 33

Dealing with Character Strings **34**
Character string basics 34
String manipulation with base R 39
String manipulation with `stringr` 42
Set operatons for character strings 46

Dealing with Regular Expressions **49**
Regex Syntax 49
Regex Functions 54
Additional resources 61

Dealing with Factors **62**
Creating, converting & inspecting factors 62
Ordering levels 63
Revalue levels 64
Dropping levels 64

Dealing with Dates **66**
Getting current date & time 66
Converting strings to dates 66
Extract & manipulate parts of dates 68
Creating date sequences 70
Calculations with dates 71
Dealing with time zones & daylight savings 73
Additional resources 74

Managing Data Structures in R 75

Data Structure Basics **76**
Identifying the Structure 76
Attributes 76

Managing Vectors **79**
Creating 79
Adding on to 80
Adding attributes 80
Subsetting 81

Managing Lists **85**
Creating 85
Adding on to 85
Adding attributes 87
Subsetting 89

Managing Matrices . . . **93**
Creating . . . 93
Adding on to . . . 95
Adding attributes . . . 95
Subsetting . . . 98

Managing Data Frames . . . **100**
Creating . . . 100
Adding on to . . . 102
Adding attributes . . . 104
Subsetting . . . 106

Dealing with Missing Values . . . **109**
Testing for missing values . . . 109
Recoding missing values . . . 110
Excluding missing values . . . 110

Importing, Scraping, and Exporting Data with R . . . **113**

Importing Data . . . **114**
Reading data from text files . . . 114
Reading data from Excel files . . . 118
Load data from saved R object file . . . 124
Additional resources . . . 124

Scraping Data . . . **126**
Importing tabular and Excel files stored online . . . 126
Scraping HTML text . . . 132
Scraping HTML table data . . . 143
Working with APIs . . . 151
Additional Resources . . . 165

Exporting Data . . . **166**
Writing data to text files . . . 166
Writing data to Excel files . . . 168
Saving data as an R object file . . . 173
Additional resources . . . 174

Creating Efficient & Readable Code in R . . . **175**

Functions . . . **176**
Function Components . . . 176
Arguments . . . 177

Scoping Rules . . . 178
Lazy Evaluation . . . 180
Returning Multiple Outputs from a Function . . . 181
Dealing with Invalid Parameters . . . 182
Saving and Sourcing Functions . . . 183
Additional Resources . . . 186

Loop Control Statements . . . 187
Basic control statements (i.e. `if`, `for`, `while`, etc.) . . . 187
Apply family . . . 195
Other useful "loop-like" functions . . . 201
Additional Resources . . . 203

Simplify Your Code with `%>%` . . . 204
Pipe (%>%) Operator . . . 204
Additional Functions . . . 208
Additional Pipe Operators . . . 209
Additional Resources . . . 212

Shaping & Transforming Your Data with R . . . 214

Reshaping Your Data with `tidyr` . . . 215
Making wide data long . . . 215
Making long data wide . . . 217
Splitting a single column into multiple columns . . . 218
Combining multiple columns into a single column . . . 219
Additional `tidyr` functions . . . 220
Sequencing your `tidyr` operations . . . 222
Additional resources . . . 223

Transforming Your Data with `dplyr` . . . 224
Selecting variables of interest . . . 225
Filtering rows . . . 226
Grouping data by categorical variables . . . 227
Performing summary statistics on variables . . . 228
Arranging variables by value . . . 230
Joining datasets . . . 232
Creating new variables . . . 234
Additional resources . . . 238

Preface

Welcome to Data Wrangling with R! In this book, I will help you learn the essentials of preprocessing data leveraging the R programming language to easily and quickly turn noisy data into usable pieces of information. Data wrangling, which is also commonly referred to as data munging, transformation, manipulation, janitor work, etc. can be a painstakenly laborious process. In fact, it has been stated that up to 80% of data analysis is spent on the process of cleaning and preparing data[1]. However, being a prerequisite to the rest of the data analysis workflow (visualization, modeling, reporting), it's essential that you become fluent *and* efficient in data wrangling techniques.

This book will guide you through the data wrangling process along with give you a solid foundation of the basics of working with data in R. My goal is to teach you how to easily wrangle your data, so you can spend more time focused on understanding the content of your data via visualization, modeling, and reporting your results. By the time you finish reading this book, you will have learned how to work with the different data types and structures, acquire and parse data from locations you may not have been able to access before, develop your own functions, manage control structures, reshape the layout of your data, and manipulate, summarize, and join data sets. In essence, you will have the data wrangling toolbox required for modern day data analysis.

Who this Book is For

This book is meant to establish the baseline R vocabulary and knowledge for the primary data wrangling processes. This captures a wide range of programming activities which covers the full spectrum from understanding basic data objects in R to writing your own functions, applying loops, and webscraping. As a result, this book can be beneficial to all levels of R programmers. Beginner R programmers will gain a basic understanding of the functionality of R along with learning how to work with data using R. Intermediate and/or advanced R programmers will likely find the early chapters reiterating established knowledge; however, these programmers will benefit from the mid and later chapters by learning newer and/or more efficient data wrangling techniques.

What You Need For this Book

Obviously to gain and retain knowledge from this book it is highly recommended that you follow along and practice the code examples yourself. Furthermore, this book assumes that you will actually be performing data wrangling in R; therefore, it is assumed that you have or plan to have R installed

[1] cf. Wickham, 2014 and Dasu and Johnson, 2003

on your computer. You will find the latest version of R for Linux, Mac OS, and Windows at cran.r-project.org/[2]. It is also recommended that you use an integrated development environment (IDE) as it will simplify and organize your coding environment greatly. There are several to choose from; however, I highly recommend RStudio[3].

Reader Feedback

Reader comments are greatly appreciated. Please send any feedback regarding typos, mistakes, confusing statements, or opportunities for improvement to wranglingdata@gmail.com.

Colophon

This book was written in Rmarkdown with Rstudio. The source code is hosted on GitHub and automatically published to LeanPub. Cover image provided by Gabriela de Queiroz[4]

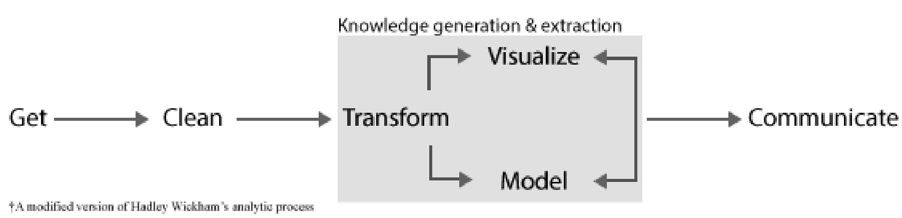
[2] https://cran.r-project.org/
[3] https://www.rstudio.com/
[4] https://twitter.com/gdequeiroz

Introduction

> *"With nothing but the power of your own mind, you operate on the symbols before you in such a way that you gradually lift yourself from a state of understanding less to one of understanding more."* - Mortimer J. Adler

Data. Our world has become increasingly reliant upon, and awash in, this resource. Businesses are increasingly seeking to capitalize on data analysis as a means for gaining competitive advantages[5]. Government agencies are using more types of data to improve operations and efficiencies[6]. Sports entities are increasing the range of data applications, from how teams are using data and analytics[7] to how data are impacting the experience for the fan base[8]. Journalism is increasing the role that numerical data are used in the production and distribution of information as evidenced by the emergining field of data journalism[9]. In fact, the need to work with data has become so prevalent that the U.S. alone is expected to have a shortage of 140,000 to 190,000 data analysts by 2018[10]. Consequently, it is safe to say there is a need for becoming fluent with the data analysis process. And I'm assuming that's why you are reading this book.

Fluency in data analysis captures a wide range of activities. At its most basic structure, data analysis fluency includes the ability to get, clean, transform, visualize, and model data along with communicating your results as depicted in the following illustration.



Analytic Process

From project to project, no analytic process will be the same. Each specific instance of data analysis includes unique, different, and often multiple requirements regarding the specific processes required

[5]See Davenport, 2006; Trkman et al., 2010; McAfee & Brynjolfsson, 2012; Waller & Fawcett, 2013 to name a few.

[6]Numerous examples exist but this article outlines several specific cases.

[7]Recent examples are illustrated in Forbes, GeekWire, and the Huffington Post.

[8]See here, here, and here for examples of how data is influencing the experience for fans.

[9]This is evidenced by the popularity of such data journalism productions such as FiveThirtyEight, UpShot, and Vox. This USAToday article further articulates this emerging demand.

[10]http://www.mckinsey.com/features/big_data

for each stage. For instance, getting data may include simply accessing an Excel file, scraping data from an HTML table, or using an API[11] to access a database. Cleaning data may include reshaping data from a wide to long format, parsing variables, and/or transforming variables to different formats. Transforming data may include filtering, summarizing, and applying common/uncommon functions to data along with joining multiple datasets. Visualizing data may range from common static exploratory data analysis plots to dynamic, interactive data visualizations in web browsers. And modeling data can be even more diverse covering the range of descriptive[12], predictive[13], and prescriptive[14] analytic techniques.

Consequently, the road to becoming an expert in data analysis can be daunting. And, in fact, obtaining expertise in the wide range of data analysis processes utilized in your own respective field is a career long process. However, the goal of this book is to help you take a step closer to fluency in the early stages of the analytic process. Why? Because before using statistical literate programming to report your results, before developing an optimization or predictive model, before performing exploratory data analysis, before visualizing your data, you need to be able to manage your data. You need to be able to import your data. You need to be able to work with the different data types. You need to be able to subset and parse your data. You need to be able to manipulate and transform your data. You need to be able to *wrangle* your data!

---

[11] https://en.wikipedia.org/wiki/Application_programming_interface

[12] https://en.wikipedia.org/wiki/Descriptive_statistics

[13] https://en.wikipedia.org/wiki/Predictive_analytics

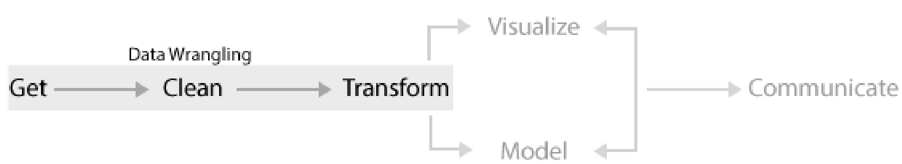
[14] https://en.wikipedia.org/wiki/Prescriptive_analytics

The Role of Data Wrangling

> *"Water, water, everywhere, nor any a drop to drink"* - Samuel Taylor Coleridge

Synonymous to Samuel Taylor Coleridge's quote in *Rime of the Ancient Mariner*, the degree to which data are useful is largely determined by an analysts ability to wrangle data. In spite of advances in technologies for working with data, analysts still spend an inordinate amount of time obtaining data, diagnosing data quality issues and pre-processing data into a usable form. Research has illustrated that this portion of the data analysis process is the most tedious and time consuming component; often consuming 50-80% of an analyst's time[15]. Despite the challenges, data wrangling remains a fundamental building block that enables visualization and statistical modeling. Only through data wrangling can we make data *useful*. Consequently, one's ability to perform data wrangling tasks effectively and efficiently is fundamental to becoming an expert data analyst in their respective domain.

So what exactly is this thing called *data wrangling*? Its the ability to take a messy, unrefined source of data and wrangle it into something useful. It's the art of using computer programming to extract raw data and creating clear and actionable bits of information for your analysis. Data wrangling is the entire front end of the analytic process and requires numerous tasks that can be categorized within the *get*, *clean*, and *transform* components.



Data Wrangling

However, learning how to wrangle your data does not necessarily follow a linear progression as suggested by the above figure. In fact, you need to start from scratch to understand how to work with data in R. Consequently, this book takes a meandering route through the data wrangling process to help build a solid data wrangling foundation.

First, modern day data wrangling requires being comfortable writing code. If you are new to writing code, R or RStudio you need to understand some of the basics of working in the "command line" environment. The next two chapters in this section will introduce you to R, discuss the benefits it provides, and then start to get you comfortable at the command line by walking you through the process of assigning and evaluating expressions, using vectorization, getting help, managing your

[15] See Dasu & Johnson, 2003; Kandel et al., 2011; Wickham, 2013.

workspace, and working with packages. Lastly, I offer some basic styling guidelines to help you write code that is easier to digest by others.

Second, data wrangling requires the ability to work with different forms of data. Analysts and organizations are finding new and unique ways to leverage all forms of data so it's important to be able to work not only with numbers but also with character strings, categorical variables, logical variables, regular expression, and dates. Section two explains how to work with these different classes of data so that when you start to learn how to manage the different data structures, which combines these data classes into multiple dimensions, you will have a strong knowledge base.

Third, modern day datasets often contain variables of different lengths and/or classes. Furthermore, many statistical and mathematical calculations operate on different types of data structures. Consequently, data wrangling requires a strong knowledge of the different structures to hold your datasets. Section three covers the different types of data structures available in R, how they differ by dimensionality and how to create, add to, and subset the various data structures. Lastly, I cover how to deal with missing values in data structures. Consequently, this section provides a robust understanding of managing various forms of datasets.

Fourth, data are arriving from multiple sources at an alarming rate and analysts and organizations are seeking ways to leverage these new sources of information. Consequently, analysts need to understand how to *get* data from these sources. Furthermore, since analysis is often a collaborative effort analysts also need to know how to share their data. Section four covers the basics of importing tabular and spreadsheet data, scraping data stored online, and exporting data for sharing purposes.

Fifth, minimizing duplication and writing simple and readable code is important to becoming an effective and efficient data analyst. Moreover, clarity should always be a goal throughout the data analysis process. Section five introduces the art of writing functions and using loop control statements to reduce redundancy in code. I also discuss how to simplify your code using pipe operators to make your code more readable. Consequently, this section will help you to perform data wrangling tasks more effectively, efficiently, and with more clarity.

Last, data wrangling is all about getting your data into the right form in order to feed it into the visualization and modeling stages. This typically requires a large amount of reshaping and transforming of your data. Section six introduces some of the fundamental functions for *"tidying"* your data and for manipulating, sorting, summarizing, and joining your data. These tasks will help to significantly reduce the time you spend on the data wrangling process.

Individually, each section will provide you important tools for performing individual data wrangling tasks. Combined, these tools will help to make you more effective and efficient in the front end of the data analysis process so that you can spend more of your time visualizing and modeling your data and communicating your results!

Introduction to R

A language for data analysis and graphics. This definition of R was used by Ross Ihaka and Robert Gentleman in the title of their 1996 paper[16] outlining their experience of designing and implementating the R software. It's safe to say this remains the essence of what R is; however, it's tough to encapsulate such a diverse programming language into a single phrase.

During the last decade, the R programming language has become one of the most widely used tools for statistics and data science. Its application runs the gamut from data preprocessing, cleaning, web scraping and visualization to a wide range of analytic tasks such as computational statistics, econometrics, optimization, and natural language processing. In 2012 R had over 2 million users[17] and continues to grow by double digit percentage points every year. R has become an essential analytic software throughout industry; being used by organizations such as Google, Facebook, New York Times, Twitter, Etsy, Department of Defense, and even in presidential political campaigns.

So what makes R such a popular tool?

Open Source

R is an *open source* software created over 20 years ago by Ihaka and Gentleman at the University of Auckland, New Zealand. However, its history is even longer as its lineage goes back to the S programming language created by John Chambers out of Bell Labs back in the 1970s.[18] R is actually a combination of S with lexical scoping semantics inspired by Scheme.[19] Whereas the resulting language is very similar in appearance to S, the underlying implementation and semantics are derived from Scheme. Unbeknownst to many the S language has been a popular vehicle for research in statistical methodology, and R provides an *open source* route to participate in that activity.

Although the history of S and R is interesting[20], the principal artifact to observe is that R is an *open source* software. Although some contest that open-source software is merely a "craze" [21], most evidence suggests that open-source is here to stay and represents a *new*[22] norm for programming languages. Open-source software such as R blurs the distinction between developer and user which provides the ability to extend and modify the analytic functionality to your, or your organization's

[16] Ihaka, R., & Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3), 299-314.

[17] http://www.oracle.com/us/corporate/press/1515738

[18] Consequently, R is named partly after its authors (Ross and Robert) and partly as a play on the name of S.

[19] Morandat, Frances; Hill, Brandon (2012). Evaluating the design of the R language: objects and functions for data analysis. *ECOOP'12 Proceedings of the 26th European conference on Object-Oriented Programming.*

[20] See Roger Peng's R programming for Data Science for further, yet concise, details on S and R's history.

[21] This was recently argued by Pollack et al. which was appropriately rebutted by Boehmke & Jackson. See my post which provides both articles.

[22] Open-source is far from new as its been around for decades (i.e. A-2 in the 1950s, IBM's ACP in the '60s, Tiny BASIC in the '70s) but has gained prominence since the late 1990s.

needs. The data analysis process is rarely restricted to just a handful of tasks with predictable input and outputs that can be pre-defined by a fixed user interface as is common in proprietary software. Rather, as previosly mentioned in the introduction, data analysis includes unique, different, and often multiple requirements regarding the specific tasks involved. Open source software allows more flexibility for you, the data analyst, to manage how data are being transformed, manipulated, and modeled "under the hood" of software rather than relying on "stiff" point and click software interfaces. Open source also allows you to operate on every major platform rather than be restricted to what your personal budget allows or the idiosyncratic purchases of organizations.

This invariably leads to new expectations for data analysts; however, organizations are proving to greatly value the increased technical abilities of open source data analysts as evidenced by a recent O'Reilly survey revealing that data analysts focusing on open source technologies make more money than those still dealing in proprietary technologies.

Flexibility

Another benefit of open source is that anybody can access the source code, modify and improve it. As a result, many excellent programmers contribute to improving existing R code and developing new capabilities. Researchers from all walks of life (academic institutions, industry, and focus groups such as RStudio[23] and rOpenSci[24]) are contributing to advancements of R's capabilities and best practices. This has resulted in some powerful tools that advance both statistical and non-statistical modeling capabilities that are taking data analysis to new levels.

Many researchers in academic institutions are using and developing R code to develop the latest techniques in statistics and machine learning. As part of their research, they often publish an R package to accompany their research articles[25]. This provides immediate access to the latest analytic techniques and implementations. And this research is not soley focused on generalized algorithms as many new capabilities are in the form of advancing analytic algorithms for tasks in specific domains. A quick assessment of the different task domains[26] for which code is being developed illustrates the wide spectrum - econometrics, finance, chemometrics & computational physics, pharmacokinetics, social sciences, etc.

Powerful tools are also being developed to perform many tasks that greatly aid the data analysis process. This is not limited to just new ways to wrangle your data but also new ways to visualize and communicate data. R packages are now making it easier than ever to create interactive graphics and websites and produce sophisticated html and pdf reports. R packages are also integrating communication with high-performance programming languages such as C, Fortran, and C++ making data analysis more powerful, efficient, and posthaste than ever.

So although the analytic montra *"use the right tool for the problem"* should always be in our prefrontal cortex, the advancements and flexibility of R is making it the right tool for many problems.

[23]https://www.rstudio.com

[24]https://ropensci.org/packages/

[25]See The Journal of Statistical Software and The R Journal

[26]https://cran.r-project.org/web/views/

Community

The R community is fantastically diverse and engaged. On a daily basis, the R community generates opportunities and resources for learning about R. These cover the full spectrum of training - books[27], online courses[28], R user groups[29], workshops[30], conferences[31], etc. And with over 2 million users and developers, finding help and technical expertise is only a simple click away. Support is available through R mailing lists[32], Q&A websites[33], social media networks[34], and numerous blogs[35].

So now that you know how awesome R is, it's time to learn how to use it.

[27]http://www.amazon.com/s/ref=nb_sb_noss_2?url=search-alias%3Daps&field-keywords=r+programming

[28]https://www.coursera.org/specializations/jhu-data-science

[29]http://blog.revolutionanalytics.com/local-r-groups.html

[30]https://www.rstudio.com/resources/training/workshops

[31]https://www.r-project.org/conferences.html

[32]https://www.r-project.org/mail.html

[33]Stack Overflow and CrossValidated are two great Q&A sources

[34]www.twitter.com/search/rstats

[35]http://www.r-bloggers.com/

R Basics

> *"Programming is like kicking yourself in the face, sooner or later your nose will bleed."*
> - Kyle Woodbury

A computer language is described by its *syntax* and *semantics*; where syntax is about the the grammar of the language and semantics the meaning behind the sentence. And jumping into a new programming language correlates to visiting a foreign country with only that 9th grade Spanish 101 class under your belt; there is no better way to learn than to immerse yourself in the environment! Although it'll be painful early on and your nose will surely bleed, eventually you'll learn the dialect and the quircks that come along with it.

Throughout this book you'll learn much of the fundamental syntax and semantics of the R programming language; and hopefully with minimal face kicking involved. However, this chapter serves to introduce you to many of the basics of R to get you comfortable. This includes understanding how to assign and evaluate expressions, the idea of vectorization, how to get help, how to manage your workspace, and how to work with packages. Finally, I offer some basic styling guidelines to help you write code that is easier to digest by others.

Assignment & Evaluation

The first operator you'll run into is the assignment operator. The assignment operator is used to *assign* a value. For instance we can assign the value `3` to the variable `x` using the `<-` assignment operator. We can then evaluate the variable by simply typing `x` at the command line which will return the value of `x`. Note that prior to the value returned you'll see `## [1]` in the command line. This simply implies that the output returned is the first output. Note that you can type any comments in your code by preceding the comment with the hashtag (`#`) symbol. Any values, symbols, and texts following `#` will not be evaluated.

```
# assignment
x <- 3

# evaluation
x
## [1] 3
```

Interestingly, R actually allows for five assignment operators:

```
# leftward assignment
x <- value
x = value
x <<- value

# rightward assignment
value -> x
value ->> x
```

The original assignment operator in R was `<-` and has continued to be the preferred among R users. The = assignment operator was added in 2001[36] primarily because it is the accepted assignment operator in many other languages and beginners to R coming from other languages were so prone to use it. However, R uses = to associate function arguments with values (i.e. f(x = 3) explicitly means to call function f and set the argument x to 3. Consequently, most R programmers prefer to keep = reserved for argument association and use `<-` for assignment.

The operators `<<-` is normally only used in functions which we will not get into the details. And the rightward assignment operators perform the same as their leftward counterparts, they just assign the value in an opposite direction.

Overwhelmed yet? Don't be. This is just meant to show you that there are options and you will likely come across them sooner or later. My suggestion is to stick with the tried and true `<-` operator. This is the most conventional assignment operator used and is what you will find in all the base R source code…which means it should be good enough for you.

Lastly, note that R is a case sensitive programming language. Meaning all variables, functions, and objects must be called by their exact spelling:

```
x <- 1
y <- 3
z <- 4
x * y * z
## [1] 12

x * Y * z
## Error in eval(expr, envir, enclos): object 'Y' not found
```

Let's move on.

Vectorization

A key difference between R and many other languages is a topic known as vectorization. What does this mean? It means that many functions that are to be applied individually to each element in a

[36] http://developer.r-project.org/equalAssign.html

vector of numbers require a *loop* assessment to evaluate; however, in R many of these functions have been coded in C to perform much faster than a `for` loop would perform. For example, let's say you want to add the elements of two seperate vectors of numbers (`x` and `y`).

```
x <- c(1, 3, 4)
y <- c(1, 2, 4)

x
## [1] 1 3 4
y
## [1] 1 2 4
```

In other languages you might have to run a loop to add two vectors together. In this `for` loop I print each iteration to show that the loop calculates the sum for the first elements in each vector, then performs the sum for the second elements, etc.

```
# empty vector
z <- as.vector(NULL)

# `for` loop to add corresponding elements in each vector
for (i in seq_along(x)) {
        z[i] <- x[i] + y[i]
        print(z)
}
## [1] 2
## [1] 2 5
## [1] 2 5 8
```

Instead, in R, + is a vectorized function which can operate on entire vectors at once. So rather than creating `for` loops for many function, you can just use simple syntax:

```
x + y
## [1] 2 5 8
x * y
## [1]  1  6 16
x > y
## [1] FALSE  TRUE FALSE
```

When performing vector operations in R, it is important to know about *recycling*. When performing an operation on two or more vectors of unequal length, R will recycle elements of the shorter vector(s) to match the longest vector. For example:

```
long <- 1:10
short <- 1:5

long
##  [1]  1  2  3  4  5  6  7  8  9 10
short
## [1] 1 2 3 4 5

long + short
##  [1]  2  4  6  8 10  7  9 11 13 15
```

The elements of `long` and `short` are added together starting from the first element of both vectors. When R reaches the end of the `short` vector, it starts again at the first element of `short` and contines until it reaches the last element of the `long` vector. This functionality is very useful when you want to perform the same operation on every element of a vector. For example, say we want to multiply every element of our vector long by 3:

```
long <- 1:10
c <- 3

long * c
##  [1]  3  6  9 12 15 18 21 24 27 30
```

Remember there are no scalars in R, so `c` is actually a vector of length 1; in order to add its value to every element of `long`, it is recycled to match the length of `long`.

When the length of the longer object is a multiple of the shorter object length, the recycling occurs silently. When the longer object length is not a multiple of the shorter object length, a warning is given:

```
even_length <- 1:10
odd_length <- 1:3

even_length + odd_length
## Warning in even_length + odd_length: longer object length is not a multiple
## of shorter object length
##  [1]  2  4  6  5  7  9  8 10 12 11
```

Getting help

Learning any new language requires lots of help. Luckily, the help documentation and support in R is comprehensive and easily accessible from the command line. To leverage general help resources you can use the following:

```
# provides general help links
help.start()

# searches the help system for documentation matching a given character string
help.search("text")
```

Note that the `help.search("some text here")` function requires a character string enclosed in quotation marks. So if you are in search of time series functions in R, using `help.search("time series")` will pull up a healthy list of vignettes and code demonstrations that illustrate packages and functions that work with time series data.

Getting Help on Functions

For more direct help on functions that are installed on your computer:

```
# provides details for specific function
help(functionname)

# provides same information as help(functionname)
?functionname

# provides examples for said function
example(functionname)
```

Note that the `help()` and `?` function calls only work for functions within loaded packages. If you want to see details on a function in a package that is installed on your computer but not loaded in the active R session you can use `help(functionname, package = "packagename")`. Another alternative is to use the `::` operator as in `help(packagename::functionname)`.

Getting Help from the Web

Typically, a problem you may be encountering is not new and others have faced, solved, and documented the same issue online. The following resources can be used to search for online help. Although, I typically just google the problem and find answers relatively quickly.

- `RSiteSearch("key phrase")`: searches for the key phrase in help manuals and archived mailing lists on the R Project website[37].
- Stack Overflow[38]: a searchable Q&A site oriented toward programming issues. 75% of my answers typically come from Stack Overflow.

[37]"http://search.r-project.org/"
[38]http://stackoverflow.com/

- Cross Validated[39]: a searchable Q&A site oriented toward statistical analysis.
- R-seek[40]: a Google custom search that is focused on R-specific websites
- R-bloggers[41]: a central hub of content collected from over 500 bloggers who provide news and tutorials about R.

Workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions). The following code provides the basics for understanding, configuring and customizing your current R environment.

Working Directory

The *working directory* is the default location for all file inputs and outputs.

```
# returns path for the current working directory
getwd()

# set the working directory to a specified directory
setwd(directory_name)
```

For example, if I call `getwd()` the file path "/Users/bradboehmke/Desktop/Personal/Data Wrangling" is returned. If I want to set the working directory to the "Workspace" folder within the "Data Wrangling" directory I would use `setwd("Workspace")`. Now if I call `getwd()` again it returns "/Users/bradboehmke/Desktop/Personal/Data Wrangling/Workspace".

Environment Objects

To identify or remove the objects (i.e. vectors, data frames, user defined functions, etc.) in your current R environment:

[39]http://stats.stackexchange.com/
[40]http://rseek.org
[41]http://www.r-bloggers.com/

```
# list all objects
ls()

# identify if an R object with a given name is present
exists("object_name")

# remove defined object from the environment
rm("object_name")

# you can remove multiple objects by using the `c()` function
rm(c("object1", "object2"))

# basically removes everything in the working environment -- use with caution!
rm(list = ls())
```

Command History

You can view previous commands one at a time by simply pressing the up arrow on your keyboard or view a defined number of previous commands with:

```
# default shows 25 most recent commands
history()

# show 100 most recent commands
history(100)

# show entire saved history
history(Inf)
```

Saving & Loading

You can save and load your workspaces. Saving your workspace will save all R files and objects within your workspace to a .RData file.

```
# save all items in workspace to a .RData file
save.image()

# save specified objects to a .RData file
save(object1, object2, file = "myfile.RData")

# load workspace into current session
load("myfile.RData")
```

Note that saving the workspace without specifying the working directory will default to saving in the current directory. You can further specify where to save the .RData by including the path: `save(object1, object2, file = "/users/name/folder/myfile.RData")`

Workspace Options

You can view and set options for the current R session:

```
# learn about available options
help(options)

# view current option settings
options()

# change a specific option (i.e. number of digits to print on output)
options(digits=3)
```

Shortcuts

To access a menu displaying all the shortcuts in RStudio you can use option + shift + k. Within RStudio you can also access them in the Help menu » Keyboard Shortcuts.

Working with packages

In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests and provides an easy method to share with others[42]. As of September 2015 there were over 7000 packages available on CRAN[43], 1000 on Bioconductor[44], and countless more available through GitHub[45]. This huge variety of packages is one of the reasons that R is so successful: chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package.

[42] Wickham, H. (2015). *R packages*. "O'Reilly Media, Inc.".
[43] https://cran.r-project.org
[44] https://www.bioconductor.org
[45] https://github.com

Installing Packages

To install packages:

```
# install packages from CRAN
install.packages("packagename")
```

As previously stated, packages are also available through Bioconductor and GitHub. To download Bioconductor packages:

```
# link to Bioconductor URL
source("http://bioconductor.org/biocLite.R")

# install core Bioconductor packages
biocLite()

# install specific Bioconductor package
biocLite("packagename")
```

And to download GitHub packages:

```
# the devtools package provides a simply function to download GitHub packages
install.packages("devtools")

# install package which exists at github.com/username/packagename
devtools::install_github("username/packagename")
```

Loading Packages

Once the package is downloaded to your computer you can access the functions and resources provided by the package in two different ways:

```
# load the package to use in the current R session
library(packagename)

# use a particular function within a package without loading the package
packagename::functionname
```

For instance, if you want to have full access to the tidyr package you would use `library(tidyr)`; however, if you just wanted to use the `gather()` function without loading the tidyr package you can use `tidyr::gather(function arguments)`.

Getting Help on Packages

For help on packages that are installed on your computer:

```r
# provides details regarding contents of a package
help(package = "packagename")

# see all packages installed
library()

# see packages currently loaded
search()

# list vignettes available for a specific package
vignette(package = "packagename")

# view specific vignette
vignette("vignettename")

# view all vignettes on your computer
vignette()
```

Note that some packages will have multiple vignettes. For instance `vignette(package = "grid")` will list the 13 vignettes available for the grid package. To access one of the specific vignettes you simply use `vignette("vignettename")`.

Useful packages

There are thousands of helpful R packages for you to use, but navigating them all can be a challenge. To help you out, RStudio compiled a guide[46] to some of the best packages for loading, manipulating, visualizing, analyzing, and reporting data. In addition, their list captures packages that specialize in spatial data, time series and financial data, increasing spead and performance, and developing your own R packages.

Style guide

> *"Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read."* - Hadley Wickham

As a medium of communication, its important to realize that the readability of code does in fact make a difference. Well styled code has many benefits to include making it easy to *i*) read, *ii*) extend, and *iii*) debug. Unfortunately, R does not come with official guidelines for code styling but such is an inconvenient truth of most open source software. However, this should not lead you to believe

[46] https://support.rstudio.com/hc/en-us/articles/201057987-Quick-list-of-useful-R-packages

there is no style to be followed and over time implicit guidelines for proper code styling have been documented. What follows are guidelines that have been widely accepted as good practice in the R community and are based on Google's[47] and Hadley Wickham's[48] R style guides.

Notation and naming

File names should be meaningful and end with a `.R` extension.

```
# Good
weather-analysis.R
emerson-text-analysis.R

# Bad
basic-stuff.r
detail.r
```

If files need to be run in sequence, prefix them with numbers:

```
0-download.R
1-preprocessing.R
2-explore.R
3-fit-model.R
```

In R, naming conventions for variables and function are famously muddled. They include the following:

```
namingconvention        # all lower case; no separator
naming.convention       # period separator
naming_convention       # underscore separator
namingConvention        # lower camel case
NamingConvention        # upper camel case
```

Historically, there has been no clearly preferred approach with multiple naming styles sometimes used within a single package. Bottom line, your naming convention will be driven by your preference but the ultimate goal should be consistency.

My personal preference is to use all lowercase with an underscore (_) to separate words within a name. This follows Hadley Wickham's suggestions in his style guide. Furthermore, variable names should be nouns and function names should be verbs to help distinguish their purpose. Also, refrain from using existing names of functions (i.e. mean, sum, true).

[47] https://google.github.io/styleguide/Rguide.xml
[48] http://adv-r.had.co.nz/Style.html

Organization

Organization of your code is also important. There's nothing like trying to decipher 2,000 lines of code that has no organization. The easiest way to achieve organization is to comment your code. The general commenting scheme I use is the following.

I break up principal sections of my code that have a common purpose with:

```
#################
# Download Data #
#################
lines of code here

###################
# Preprocess Data #
###################
lines of code here

########################
# Exploratory Analysis #
########################
lines of code here
```

Then comments for specific lines of code can be done as follows:

```
code_1  # short comments can be placed to the right of code
code_2  # blah
code_3  # blah

# or comments can be placed above a line of code
code_4

# Or extremely long lines of commentary that go beyond the suggested 80
# characters per line can be broken up into multiple lines. Just don't forget
# to use the hash on each.
code_5
```

Syntax

The maximum number of characters on a single line of code should be 80 or less. If you are using RStudio you can have a margin displayed so you know when you need to break to a new line.[49]

[49]Go to *RStudio* on the menu bar then *Preferences* > *Code* > *Display* and you can select the "show margin" option and set the margin to 80.

This allows your code to be printed on a normal 8.5 x 11 page with a reasonably sized font. Also, when indenting your code use two spaces rather than using tabs. The only exception is if a line break occurs inside parentheses. In this case align the wrapped line with the first character inside the parenthesis:

```
super_long_name <- seq(ymd_hm("2015-1-1 0:00"),
                       ymd_hm("2015-1-1 12:00"),
                       by = "hour")
```

Proper spacing within your code also helps with readability. The following pulls straight from Hadley Wickham's suggestions[50]. Place spaces around all infix operators (=, +, -, <-, etc.). The same rule applies when using = in function calls. Always put a space after a comma, and never before.

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

There's a small exception to this rule: `:`, `::` and `:::` don't need spaces around them.

```
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

It is important to think about style when communicating any form of language. Writing code is no exception and is especially important if your code will be read by others. Following these basic style guides will get you on the right track for writing code that can be easily communicated to others.

[50] http://adv-r.had.co.nz/Style.html

Working with Different Types of Data in R

Wait, there are different types of data?

R is a flexible language that allows you to work with many different *forms* of data. This includes numeric, character, categorical, dates, and logical. Technically, R classifies all the different types of data into five classes:

- integer
- numeric
- character
- complex
- logical

Modern day analysis typically deals with every class so its important to gain fluency in dealing with these data forms. This section covers the fundamentals of handling the different data classes. First I cover the basics of dealing with numbers so you understand the different classes of numbers, how to generate number sequences, compare numeric values, and round. I then provide an introduction to working with characters to get you comfortable with character string manipulation and set operations. This prepares you to then learn about regular expressions which deals with search patterns for character classes. I then introduce factors, also referred to as categorical variables, and how to create, convert, order, and re-level this data class. Lastly, I cover how to manage dates as this can be a persnickety type of variable when performing data analysis. Throughout several of these chapters you'll also gain an understanding of the `TRUE`/`FALSE` logical variables.

Together, this will give you a solid foundation for dealing with the basic data classes in R so that when you start to learn how to manage the different data structures, which combines these data classes into multiple dimensions, you will have a strong base from which to start.

Dealing with Numbers

In this chapter you will learn the basics of working with numbers in R. This includes understanding how to manage the numeric type (integer vs. double), the different ways of generating non-random and random numbers, how to set seed values for reproducible random number generation, and the different ways to compare and round numeric values.

Integer vs. Double

The two most common numeric classes used in R are integer and double (for double precision floating point numbers). R automatically converts between these two classes when needed for mathematical purposes. As a result, it's feasible to use R and perform analyses for years without specifying these differences. To check whether a pre-existing vector is made up of integer or double values you can use `typeof(x)` which will tell you if the vector is a double, integer, logical, or character type.

Creating Integer and Double Vectors

By default, when you create a numeric vector using the `c()` function it will produce a vector of double precision numeric values. To create a vector of integers using `c()` you must specify explicity by placing an `L` directly after each number.

```
# create a string of double-precision values
dbl_var <- c(1, 2.5, 4.5)
dbl_var
## [1] 1.0 2.5 4.5

# placing an L after the values creates a string of integers
int_var <- c(1L, 6L, 10L)
int_var
## [1]  1  6 10
```

Converting Between Integer and Double Values

By default, if you read in data that has no decimal points or you create numeric values using the `x <- 1:10` method the numeric values will be coded as integer. If you want to change a double to an integer or vice versa you can specify one of the following:

```
# converts integers to double-precision values
as.double(int_var)
## [1]  1  6 10

# identical to as.double()
as.numeric(int_var)
## [1]  1  6 10

# converts doubles to integers
as.integer()
## integer(0)
```

Generating sequence of non-random numbers

There are a few R operators and functions that are especially useful for creating vectors of non-random numbers. These functions provide multiple ways for generating sequences of numbers.

Specifing Numbers within a Sequence

To explicitly specify numbers in a sequence you can use the colon `:` operator to specify all integers between two specified numbers or the combine `c()` function to explicity specify all numbers in the sequence.

```
# create a vector of integers between 1 and 10
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10

# create a vector consisting of 1, 5, and 10
c(1, 5, 10)
## [1]  1  5 10

# save the vector of integers between 1 and 10 as object x
x <- 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Generating Regular Sequences

A generalization of `:` is the `seq()` function, which generates a sequence of numbers with a specified arithmetic progression.

```
# generate a sequence of numbers from 1 to 21 by increments of 2
seq(from = 1, to = 21, by = 2)
##  [1]  1  3  5  7  9 11 13 15 17 19 21

# generate a sequence of numbers from 1 to 21 that has 15 equal incremented
# numbers
seq(0, 21, length.out = 15)
##  [1]  0.0  1.5  3.0  4.5  6.0  7.5  9.0 10.5 12.0 13.5 15.0 16.5 18.0 19.5
## [15] 21.0
```

The `rep()` function allows us to conveniently repeat specified constants into long vectors. This function allows for collated and non-collated repetitions.

```
# replicates the values in x a specified number of times
rep(1:4, times = 2)
## [1] 1 2 3 4 1 2 3 4

# replicates the values in x in a collated fashion
rep(1:4, each = 2)
## [1] 1 1 2 2 3 3 4 4
```

Generating sequence of random numbers

Simulation is a common practice in data analysis. Sometimes your analysis requires the implementation of a statistical procedure that requires random number generation or sampling (i.e. Monte Carlo simulation, bootstrap sampling, etc). R comes with a set of pseudo-random number generators that allow you to simulate the most common probability distributions such as Uniform, Normal, Binomial, Poisson, Exponential and Gamma.

Uniform numbers

To generate random numbers from a uniform distribution you can use the `runif()` function. Alternatively, you can use `sample()` to take a random sample using with or without replacements.

```
# generate n random numbers between the default values of 0 and 1
runif(n)

# generate n random numbers between 0 and 25
runif(n, min = 0, max = 25)

# generate n random numbers between 0 and 25 (with replacement)
sample(0:25, n, replace = TRUE)

# generate n random numbers between 0 and 25 (without replacement)
sample(0:25, n, replace = FALSE)
```

For example, to generate 25 random numbers between the values 0 and 10:

```
runif(25, min = 0, max = 10)
##  [1] 9.2494720 1.0276421 9.6061007 7.4582455 8.3666868 0.8090925 7.5638221
##  [8] 4.2810155 2.5850736 9.7962788 6.1705894 0.7037997 9.5056240 4.7589622
## [15] 7.9750129 5.3932881 5.1624935 1.2704098 8.7064680 8.6649293 0.1049461
## [22] 1.4827342 2.7337917 7.5236131 3.9803653
```

For each non-uniform probability distribution there are four primary functions available to generate random numbers, density (aka probability mass function), cumulative density, and quantiles. The prefixes for these functions are:

- `r`: random number generation
- `d`: density or probability mass function
- `p`: cumulative distribution
- `q`: quantiles

Normal Distribution Numbers

The normal (or Gaussian) distribution is the most common and well know distribution. Within R, the normal distribution functions are written as <prefix>`norm()`.

```
# generate n random numbers from a normal distribution with given mean & st. dev.
rnorm(n, mean = 0, sd = 1)

# generate CDF probabilities for value(s) in vector q
pnorm(q, mean = 0, sd = 1)

# generate quantile for probabilities in vector p
qnorm(p, mean = 0, sd = 1)

# generate density function probabilites for value(s) in vector x
dnorm(x, mean = 0, sd = 1)
```

For example, to generate 25 random numbers from a normal distribution with `mean = 100` and `standard deviation = 15`:

```
x <- rnorm(25, mean = 100, sd = 15)
x
##  [1] 107.84214 101.10742  73.67151 113.94035 108.47938  77.48445  73.02016
##  [8]  81.02323 101.64169 112.67715 105.28478  92.35393  85.96284 108.83169
## [15]  88.71057 115.13657 141.69830  99.91198 118.69664 110.61667  83.20282
## [22] 113.91008 109.10879  93.45276 109.01996

summary(x)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   73.02   88.71  105.30  101.10  110.60  141.70
```

You can also pass a vector of values. For instance, say you want to know the CDF probabilities for each value in the vector `x` created above:

```
pnorm(x, mean = 100, sd = 15)
##  [1] 0.69944664 0.52942643 0.03960976 0.82364789 0.71406244 0.06667308
##  [7] 0.03603657 0.10291447 0.54357552 0.80098468 0.63770038 0.30511760
## [13] 0.17468526 0.72199534 0.22583658 0.84353778 0.99728111 0.49765904
## [19] 0.89369904 0.76045844 0.13139693 0.82312464 0.72815841 0.33124331
## [25] 0.72619004
```

Binomial Distribution Numbers

This is conventionally interpreted as the number of successes in `size = x` trials and with `prob = p` probability of success:

```r
# generate a vector of length n displaying the number of successes from a trial
# size = 100 with a probabilty of success = 0.5
rbinom(n, size = 100, prob = 0.5)

# generate CDF probabilities for value(s) in vector q
pbinom(q, size = 100, prob = 0.5)

# generate quantile for probabilities in vector p
qbinom(p, size = 100, prob = 0.5)

# generate density function probabilites for value(s) in vector x
dbinom(x, size = 100, prob = 0.5)
```

Poisson Distribution Numbers

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occuring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event.

```r
# generate a vector of length n displaying the random number of events occuring
# when lambda (mean rate) equals 4.
rpois(n, lambda = 4)

# generate CDF probabilities for value(s) in vector q when lambda (mean rate)
# equals 4.
ppois(q, lambda = 4)

# generate quantile for probabilities in vector p when lambda (mean rate)
# equals 4.
qpois(p, lambda = 4)

# generate density function probabilites for value(s) in vector x when lambda
# (mean rate) equals 4.
dpois(x, lambda = 4)
```

Exponential Distribution Numbers

The Exponential probability distribution describes the time between events in a Poisson process.

```
# generate a vector of length n with rate = 1
rexp(n, rate = 1)

# generate CDF probabilities for value(s) in vector q when rate = 4.
pexp(q, rate = 1)

# generate quantile for probabilities in vector p when rate = 4.
qexp(p, rate = 1)

# generate density function probabilites for value(s) in vector x when rate = 4.
dexp(x, rate = 1)
```

Gamma Distribution Numbers

The Gamma probability distribution is related to the Beta distribution and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

```
# generate a vector of length n with shape parameter = 1
rgamma(n, shape = 1)

# generate CDF probabilities for value(s) in vector q when shape parameter = 1.
pgamma(q, shape = 1)

# generate quantile for probabilities in vector p when shape parameter = 1.
qgamma(p, shape = 1)

# generate density function probabilites for value(s) in vector x when shape
# parameter = 1.
dgamma(x, shape = 1)
```

Setting the seed for reproducible random numbers

If you want to generate a sequence of random numbers and then be able to reproduce that same sequence of random numbers later you can set the random number seed generator with `set.seed()`. This is a critical aspect of reproducible research[51].

For example, we can reproduce a random generation of 10 values from a normal distribution:

[51] https://en.wikipedia.org/wiki/Reproducibility

```
set.seed(197)
rnorm(n = 10, mean = 0, sd = 1)
##  [1]  0.6091700 -1.4391423  2.0703326  0.7089004  0.6455311  0.7290563
##  [7] -0.4658103  0.5971364 -0.5135480 -0.1866703
```

```
set.seed(197)
rnorm(n = 10, mean = 0, sd = 1)
##  [1]  0.6091700 -1.4391423  2.0703326  0.7089004  0.6455311  0.7290563
##  [7] -0.4658103  0.5971364 -0.5135480 -0.1866703
```

Comparing numeric values

There are multiple ways to compare numeric values and vectors. This includes logical operators along with testing for exact equality and also near equality.

Comparison Operators

The normal binary operators allow you to compare numeric values and provides the answer in logical form:

```
x < y     # is x less than y
x > y     # is x greater than y
x <= y    # is x less than or equal to y
x >= y    # is x greater than or equal to y
x == y    # is x equal to y
x != y    # is x not equal to y
```

These operations can be used for single number comparison:

```
x <- 9
y <- 10

x == y
## [1] FALSE
```

and also for comparison of numbers within vectors:

```
x <- c(1, 4, 9, 12)
y <- c(4, 4, 9, 13)

x == y
## [1] FALSE  TRUE  TRUE FALSE
```

Note that logical values `TRUE` and `FALSE` equate to `1` and `0` respectively. So if you want to identify the number of equal values in two vectors you can wrap the operation in the `sum()` function:

```
# How many pairwise equal values are in vectors x and y
sum(x == y)
## [1] 2
```

If you need to identify the location of pairwise equalities in two vectors you can wrap the operation in the `which()` function:

```
# Where are the pairwise equal values located in vectors x and y
which(x == y)
## [1] 2 3
```

Exact Equality

To test if two objects are exactly equal:

```
x <- c(4, 4, 9, 12)
y <- c(4, 4, 9, 13)

identical(x, y)
## [1] FALSE
```

```
x <- c(4, 4, 9, 12)
y <- c(4, 4, 9, 12)

identical(x, y)
## [1] TRUE
```

Floating Point Comparison

Sometimes you wish to test for ‘near equality’. The `all.equal()` function allows you to test for equality with a difference tolerance of 1.5e-8.

```
x <- c(4.00000005, 4.00000008)
y <- c(4.00000002, 4.00000006)

all.equal(x, y)
## [1] TRUE
```

If the difference is greater than the tolerance level the function will return the mean relative difference:

```
x <- c(4.005, 4.0008)
y <- c(4.002, 4.0006)

all.equal(x, y)
## [1] "Mean relative difference: 0.0003997102"
```

Rounding numbers

There are many ways of rounding to the nearest integer, up, down, or toward a specified decimal place. Lets assume `x = 1, 1.35, 1.7, 2.05, 2.4, 2.75, 3.1, 3.45, 3.8, 4.15, 4.5, 4.85, 5.2, 5.55, 5.9`. The following illustrates the common ways to round `x`.

```
# Round to the nearest integer
round(x)
##  [1] 1 1 2 2 2 3 3 3 4 4 4 5 5 6 6

# Round up
ceiling(x)
##  [1] 1 2 2 3 3 3 4 4 4 5 5 5 6 6 6

# Round down
floor(x)
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5

# Round to a specified decimal
round(x, digits = 1)
##  [1] 1.0 1.4 1.7 2.0 2.4 2.8 3.1 3.4 3.8 4.2 4.5 4.8 5.2 5.5 5.9
```

Dealing with Character Strings

Dealing with character strings is often under-emphasized in data analysis training. The focus typically remains on numeric values; however, the growth in data collection is also resulting in greater bits of information embedded in character strings. Consequently, handling, cleaning and processing character strings is becoming a prerequisite in daily data analysis. This chapter is meant to give you the foundation of working with characters by covering some basics followed by learning how to manipulate strings using base R functions along with using the simplified `stringr` package.

Character string basics

In this section you'll learn the basics of creating, converting and printing character strings followed by how to assess the number of elements and characters in a string.

Creating Strings

The most basic way to create strings is to use quotation marks and assign a string to an object similar to creating number sequences.

```
a <- "learning to create"    # create string a
b <- "character strings"     # create string b
```

The `paste()` function provides a versatile means for creating and building strings. It takes one or more R objects, converts them to "character", and then it concatenates (pastes) them to form one or several character strings.

```
# paste together string a & b
paste(a, b)
## [1] "learning to create character strings"
```

```
# paste character and number strings (converts numbers to character class)
paste("The life of", pi)
## [1] "The life of 3.14159265358979"

# paste multiple strings
paste("I", "love", "R")
## [1] "I love R"

# paste multiple strings with a separating character
paste("I", "love", "R", sep = "-")
## [1] "I-love-R"

# use paste0() to paste without spaces btwn characters
paste0("I", "love", "R")
## [1] "IloveR"

# paste objects with different lengths
paste("R", 1:5, sep = " v1.")
## [1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"
```

Converting to Strings

Test if strings are characters with `is.character()` and convert strings to character with `as.character()` or with `toString()`.

```
a <- "The life of"
b <- pi

is.character(a)
## [1] TRUE

is.character(b)
## [1] FALSE

c <- as.character(b)
is.character(c)
## [1] TRUE

toString(c("Aug", 24, 1980))
## [1] "Aug, 24, 1980"
```

Printing Strings

The common printing methods include:

- `print()`: generic printing
- `noquote()`: print with no quotes
- `cat()`: concatenate and print with no quotes
- `sprintf()`: a wrapper for the C function `sprintf`, that returns a character vector containing a formatted combination of text and variable values

The primary printing function in R is `print()`

```
x <- "learning to print strings"

# basic printing
print(x)
## [1] "learning to print strings"

# print without quotes
print(x, quote = FALSE)
## [1] learning to print strings
```

An alternative to printing a string without quotes is to use `noquote()`

```
noquote(x)
## [1] learning to print strings
```

Another very useful function is `cat()` which allows us to concatenate objects and print them either on screen or to a file. The output result is very similar to `noquote()`; however, `cat()` does not print the numeric line indicator. As a result, `cat()` can be useful for printing nicely formated responses to users.

```
# basic printing (similar to noquote)
cat(x)
## learning to print strings

# combining character strings
cat(x, "in R")
## learning to print strings in R

# basic printing of alphabet
cat(letters)
## 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

# specify a seperator between the combined characters
cat(letters, sep = "-")
## 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

# collapse the space between the combine characters
cat(letters, sep = "")
## abcdefghijklmnopqrstuvwxyz
```

You can also format the line width for printing long strings using the `fill` argument:

```
x <- "Today I am learning how to print strings."
y <- "Tomorrow I plan to learn about textual analysis."
z <- "The day after I will take a break and drink a beer."

cat(x, y, z, fill = 0)
## Today I am learning how to print strings. Tomorrow I plan to learn about text\
ual analysis. The day after I will take a break and drink a beer.

cat(x, y, z, fill = 5)
## Today I am learning how to print strings.
## Tomorrow I plan to learn about textual analysis.
## The day after I will take a break and drink a beer.
```

`sprintf()` is a useful printing function for precise control of the output. It is a wrapper for the C function `sprintf` and returns a character vector containing a formatted combination of text and variable values.

To substitute in a string or string variable, use `%s`:

```
x <- "print strings"

# substitute a single string/variable
sprintf("Learning to %s in R", x)
## [1] "Learning to print strings in R"

# substitute multiple strings/variables
y <- "in R"
sprintf("Learning to %s %s", x, y)
## [1] "Learning to print strings in R"
```

For integers, use `%d` or a variant:

```
version <- 3

# substitute integer
sprintf("This is R version:%d", version)
## [1] "This is R version:3"

# print with leading spaces
sprintf("This is R version:%4d", version)
## [1] "This is R version:   3"

# can also lead with zeros
sprintf("This is R version:%04d", version)
## [1] "This is R version:0003"
```

For floating-point numbers, use `%f` for standard notation, and `%e` or `%E` for exponential notation:

```
sprintf("%f", pi)          # '%f' indicates 'fixed point' decimal notation
## [1] "3.141593"

sprintf("%.3f", pi)        # decimal notation with 3 decimal digits
## [1] "3.142"

sprintf("%1.0f", pi)       # 1 integer and 0 decimal digits
## [1] "3"

sprintf("%5.1f", pi)       # decimal notation with 5 total decimal digits and
## [1] "  3.1"             # only 1 to the right of the decimal point

sprintf("%05.1f", pi)      # same as above but fill empty digits with zeros
```

```
## [1] "003.1"

sprintf("%+f", pi)        # print with sign (positive)
## [1] "+3.141593"

sprintf("% f", pi)        # prefix a space
## [1] " 3.141593"

sprintf("%e", pi)         # exponential decimal notation 'e'
## [1] "3.141593e+00"

sprintf("%E", pi)         # exponential decimal notation 'E'
## [1] "3.141593E+00"
```

Counting string elements and characters

To count the number of elements in a string use `length()`:

```
length("How many elements are in this string?")
## [1] 1

length(c("How", "many", "elements", "are", "in", "this", "string?"))
## [1] 7
```

To count the number of characters in a string use `nchar()`:

```
nchar("How many characters are in this string?")
## [1] 39

nchar(c("How", "many", "characters", "are", "in", "this", "string?"))
## [1]  3  4 10  3  2  4  7
```

String manipulation with base R

Basic string manipulation typically inludes case conversion, simple character and substring replacement, adding/removing whitespace, and performing set operations to compare similarities and differences between two character vectors. These operations can all be performed with base R functions; however, some operations (or at least their syntax) are simplified with the `stringr` package which we will discuss in the next section. This section illustrates the base R string manipulation capabilities.

Case conversion

To convert all upper case characters to lower case use `tolower()`:

```
x <- "Learning To MANIPULATE strinGS in R"

tolower(x)
## [1] "learning to manipulate strings in r"
```

To convert all lower case characters to upper case use `toupper()`:

```
toupper(x)
## [1] "LEARNING TO MANIPULATE STRINGS IN R"
```

Simple Character Replacement

To replace a character (or multiple characters) in a string you can use `chartr()`:

```
# replace 'A' with 'a'
x <- "This is A string."
chartr(old = "A", new = "a", x)
## [1] "This is a string."

# multiple character replacements
# replace any 'd' with 't' and any 'z' with 'a'
y <- "Tomorrow I plzn do lezrn zbout dexduzl znzlysis."
chartr(old = "dz", new = "ta", y)
## [1] "Tomorrow I plan to learn about textual analysis."
```

Note that `chartr()` replaces every identified letter for replacement so the only time I use it is when I am certain that I want to change every possible occurence of a letter.

String Abbreviations

To abbreviate strings you can use `abbreviate()`:

```
streets <- c("Main", "Elm", "Riverbend", "Mario", "Frederick")

# default abbreviations
abbreviate(streets)
##      Main       Elm Riverbend     Mario Frederick
##    "Main"     "Elm"    "Rvrb"    "Mari"    "Frdr"

# set minimum length of abbreviation
abbreviate(streets, minlength = 2)
##      Main       Elm Riverbend     Mario Frederick
##      "Mn"      "El"      "Rv"      "Mr"      "Fr"
```

Note that if you are working with U.S. states, R already has a pre-built vector with state names (`state.name`). Also, there is a pre-built vector of abbreviated state names (`state.abb`).

Extract/Replace Substrings

To extract or replace substrings in a character vector there are three primary base R functions to use: `substr()`, `substring()`, and `strsplit()`. The purpose of `substr()` is to extract and replace substrings with specified starting and stopping characters:

```
alphabet <- paste(LETTERS, collapse = "")

# extract 18th character in string
substr(alphabet, start = 18, stop = 18)
## [1] "R"

# extract 18-24th characters in string
substr(alphabet, start = 18, stop = 24)
## [1] "RSTUVWX"

# replace 1st-17th characters with `R`
substr(alphabet, start = 19, stop = 24) <- "RRRRRR"
alphabet
## [1] "ABCDEFGHIJKLMNOPQRRRRRRRYZ"
```

The purpose of `substring()` is to extract and replace substrings with only a specified starting point. `substring()` also allows you to extract/replace in a recursive fashion:

```
alphabet <- paste(LETTERS, collapse = "")

# extract 18th through last character
substring(alphabet, first = 18)
## [1] "RSTUVWXYZ"

# recursive extraction; specify start position only
substring(alphabet, first = 18:24)
## [1] "RSTUVWXYZ" "STUVWXYZ"  "TUVWXYZ"   "UVWXYZ"    "VWXYZ"     "WXYZ"
## [7] "XYZ"

# recursive extraction; specify start and stop positions
substring(alphabet, first = 1:5, last = 3:7)
## [1] "ABC" "BCD" "CDE" "DEF" "EFG"
```

To split the elements of a character string use `strsplit()`:

```
z <- "The day after I will take a break and drink a beer."
strsplit(z, split = " ")
## [[1]]
##  [1] "The"    "day"    "after" "I"      "will"   "take"   "a"      "break"
##  [9] "and"    "drink" "a"      "beer."

a <- "Alabama-Alaska-Arizona-Arkansas-California"
strsplit(a, split = "-")
## [[1]]
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

Note that the output of `strsplit()` is a list. To convert the output to a simple atomic vector simply wrap in `unlist()`:

```
unlist(strsplit(a, split = "-"))
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

String manipulation with `stringr`

The `stringr`[52] package was developed by Hadley Wickham to act as simple wrappers that make R's string functions more consistent, simple, and easier to use. To replicate the functions in this section you will need to install and load the `stringr` package:

[52] http://cran.r-project.org/web/packages/stringr/index.html

```
# install stringr package
install.packages("stringr")

# load package
library(stringr)
```

For more information on getting help with packages visit the working with packages section.

Basic Operations

There are three string functions that are closely related to their base R equivalents, but with a few enhancements:

- Concatenate with `str_c()`
- Number of characters with `str_length()`
- Substring with `str_sub()`

`str_c()` is equivalent to the `paste()` functions:

```
# same as paste0()
str_c("Learning", "to", "use", "the", "stringr", "package")
## [1] "Learningtousethestringrpackage"

# same as paste()
str_c("Learning", "to", "use", "the", "stringr", "package", sep = " ")
## [1] "Learning to use the stringr package"

# allows recycling
str_c(letters, " is for", "...")
##  [1] "a is for..." "b is for..." "c is for..." "d is for..." "e is for..."
##  [6] "f is for..." "g is for..." "h is for..." "i is for..." "j is for..."
## [11] "k is for..." "l is for..." "m is for..." "n is for..." "o is for..."
## [16] "p is for..." "q is for..." "r is for..." "s is for..." "t is for..."
## [21] "u is for..." "v is for..." "w is for..." "x is for..." "y is for..."
## [26] "z is for..."
```

`str_length()` is similiar to the `nchar()` function; however, `str_length()` behaves more appropriately with missing ('NA') values:

```
# some text with NA
text = c("Learning", "to", NA, "use", "the", NA, "stringr", "package")

# compare `str_length()` with `nchar()`
nchar(text)
## [1] 8 2 2 3 3 2 7 7

str_length(text)
## [1]  8  2 NA  3  3 NA  7  7
```

`str_sub()` is similar to `substr()`; however, it returns a zero length vector if any of its inputs are zero length, and otherwise expands each argument to match the longest. It also accepts negative positions, which are calculated from the left of the last character.

```
x <- "Learning to use the stringr package"

# alternative indexing
str_sub(x, start = 1, end = 15)
## [1] "Learning to use"

str_sub(x, end = 15)
## [1] "Learning to use"

str_sub(x, start = 17)
## [1] "the stringr package"

str_sub(x, start = c(1, 17), end = c(15, 35))
## [1] "Learning to use"     "the stringr package"

# using negative indices for start/end points from end of string
str_sub(x, start = -1)
## [1] "e"

str_sub(x, start = -19)
## [1] "the stringr package"

str_sub(x, end = -21)
## [1] "Learning to use"

# Replacement
str_sub(x, end = 15) <- "I know how to use"
x
## [1] "I know how to use the stringr package"
```

Duplicate Characters within a String

A new functionality that stringr provides in which base R does not have a specific function for is character duplication:

```
str_dup("beer", times = 3)
## [1] "beerbeerbeer"

str_dup("beer", times = 1:3)
## [1] "beer"         "beerbeer"     "beerbeerbeer"


# use with a vector of strings
states_i_luv <- state.name[c(6, 23, 34, 35)]
str_dup(states_i_luv, times = 2)
## [1] "ColoradoColorado"         "MinnesotaMinnesota"
## [3] "North DakotaNorth Dakota" "OhioOhio"
```

Remove Leading and Trailing Whitespace

A common task of string processing is that of parsing text into individual words. Often, this results in words having blank spaces (whitespaces) on either end of the word. The `str_trim()` can be used to remove these spaces:

```
text <- c("Text ", "  with", " whitespace ", " on", "both ", " sides ")

# remove whitespaces on the left side
str_trim(text, side = "left")
## [1] "Text "        "with"         "whitespace " "on"          "both "
## [6] "sides "

# remove whitespaces on the right side
str_trim(text, side = "right")
## [1] "Text"         "  with"       " whitespace" " on"          "both"
## [6] " sides"

# remove whitespaces on both sides
str_trim(text, side = "both")
## [1] "Text"        "with"        "whitespace" "on"          "both"
## [6] "sides"
```

Pad a String with Whitespace

To add whitespace, or to *pad* a string, use `str_pad()`. You can also use `str_pad()` to pad a string with specified characters.

```
str_pad("beer", width = 10, side = "left")
## [1] "      beer"

str_pad("beer", width = 10, side = "both")
## [1] "   beer   "

str_pad("beer", width = 10, side = "right", pad = "!")
## [1] "beer!!!!!!"
```

Set operatons for character strings

There are also base R functions that allows for assessing the set union, intersection, difference, equality, and membership of two vectors.

Set Union

To obtain the elements of the union between two character vectors use `union()`:

```
set_1 <- c("lagunitas", "bells", "dogfish", "summit", "odell")
set_2 <- c("sierra", "bells", "harpoon", "lagunitas", "founders")

union(set_1, set_2)
## [1] "lagunitas" "bells"     "dogfish"   "summit"    "odell"     "sierra"
## [7] "harpoon"   "founders"
```

Set Intersection

To obtain the common elements of two character vectors use `intersect()`:

```
intersect(set_1, set_2)
## [1] "lagunitas" "bells"
```

Identifying Different Elements

To obtain the non-common elements, or the difference, of two character vectors use `setdiff()`:

```
# returns elements in set_1 not in set_2
setdiff(set_1, set_2)
## [1] "dogfish" "summit"  "odell"

# returns elements in set_2 not in set_1
setdiff(set_2, set_1)
## [1] "sierra"   "harpoon"  "founders"
```

Testing for Element Equality

To test if two vectors contain the same elements regardless of order use `setequal()`:

```
set_3 <- c("woody", "buzz", "rex")
set_4 <- c("woody", "andy", "buzz")
set_5 <- c("andy", "buzz", "woody")

setequal(set_3, set_4)
## [1] FALSE

setequal(set_4, set_5)
## [1] TRUE
```

Testing for *Exact* Equality

To test if two character vectors are equal in content and order use `identical()`:

```
set_6 <- c("woody", "andy", "buzz")
set_7 <- c("andy", "buzz", "woody")
set_8 <- c("woody", "andy", "buzz")

identical(set_6, set_7)
## [1] FALSE

identical(set_6, set_8)
## [1] TRUE
```

Identifying if Elements are Contained in a String

To test if an element is contained within a character vector use `is.element()` or `%in%`:

```
good <- "andy"
bad <- "sid"

is.element(good, set_8)
## [1] TRUE

good %in% set_8
## [1] TRUE

bad %in% set_8
## [1] FALSE
```

Sorting a String

To sort a character vector use `sort()`:

```
sort(set_8)
## [1] "andy"  "buzz"  "woody"

sort(set_8, decreasing = TRUE)
## [1] "woody" "buzz"  "andy"
```

Dealing with Regular Expressions

A regular expression (aka regex) is a sequence of characters that define a search pattern, mainly for use in pattern matching with text strings. Typically, regex patterns consist of a combination of alphanumeric characters as well as special characters. The pattern can also be as simple as a single character or it can be more complex and include several characters.

To understand how to work with regular expressions in R, we need to consider two primary features of regular expressions. One has to do with the *syntax*, or the way regex patterns are expressed in R. The other has to do with the *functions* used for regex matching in R. In this chapter, we will cover both of these aspects. First, I cover the syntax that allow you to perform pattern matching functions with meta characters, character and POSIX classes, and quantifiers. This will provide you with the basic understanding of the syntax required to establish the pattern to find. Then I cover the functions you can apply to identify, extract, replace, and split parts of character strings based on the regex pattern specified.

Regex Syntax

At first glance (and second, third,…) the regex syntax can appear quite confusing. This section will provide you with the basic foundation of regex syntax; however, realize that there is a plethora of resources available that will give you far more detailed, and advanced, knowledge of regex syntax. To read more about the specifications and technicalities of regex in R you can find help at `help(regex)` or `help(regexp)`.

Metacharacters

Metacharacters consist of non-alphanumeric symbols such as:

```
.   \\   |   (   )   [   {   $   *   +   ?
```

To match metacharacters in R you need to escape them with a double backslash "\\". The following displays the general escape syntax for the most common metacharacters:

Metacharacter	Literal Meaning	Escape Syntax
.	period or dot	\\.
$	dollar sign	\\$
*	asterisk	*
+	plus sign	\\+
?	question mark	\\?
|	vertical bar	\\\|
\\	double backslash	\\\\
^	caret	\\^
[	square bracket	\\[
{	curly brace	\\{
(	parenthesis	\\(

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Escape syntax for common metacharacters

The following provides examples to show how to use the escape syntax to find and replace metacharacters. For information on the `sub` and `gsub` functions used in this example visit the main regex functions page.

```
# substitute $ with !
sub(pattern = "\\$", "\\!", "I love R$")
## [1] "I love R!"

# substitute ^ with carrot
sub(pattern = "\\^", "carrot", "My daughter has a ^ with almost every meal!")
## [1] "My daughter has a carrot with almost every meal!"

# substitute \\ with whitespace
gsub(pattern = "\\\\", " ", "I\\need\\space")
## [1] "I need space"
```

Sequences

To match a sequence of characters we can apply short-hand notation which captures the fundamental types of sequences. The following displays the general syntax for these common sequences:

Anchor	Description
\\d	match a digit character
\\D	match a non-digit character
\\s	match a space character
\\S	match a non-space character
\\w	match a word
\\W	match a non-word
\\b	match a word boundary
\\B	match a non-word boundary
\\h	match a horizontal space
\\H	match a non-horizontal space
\\v	match a vertical space
\\V	match a non-vertical space

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Anchors for common sequences

The following provides examples to show how to use the anchor syntax to find and replace sequences. For information on the `gsub` function used in this example visit the main regex functions page.

```
# substitute any digit with an underscore
gsub(pattern = "\\d", "_", "I'm working in RStudio v.0.99.484")
## [1] "I'm working in RStudio v._.__.___"

# substitute any non-digit with an underscore
gsub(pattern = "\\D", "_", "I'm working in RStudio v.0.99.484")
## [1] "_________________________0_99_484"

# substitute any whitespace with underscore
gsub(pattern = "\\s", "_", "I'm working in RStudio v.0.99.484")
## [1] "I'm_working_in_RStudio_v.0.99.484"

# substitute any wording with underscore
gsub(pattern = "\\w", "_", "I'm working in RStudio v.0.99.484")
## [1] "_'_ _______ __ _______ _._.__.___"
```

Character classes

To match one of several characters in a specified set we can enclose the characters of concern with square brackets [ ]. In addition, to match any characters **not** in a specified character set we can include the caret ^ at the beginning of the set within the brackets. The following displays the general syntax for common character classes but these can be altered easily as shown in the examples that follow:

Anchor	Description
[aeiou]	match any specified lower case vowel
[AEIOU]	match any specified upper case vowel
[0123456789]	match any specified numeric value
[0-9]	match any range of specified numeric values
[a-z]	match any range of lower case letter
[A-Z]	match any range of upper case letter
[a-zA-Z0-9]	match any of the above
[^aeiou]	match anything other than a lowercase vowel
[^0-9]	match anything other than the specified numeric values

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Anchors for common character classes

The following provides examples to show how to use the anchor syntax to match character classes. For information on the `grep` function used in this example visit the main regex functions page.

```
x <- c("RStudio", "v.0.99.484", "2015", "09-22-2015", "grep vs. grepl")

# find any strings with numeric values between 0-9
grep(pattern = "[0-9]", x, value = TRUE)
## [1] "v.0.99.484" "2015"       "09-22-2015"

# find any strings with numeric values between 6-9
grep(pattern = "[6-9]", x, value = TRUE)
## [1] "v.0.99.484" "09-22-2015"

# find any strings with the character R or r
grep(pattern = "[Rr]", x, value = TRUE)
## [1] "RStudio"        "grep vs. grepl"

# find any strings that have non-alphanumeric characters
grep(pattern = "[^0-9a-zA-Z]", x, value = TRUE)
## [1] "v.0.99.484"     "09-22-2015"     "grep vs. grepl"
```

POSIX character classes

Closely related to regex character classes are POSIX character classes which are expressed in double brackets [[ ]].

Anchor	Description	
[[:lower:]]	lower-case letters	
[[:upper:]]	upper-case letters	
[[:alpha:]]	alphabetic characters	[[:lower:]] + [[:upper:]]
[[:digit:]]	numeric values	
[[:alnum:]]	alphanumeric characters	[[:alpha:]] + [[:digit:]]
[[:blank:]]	blank characters (space & tab)	
[[:cntrl:]]	control characters	
[[:punct:]]	punctuation characters: ! " # % & ' ( ) * + , - . / : ;	
[[:space:]]	space characters: tab, newline, vertical tab, space, etc	
[[:xdigit:]]	hexadecimal digits: 0-9 A B C D E F a b c d e f	
[[:print:]]	printable characters	[[:alpha:]] + [[:punct:]] + space
[[:graph:]]	graphical characters	[[:alpha:]] + [[:punct:]]

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Anchors for POSIX character classes

The following provides examples to show how to use the anchor syntax to match POSIX character classes. For information on the `grep` function used in this example visit the main regex functions page.

```
x <- "I like beer! #beer, @wheres_my_beer, I like R (v3.2.2) #rrrrrrr2015"

# remove space or tabs
gsub(pattern = "[[:blank:]]", replacement = "", x)
## [1] "Ilikebeer!#beer,@wheres_my_beer,IlikeR(v3.2.2)#rrrrrrr2015"

# replace punctuation with whitespace
gsub(pattern = "[[:punct:]]", replacement = " ", x)
## [1] "I like beer   beer   wheres my beer  I like R  v3 2 2   rrrrrrr2015"

# remove alphanumeric characters
gsub(pattern = "[[:alnum:]]", replacement = "", x)
## [1] "  ! #, @__,    (..) #"
```

Quantifiers

When we want to match a **certain number** of characters that meet a certain criteria we can apply quantifiers to our pattern searches. The quantifiers we can use are:

Quantifier	Description
?	the preceding item is optional and will be matched at most once
*	the preceding item will be matched zero or more times
+	the preceding item will be matched one or more times
{n}	the preceding item is matched exactly n times
{n,}	the preceding item is matched n or more times
{n,m}	the preceding item is matched at least n times, but not more than m times

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Quantifiers

The following provides examples to show how to use the quantifier syntax to match a **certain number** of characters patterns. For information on the `grep` function used in this example visit the main regex functions page. Note that `state.name` is a built in dataset within R that contains all the U.S. state names.

```
# match states that contain z
grep(pattern = "z+", state.name, value = TRUE)
## [1] "Arizona"

# match states with two s
grep(pattern = "s{2}", state.name, value = TRUE)
## [1] "Massachusetts" "Mississippi"   "Missouri"      "Tennessee"

# match states with one or two s
grep(pattern = "s{1,2}", state.name, value = TRUE)
##  [1] "Alaska"        "Arkansas"      "Illinois"      "Kansas"
##  [5] "Louisiana"     "Massachusetts" "Minnesota"     "Mississippi"
##  [9] "Missouri"      "Nebraska"      "New Hampshire" "New Jersey"
## [13] "Pennsylvania"  "Rhode Island"  "Tennessee"     "Texas"
## [17] "Washington"    "West Virginia" "Wisconsin"
```

Regex Functions

Now that I've illustrated how R handles some of the most common regular expression elements, it's time to present the functions you can use for working with regular expression. R contains a set of functions in the base package that we can use to find pattern matches. Alternatively, the R package stringr also provides several functions for regex operations. We will cover both these alternatives.

Main regex functions in R

The primary base R regex functions serve three primary purposes: pattern matching, pattern replacement, and character splitting.

Pattern matching

There are five functions that provide pattern matching capabilities. The three functions that I provide examples for (`grep()`, `grepl()`, and `regexpr()`) are ones that are most common. The primary difference in between these three functions is the output they provide. The two other functions which I do not illustrate are `gregexpr()` and `regexec()`. These two functions provide similar capabilities as `regexpr()` but with the output in list form.

To find a pattern in a character vector and to have the element values or indices as the output use `grep()`:

```
# use the built in data set `state.division`
head(as.character(state.division))
## [1] "East South Central" "Pacific"            "Mountain"
## [4] "West South Central" "Pacific"            "Mountain"
```

```
# find the elements which match the pattern
grep("North", state.division)
##  [1] 13 14 15 16 22 23 25 27 34 35 41 49
```

```
# use 'value = TRUE' to show the element value
grep("North", state.division, value = TRUE)
##  [1] "East North Central" "East North Central" "West North Central"
##  [4] "West North Central" "East North Central" "West North Central"
##  [7] "West North Central" "West North Central" "West North Central"
## [10] "East North Central" "West North Central" "East North Central"
```

```
# can use the 'invert' argument to show the non-matching elements
grep("North | South", state.division, invert = TRUE)
##  [1]  2  3  5  6  7  8  9 10 11 12 19 20 21 26 28 29 30 31 32 33 37 38 39
## [24] 40 44 45 46 47 48 50
```

To find a pattern in a character vector and to have logical (TRUE/FALSE) outputs use `grepl()`:

```
grepl("North | South", state.division)
##  [1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
## [23]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [34]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
## [45] FALSE FALSE FALSE FALSE  TRUE FALSE

# wrap in sum() to get the count of matches
sum(grepl("North | South", state.division))
## [1] 20
```

To find exactly where the pattern exists in a string use `regexpr()`:

```
x <- c("v.111", "0v.11", "00v.1", "000v.", "00000")

regexpr("v.", x)
## [1]  1  2  3  4 -1
## attr(,"match.length")
## [1]  2  2  2  2 -1
## attr(,"useBytes")
## [1] TRUE
```

The output of `regexpr()` can be interepreted as follows. The first element provides the starting position of the match in each element. Note that the value **-1** means there is no match. The second element (attribute "match length") provides the length of the match. The third element (attribute "useBytes") has a value TRUE meaning matching was done byte-by-byte rather than character-by-character.

Pattern Replacement Functions

In addition to finding patterns in character vectors, its also common to want to *replace* a pattern in a string with a new pattern. Base R regex functions provide two options for this: *i*) replace the first matching occurrence or *ii*) replace all occurrences.

To replace the **first** matching occurrence of a pattern use `sub()`:

```
new <- c("New York", "new new York", "New New New York")
new
## [1] "New York"         "new new York"     "New New New York"

# Default is case sensitive
sub("New", replacement = "Old", new)
## [1] "Old York"         "new new York"     "Old New New York"

# use 'ignore.case = TRUE' to perform the obvious
sub("New", replacement = "Old", new, ignore.case = TRUE)
## [1] "Old York"         "Old new York"     "Old New New York"
```

To replace **all** matching occurrences of a pattern use `gsub()`:

```
# Default is case sensitive
gsub("New", replacement = "Old", new)
## [1] "Old York"         "new new York"     "Old Old Old York"

# use 'ignore.case = TRUE' to perform the obvious
gsub("New", replacement = "Old", new, ignore.case = TRUE)
## [1] "Old York"         "Old Old York"     "Old Old Old York"
```

Splitting Character Vectors

There will be times when you want to to split the elements of a character string into seperate elements. To divide the characters in a vector into individual components use `strsplit()`:

```
x <- paste(state.name[1:10], collapse = " ")

# output will be a list
strsplit(x, " ")
## [[1]]
##  [1] "Alabama"     "Alaska"      "Arizona"     "Arkansas"    "California"
##  [6] "Colorado"    "Connecticut" "Delaware"    "Florida"     "Georgia"

# output as a vector rather than a list
unlist(strsplit(x, " "))
##  [1] "Alabama"     "Alaska"      "Arizona"     "Arkansas"    "California"
##  [6] "Colorado"    "Connecticut" "Delaware"    "Florida"     "Georgia"
```

Regex functions in `stringr`

Similar to basic string manipulation, the `stringr` package also offers regex functionality. In some cases the `stringr` performs the same functions as certain base R functions but with more consistent syntax. In other cases `stringr` offers additional functionality that is not available in base R functions.

```
# install stringr package
install.packages("stringr")

# load package
library(stringr)
```

Detecting Patterns

To *detect* whether a pattern is present (or absent) in a string vector use the `str_detect()`. This function is a wrapper for `grepl()`.

```
# use the built in data set 'state.name'
head(state.name)
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
## [6] "Colorado"

str_detect(state.name, pattern = "New")
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE FALSE

# count the total matches by wrapping with sum
sum(str_detect(state.name, pattern = "New"))
## [1] 4
```

Locating Patterns

To *locate* the occurrences of patterns `stringr` offers two options: *i*) locate the first matching occurrence or *ii*) locate all occurrences. To locate the position of the first occurrence of a pattern in a string vector use `str_locate()`. The output provides the starting and ending position of the first match found within each element.

```
x <- c("abcd", "a22bc1d", "ab3453cd46", "a1bc44d")

# locate 1st sequence of 1 or more consecutive numbers
str_locate(x, "[0-9]+")
##      start end
## [1,]    NA  NA
## [2,]     2   3
## [3,]     3   6
## [4,]     2   2
```

To locate the positions of all pattern match occurrences in a character vector use `str_locate_all()`. The output provides a list the same length as the number of elements in the vector. Each list item will provide the starting and ending positions for each pattern match occurrence in its respective element.

```
# locate all sequences of 1 or more consecutive numbers
str_locate_all(x, "[0-9]+")
## [[1]]
##      start end
##
## [[2]]
##      start end
## [1,]     2   3
## [2,]     6   6
##
## [[3]]
##      start end
## [1,]     3   6
## [2,]     9  10
##
## [[4]]
##      start end
## [1,]     2   2
## [2,]     5   6
```

Extracting Patterns

For extracting a string containing a pattern, `stringr` offers two primary options: *i*) extract the first matching occurrence or *ii*) extract all occurrences. To extract the first occurrence of a pattern in a character vector use `str_extract()`. The output will be the same length as the string and if no match is found the output will be `NA` for that element.

```
y <- c("I use R #useR2014", "I use R and love R #useR2015", "Beer")

str_extract(y, pattern = "R")
## [1] "R" "R" NA
```

To extract all occurrences of a pattern in a character vector use `str_extract_all()`. The output provides a list the same length as the number of elements in the vector. Each list item will provide the matching pattern occurrence within that relative vector element.

```
str_extract_all(y, pattern = "[[:punct:]]*[a-zA-Z0-9]*R[a-zA-Z0-9]*")
## [[1]]
## [1] "R"         "#useR2014"
##
## [[2]]
## [1] "R"         "R"         "#useR2015"
##
## [[3]]
## character(0)
```

Replacing Patterns

For extracting a string containing a pattern, `stringr` offers two options: *i*) replace the first matching occurrence or *ii*) replace all occurrences. To replace the first occurrence of a pattern in a character vector use `str_replace()`. This function is a wrapper for `sub()`.

```
cities <- c("New York", "new new York", "New New New York")
cities
## [1] "New York"         "new new York"     "New New New York"

# case sensitive
str_replace(cities, pattern = "New", replacement = "Old")
## [1] "Old York"         "new new York"     "Old New New York"

# to deal with case sensitivities use Regex syntax in the 'pattern' argument
str_replace(cities, pattern = "[N]*[n]*ew", replacement = "Old")
## [1] "Old York"         "Old new York"     "Old New New York"
```

To extract all occurrences of a pattern in a character vector use `str_replace_all()`. This function is a wrapper for `gsub()`.

```
str_replace_all(cities, pattern = "[N]*[n]*ew", replacement = "Old")
## [1] "Old York"         "Old Old York"     "Old Old Old York"
```

String Splitting

To split the elements of a character string use `str_split()`. This function is a wrapper for `strsplit()`.

```
z <- "The day after I will take a break and drink a beer."
str_split(z, pattern = " ")
## [[1]]
##  [1] "The"    "day"    "after" "I"      "will"   "take"   "a"      "break"
##  [9] "and"    "drink" "a"      "beer."

a <- "Alabama-Alaska-Arizona-Arkansas-California"
str_split(a, pattern = "-")
## [[1]]
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

Note that the output of `strs_plit()` is a list. To convert the output to a simple atomic vector simply wrap in `unlist()`:

```
unlist(str_split(a, pattern = "-"))
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
```

Additional resources

Character string data is often considered semi-structured data. Text can be structured in a specified field; however, the quality and consistency of the text input can be far from structured. Consequently, managing and manipulating character strings can be extremely tedious and unique to each data wrangling process. As a result, taking the time to learn the nuances of dealing with character strings and regex functions can provide a great return on investment; however, the functions and techniques required will likey be greater than what I could offer here. So here are additional resources that are worth reading and learning from:

- Handling and Processing Strings in R[53]
- `stringr` Package Vignette[54]
- Regular Expressions[55]

[53] http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf
[54] https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html
[55] http://www.regular-expressions.info/

Dealing with Factors

Factors are variables in R which take on a limited number of different values; such variables are often referred to as categorical variables[56]. One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models such as `lm` and `glm` differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.

One can think of a factor as an integer vector where each integer has a label[57]. In fact, factors are built on top of integer vectors using two attributes: the `class()` "factor", which makes them behave differently from regular integer vectors, and the `levels()`, which defines the set of allowed values[58].

In this chapter I will cover the basics of dealing with factors which includes Creating, converting & inspecting factors, Ordering levels, Revaluing levels, and Dropping levels.

Creating, converting & inspecting factors

Factor objects can be created with the `factor()` function:

```
# create a factor string
gender <- factor(c("male", "female", "female", "male", "female"))
gender
## [1] male   female female male   female
## Levels: female male

# inspect to see if it is a factor class
class(gender)
## [1] "factor"

# show that factors are just built on top of integers
typeof(gender)
## [1] "integer"

# See the underlying representation of factor
unclass(gender)
## [1] 2 1 1 2 1
```

[56] https://en.wikipedia.org/wiki/Categorical_variable
[57] https://leanpub.com/rprogramming
[58] http://adv-r.had.co.nz/Data-structures.html

```
## attr(,"levels")
## [1] "female" "male"

# what are the factor levels?
levels(gender)
## [1] "female" "male"

# show summary of counts
summary(gender)
## female   male
##      3      2
```

If we have a vector of character strings or integers we can easily convert to factors:

```
group <- c("Group1", "Group2", "Group2", "Group1", "Group1")
str(group)
##  chr [1:5] "Group1" "Group2" "Group2" "Group1" "Group1"

# convert from characters to factors
as.factor(group)
## [1] Group1 Group2 Group2 Group1 Group1
## Levels: Group1 Group2
```

Ordering levels

When creating a factor we can control the ordering of the levels by using the `levels` argument:

```
# when not specified the default puts order as alphabetical
gender <- factor(c("male", "female", "female", "male", "female"))
gender
## [1] male   female female male   female
## Levels: female male

# specifying order
gender <- factor(c("male", "female", "female", "male", "female"),
                 levels = c("male", "female"))
gender
## [1] male   female female male   female
## Levels: male female
```

We can also create ordinal factors in which a specific order is desired by using the `ordered = TRUE` argument. This will be reflected in the output of the levels as shown below in which `low < middle < high`:

```
ses <- c("low", "middle", "low", "low", "low", "low", "middle", "low", "middle",
    "middle", "middle", "middle", "middle", "high", "high", "low", "middle",
    "middle", "low", "high")

# create ordinal levels
ses <- factor(ses, levels = c("low", "middle", "high"), ordered = TRUE)
ses
##  [1] low    middle low    low    low    low    middle low    middle middle
## [11] middle middle middle high   high   low    middle middle low    high
## Levels: low < middle < high

# you can also reverse the order of levels if desired
factor(ses, levels = rev(levels(ses)))
##  [1] low    middle low    low    low    low    middle low    middle middle
## [11] middle middle middle high   high   low    middle middle low    high
## Levels: high < middle < low
```

Revalue levels

To recode factor levels I usually use the `revalue()` function from the `plyr` package.

```
plyr::revalue(ses, c("low" = "small", "middle" = "medium", "high" = "large"))
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium large  large  small  medium medium small  large
## Levels: small < medium < large
```

Note that Using the `::` notation allows you to access the `revalue()` function without having to fully load the `plyr` package.

Dropping levels

When you want to drop unused factor levels, use `droplevels()`:

```
ses2 <- ses[ses != "middle"]

# lets say you have no observations in one level
summary(ses2)
##    low middle   high 
##      8      0      3

# you can drop that level if desired
droplevels(ses2)
##  [1] low  low  low  low  low  low  high high low  low  high
## Levels: low < high
```

Dealing with Dates

Real world data are often associated with dates and time; however, dealing with dates accurately can appear to be a complicated task due to the variety in formats and accounting for time-zone differences and leap years. R has a range of functions that allow you to work with dates and times. Furthermore, packages such as `lubridate`[59] make it easier to work with dates and times.

In this chapter I will introduce you to the basics of dealing with dates. This includes printing the current date and time stamp, converting strings to dates, extracting and manipulating parts of dates, creating date sequences, performing calculations with dates, and dealing with time zone and daylight savings differences. I end with offering additional resources to learn and deal with date and time data.

Getting current date & time

To get current date and time information:

```
Sys.timezone()
## [1] "America/New_York"

Sys.Date()
## [1] "2015-09-24"

Sys.time()
## [1] "2015-09-24 15:08:57 EDT"
```

If using the `lubridate` package:

```
library(lubridate)

now()
## [1] "2015-09-24 15:08:57 EDT"
```

Converting strings to dates

When date and time data are imported into R they will often default to a character string. This requires us to convert strings to dates. We may also have multiple strings that we want to merge to create a date variable.

[59]https://cran.r-project.org/web/packages/lubridate/index.html

Convert Strings to Dates

To convert a string that is already in a date format (YYYY-MM-DD) into a date object use `as.Date()`:

```
x <- c("2015-07-01", "2015-08-01", "2015-09-01")

as.Date(x)
## [1] "2015-07-01" "2015-08-01" "2015-09-01"
```

Note that the default date format is YYYY-MM-DD; therefore, if your string is of different format you must incorporate the `format` argument. There are multiple formats that dates can be in; for a complete list of formatting code options in R type `?strftime` in your console.

```
y <- c("07/01/2015", "07/01/2015", "07/01/2015")

as.Date(y, format = "%m/%d/%Y")
## [1] "2015-07-01" "2015-07-01" "2015-07-01"
```

If using the `lubridate` package:

```
library(lubridate)
ymd(x)
## [1] "2015-07-01 UTC" "2015-08-01 UTC" "2015-09-01 UTC"

mdy(y)
## [1] "2015-07-01 UTC" "2015-07-01 UTC" "2015-07-01 UTC"
```

One of the many benefits of the `lubricate` package is that it automatically recognizes the common separators used when recording dates (“-“, “/”, “.”, and “”). As a result, you only need to focus on specifying the order of the date elements to determine the parsing function applied:

Order of elements in date-time	Parse function
year, month, day	`ymd()`
year, day, month	`ydm()`
month, day, year	`mdy()`
day, month, year	`dmy()`
hour, minute	`hm()`
hour, minute, second	`hms()`
year, month, day, hour, minute, second	`ymd_hms()`

*adapted from *Dates and Times Made Easy with lubridate* (Grolemund & Wickham, 2011)

Create Dates by Merging Data

Sometimes your date data are collected in separate elements. To convert these separate data into one date object incorporate the `ISOdate()` function:

```r
yr <- c("2012", "2013", "2014", "2015")
mo <- c("1", "5", "7", "2")
day <- c("02", "22", "15", "28")

# ISOdate converts to a POSIXct object
ISOdate(year = yr, month = mo, day = day)
## [1] "2012-01-02 12:00:00 GMT" "2013-05-22 12:00:00 GMT"
## [3] "2014-07-15 12:00:00 GMT" "2015-02-28 12:00:00 GMT"

# truncate the unused time data by converting with as.Date
as.Date(ISOdate(year = yr, month = mo, day = day))
## [1] "2012-01-02" "2013-05-22" "2014-07-15" "2015-02-28"
```

Note that `ISODate()` also has arguments to accept data for hours, minutes, seconds, and time-zone if you need to merge all these separate components.

Extract & manipulate parts of dates

To extract and manipulate individual elements of a date I typically use the `lubridate` package due to its simplistic function syntax. The functions provided by `lubridate` to perform extraction and manipulation of dates include:

Date component	Accessor
Year	`year()`
Month	`month()`
Week	`week()`
Day of year	`yday()`
Day of month	`mday()`
Day of week	`wday()`
Hour	`hour()`
Minute	`minute()`
Second	`second()`
Time zone	`tz()`

*adapted from *Dates and Times Made Easy with lubridate* (Grolemund & Wickham, 2011)

To extract an individual element of the date variable you simply use the accessor function desired. Note that the accessor variables have additional arguments that can be used to show the name of the date element in full or abbreviated form.

```
library(lubridate)

x <- c("2015-07-01", "2015-08-01", "2015-09-01")

year(x)
## [1] 2015 2015 2015

# default is numerical value
month(x)
## [1] 7 8 9

# show abbreviated name
month(x, label = TRUE)
## [1] Jul Aug Sep
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec

# show unabbreviated name
month(x, label = TRUE, abbr = FALSE)
## [1] July      August    September
## 12 Levels: January < February < March < April < May < June < ... < December


wday(x, label = TRUE, abbr = FALSE)
## [1] Wednesday Saturday  Tuesday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

To manipulate or change the values of date elements we simply use the accessor function to extract the element of choice and then use the assignment function to assign a new value.

```
# convert to date format
x <- ymd(x)
x
## [1] "2015-07-01 UTC" "2015-08-01 UTC" "2015-09-01 UTC"

# change the days for the dates
mday(x)
## [1] 1 1 1

mday(x) <- c(3, 10, 22)
x
## [1] "2015-07-03 UTC" "2015-08-10 UTC" "2015-09-22 UTC"
```

```
# can also use 'update()' function
update(x, year = c(2013, 2014, 2015), month = 9)
## [1] "2013-09-03 UTC" "2014-09-10 UTC" "2015-09-22 UTC"

# can also add/subtract units
x + years(1) - days(c(2, 9, 21))
## [1] "2016-07-01 UTC" "2016-08-01 UTC" "2016-09-01 UTC"
```

Creating date sequences

To create a sequence of dates we can leverage the `seq()` function. As with numeric vectors, you have to specify at least three of the four arguments (`from`, `to`, `by`, and `length.out`).

```
seq(as.Date("2010-1-1"), as.Date("2015-1-1"), by = "years")
## [1] "2010-01-01" "2011-01-01" "2012-01-01" "2013-01-01" "2014-01-01"
## [6] "2015-01-01"

seq(as.Date("2015/1/1"), as.Date("2015/12/30"), by = "quarter")
## [1] "2015-01-01" "2015-04-01" "2015-07-01" "2015-10-01"

seq(as.Date('2015-09-15'), as.Date('2015-09-30'), by = "2 days")
## [1] "2015-09-15" "2015-09-17" "2015-09-19" "2015-09-21" "2015-09-23"
## [6] "2015-09-25" "2015-09-27" "2015-09-29"
```

Using the `lubridate` package is very similar. The only difference is `lubridate` changes the way you specify the first two arguments in the `seq()` function.

```
library(lubridate)

seq(ymd("2010-1-1"), ymd("2015-1-1"), by = "years")
## [1] "2010-01-01 UTC" "2011-01-01 UTC" "2012-01-01 UTC" "2013-01-01 UTC"
## [5] "2014-01-01 UTC" "2015-01-01 UTC"

seq(ymd("2015/1/1"), ymd("2015/12/30"), by = "quarter")
## [1] "2015-01-01 UTC" "2015-04-01 UTC" "2015-07-01 UTC" "2015-10-01 UTC"

seq(ymd('2015-09-15'), ymd('2015-09-30'), by = "2 days")
## [1] "2015-09-15 UTC" "2015-09-17 UTC" "2015-09-19 UTC" "2015-09-21 UTC"
## [5] "2015-09-23 UTC" "2015-09-25 UTC" "2015-09-27 UTC" "2015-09-29 UTC"
```

Creating sequences with time is very similar; however, we need to make sure our date object is POSIXct rather than just a Date object (as produced by `as.Date`):

```
seq(as.POSIXct("2015-1-1 0:00"), as.POSIXct("2015-1-1 12:00"), by = "hour")
##  [1] "2015-01-01 00:00:00 EST" "2015-01-01 01:00:00 EST"
##  [3] "2015-01-01 02:00:00 EST" "2015-01-01 03:00:00 EST"
##  [5] "2015-01-01 04:00:00 EST" "2015-01-01 05:00:00 EST"
##  [7] "2015-01-01 06:00:00 EST" "2015-01-01 07:00:00 EST"
##  [9] "2015-01-01 08:00:00 EST" "2015-01-01 09:00:00 EST"
## [11] "2015-01-01 10:00:00 EST" "2015-01-01 11:00:00 EST"
## [13] "2015-01-01 12:00:00 EST"

# with lubridate
seq(ymd_hm("2015-1-1 0:00"), ymd_hm("2015-1-1 12:00"), by = "hour")
##  [1] "2015-01-01 00:00:00 UTC" "2015-01-01 01:00:00 UTC"
##  [3] "2015-01-01 02:00:00 UTC" "2015-01-01 03:00:00 UTC"
##  [5] "2015-01-01 04:00:00 UTC" "2015-01-01 05:00:00 UTC"
##  [7] "2015-01-01 06:00:00 UTC" "2015-01-01 07:00:00 UTC"
##  [9] "2015-01-01 08:00:00 UTC" "2015-01-01 09:00:00 UTC"
## [11] "2015-01-01 10:00:00 UTC" "2015-01-01 11:00:00 UTC"
## [13] "2015-01-01 12:00:00 UTC"
```

Calculations with dates

Since R stores date and time objects as numbers, this allows you to perform various calculations such as logical comparisons, addition, subtraction, and working with durations.

```
x <- Sys.Date()
x
## [1] "2015-09-26"

y <- as.Date("2015-09-11")

x > y
## [1] TRUE

x - y
## Time difference of 15 days
```

The nice thing about the date/time classes is that they keep track of leap years, leap seconds, daylight savings, and time zones. Use `OlsonNames()` for a full list of acceptable time zone specifications.

```
# last leap year
x <- as.Date("2012-03-1")
y <- as.Date("2012-02-28")

x - y
## Time difference of 2 days


# example with time zones
x <- as.POSIXct("2015-09-22 01:00:00", tz = "US/Eastern")
y <- as.POSIXct("2015-09-22 01:00:00", tz = "US/Pacific")

y == x
## [1] FALSE

y - x
## Time difference of 3 hours
```

Similarly, the same functionality exists with the `lubridate` package with the only difference being the accessor function(s) used.

```
library(lubridate)

x <- now()
x
## [1] "2015-09-26 10:08:18 EDT"

y <- ymd("2015-09-11")

x > y
## [1] TRUE

x - y
## Time difference of 15.5891 days

y + days(4)
## [1] "2015-09-15 UTC"

x - hours(4)
## [1] "2015-09-26 06:08:18 EDT"
```

We can also deal with time spans by using the duration functions in `lubridate`. Durations simply measure the time span between start and end dates. Using base R date functions for duration calculations is tedious and often results in wrong measurements. `lubridate` provides simplistic syntax to calculate durations with the desired measurement (seconds, minutes, hours, etc.).

```
# create new duration (represented in seconds)
new_duration(60)
## [1] "60s"

# create durations for minutes, hours, years
dminutes(1)
## [1] "60s"

dhours(1)
## [1] "3600s (~1 hours)"

dyears(1)
## [1] "31536000s (~365 days)"

# add/subtract durations from date/time object
x <- ymd_hms("2015-09-22 12:00:00")

x + dhours(10)
## [1] "2015-09-22 22:00:00 UTC"

x + dhours(10) + dminutes(33) + dseconds(54)
## [1] "2015-09-22 22:33:54 UTC"
```

Dealing with time zones & daylight savings

To change the time zone for a date/time we can use the `with_tz()` function which will also update the clock time to align with the updated time zone:

```
library(lubridate)

time <- now()
time
## [1] "2015-09-26 10:30:32 EDT"

with_tz(time, tzone = "MST")
## [1] "2015-09-26 07:30:32 MST"
```

If the time zone is incorrect or for some reason you need to change the time zone without changing the clock time you can force it with `force_tz()`:

```
time
## [1] "2015-09-26 10:30:32 EDT"

force_tz(time, tzone = "MST")
## [1] "2015-09-26 10:30:32 MST"
```

We can also easily work with daylight savings times to eliminate impacts on date/time calculations:

```
# most recent daylight savings time
ds <- ymd_hms("2015-03-08 01:59:59", tz = "US/Eastern")

# if we add a duration of 1 sec we gain an extra hour
ds + dseconds(1)
## [1] "2015-03-08 03:00:00 EDT"

# add a duration of 2 hours will reflect actual daylight savings clock time
# that occured 2 hours after 01:59:59 on 2015-03-08
ds + dhours(2)
## [1] "2015-03-08 04:59:59 EDT"

# add a period of two hours will reflect clock time that normally occurs after
# 01:59:59 and is not influenced by daylight savings time.
ds + hours(2)
## [1] "2015-03-08 03:59:59 EDT"
```

Additional resources

For additional resources on learning and dealing with dates I recommend the following:

- Dates and times made easy with `lubridate`[60]
- Date and time classes in R[61]

---

[60] http://www.jstatsoft.org/article/view/v040i03
[61] https://www.r-project.org/doc/Rnews/Rnews_2004-1.pdf

Managing Data Structures in R

> *"Smart data structures and dumb code works a lot better than the other way around" -* Eric S. Raymond

In the previous section I illustrated how to work with different types of data; however, we primarily focused on data in a one-dimensional structure. In typical data analyses you often need more than one dimension. Many datasets can contain variables of different length and or types of values (i.e. numeric vs character). Furthermore, many statistical and mathematical calculations are based on matrices. R provides multiple types of data structures to deal with these different needs.

The basic data structures in R can be organized by their dimensionality (1D, 2D, …, *n*D) and their "likeness" (homogenous vs. heterogeneous). This results in five data structure types most often used in data analysis; and almost all other objects in R are built from these foundational types:

Dimensions	Homogenous	Heterogeneous
1D	Atomic Vector	List
2D	Matrix	Data frame
*n*D	Array	

*adapted from *Advanced R* (H. Wickham

Basic Data Structures in R

In this section I will cover the basics of these data structures. I have not had the need to use multi-dimensional arrays, therefore, the topics I will go into details on will include vectors, lists, matrices, and data frames. These types represent the most commonly used data structures for day-to-day analyses. For each data structure I will illustrate how to create the structure, add additional elements to a pre-existing structure, add attributes to structures, and how to subset the various data structures. Lastly, I will cover how to deal with missing values in data structures. Consequently, this section will provide a robust understanding of managing various forms of datasets depending on dimensionality needs.

Data Structure Basics

Prior to jumping into the data structures, it's beneficial to understand two components of data structures - the structure and attributes.

Identifying the Structure

Given an object, the best way to understand what data structure it represents is to use the structure function `str()`. `str()` stands for structure and provides a compact display of the internal **str**ucture of an R object.

```
# different data structures
vector <- 1:10
list <- list(item1 = 1:10, item2 = LETTERS[1:18])
matrix <- matrix(1:12, nrow = 4)
df <- data.frame(item1 = 1:18, item2 = LETTERS[1:18])

# identify the structure of each object
str(vector)
##  int [1:10] 1 2 3 4 5 6 7 8 9 10

str(list)
## List of 2
##  $ item1: int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ item2: chr [1:18] "A" "B" "C" "D" ...

str(matrix)
##  int [1:4, 1:3] 1 2 3 4 5 6 7 8 9 10 ...

str(df)
## 'data.frame':        18 obs. of  2 variables:
##  $ item1: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ item2: Factor w/ 18 levels "A","B","C","D",..: 1 2 3 4 5 6 7 8 9 10 ...
```

Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the `attributes()` function. Not all R objects contain attributes, in which case the `attributes()` function returns NULL.

```
# assess attributes of an object
attributes(df)
## $names
## [1] "item1" "item2"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
##
## $class
## [1] "data.frame"

attributes(matrix)
## $dim
## [1] 4 3

# assess names of an object
names(df)
## [1] "item1" "item2"

# assess the dimensions of an object
dim(matrix)
## [1] 4 3

# assess the class of an object
class(list)
## [1] "list"

# access the length of an object
length(vector)
## [1] 10

# note that length will measure the number of items in
# a list or number of columns in a data frame
```

```
length(list)
## [1] 2

length(df)
## [1] 2
```

This chapter only shows you functions to assess these attributes. In the chapters that follow more details are provided on how to view and create attributes for each type of data structure.

Managing Vectors

The basic structure in R is the vector. A vector is a sequence of data elements of the same basic type: integer, double, logical, or character.[62] The one-dimensional examples illustrated in the previous section are considered vectors. In this chapter I will illustrate how to create vectors, add additional elements to pre-existing vectors, add attributes to vectors, and subset vectors.

Creating

The colon `:` operator can be used to create a vector of integers between two specified numbers or the `c()` function can be used to create vectors of objects by concatenating elements together:

```
# integer vector
w <- 8:17
w
##  [1]  8  9 10 11 12 13 14 15 16 17

# double vector
x <- c(0.5, 0.6, 0.2)
x
## [1] 0.5 0.6 0.2

# logical vector
y1 <- c(TRUE, FALSE, FALSE)
y1
## [1]  TRUE FALSE FALSE

# logical vector in shorthand
y2 <- c(T, F, F)
y2
## [1]  TRUE FALSE FALSE

# Character vector
z <- c("a", "b", "c")
z
## [1] "a" "b" "c"
```

You can also use the `as.vector()` function to initialize vectors or change the vector type:

[62] There are two additional vector types which I will not discuss - complex and raw.

```
v <- as.vector(8:17)
v
##  [1]  8  9 10 11 12 13 14 15 16 17

# turn numerical vector to character
as.vector(v, mode = "character")
##  [1] "8"  "9"  "10" "11" "12" "13" "14" "15" "16" "17"
```

All elements of a vector must be the same type, so when you attempt to combine different types of elements they will be coerced to the most flexible type possible:

```
# numerics are turned to characters
str(c("a", "b", "c", 1, 2, 3))
##  chr [1:6] "a" "b" "c" "1" "2" "3"

# logical are turned to numerics...
str(c(1, 2, 3, TRUE, FALSE))
##  num [1:5] 1 2 3 1 0

# or character
str(c("A", "B", "C", TRUE, FALSE))
##  chr [1:5] "A" "B" "C" "TRUE" "FALSE"
```

Adding on to

To add additional elements to a pre-existing vector we can continue to leverage the `c()` function. Also, note that vectors are always flat so nested `c()` functions will not add additional dimensions to the vector:

```
v1 <- 8:17

c(v1, 18:22)
##  [1]  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22

# same as
c(v1, c(18, c(19, c(20, c(21:22)))))
##  [1]  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

Adding attributes

The attributes that you can add to vectors includes names and comments. If we continue with our vector `v1` we can see that the vector currently has no attributes:

```
attributes(v1)
## NULL
```

We can add names to vectors using two approaches. The first uses `names()` to assign names to each element of the vector. The second approach is to assign names when creating the vector.

```
# assigning names to a pre-existing vector
names(v1) <- letters[1:length(v1)]
v1
##  a  b  c  d  e  f  g  h  i  j
##  8  9 10 11 12 13 14 15 16 17
attributes(v1)
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

# adding names when creating vectors
v2 <- c(name1 = 1, name2 = 2, name3 = 3)
v2
## name1 name2 name3
##     1     2     3
attributes(v2)
## $names
## [1] "name1" "name2" "name3"
```

We can also add comments to vectors to act as a note to the user. This does not change how the vector behaves; rather, it simply acts as a form of metadata for the vector.

```
comment(v1) <- "This is a comment on a vector"
v1
##  a  b  c  d  e  f  g  h  i  j
##  8  9 10 11 12 13 14 15 16 17
attributes(v1)
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
##
## $comment
## [1] "This is a comment on a vector"
```

Subsetting

The four main ways to subset a vector include combining square brackets [ ] with:

- Positive integers
- Negative integers
- Logical values
- Names

You can also subset with double brackets `[[ ]]` for simplifying subsets.

Subsetting with positive integers

Subsetting with positive integers returns the elements at the specified positions:

```
v1
##  a  b  c  d  e  f  g  h  i  j
##  8  9 10 11 12 13 14 15 16 17

v1[2]
## b
## 9

v1[2:4]
##  b  c  d
##  9 10 11

v1[c(2, 4, 6, 8)]
##  b  d  f  h
##  9 11 13 15

# note that you can duplicate index positions
v1[c(2, 2, 4)]
##  b  b  d
##  9  9 11
```

Subsetting with negative integers

Subsetting with negative integers will omit the elements at the specified positions:

```
v1[-1]
##  b  c  d  e  f  g  h  i  j
##  9 10 11 12 13 14 15 16 17
```

```
v1[-c(2, 4, 6, 8)]
##  a  c  e  g  i  j
##  8 10 12 14 16 17
```

Subsetting with logical values

Subsetting with logical values will select the elements where the corresponding logical value is `TRUE`:

```
v1[c(TRUE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE)]
##  a  c  e  f  g  j
##  8 10 12 13 14 17

v1[v1 < 12]
##  a  b  c  d
##  8  9 10 11

v1[v1 < 12 | v1 > 15]
##  a  b  c  d  i  j
##  8  9 10 11 16 17

# if logical vector is shorter than the length of the vector being
# subsetted, it will be recycled to be the same length
v1[c(TRUE, FALSE)]
##  a  c  e  g  i
##  8 10 12 14 16
```

Subsetting with names

Subsetting with names will return the elements with the matching names specified:

```
v1["b"]
## b
## 9

v1[c("a", "c", "h")]
##  a  c  h
##  8 10 15
```

Simplifying vs. Preserving

Its also important to understand the difference between simplifying and preserving when subsetting. **Simplifying** subsets returns the simplest possible data structure that can represent the output. **Preserving** subsets keeps the structure of the output the same as the input.

For vectors, subsetting with single brackets `[ ]` preserves while subsetting with double brackets `[[ ]]` simplifies. The change you will notice when simplifying vectors is the removal of names.

```
v1[1]
## a
## 8

v1[[1]]
## [1] 8
```

Managing Lists

A list is an R structure that allows you to combine elements of different types, including lists embedded in a list, and length. Many statistical outputs are provided as a list as well; therefore, its critical to understand how to work with lists. In this chapter I will illustrate how to create lists, add additional elements to pre-existing lists, add attributes to lists, and subset lists.

Creating

To create a list we can use the `list()` function. Note how each of the four list items are of different classes (integer, character, logical, and numeric) and different length.

```
l <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.5, 4.2))
str(l)
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.5 4.2

# a list containing a list
l <- list(1:3, list(letters[1:5], c(TRUE, FALSE, TRUE)))
str(l)
## List of 2
##  $ : int [1:3] 1 2 3
##  $ :List of 2
##   ..$ : chr [1:5] "a" "b" "c" "d" ...
##   ..$ : logi [1:3] TRUE FALSE TRUE
```

Adding on to

To add additional list components to a list we can leverage the `list()` and `append()` functions. We can illustrate with the following list.

```
l1 <- list(1:3, "a", c(TRUE, FALSE, TRUE))
str(l1)
## List of 3
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
```

If we add the new elements with `list()` it will create a list of two components, component 1 will be a nested list of the original list and component 2 will be the new elements added:

```
l2 <- list(l1, c(2.5, 4.2))
str(l2)
## List of 2
##  $ :List of 3
##   ..$ : int [1:3] 1 2 3
##   ..$ : chr "a"
##   ..$ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.5 4.2
```

To simply add a 4th list component without creating nested lists we use the `append()` function:

```
l3 <- append(l1, list(c(2.5, 4.2)))
str(l3)
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.5 4.2
```

Alternatively, we can also add a new list component by utilizing the '$' sign and naming the new item:

```
l3$item4 <- "new list item"
str(l3)
## List of 5
##  $      : int [1:3] 1 2 3
##  $      : chr "a"
##  $      : logi [1:3] TRUE FALSE TRUE
##  $      : num [1:2] 2.5 4.2
##  $ item4: chr "new list item"
```

To add individual elements to a specific list component we need to introduce some subsetting which is further discussed later in the chapter in the Subsetting section. We'll continue with our original `l1` list:

```
str(l1)
## List of 3
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
```

To add additional values to a list item you need to subset for that specific list item and then you can use the `c()` function to add the additional elements to that list item:

```
l1[[1]] <- c(l1[[1]], 4:6)
str(l1)
## List of 3
##  $ : int [1:6] 1 2 3 4 5 6
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE

l1[[2]] <- c(l1[[2]], c("dding", "to a", "list"))
str(l1)
## List of 3
##  $ : int [1:6] 1 2 3 4 5 6
##  $ : chr [1:4] "a" "dding" "to a" "list"
##  $ : logi [1:3] TRUE FALSE TRUE
```

Adding attributes

The attributes that you can add to lists include names, general comments, and specific list item comments. Currently, our `l1` list has no attributes:

```
attributes(l1)
## NULL
```

We can add names to lists in two ways. First, we can use `names()` to assign names to list items in a pre-existing list. Second, we can add names to a list when we are creating a list.

```
# adding names to a pre-existing list
names(l1) <- c("item1", "item2", "item3")
str(l1)
## List of 3
##  $ item1: int [1:6] 1 2 3 4 5 6
##  $ item2: chr [1:4] "a" "dding" "to a" "list"
##  $ item3: logi [1:3] TRUE FALSE TRUE
attributes(l1)
## $names
## [1] "item1" "item2" "item3"

# adding names when creating lists
l2 <- list(item1 = 1:3, item2 = letters[1:5], item3 = c(T, F, T, T))
str(l2)
## List of 3
##  $ item1: int [1:3] 1 2 3
##  $ item2: chr [1:5] "a" "b" "c" "d" ...
##  $ item3: logi [1:4] TRUE FALSE TRUE TRUE
attributes(l2)
## $names
## [1] "item1" "item2" "item3"
```

We can also add comments to lists. As previously mentioned, comments act as a note to the user without changing how the object behaves. With lists, we can add a general comment to the list using `comment()` and we can also add comments to specific list items with `attr()`.

```
# adding a general comment to list l2 with comment()
comment(l2) <- "This is a comment on a list"
str(l2)
## List of 3
##  $ item1: int [1:3] 1 2 3
##  $ item2: chr [1:5] "a" "b" "c" "d" ...
##  $ item3: logi [1:4] TRUE FALSE TRUE TRUE
##  - attr(*, "comment")= chr "This is a comment on a list"
attributes(l2)
## $names
## [1] "item1" "item2" "item3"
##
## $comment
## [1] "This is a comment on a list"

# adding a comment to a specific list item with attr()
```

```
attr(l2, "item2") <- "Comment for item2"
str(l2)
## List of 3
##  $ item1: int [1:3] 1 2 3
##  $ item2: chr [1:5] "a" "b" "c" "d" ...
##  $ item3: logi [1:4] TRUE FALSE TRUE TRUE
##  - attr(*, "comment")= chr "This is a comment on a list"
##  - attr(*, "item2")= chr "Comment for item2"
attributes(l2)
## $names
## [1] "item1" "item2" "item3"
##
## $comment
## [1] "This is a comment on a list"
##
## $item2
## [1] "Comment for item2"
```

Subsetting

> *"If list x is a train carrying objects, then x[[5]] is the object in car 5; x[4:6] is a train of cars 4-6"* - @RLangTip

To subset lists we can utilize the single bracket `[ ]`, double brackets `[[ ]]`, and dollar sign `$` operators. Each approach provides a specific purpose and can be combined in different ways to achieve the following subsetting objectives:

- Subset list and preserve output as a list
- Subset list and simplify output
- Subset list to get elements out of a list
- Subset list with a nested list

Subset list and preserve output as a list

To extract one or more list items while **preserving**[63] the output in list format use the `[ ]` operator:

[63] Its important to understand the difference between simplifying and preserving subsetting. **Simplifying** subsets returns the simplest possible data structure that can represent the output. **Preserving** subsets keeps the structure of the output the same as the input. See Hadley Wickham's section on Simplifying vs. Preserving Subsetting to learn more.

```
# extract first list item
l2[1]
## $item1
## [1] 1 2 3

# same as above but using the item's name
l2["item1"]
## $item1
## [1] 1 2 3

# extract multiple list items
l2[c(1,3)]
## $item1
## [1] 1 2 3
##
## $item3
## [1]  TRUE FALSE  TRUE  TRUE

# same as above but using the items' names
l2[c("item1", "item3")]
## $item1
## [1] 1 2 3
##
## $item3
## [1]  TRUE FALSE  TRUE  TRUE
```

Subset list and simplify output

To extract one or more list items while **simplifying**[64] the output use the `[[ ]]` or `$` operator:

```
# extract first list item and simplify to a vector
l2[[1]]
## [1] 1 2 3

# same as above but using the item's name
l2[["item1"]]
## [1] 1 2 3

# same as above but using the `$` operator
```

[64]Its important to understand the difference between simplifying and preserving subsetting. **Simplifying** subsets returns the simplest possible data structure that can represent the output. **Preserving** subsets keeps the structure of the output the same as the input. See Hadley Wickham's section on Simplifying vs. Preserving Subsetting to learn more.

```
l2$item1
## [1] 1 2 3
```

One thing that differentiates the [[operator from the $ is that the [[operator can be used with computed indices. The $ operator can only be used with literal names.

Subset list to get elements out of a list

To extract individual elements out of a specific list item combine the `[[` (or $) operator with the `[` operator:

```
# extract third element from the second list item
l2[[2]][3]
## [1] "c"

# same as above but using the item's name
l2[["item2"]][3]
## [1] "c"

# same as above but using the `$` operator
l2$item2[3]
## [1] "c"
```

Subset list with a nested list

If you have nested lists you can expand the ideas above to extract items and elements. We'll use the following list `l3` which has a nested list in item `2`.

```
l3 <- list(item1 = 1:3,
           item2 = list(item2a = letters[1:5],
                        item3b = c(T, F, T, T)))
str(l3)
## List of 2
##  $ item1: int [1:3] 1 2 3
##  $ item2:List of 2
##   ..$ item2a: chr [1:5] "a" "b" "c" "d" ...
##   ..$ item3b: logi [1:4] TRUE FALSE TRUE TRUE
```

If the goal is to subset `l3` to extract the nested list item `item2a` from `item2`, we can perform this multiple ways.

```
# preserve the output as a list
l3[[2]][1]
## $item2a
## [1] "a" "b" "c" "d" "e"

# same as above but simplify the output
l3[[2]][[1]]
## [1] "a" "b" "c" "d" "e"

# same as above with names
l3[["item2"]][["item2a"]]
## [1] "a" "b" "c" "d" "e"

# same as above with `$` operator
l3$item2$item2a
## [1] "a" "b" "c" "d" "e"

# extract individual element from a nested list item
l3[[2]][[1]][3]
## [1] "c"
```

Managing Matrices

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. In R, the elements that make up a matrix must be of a consistant mode (i.e. all elements must be numeric, or character, etc.). Therefore, a matrix can be thought of as an atomic vector with a dimension attribute. Furthermore, all rows of a matrix must be of same length. In this chapter I will illustrate how to create matrices, add additional elements to pre-existing matrices, add attributes to matrices, and subset matrices.

Creating

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns. We can create a matrix using the `matrix()` function and specifying the values to fill in the matrix and the number of rows and columns to make the matrix.

```
# numeric matrix
m1 <- matrix(1:6, nrow = 2, ncol = 3)
m1
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

The underlying structure of this matrix is simply an integer vector with an added 2x3 dimension attribute.

```
str(m1)
##  int [1:2, 1:3] 1 2 3 4 5 6
attributes(m1)
## $dim
## [1] 2 3
```

Matrices can also contain character values. Whether a matrix contains data that are of numeric or character type, all the elements must be of the same class.

```
# a character matrix
m2 <- matrix(letters[1:6], nrow = 2, ncol = 3)
m2
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"

# structure of m2 is simply character vector with 2x3 dimension
str(m2)
##  chr [1:2, 1:3] "a" "b" "c" "d" "e" "f"
attributes(m2)
## $dim
## [1] 2 3
```

Matrices can also be created using the column-bind `cbind()` and row-bind `rbind()` functions. However, keep in mind that the vectors that are being binded must be of equal length and mode.

```
v1 <- 1:4
v2 <- 5:8

cbind(v1, v2)
##      v1 v2
## [1,]  1  5
## [2,]  2  6
## [3,]  3  7
## [4,]  4  8

rbind(v1, v2)
##    [,1] [,2] [,3] [,4]
## v1    1    2    3    4
## v2    5    6    7    8

# bind several vectors together
v3 <- 9:12

cbind(v1, v2, v3)
##      v1 v2 v3
## [1,]  1  5  9
## [2,]  2  6 10
## [3,]  3  7 11
## [4,]  4  8 12
```

Adding on to

We can leverage the `cbind()` and `rbind()` functions for adding onto matrices as well. Again, its important to keep in mind that the vectors that are being binded must be of equal length and mode to the pre-existing matrix.

```
m1 <- cbind(v1, v2)
m1
##      v1 v2
## [1,]  1  5
## [2,]  2  6
## [3,]  3  7
## [4,]  4  8

# add a new column
cbind(m1, v3)
##      v1 v2 v3
## [1,]  1  5  9
## [2,]  2  6 10
## [3,]  3  7 11
## [4,]  4  8 12

# or add a new row
rbind(m1, c(4.1, 8.1))
##       v1  v2
## [1,] 1.0 5.0
## [2,] 2.0 6.0
## [3,] 3.0 7.0
## [4,] 4.0 8.0
## [5,] 4.1 8.1
```

Adding attributes

As previously mentioned, matrices by default will have a dimension attribute as illustrated in the following matrix `m2`.

```
# basic matrix
m2 <- matrix(1:12, nrow = 4, ncol = 3)
m2
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12

# the dimension attribute shows this matrix has 4 rows and 3 columns
attributes(m2)
## $dim
## [1] 4 3
```

However, matrices can also have additional attributes such as row names, column names, and comments. Adding names can be done individually, meaning we can add row names or column names separately.

```
# add row names as an attribute
rownames(m2) <- c("row1", "row2", "row3", "row4")
m2
##      [,1] [,2] [,3]
## row1    1    5    9
## row2    2    6   10
## row3    3    7   11
## row4    4    8   12

# attributes displayed will now show the dimension, list the row names
# and will show the column names as NULL
attributes(m2)
## $dim
## [1] 4 3
##
## $dimnames
## $dimnames[[1]]
## [1] "row1" "row2" "row3" "row4"
##
## $dimnames[[2]]
## NULL

# add column names
colnames(m2) <- c("col1", "col2", "col3")
```

```
m2
##      col1 col2 col3
## row1    1    5    9
## row2    2    6   10
## row3    3    7   11
## row4    4    8   12
attributes(m2)
## $dim
## [1] 4 3
##
## $dimnames
## $dimnames[[1]]
## [1] "row1" "row2" "row3" "row4"
##
## $dimnames[[2]]
## [1] "col1" "col2" "col3"
```

Another option is to use the `dimnames()` function. To add row names you assign the names to `dimnames(m2)[[1]]` and to add column names you assign the names to `dimnames(m2)[[2]]`.

```
dimnames(m2)[[1]] <- c("row_1", "row_2", "row_3", "row_4")
m2
##       col1 col2 col3
## row_1    1    5    9
## row_2    2    6   10
## row_3    3    7   11
## row_4    4    8   12

# column names are contained in the second list item
dimnames(m2)[[2]] <- c("col_1", "col_2", "col_3")
m2
##       col_1 col_2 col_3
## row_1     1     5     9
## row_2     2     6    10
## row_3     3     7    11
## row_4     4     8    12
```

Lastly, similar to lists and vectors you can add a comment attribute to a list.

```
comment(m2) <- "adding a comment to a matrix"
attributes(m2)
## $dim
## [1] 4 3
##
## $dimnames
## $dimnames[[1]]
## [1] "row_1" "row_2" "row_3" "row_4"
##
## $dimnames[[2]]
## [1] "col_1" "col_2" "col_3"
##
##
## $comment
## [1] "adding a comment to a matrix"
```

Subsetting

To subset matrices we use the `[` operator; however, since matrices have `2` dimensions we need to incorporate subsetting arguments for both row and column dimensions. A generic form of matrix subsetting looks like: `matrix[rows, columns]`. We can illustrate with matrix `m2`:

```
m2
##       col_1 col_2 col_3
## row_1     1     5     9
## row_2     2     6    10
## row_3     3     7    11
## row_4     4     8    12
```

By using different values in the `rows` and `columns` argument of `m2[rows, columns]`, we can subset `m2` in multiple ways.

```
# subset for rows 1 and 2 but keep all columns
m2[1:2, ]
##       col_1 col_2 col_3
## row_1     1     5     9
## row_2     2     6    10

# subset for columns 1 and 3 but keep all rows
m2[ , c(1, 3)]
##       col_1 col_3
```

```
## row_1     1     9
## row_2     2    10
## row_3     3    11
## row_4     4    12

# subset for both rows and columns
m2[1:2, c(1, 3)]
##       col_1 col_3
## row_1     1     9
## row_2     2    10

# use a vector to subset
v <- c(1, 2, 4)
m2[v, c(1, 3)]
##       col_1 col_3
## row_1     1     9
## row_2     2    10
## row_4     4    12

# use names to subset
m2[c("row_1", "row_3"), ]
##       col_1 col_2 col_3
## row_1     1     5     9
## row_3     3     7    11
```

Note that subsetting matrices with the `[` operator will simplify the results to the lowest possible dimension. To avoid this you can introduce the `drop = FALSE` argument:

```
# simplifying results in a named vector
m2[, 2]
## row_1 row_2 row_3 row_4
##     5     6     7     8

# preserving results in a 4x1 matrix
m2[, 2, drop = FALSE]
##       col_2
## row_1     5
## row_2     6
## row_3     7
## row_4     8
```

Managing Data Frames

A data frame is the most common way of storing data in R and, generally, is the data structure most often used for data analyses. Under the hood, a data frame is a list of equal-length vectors. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows. As a result, data frames can store different classes of objects in each column (i.e. numeric, character, factor). In essence, the easiest way to think of a data frame is as an Excel worksheet that contains columns of different types of data but are all of equal length rows. In this chapter I will illustrate how to create data frames, add additional elements to pre-existing data frames, add attributes to data frames, and subset data frames.

Creating

Data frames are usually created by reading in a dataset using the `read.table()` or `read.csv()`; this will be covered in the importing and scraping data chapters. However, data frames can also be created explicitly with the `data.frame()` function or they can be coerced from other types of objects like lists. In this case I'll create a simple data frame `df` and assess its basic structure:

```
df <- data.frame(col1 = 1:3,
                 col2 = c("this", "is", "text"),
                 col3 = c(TRUE, FALSE, TRUE),
                 col4 = c(2.5, 4.2, pi))

# assess the structure of a data frame
str(df)
## 'data.frame':        3 obs. of  4 variables:
##  $ col1: int  1 2 3
##  $ col2: Factor w/ 3 levels "is","text","this": 3 1 2
##  $ col3: logi  TRUE FALSE TRUE
##  $ col4: num  2.5 4.2 3.14

# number of rows
nrow(df)
## [1] 3

# number of columns
ncol(df)
## [1] 4
```

Note how col2 in df was converted to a column of factors. This is because there is a default setting in data.frame() that converts character columns to factors. We can turn this off by setting the stringsAsFactors = FALSE argument:

```
df <- data.frame(col1 = 1:3,
                 col2 = c("this", "is", "text"),
                 col3 = c(TRUE, FALSE, TRUE),
                 col4 = c(2.5, 4.2, pi),
                 stringsAsFactors = FALSE)

# note how col2 now is of a character class
str(df)
## 'data.frame':        3 obs. of  4 variables:
##  $ col1: int  1 2 3
##  $ col2: chr  "this" "is" "text"
##  $ col3: logi  TRUE FALSE TRUE
##  $ col4: num  2.5 4.2 3.14
```

We can also convert pre-existing structures to a data frame. The following illustrates how we can turn multiple vectors, a list, or a matrix into a data frame:

```
v1 <- 1:3
v2 <-c("this", "is", "text")
v3 <- c(TRUE, FALSE, TRUE)

# convert same length vectors to a data frame using data.frame()
data.frame(col1 = v1, col2 = v2, col3 = v3)
##   col1 col2  col3
## 1    1 this  TRUE
## 2    2   is FALSE
## 3    3 text  TRUE

# convert a list to a data frame using as.data.frame()
l <- list(item1 = 1:3, item2 = c("this", "is", "text"), item3 = c(2.5, 4.2, 5.1))
l
## $item1
## [1] 1 2 3
##
## $item2
## [1] "this" "is"   "text"
##
## $item3
```

```
## [1] 2.5 4.2 5.1

as.data.frame(l)
##   item1 item2 item3
## 1     1  this   2.5
## 2     2    is   4.2
## 3     3  text   5.1

# convert a matrix to a data frame using as.data.frame()
m1 <- matrix(1:12, nrow = 4, ncol = 3)
m1
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12

as.data.frame(m1)
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

Adding on to

We can leverage the `cbind()` function for adding columns to a data frame. Note that one of the objects being combined must already be a data frame otherwise `cbind()` could produce a matrix.

```
df
##   col1 col2  col3     col4
## 1    1 this  TRUE 2.500000
## 2    2   is FALSE 4.200000
## 3    3 text  TRUE 3.141593

# add a new column
v4 <- c("A", "B", "C")
cbind(df, v4)
##   col1 col2  col3     col4 v4
## 1    1 this  TRUE 2.500000  A
## 2    2   is FALSE 4.200000  B
## 3    3 text  TRUE 3.141593  C
```

We can also use the `rbind()` function to add data frame rows together. However, severe caution should be taken because this can cause changes in the classes of the columns. For instance, our data frame `df` currently consists of an integer, character, logical, and numeric variables.

```
df
##   col1 col2  col3     col4
## 1    1 this  TRUE 2.500000
## 2    2   is FALSE 4.200000
## 3    3 text  TRUE 3.141593
str(df)
## 'data.frame':        3 obs. of  4 variables:
##  $ col1: int  1 2 3
##  $ col2: chr  "this" "is" "text"
##  $ col3: logi  TRUE FALSE TRUE
##  $ col4: num  2.5 4.2 3.14
```

If we attempt to add a row using `rbind()` and `c()` it converts all columns to a character class. This is because all elements in the vector created by `c()` must be of the same class so they are all coerced to the character class which coerces all the variables in the data frame to the character class.

```
df2 <- rbind(df, c(4, "R", F, 1.1))
df2
##   col1 col2  col3             col4
## 1    1 this  TRUE              2.5
## 2    2   is FALSE              4.2
## 3    3 text  TRUE 3.14159265358979
## 4    4    R FALSE              1.1
str(df2)
## 'data.frame':        4 obs. of  4 variables:
##  $ col1: chr  "1" "2" "3" "4"
##  $ col2: chr  "this" "is" "text" "R"
##  $ col3: chr  "TRUE" "FALSE" "TRUE" "FALSE"
##  $ col4: chr  "2.5" "4.2" "3.14159265358979" "1.1"
```

To add rows appropriately, we need to convert the items being added to a data frame and make sure the columns are the same class as the original data frame.

```
adding_df <- data.frame(col1 = 4, col2 = "R", col3 = FALSE, col4 = 1.1,
                 stringsAsFactors = FALSE)

df3 <- rbind(df, adding_df)
df3
##   col1 col2  col3     col4
## 1    1 this  TRUE 2.500000
## 2    2   is FALSE 4.200000
## 3    3 text  TRUE 3.141593
## 4    4    R FALSE 1.100000
str(df3)
## 'data.frame':        4 obs. of  4 variables:
##  $ col1: num  1 2 3 4
##  $ col2: chr  "this" "is" "text" "R"
##  $ col3: logi  TRUE FALSE TRUE FALSE
##  $ col4: num  2.5 4.2 3.14 1.1
```

There are better ways to join data frames together than to use `cbind()` and `rbind()`. These are covered later on in the transforming your data with `dplyr` chapter.

Adding attributes

Similar to matrices, data frames will have a dimension attribute. In addition, data frames can also have additional attributes such as row names, column names, and comments. We can illustrate with data frame `df`.

```
# basic matrix
df
##   col1 col2  col3     col4
## 1    1 this  TRUE 2.500000
## 2    2   is FALSE 4.200000
## 3    3 text  TRUE 3.141593
dim(df)
## [1] 3 4
attributes(df)
## $names
## [1] "col1" "col2" "col3" "col4"
##
## $row.names
## [1] 1 2 3
##
```

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

Currently `df` does not have row names but we can add them with `rownames()`:

```
# add row names
rownames(df) <- c("row1", "row2", "row3")
df
##      col1 col2  col3     col4
## row1    1 this  TRUE 2.500000
## row2    2   is FALSE 4.200000
## row3    3 text  TRUE 3.141593
attributes(df)
## $names
## [1] "col1" "col2" "col3" "col4"
##
## $row.names
## [1] "row1" "row2" "row3"
##
## $class
## [1] "data.frame"
```

We can also also change the existing column names by using `colnames()` or `names()`:

```
# add/change column names with colnames()
colnames(df) <- c("col_1", "col_2", "col_3", "col_4")
df
##      col_1 col_2 col_3    col_4
## row1     1  this  TRUE 2.500000
## row2     2    is FALSE 4.200000
## row3     3  text  TRUE 3.141593
attributes(df)
## $names
## [1] "col_1" "col_2" "col_3" "col_4"
##
## $row.names
## [1] "row1" "row2" "row3"
##
## $class
## [1] "data.frame"

# add/change column names with names()
```

```
names(df) <- c("col.1", "col.2", "col.3", "col.4")
df
##      col.1 col.2 col.3    col.4
## row1     1  this  TRUE 2.500000
## row2     2    is FALSE 4.200000
## row3     3  text  TRUE 3.141593
attributes(df)
## $names
## [1] "col.1" "col.2" "col.3" "col.4"
##
## $row.names
## [1] "row1" "row2" "row3"
##
## $class
## [1] "data.frame"
```

Lastly, just like vectors, lists, and matrices, we can add a comment to a data frame without affecting how it operates.

```
# adding a comment attribute
comment(df) <- "adding a comment to a data frame"
attributes(df)
## $names
## [1] "col.1" "col.2" "col.3" "col.4"
##
## $row.names
## [1] "row1" "row2" "row3"
##
## $class
## [1] "data.frame"
##
## $comment
## [1] "adding a comment to a data frame"
```

Subsetting

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists and will return the selected columns with all rows; if you subset with two vectors, they behave like matrices and can be subset by row and column:

```
df
##      col.1 col.2 col.3    col.4
## row1     1  this  TRUE 2.500000
## row2     2    is FALSE 4.200000
## row3     3  text  TRUE 3.141593

# subsetting by row numbers
df[2:3, ]
##      col.1 col.2 col.3    col.4
## row2     2    is FALSE 4.200000
## row3     3  text  TRUE 3.141593

# subsetting by row names
df[c("row2", "row3"), ]
##      col.1 col.2 col.3    col.4
## row2     2    is FALSE 4.200000
## row3     3  text  TRUE 3.141593

# subsetting columns like a list
df[c("col.2", "col.4")]
##      col.2    col.4
## row1  this 2.500000
## row2    is 4.200000
## row3  text 3.141593

# subsetting columns like a matrix
df[ , c("col.2", "col.4")]
##      col.2    col.4
## row1  this 2.500000
## row2    is 4.200000
## row3  text 3.141593

# subset for both rows and columns
df[1:2, c(1, 3)]
##      col.1 col.3
## row1     1  TRUE
## row2     2 FALSE

# use a vector to subset
v <- c(1, 2, 4)
df[ , v]
##      col.1 col.2    col.4
```

```
## row1     1  this 2.500000
## row2     2    is 4.200000
## row3     3  text 3.141593
```

Note that subsetting data frames with the `[` operator will simplify the results to the lowest possible dimension. To avoid this you can introduce the `drop = FALSE` argument:

```
# simplifying results in a named vector
df[, 2]
## [1] "this" "is"   "text"

# preserving results in a 3x1 data frame
df[, 2, drop = FALSE]
##      col.2
## row1  this
## row2    is
## row3  text
```

Dealing with Missing Values

A common task in data analysis is dealing with missing values. In R, missing values are often represented by `NA` or some other value that represents missing values (i.e. `99`). We can easily work with missing values and in this chapter I illustrate how to test for, recode, and exclude missing values in your data.

Testing for missing values

To identify missing values use `is.na()` which returns a logical vector with `TRUE` in the element locations that contain missing values represented by `NA`. `is.na()` will work on vectors, lists, matrices, and data frames.

```
# vector with missing data
x <- c(1:4, NA, 6:7, NA)
x
## [1]  1  2  3  4 NA  6  7 NA

is.na(x)
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE

# data frame with missing data
df <- data.frame(col1 = c(1:3, NA),
                 col2 = c("this", NA,"is", "text"),
                 col3 = c(TRUE, FALSE, TRUE, TRUE),
                 col4 = c(2.5, 4.2, 3.2, NA),
                 stringsAsFactors = FALSE)

# identify NAs in full data frame
is.na(df)
##       col1  col2  col3  col4
## [1,] FALSE FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE FALSE
## [3,] FALSE FALSE FALSE FALSE
## [4,]  TRUE FALSE FALSE  TRUE

# identify NAs in specific data frame column
is.na(df$col4)
## [1] FALSE FALSE FALSE  TRUE
```

To identify the location or the number of NAs we can leverage the `which()` and `sum()` functions:

```
# identify location of NAs in vector
which(is.na(x))
## [1] 5 8

# identify count of NAs in data frame
sum(is.na(df))
## [1] 3
```

Recoding missing values

To recode missing values; or recode specific indicators that represent missing values, we can use normal subsetting and assignment operations. For example, we can recode missing values in vector `x` with the mean values in `x` by first subsetting the vector to identify `NA`s and then assign these elements a value. Similarly, if missing values are represented by another value (i.e. `99`) we can simply subset the data for the elements that contain that value and then assign a desired value to those elements.

```
# recode missing values with the mean
x[is.na(x)] <- mean(x, na.rm = TRUE)
round(x, 2)
## [1] 1.00 2.00 3.00 4.00 3.83 6.00 7.00 3.83

# data frame that codes missing values as 99
df <- data.frame(col1 = c(1:3, 99), col2 = c(2.5, 4.2, 99, 3.2))

# change 99s to NAs
df[df == 99] <- NA
df
##   col1 col2
## 1    1  2.5
## 2    2  4.2
## 3    3   NA
## 4   NA  3.2
```

Excluding missing values

We can exclude missing values in a couple different ways. First, if we want to exclude missing values from mathematical operations use the `na.rm = TRUE` argument. If you do not exclude these values most functions will return an `NA`.

```
# A vector with missing values
x <- c(1:4, NA, 6:7, NA)

# including NA values will produce an NA output
mean(x)
## [1] NA

# excluding NA values will calculate the mathematical
# operation for all non-missing values
mean(x, na.rm = TRUE)
## [1] 3.833333
```

We may also desire to subset our data to obtain complete observations, those observations (rows) in our data that contain no missing data. We can do this a few different ways.

```
# data frame with missing values
df <- data.frame(col1 = c(1:3, NA),
                 col2 = c("this", NA,"is", "text"),
                 col3 = c(TRUE, FALSE, TRUE, TRUE),
                 col4 = c(2.5, 4.2, 3.2, NA),
                 stringsAsFactors = FALSE)
df
##   col1 col2  col3 col4
## 1    1 this  TRUE  2.5
## 2    2 <NA> FALSE  4.2
## 3    3   is  TRUE  3.2
## 4   NA text  TRUE   NA
```

First, to find complete cases we can leverage the `complete.cases()` function which returns a logical vector identifying rows which are complete cases. So in the following case rows 1 and 3 are complete cases. We can use this information to subset our data frame which will return the rows which `complete.cases()` found to be `TRUE`.

```
complete.cases(df)
## [1]  TRUE FALSE  TRUE FALSE

# subset with complete.cases to get complete cases
df[complete.cases(df), ]
##   col1 col2 col3 col4
## 1    1 this TRUE  2.5
## 3    3   is TRUE  3.2

# or subset with `!` operator to get incomplete cases
df[!complete.cases(df), ]
##   col1 col2  col3 col4
## 2    2 <NA> FALSE  4.2
## 4   NA text  TRUE   NA
```

An shorthand alternative is to simply use `na.omit()` to omit all rows containing missing values.

```
# or use na.omit() to get same as above
na.omit(df)
##   col1 col2 col3 col4
## 1    1 this TRUE  2.5
## 3    3   is TRUE  3.2
```

Importing, Scraping, and Exporting Data with R

> *"What we have is a data glut."* - Vernon Vinge

Data are being generated by everything around us at all times. Every digital process and social media exchange produces it. Systems, sensors and mobile devices transmit it. Countless databases collect it. Data are arriving from multiple sources at an alarming rate and analysts and organizations are seeking ways to leverage these new sources of information. Consequently, analysts need to understand how to *get* data from these data sources. Furthermore, since analysis is often a collaborative effort analysts also need to know how to share their data.

This section covers the process of importing, scraping, and exporting data. First, I cover the basics of importing tabular and spreadsheet data. Second, since modern day data wrangling often includes scraping data from the flood of web-based data becoming available to organizations and analysts, I cover the fundamentals of web-scraping with R. This includes importing spreadsheet data files stored online, scraping HTML text and data tables, and leveraging APIs. Third, although getting data into R is essential, I also cover the equally important process of getting data out of R. Consequently, this section will give you a strong foundation for the different ways to get your data into and out of R.

Importing Data

The first step to any data analysis process is to *get* the data. Data can come from many sources but two of the most common include text and Excel files. This chapter covers how to import data into R by reading data from common text files and Excel spreadsheets. In addition, I cover how to load data from saved R object files for holding or transferring data that has been processed in R. In addition to the the commonly used base R functions to perform data importing, I will also cover functions from the popular `readr`[65], `xlsx`[66], and `readxl`[67] packages.

Reading data from text files

Text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Text file formats use delimiters to separate the different elements in a line, and each line of data is in its own line in the text file. Therefore, importing different kinds of text files can follow a fairly consistent process once you've identified the delimiter.

There are two main groups of functions that we can use to read in text files:

- Base R functions
- `readr` package functions

Base R functions

`read.table()` is a multipurpose work-horse function in base R for importing data. The functions `read.csv()` and `read.delim()` are special cases of `read.table()` in which the defaults have been adjusted for efficiency. To illustrate these functions let's work with a CSV file that is saved in our working directory which looks like:

```
variable 1,variable 2,variable 3
10,beer,TRUE
25,wine,TRUE
8,cheese,FALSE
```

To read in the CSV file we can use `read.csv()`. Note that when we assess the structure of the data set that we read in, `variable.2` is automatically coerced to a factor variable and `variable.3` is automatically coerced to a logical variable. Furthermore, any whitespace in the column names are replaced with a ".".

[65] https://cran.rstudio.com/web/packages/readr/
[66] https://cran.rstudio.com/web/packages/xlsx/
[67] https://cran.rstudio.com/web/packages/readxl/

```
mydata = read.csv("mydata.csv")
mydata
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

str(mydata)
## 'data.frame':        3 obs. of  3 variables:
##  $ variable.1: int  10 25 8
##  $ variable.2: Factor w/ 3 levels "beer","cheese",..: 1 3 2
##  $ variable.3: logi  TRUE TRUE FALSE
```

However, we may want to read in `variable.2` as a character variable rather then a factor. We can take care of this by changing the `stringsAsFactors` argument. The default has `stringsAsFactors = TRUE`; however, setting it equal to `FALSE` will read in the variable as a character variable.

```
mydata_2 = read.csv("mydata.csv", stringsAsFactors = FALSE)
mydata_2
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

str(mydata_2)
## 'data.frame':        3 obs. of  3 variables:
##  $ variable.1: int  10 25 8
##  $ variable.2: chr  "beer" "wine" "cheese"
##  $ variable.3: logi  TRUE TRUE FALSE
```

As previously stated `read.csv` is just a wrapper for `read.table` but with adjusted default arguments. Therefore, we can use `read.table` to read in this same data. The two arguments we need to be aware of are the field separator (`sep`) and the argument indicating whether the file contains the names of the variables as its first line (`header`). In `read.table` the defaults are `sep = ""` and `header = FALSE` whereas in `read.csv` the defaults are `sep = ","` and `header = TRUE`. There are multiple other arguments we can use for certain situations which we illustrate below:

```
# provides same results as read.csv above
read.table("mydata.csv", sep=",", header = TRUE, stringsAsFactors = FALSE)
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

# set column and row names
read.table("mydata.csv", sep=",", header = TRUE, stringsAsFactors = FALSE,
           col.names = c("Var 1", "Var 2", "Var 3"),
           row.names = c("Row 1", "Row 2", "Row 3"))
##       Var.1  Var.2 Var.3
## Row 1    10   beer  TRUE
## Row 2    25   wine  TRUE
## Row 3     8 cheese FALSE

# manually set the classes of the columns
set_classes <- read.table("mydata.csv", sep=",", header = TRUE,
                          colClasses = c("numeric", "character", "character"))
str(set_classes)
## 'data.frame':        3 obs. of  3 variables:
##  $ variable.1: num  10 25 8
##  $ variable.2: chr  "beer" "wine" "cheese"
##  $ variable.3: chr  "TRUE" "TRUE" "FALSE"

# limit the number of rows to read in
read.table("mydata.csv", sep=",", header = TRUE, nrows = 2)
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
```

In addition to CSV files, there are other text files that `read.table` works with. The primary difference is what separates the elements. For example, tab delimited text files typically end with the `.txt` extension. You can also use the `read.delim()` function as, similiar to `read.csv()`, `read.delim()` is a wrapper of `read.table()` with defaults set specifically for tab delimited files.

```
# reading in tab delimited text files
read.delim("mydata.txt")
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

# provides same results as read.delim
read.table("mydata.txt", sep="\t", header = TRUE)
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE
```

readr package

Compared to the equivalent base functions, `readr`[68] functions are around 10x faster. They bring consistency to importing functions, they produce data frames in a `data.table` format which are easier to view for large data sets, the default settings removes the "hassels" of `stringsAsFactors`, and they have a more flexible column specification.

To illustrate, we can use `read_csv()` which is equivalent to base R's `read.csv()` function. However, note that `read_csv()` maintains the full variable name (whereas `read.csv` eliminates any spaces in variable names and fills it with '.'). Also, `read_csv()` automatically sets `stringsAsFactors = FALSE`, which can be a controversial topic[69].

```
library(readr)
mydata_3 = read_csv("mydata.csv")
mydata_3
##   variable 1 variable 2 variable 3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

str(mydata_3)
## Classes 'tbl_df', 'tbl' and 'data.frame':       3 obs. of  3 variables:
##  $ variable 1: int  10 25 8
##  $ variable 2: chr  "beer" "wine" "cheese"
##  $ variable 3: logi  TRUE TRUE FALSE
```

`read_csv` also offers many additional arguments for making adjustments to your data as you read it in:

[68] https://cran.rstudio.com/web/packages/readr/
[69] http://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/

```
# specify the column class using col_types
read_csv("mydata.csv", col_types = list(col_double(),
                                        col_character(),
                                        col_character()))
##   variable 1 variable 2 variable 3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

# we can also specify column classes with a string
# in this example d = double, _ skips column, c = character
read_csv("mydata.csv", col_types = "d_c")
##   variable 1 variable 3
## 1         10       TRUE
## 2         25       TRUE
## 3          8      FALSE

# set column names
read_csv("mydata.csv", col_names = c("Var 1", "Var 2", "Var 3"), skip = 1)
##   Var 1  Var 2 Var 3
## 1    10   beer  TRUE
## 2    25   wine  TRUE
## 3     8 cheese FALSE

# set the maximum number of lines to read in
read_csv("mydata.csv", n_max = 2)
##   variable 1 variable 2 variable 3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
```

Similar to base R, `readr` also offers functions to import .txt files (`read_delim()`), fixed-width files (`read_fwf()`), general text files (`read_table()`), and more.

These examples provide the basics for reading in text files. However, sometimes even text files can offer unanticipated difficulties with their formatting. Both the base R and `readr` functions offer many arguments to deal with different formatting issues and I suggest you take time to look at the help files for these functions to learn more (i.e. `?read.table`). Also, you will find more resources at the end of this chapter for importing files.

Reading data from Excel files

With Excel still being the spreadsheet software of choice its important to be able to efficiently import and export data from these files. Often, R users will simply resort to exporting the Excel file as a

CSV file and then import into R using `read.csv`; however, this is far from efficient. This section will teach you how to eliminate the CSV step and to import data directly from Excel using two different packages:

- `xlsx` package
- `readxl` package

Note that there are several packages available to connect R with Excel (i.e. `gdata`, `RODBC`, `XLConnect`, `RExcel`, etc.); however, I am only going to cover the two main packages that I use which provide all the fundamental requirements I've needed for dealing with Excel.

xlsx package

The `xlsx`[70] package provides tools neccessary to interact with Excel 2007 (and older) files from R. Many of the benefits of the `xlsx` come from being able to *export* and *format* Excel files from R. Some of these capabilities will be covered in the Exporting Data chapter; however, in this section we will simply cover *importing* data from Excel with the `xlsx` package.

To illustrate, we'll use similar data from the previous section; however, saved as an .xlsx file in our working director. To import the Excel data we simply use the `read.xlsx()` function:

```
library(xlsx)

# read in first worksheet using a sheet index or name
read.xlsx("mydata.xlsx", sheetName = "Sheet1")
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

read.xlsx("mydata.xlsx", sheetIndex = 1)
##   variable.1 variable.2 variable.3
## 1         10       beer       TRUE
## 2         25       wine       TRUE
## 3          8     cheese      FALSE

# read in second worksheet
read.xlsx("mydata.xlsx", sheetName = "Sheet2")
##   variable.4 variable.5
## 1     Dayton     johnny
```

[70] https://cran.rstudio.com/web/packages/xlsx/

```
## 2   Columbus      amber
## 3  Cleveland       tony
## 4 Cincinnati      alice
```

Since Excel is such a flexible spreadsheet software, people often make notes, comments, headers, etc. at the beginning or end of the files which we may not want to include. If we want to read in data that starts further down in the Excel worksheet we can include the `startRow` argument. If we have a specific range of rows (or columns) to include we can use the `rowIndex` (or `colIndex`) argument.

```
# a worksheet with comments in the first two lines
read.xlsx("mydata.xlsx", sheetName = "Sheet3")
##                                          HEADER..COMPANY.A        NA.
## 1 What if we want to disregard header text in Excel file?       <NA>
## 2                                               variable 6 variable 7
## 3                                                      200       Male
## 4                                                      225     Female
## 5                                                      400     Female
## 6                                                      310       Male

# read in all data below the second line
read.xlsx("mydata.xlsx", sheetName = "Sheet3", startRow = 3)
##   variable.6 variable.7
## 1        200       Male
## 2        225     Female
## 3        400     Female
## 4        310       Male

# read in a range of rows
read.xlsx("mydata.xlsx", sheetName = "Sheet3", rowIndex = 3:5)
##   variable.6 variable.7
## 1        200       Male
## 2        225     Female
```

We can also change the class type of the columns when we read them in:

```
# read in data without changing class type
mydata_sheet1.1 <- read.xlsx("mydata.xlsx", sheetName = "Sheet1")

str(mydata_sheet1.1)
## 'data.frame':        3 obs. of  3 variables:
##  $ variable.1: num  10 25 8
##  $ variable.2: Factor w/ 3 levels "beer","cheese",..: 1 3 2
##  $ variable.3: logi  TRUE TRUE FALSE

# read in data and change class type
mydata_sheet1.2 <- read.xlsx("mydata.xlsx", sheetName = "Sheet1",
                             stringsAsFactors = FALSE,
                             colClasses = c("double", "character", "logical"))

str(mydata_sheet1.2)
## 'data.frame':        3 obs. of  3 variables:
##  $ variable.1: num  10 25 8
##  $ variable.2: chr  "beer" "wine" "cheese"
##  $ variable.3: logi  TRUE TRUE FALSE
```

Another useful argument is `keepFormulas` which allows you to see the text of any formulas in the Excel spreadsheet:

```
# by default keepFormula is set to FALSE so only
# the formula output will be read in
read.xlsx("mydata.xlsx", sheetName = "Sheet4")
##   Future.Value  Rate Periods Present.Value
## 1          500 0.065      10      266.3630
## 2          600 0.085       6      367.7671
## 3          750 0.080      11      321.6621
## 4         1000 0.070      16      338.7346

# changing the keepFormula to TRUE will display the equations
read.xlsx("mydata.xlsx", sheetName = "Sheet4", keepFormulas = TRUE)
##   Future.Value  Rate Periods Present.Value
## 1          500 0.065      10  A2/(1+B2)^C2
## 2          600 0.085       6  A3/(1+B3)^C3
## 3          750 0.080      11  A4/(1+B4)^C4
## 4         1000 0.070      16  A5/(1+B5)^C5
```

readxl package

`readxl`[71] is one of the newest packages for accessing Excel data with R and was developed by Hadley Wickham[72] and the RStudio[73] team who also developed the `readr` package. This package works with both legacy .xls formats and the modern xml-based .xlsx format. Similar to `readr` the `readxl` functions are based on a C++ library so they are extremely fast. Unlike most other packages that deal with Excel, `readxl` has no external dependencies, so you can use it to read Excel data on just about any platform. Additional benefits `readxl` provides includes the ability to load dates and times as POSIXct formatted dates, automatically drops blank columns, and returns outputs as data.table formatted which provides easier viewing for large data sets.

To read in Excel data with `readxl` you use the `read_excel()` function which has very similar operations and arguments as `xlsx`. A few important differences you will see below include: `readxl` will automatically convert date and date-time variables to POSIXct formatted variables, character variables will not be coerced to factors, and logical variables will be read in as integers.

```
library(readxl)

mydata <- read_excel("mydata.xlsx", sheet = "Sheet5")
mydata
##   variable 1 variable 2 variable 3 variable 4          variable 5
## 1         10       beer          1 2015-11-20 2015-11-20 13:30:00
## 2         25       wine          1       <NA> 2015-11-21 16:30:00
## 3          8       <NA>          0 2015-11-22 2015-11-22 14:45:00

str(mydata)
## Classes 'tbl_df', 'tbl' and 'data.frame':        3 obs. of  5 variables:
##  $ variable 1: num  10 25 8
##  $ variable 2: chr  "beer" "wine" NA
##  $ variable 3: num  1 1 0
##  $ variable 4: POSIXct, format: "2015-11-20" NA ...
##  $ variable 5: POSIXct, format: "2015-11-20 13:30:00" "2015-11-21 16:30:00" .\
..
```

The available arguments allow you to change the data as you import it. Some examples are provided:

[71] https://cran.rstudio.com/web/packages/readxl/
[72] https://twitter.com/hadleywickham
[73] https://www.rstudio.com/

```
# change variable names by skipping the first row
# and using col_names to set the new names
read_excel("mydata.xlsx", sheet = "Sheet5", skip = 1,
           col_names = paste("Var", 1:5))
##   Var 1 Var 2 Var 3 Var 4               Var 5
## 1    10  beer     1 42328 2015-11-20 13:30:00
## 2    25  wine     1    NA 2015-11-21 16:30:00
## 3     8  <NA>     0 42330 2015-11-22 14:45:00

# sometimes missing values are set as a sentinel value
# rather than just left blank - (i.e. "999")
read_excel("mydata.xlsx", sheet = "Sheet6")
##   variable 1 variable 2 variable 3 variable 4
## 1         10       beer          1      42328
## 2         25       wine          1        999
## 3          8        999          0      42330

# we can change these to missing values with na argument
read_excel("mydata.xlsx", sheet = "Sheet6", na = "999")
##   variable 1 variable 2 variable 3 variable 4
## 1         10       beer          1      42328
## 2         25       wine          1         NA
## 3          8       <NA>          0      42330
```

One unique difference between `readxl` and `xlsx` is how to deal with column types. Whereas `read.xlsx()` allows you to change the column types to integer, double, numeric, character, or logical; `read_excel()` restricts you to changing column types to blank, numeric, date, or text. The "blank" option allows you to skip columns; however, to change variable 3 to a logical `TRUE`/`FALSE` variable requires a second step.

```
mydata_ex <- read_excel("mydata.xlsx", sheet = "Sheet5",
                        col_types = c("numeric", "blank", "numeric",
                                      "date", "blank"))
mydata_ex
##   variable 1 variable 3 variable 4
## 1         10          1 2015-11-20
## 2         25          1       <NA>
## 3          8          0 2015-11-22

# change variable 3 to a logical variable
mydata_ex$`variable 3` <- as.logical(mydata_ex$`variable 3`)
mydata_ex
```

```
##   variable 1 variable 3 variable 4
## 1         10       TRUE 2015-11-20
## 2         25       TRUE       <NA>
## 3          8      FALSE 2015-11-22
```

Load data from saved R object file

Sometimes you may need to save data or other R objects outside of your workspace. You may want to share R data/objects with co-workers, transfer between projects or computers, or simply archive them. There are three primary ways that people tend to save R data/objects: as .RData, .rda, or as .rds files. The differences behind when you use each will be covered in the Saving data as an R object file section. This section will simply shows how to load these data/object forms.

```
load("mydata.RData")

load(file = "mydata.rda")

name <- readRDS("mydata.rds")
```

Additional resources

In addition to text and Excel files, there are multiple other ways that data are stored and exchanged. Commercial statistical software such as SPSS, SAS, Stata, and Minitab often have the option to store data in a specific format for that software. In addition, analysts commonly use databases to store large quantities of data. R has good support to work with these additional options which we did not cover here. The following provides a list of additional resources to learn about data importing for these specific cases:

- R data import/export manual[74]
- Working with databases[75]
  - MySQL[76]
  - Oracle[77]
  - PostgreSQL[78]
  - SQLite[79]

[74] https://cran.r-project.org/doc/manuals/R-data.html
[75] https://cran.r-project.org/doc/manuals/R-data.html#Relational-databases
[76] https://cran.r-project.org/web/packages/RMySQL/index.html
[77] https://cran.r-project.org/web/packages/ROracle/index.html
[78] https://cran.r-project.org/web/packages/RPostgreSQL/index.html
[79] https://cran.r-project.org/web/packages/RSQLite/index.html

  - Open Database Connectivity databases[80]
- Importing data from commercial software[81]
  - The `foreign`[82] package provides functions that help you load data files from other programs such as SPSS[83], SAS[84], Stata[85], and others into R.

---

[80]https://cran.rstudio.com/web/packages/RODBC/

[81]https://cran.r-project.org/doc/manuals/R-data.html#Importing-from-other-statistical-systems

[82]http://www.rdocumentation.org/packages/foreign

[83]http://www.r-bloggers.com/how-to-open-an-spss-file-into-r/

[84]http://rconvert.com/sas-vs-r-code-compare/5-ways-to-convert-sas-data-to-r/

[85]http://www.r-bloggers.com/how-to-read-and-write-stata-data-dta-files-into-r/

Scraping Data

Rapid growth of the World Wide Web has significantly changed the way we share, collect, and publish data. Vast amount of information is being stored online, both in structured and unstructured forms. Regarding certain questions or research topics, this has resulted in a new problem - no longer is the concern of data scarcity and inaccessibility but, rather, one of overcoming the tangled masses of online data.

Collecting data from the web is not an easy process as there are many technologies used to distribute web content (i.e. HTML[86], XML[87], JSON[88]). Therefore, dealing with more advanced web scraping requires familiarity in accessing data stored in these technologies via R. Through this chapter I will provide an introduction to some of the fundamental tools required to perform basic web scraping. This includes importing spreadsheet data files stored online, scraping HTML text, scraping HTML table data, and leveraging APIs to scrape data.

My purpose in the following sections is to discuss these topics at a level meant to get you started in web scraping; however, this area is vast and complex and this chapter will far from provide you expertise level insight. To advance your knowledge I highly recommend getting copies of *XML and Web Technologies for Data Sciences with R*[89] and *Automated Data Collection with R*[90].

Importing tabular and Excel files stored online

The most basic form of getting data from online is to import tabular (i.e. .txt, .csv) or Excel files that are being hosted online. This is often not considered *web scraping*[91]; however, I think its a good place to start introducing the user to interacting with the web for obtaining data. Importing tabular data is especially common for the many types of government data available online. A quick perusal of Data.gov[92] illustrates nearly 188,510 examples. In fact, we can provide our first example of importing online tabular data by downloading the Data.gov CSV file that lists all the federal agencies that supply data to Data.gov.

[86]https://en.wikipedia.org/wiki/HTML

[87]https://en.wikipedia.org/wiki/XML

[88]https://en.wikipedia.org/wiki/JSON

[89]http://www.amazon.com/XML-Web-Technologies-Data-Sciences/dp/1461478995

[90]http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX

[91]In Automated Data Collection with R Munzert et al. state that "[t]he first way to get data from the web is almost too banal to be considered here and actually not a case of web scraping in the narrower sense."

[92]https://www.data.gov/

```
# the url for the online CSV
url <- "https://www.data.gov/media/federal-agency-participation.csv"

# use read.csv to import
data_gov <- read.csv(url, stringsAsFactors = FALSE)

# for brevity I only display first 6 rows
data_gov[1:6, c(1,3:4)]
##                                       Agency.Name Datasets Last.Entry
## 1           Commodity Futures Trading Commission        3 01/12/2014
## 2           Consumer Financial Protection Bureau        2 09/26/2015
## 3           Consumer Financial Protection Bureau        2 09/26/2015
## 4 Corporation for National and Community Service        3 01/12/2014
## 5 Court Services and Offender Supervision Agency        1 01/12/2014
## 6                       Department of Agriculture      698 12/01/2015
```

Downloading Excel spreadsheets hosted online can be performed just as easily. Recall that there is not a base R function for importing Excel data; however, several packages exist to handle this capability. One package that works smoothly with pulling Excel data from urls is `gdata`[93]. With `gdata` we can use `read.xls()` to download this Fair Market Rents for Section 8 Housing[94] Excel file from the given url.

```
library(gdata)

# the url for the online Excel file
url <- "http://www.huduser.org/portal/datasets/fmr/fmr2015f/FY2015F_4050_Final.x\
ls"

# use read.xls to import
rents <- read.xls(url)

rents[1:6, 1:10]
##    fips2000  fips2010 fmr2 fmr0 fmr1 fmr3 fmr4 county State CouSub
## 1 100199999 100199999  788  628  663 1084 1288      1     1  99999
## 2 100399999 100399999  762  494  643 1123 1318      3     1  99999
## 3 100599999 100599999  670  492  495  834  895      5     1  99999
## 4 100799999 100799999  773  545  652 1015 1142      7     1  99999
## 5 100999999 100999999  773  545  652 1015 1142      9     1  99999
## 6 101199999 101199999  599  481  505  791 1061     11     1  99999
```

[93] https://cran.r-project.org/web/packages/gdata/index.html

[94] http://catalog.data.gov/dataset/fair-market-rents-for-the-section-8-housing-assistance-payments-program

Note that many of the arguments covered in the Importing Data chapter (i.e. specifying sheets to read from, skipping lines) also apply to `read.xls()`. In addition, `gdata` provides some useful functions (`sheetCount()` and `sheetNames()`) for identifying if multiple sheets exist prior to downloading.

Another common form of file storage is using zip files. For instance, the Bureau of Labor Statistics[95] (BLS) stores their public-use microdata[96] for the Consumer Expenditure Survey[97] in .zip files. We can use `download.file()` to download the file to your working directory and then work with this data as desired.

```
url <- "http://www.bls.gov/cex/pumd/data/comma/diary14.zip"

# download .zip file and unzip contents
download.file(url, dest="dataset.zip", mode="wb")
unzip ("dataset.zip", exdir = "./")

# assess the files contained in the .zip file which
# unzips as a folder named "diary14"
list.files("diary14")
##  [1] "dtbd141.csv" "dtbd142.csv" "dtbd143.csv" "dtbd144.csv" "dtid141.csv"
##  [6] "dtid142.csv" "dtid143.csv" "dtid144.csv" "expd141.csv" "expd142.csv"
## [11] "expd143.csv" "expd144.csv" "fmld141.csv" "fmld142.csv" "fmld143.csv"
## [16] "fmld144.csv" "memd141.csv" "memd142.csv" "memd143.csv" "memd144.csv"

# alternatively, if we know the file we want prior to unzipping
# we can extract the file without unzipping using unz():
zip_data <- read.csv(unz("dataset.zip", "diary14/expd141.csv"))
zip_data[1:5, 1:10]
##     NEWID ALLOC COST GIFT PUB_FLAG    UCC EXPNSQDY EXPN_QDY EXPNWKDY   EXPN_K\
DY
## 1 2825371     0 6.26    2        2 190112        1        D        3        D
## 2 2825371     0 1.20    2        2 190322        1        D        3        D
## 3 2825381     0 0.98    2        2  20510        3        D        2        D
## 4 2825381     0 0.98    2        2  20510        3        D        2        D
## 5 2825381     0 2.50    2        2  20510        3        D        2        D
```

The .zip archive file format is meant to compress files and are typically used on files of significant size. For instance, the Consumer Expenditure Survey data we downloaded in the previous example is over 10MB. Obviously there may be times in which we want to get specific data in the .zip file to analyze but not always permanently store the entire .zip file contents. In these instances we can

[95] http://www.bls.gov/home.htm
[96] http://www.bls.gov/cex/pumdhome.htm
[97] http://www.bls.gov/cex/home.htm

use the following process[98] proposed by Dirk Eddelbuettel[99] to temporarily download the .zip file, extract the desired data, and then discard the .zip file.

```
# Create a temp. file name
temp <- tempfile()

# Use download.file() to fetch the file into the temp. file
download.file("http://www.bls.gov/cex/pumd/data/comma/diary14.zip",temp)

# Use unz() to extract the target file from temp. file
zip_data2 <- read.csv(unz(temp, "diary14/expd141.csv"))

# Remove the temp file via unlink()
unlink(temp)

zip_data2[1:5, 1:10]
##     NEWID ALLOC COST GIFT PUB_FLAG    UCC EXPNSQDY EXPN_QDY EXPNWKDY   EXPN_K\
DY
## 1 2825371     0 6.26    2        2 190112        1        D        3        D
## 2 2825371     0 1.20    2        2 190322        1        D        3        D
## 3 2825381     0 0.98    2        2  20510        3        D        2        D
## 4 2825381     0 0.98    2        2  20510        3        D        2        D
## 5 2825381     0 2.50    2        2  20510        3        D        2        D
```

One last common scenario I'll cover when importing spreadsheet data from online is when we identify multiple data sets that we'd like to download but are not centrally stored in a .zip format or the like. As a simple example lets look at the average consumer price data[100] from the BLS. The BLS holds multiple data sets for different types of commodities within one url[101]; however, there are separate links for each individual data set. More complicated cases of this will have the links to tabular data sets scattered throughout a webpage[102]. The `XML`[103] package provides the useful `getHTMLLinks()` function to identify these links.

[98] http://stackoverflow.com/questions/3053833/using-r-to-download-zipped-data-file-extract-and-import-data

[99] https://twitter.com/eddelbuettel

[100] http://www.bls.gov/data/#prices

[101] http://download.bls.gov/pub/time.series/ap/

[102] An example is provided in Automated Data Collection with R in which they use a similar approach to extract desired CSV files scattered throughout the Maryland State Board of Elections websiteMaryland State Board of Elections website.

[103] https://cran.r-project.org/web/packages/XML/index.html

```
library(XML)

# url hosting multiple links to data sets
url <- "http://download.bls.gov/pub/time.series/ap/"

# identify the links available
links <- getHTMLLinks(url)

links
##  [1] "/pub/time.series/"
##  [2] "/pub/time.series/ap/ap.area"
##  [3] "/pub/time.series/ap/ap.contacts"
##  [4] "/pub/time.series/ap/ap.data.0.Current"
##  [5] "/pub/time.series/ap/ap.data.1.HouseholdFuels"
##  [6] "/pub/time.series/ap/ap.data.2.Gasoline"
##  [7] "/pub/time.series/ap/ap.data.3.Food"
##  [8] "/pub/time.series/ap/ap.footnote"
##  [9] "/pub/time.series/ap/ap.item"
## [10] "/pub/time.series/ap/ap.period"
## [11] "/pub/time.series/ap/ap.series"
## [12] "/pub/time.series/ap/ap.txt"
```

This allows us to assess which files exist that may be of interest. In this case the links that we are primarily interested in are the ones that contain "data" in their name (links 4-7 listed above). We can use the `stringr`[104] package to extract these desired links which we will use to download the data.

```
library(stringr)

# extract names for desired links and paste to url
links_data <- links[str_detect(links, "data")]

# paste url to data links to have full url for data sets
# use str_sub and regexpr to paste links at appropriate
# starting point
filenames <- paste0(url, str_sub(links_data,
                    start = regexpr("ap.data", links_data)))

filenames
## [1] "http://download.bls.gov/pub/time.series/ap/ap.data.0.Current"
## [2] "http://download.bls.gov/pub/time.series/ap/ap.data.1.HouseholdFuels"
```

[104] https://cran.r-project.org/web/packages/stringr/index.html

```
## [3] "http://download.bls.gov/pub/time.series/ap/ap.data.2.Gasoline"
## [4] "http://download.bls.gov/pub/time.series/ap/ap.data.3.Food"
```

We can now proceed to develop a simple `for` loop function (which you will learn about in the loop control statements chapter) to download each data set. We store the results in a list which contains 4 items, one item for each data set. Each list item contains the url in which the data was extracted from and the dataframe containing the downloaded data. We're now ready to analyze these data sets as necessary.

```
# create empty list to dump data into
data_ls <- list()

for(i in 1:length(filenames)){
        url <- filenames[i]
        data <- read.delim(url)
        data_ls[[length(data_ls) + 1]] <- list(url = filenames[i], data = data)
}

str(data_ls)
## List of 4
##  $ :List of 2
##   ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.0.Current"
##   ..$ data:'data.frame':        144712 obs. of  5 variables:
##   .. ..$ series_id     : Factor w/ 878 levels "APU0000701111    ",..: 1 1 ...
##   .. ..$ year          : int [1:144712] 1995 1995 1995 1995 1995 1995 ...
##   .. ..$ period        : Factor w/ 12 levels "M01","M02","M03",..: 1 2 3 4 ...
##   .. ..$ value         : num [1:144712] 0.238 0.242 0.242 0.236 0.244 ...
##   .. ..$ footnote_codes: logi [1:144712] NA NA NA NA NA NA ...
##  $ :List of 2
##   ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.1.Hou..."
##   ..$ data:'data.frame':        90339 obs. of  5 variables:
##   .. ..$ series_id     : Factor w/ 343 levels "APU000072511     ",..: 1 1 ...
##   .. ..$ year          : int [1:90339] 1978 1978 1979 1979 1979 1979 1979 ...
##   .. ..$ period        : Factor w/ 12 levels "M01","M02","M03",..: 11 12 ...
##   .. ..$ value         : num [1:90339] 0.533 0.545 0.555 0.577 0.605 0.627 ...
##   .. ..$ footnote_codes: logi [1:90339] NA NA NA NA NA NA ...
##  $ :List of 2
##   ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.2.Gas..."
##   ..$ data:'data.frame':        69357 obs. of  5 variables:
##   .. ..$ series_id     : Factor w/ 341 levels "APU000074712     ",..: 1 1 ...
##   .. ..$ year          : int [1:69357] 1973 1973 1973 1974 1974 1974 1974 ...
##   .. ..$ period        : Factor w/ 12 levels "M01","M02","M03",..: 10 11 ...
```

```
##   .. ..$ value         : num [1:69357] 0.402 0.418 0.437 0.465 0.491 0.528 ...
##   .. ..$ footnote_codes: logi [1:69357] NA NA NA NA NA NA ...
##  $ :List of 2
##   ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.3.Food"
##   ..$ data:'data.frame':       122302 obs. of  5 variables:
##   .. ..$ series_id     : Factor w/ 648 levels "APU0000701111    ",..: 1 1 ...
##   .. ..$ year          : int [1:122302] 1980 1980 1980 1980 1980 1980 1980 ...
##   .. ..$ period        : Factor w/ 12 levels "M01","M02","M03",..: 1 2 3 4 ...
##   .. ..$ value         : num [1:122302] 0.203 0.205 0.211 0.206 0.207 0.21 ...
##   .. ..$ footnote_codes: logi [1:122302] NA NA NA NA NA NA ...
```

These examples provide the basics required for downloading most tabular and Excel files from online. However, this is just the beginning of importing/scraping data from the web. Next, we'll start exploring the more conventional forms scraping text and data stored in HTML webpages.

Scraping HTML text

Vast amount of information exists across the interminable webpages that exist online. Much of this information are "unstructured" text that may be useful in our analyses. This section covers the basics of scraping these texts from online sources. Throughout this section I will illustrate how to extract different text components of webpages by dissecting the Wikipedia page on web scraping[105]. However, its important to first cover one of the basic components of HTML elements as we will leverage this information to pull desired information. I offer only enough insight required to begin scraping; I highly recommend *XML and Web Technologies for Data Sciences with R*[106] and *Automated Data Collection with R*[107] to learn more about HTML and XML element structures.

HTML elements are written with a start tag, an end tag, and with the content in between: `<tagname>content</tagname>`. The tags which typically contain the textual content we wish to scrape, and the tags we will leverage in the next two sections, include:

- `<h1>`, `<h2>`,…,`<h6>`: Largest heading, second largest heading, etc.
- `<p>`: Paragraph elements
- `<ul>`: Unordered bulleted list
- `<ol>`: Ordered list
- `<li>`: Individual List item
- `<div>`: Division or section
- `<table>`: Table

For example, text in paragraph form that you see online is wrapped with the HTML paragraph tag `<p>` as in:

[105]https://en.wikipedia.org/wiki/Web_scraping

[106]http://www.amazon.com/XML-Web-Technologies-Data-Sciences/dp/1461478995

[107]http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX

```
<p>
This paragraph represents
a typical text paragraph
in HTML form
</p>
```

It is through these tags that we can start to extract textual components (also referred to as nodes) of HTML webpages.

Scraping HTML Nodes

To scrape online text we'll make use of the relatively newer `rvest`[108] package. `rvest` was created by the RStudio team inspired by libraries such as beautiful soup[109] which has greatly simplified web scraping. `rvest` provides multiple functionalities; however, in this section we will focus only on extracting HTML text with `rvest`. Its important to note that `rvest` makes use of of the pipe operator (`%>%`) developed through the `magrittr` package[110]. If you are not familiar with the functionality of `%>%` I recommend you jump to the chapter on Simplifying Your Code with `%>%` so that you have a better understanding of what's going on with the code.

To extract text from a webpage of interest, we specify what HTML elements we want to select by using `html_nodes()`. For instance, if we want to scrape the primary heading for the Web Scraping Wikipedia webpage[111] we simply identify the `<h1>` node as the node we want to select. `html_nodes()` will identify all `<h1>` nodes on the webpage and return the HTML element. In our example we see there is only one `<h1>` node on this webpage.

```
library(rvest)

scraping_wiki <- read_html("https://en.wikipedia.org/wiki/Web_scraping")

scraping_wiki %>%
        html_nodes("h1")
## {xml_nodeset (1)}
## [1] <h1 id="firstHeading" class="firstHeading" lang="en">Web scraping</h1>
```

To extract only the heading text for this `<h1>` node, and not include all the HTML syntax we use `html_text()` which returns the heading text we see at the top of the Web Scraping Wikipedia page[112].

[108] https://cran.r-project.org/web/packages/rvest/index.html
[109] http://www.crummy.com/software/BeautifulSoup/
[110] https://cran.r-project.org/web/packages/magrittr/index.html
[111] https://en.wikipedia.org/wiki/Web_scraping
[112] https://en.wikipedia.org/wiki/Web_scraping

```
scraping_wiki %>%
        html_nodes("h1") %>%
        html_text()
## [1] "Web scraping"
```

If we want to identify all the second level headings on the webpage we follow the same process but instead select the `<h2>` nodes. In this example we see there are 10 second level headings on the Web Scraping Wikipedia page[113].

```
scraping_wiki %>%
        html_nodes("h2") %>%
        html_text()
##  [1] "Contents"
##  [2] "Techniques[edit]"
##  [3] "Legal issues[edit]"
##  [4] "Notable tools[edit]"
##  [5] "See also[edit]"
##  [6] "Technical measures to stop bots[edit]"
##  [7] "Articles[edit]"
##  [8] "References[edit]"
##  [9] "See also[edit]"
## [10] "Navigation menu"
```

Next, we can move on to extracting much of the text on this webpage which is in paragraph form. We can follow the same process illustrated above but instead we'll select all `<p>` nodes. This selects the 17 paragraph elements from the web page; which we can examine by subsetting the list `p_nodes` to see the first line of each paragraph along with the HTML syntax. Just as before, to extract the text from these nodes and coerce them to a character string we simply apply `html_text()`.

```
p_nodes <- scraping_wiki %>%
        html_nodes("p")

length(p_nodes)
## [1] 17

p_nodes[1:6]
## {xml_nodeset (6)}
## [1] <p><b>Web scraping</b> (<b>web harvesting</b> or <b>web data extract ...
## [2] <p>Web scraping is closely related to <a href="/wiki/Web_indexing" t ...
## [3] <p/>
```

[113]https://en.wikipedia.org/wiki/Web_scraping

```
## [4] <p/>
## [5] <p>Web scraping is the process of automatically collecting informati ...
## [6] <p>Web scraping may be against the <a href="/wiki/Terms_of_use" titl ...
```

```
p_text <- scraping_wiki %>%
        html_nodes("p") %>%
        html_text()

p_text[1]
## [1] "Web scraping (web harvesting or web data extraction) is a computer softw\
are technique of extracting information from websites. Usually, such software pr\
ograms simulate human exploration of the World Wide Web by either implementing l\
ow-level Hypertext Transfer Protocol (HTTP), or embedding a fully-fledged web br\
owser, such as Mozilla Firefox."
```

Not too bad; however, we may not have captured all the text that we were hoping for. Since we extracted text for all `<p>` nodes, we collected all identified paragraph text; however, this does not capture the text in the bulleted lists. For example, when you look at the Web Scraping Wikipedia page[114] you will notice a significant amount of text in bulleted list format following the third paragraph under the **Techniques**[115] heading. If we look at our data we'll see that that the text in this list format are not capture between the two paragraphs:

```
p_text[5]
## [1] "Web scraping is the process of automatically collecting information from\
 the World Wide Web. It is a field with active developments sharing a common goa\
l with the semantic web vision, an ambitious initiative that still requires brea\
kthroughs in text processing, semantic understanding, artificial intelligence an\
d human-computer interactions. Current web scraping solutions range from the ad-\
hoc, requiring human effort, to fully automated systems that are able to convert\
 entire web sites into structured information, with limitations."

p_text[6]
## [1] "Web scraping may be against the terms of use of some websites. The enfor\
ceability of these terms is unclear.[4] While outright duplication of original e\
xpression will in many cases be illegal, in the United States the courts ruled i\
n Feist Publications v. Rural Telephone Service that duplication of facts is all\
owable. U.S. courts have acknowledged that users of \"scrapers\" or \"robots\" m\
ay be held liable for committing trespass to chattels,[5][6] which involves a co\
mputer system itself being considered personal property upon which the user of a\
```

[114] https://en.wikipedia.org/wiki/Web_scraping
[115] https://en.wikipedia.org/wiki/Web_scraping#Techniques

```
 scraper is trespassing. The best known of these cases, eBay v. Bidder's Edge, r\
esulted in an injunction ordering Bidder's Edge to stop accessing, collecting, a\
nd indexing auctions from the eBay web site. This case involved automatic placin\
g of bids, known as auction sniping. However, in order to succeed on a claim of \
trespass to chattels, the plaintiff must demonstrate that the defendant intentio\
nally and without authorization interfered with the plaintiff's possessory inter\
est in the computer system and that the defendant's unauthorized use caused dama\
ge to the plaintiff. Not all cases of web spidering brought before the courts ha\
ve been considered trespass to chattels.[7]"
```

This is because the text in this list format are contained in `<ul>` nodes. To capture the text in lists, we can use the same steps as above but we select specific nodes which represent HTML lists components. We can approach extracting list text two ways.

First, we can pull all list elements (`<ul>`). When scraping all `<ul>` text, the resulting data structure will be a character string vector with each element representing a single list consisting of all list items in that list. In our running example there are 21 list elements as shown in the example that follows. You can see the first list scraped is the table of contents and the second list scraped is the list in the Techniques[116] section.

```
ul_text <- scraping_wiki %>%
        html_nodes("ul") %>%
        html_text()

length(ul_text)
## [1] 21

ul_text[1]
## [1] "\n1 Techniques\n2 Legal issues\n3 Notable tools\n4 See also\n5 Technical\
 measures to stop bots\n6 Articles\n7 References\n8 See also\n"

# read the first 200 characters of the second list
substr(ul_text[2], start = 1, stop = 200)
## [1] "\nHuman copy-and-paste: Sometimes even the best web-scraping technology \
cannot replace a human's manual examination and copy-and-paste, and sometimes th\
is may be the only workable solution when the web"
```

An alternative approach is to pull all `<li>` nodes. This will pull the text contained in each list item for all the lists. In our running example there's 146 list items that we can extract from this Wikipedia page. The first eight list items are the list of contents we see towards the top of the page. List items

[116] https://en.wikipedia.org/wiki/Web_scraping#Techniques

9-17 are the list elements contained in the "Techniques[117]" section, list items 18-44 are the items listed under the "Notable Tools[118]" section, and so on.

```
li_text <- scraping_wiki %>%
        html_nodes("li") %>%
        html_text()

length(li_text)
## [1] 147

li_text[1:8]
## [1] "1 Techniques"                      "2 Legal issues"
## [3] "3 Notable tools"                   "4 See also"
## [5] "5 Technical measures to stop bots" "6 Articles"
## [7] "7 References"                      "8 See also"
```

At this point we may believe we have all the text desired and proceed with joining the paragraph (`p_text`) and list (`ul_text` or `li_text`) character strings and then perform the desired textual analysis. However, we may now have captured *more* text than we were hoping for. For example, by scraping all lists we are also capturing the listed links in the left margin of the webpage. If we look at the 104-136 list items that we scraped, we'll see that these texts correspond to the left margin text.

```
li_text[104:136]
##  [1] "Main page"          "Contents"            "Featured content"
##  [4] "Current events"     "Random article"      "Donate to Wikipedia"
##  [7] "Wikipedia store"    "Help"                "About Wikipedia"
## [10] "Community portal"   "Recent changes"      "Contact page"
## [13] "What links here"    "Related changes"     "Upload file"
## [16] "Special pages"      "Permanent link"      "Page information"
## [19] "Wikidata item"      "Cite this page"      "Create a book"
## [22] "Download as PDF"    "Printable version"   "Català"
## [25] "Deutsch"            "Español"             "Français"
## [28] "Íslenska"           "Italiano"            "Latviešu"
## [31] "Nederlands"         "□□□"              "Српски / srpski"
```

If we desire to scrape every piece of text on the webpage than this won't be of concern. In fact, if we want to scrape all the text regardless of the content they represent there is an easier approach. We can capture all the content to include text in paragraph (`<p>`), lists (`<ul>`, `<ol>`, and `<li>`), and even data in tables (`<table>`) by using `<div>`. This is because these other elements are usually a subsidiary of an HTML division or section so pulling all `<div>` nodes will extract all text contained in that division or section regardless if it is also contained in a paragraph or list.

117 https://en.wikipedia.org/wiki/Web_scraping#Techniques

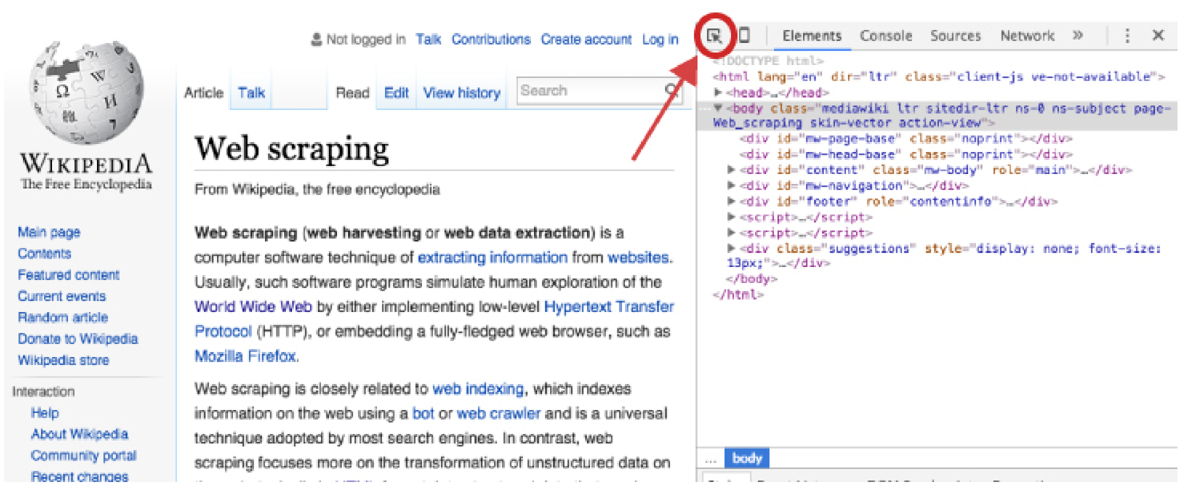
118 https://en.wikipedia.org/wiki/Web_scraping#Notable_tools

```
all_text <- scraping_wiki %>%
        html_nodes("div") %>%
        html_text()
```

Scraping Specific HTML Nodes

However, if we are concerned only with specific content on the webpage then we need to make our HTML node selection process a little more focused. To do this we, we can use our browser's developer tools to examine the webpage we are scraping and get more details on specific nodes of interest. If you are using Chrome or Firefox you can open the developer tools by clicking F12 (Cmd + Opt + I for Mac) or for Safari you would use Command-Option-I. An additional option which is recommended by Hadley Wickham is to use selectorgadget.com[119], a Chrome extension, to help identify the web page elements you need[120].

Once the developer's tools are opened your primary concern is with the element selector. This is located in the top lefthand corner of the developers tools window.

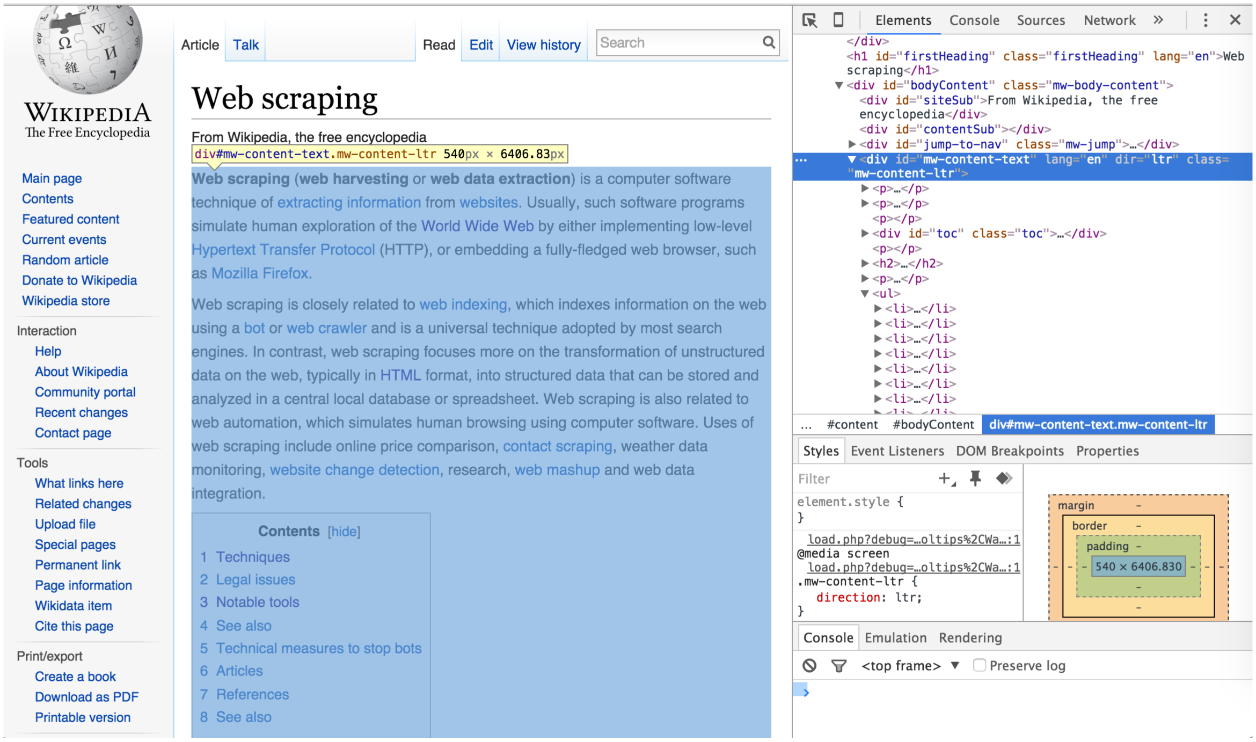


Developer Tools: Element Selector

Once you've selected the element selector you can now scroll over the elements of the webpage which will cause each element you scroll over to be highlighted. Once you've identified the element you want to focus on, select it. This will cause the element to be identified in the developer tools window. For example, if I am only interested in the main body of the Web Scraping content on the Wikipedia page then I would select the element that highlights the entire center component of the webpage. This highlights the corresponding element `<div id="bodyContent" class="mw-body-content">` in the developer tools window as the following illustrates.

[119] http://selectorgadget.com/

[120] You can learn more about selectors at flukeout.github.io



Selecting Content of Interest

I can now use this information to select and scrape all the text from this specific <div> node by calling the ID name ("#mw-content-text") in `html_nodes()`[121]. As you can see below, the text that is scraped begins with the first line in the main body of the Web Scraping content and ends with the text in the See Also[122] section which is the last bit of text directly pertaining to Web Scraping on the webpage. Explicitly, we have pulled the specific text associated with the web content we desire.

```
body_text <- scraping_wiki %>%
        html_nodes("#mw-content-text") %>%
        html_text()

# read the first 207 characters
substr(body_text, start = 1, stop = 207)
## [1] "Web scraping (web harvesting or web data extraction) is a computer softw\
are technique of extracting information from websites. Usually, such software pr\
ograms simulate human exploration of the World Wide Web"

# read the last 73 characters
substr(body_text, start = nchar(body_text)-73, stop = nchar(body_text))
```

[121] You can simply assess the name of the ID in the highlighted element or you can right click the highlighted element in the developer tools window and select *Copy selector*. You can then paste directly into 'html_nodes() as it will paste the exact ID name that you need for that element.

[122] https://en.wikipedia.org/wiki/Web_scraping#See_also_2

```
## [1] "See also[edit]\n\nData scraping\nData wrangling\nKnowledge extraction\n\\
n\n\n\n\n\n\n\n\n"
```

Using the developer tools approach allows us to be as specific as we desire. We can identify the class name for a specific HTML element and scrape the text for only that node rather than all the other elements with similar tags. This allows us to scrape the main body of content as we just illustrated or we can also identify specific headings, paragraphs, lists, and list components if we desire to scrape only these specific pieces of text:

```
# Scraping a specific heading
scraping_wiki %>%
        html_nodes("#Techniques") %>%
        html_text()
## [1] "Techniques"

# Scraping a specific paragraph
scraping_wiki %>%
        html_nodes("#mw-content-text > p:nth-child(20)") %>%
        html_text()
## [1] "In Australia, the Spam Act 2003 outlaws some forms of web harvesting, al\
though this only applies to email addresses.[20][21]"

# Scraping a specific list
scraping_wiki %>%
        html_nodes("#mw-content-text > div:nth-child(22)") %>%
        html_text()
## [1] "\n\nApache Camel\nArchive.is\nAutomation Anywhere\nConvertigo\ncURL\nDat\
a Toolbar\nDiffbot\nFirebug\nGreasemonkey\nHeritrix\nHtmlUnit\nHTTrack\niMacros\\
nImport.io\nJaxer\nNode.js\nnokogiri\nPhantomJS\nScraperWiki\nScrapy\nSelenium\n\
SimpleTest\nwatir\nWget\nWireshark\nWSO2 Mashup Server\nYahoo! Query Language (Y\
QL)\n\n"

# Scraping a specific reference list item
scraping_wiki %>%
        html_nodes("#cite_note-22") %>%
        html_text()
## [1] "^ \"Web Scraping: Everything You Wanted to Know (but were afraid to ask)\
\". Distil Networks. 2015-07-22. Retrieved 2015-11-04. "
```

Cleaning up

With any webscraping activity, especially involving text, there is likely to be some clean up involved. For example, in the previous example we saw that we can specifically pull the list of **Notable Tools**[123]; however, you can see that in between each list item rather than a space there contains one or more `\n` which is used in HTML to specify a new line. We can clean this up quickly with a little character string manipulation.

```
library(magrittr)

scraping_wiki %>%
        html_nodes("#mw-content-text > div:nth-child(22)") %>%
        html_text()
## [1] "\n\nApache Camel\nArchive.is\nAutomation Anywhere\nConvertigo\ncURL\nDat\
a Toolbar\nDiffbot\nFirebug\nGreasemonkey\nHeritrix\nHtmlUnit\nHTTrack\niMacros\\
nImport.io\nJaxer\nNode.js\nnokogiri\nPhantomJS\nScraperWiki\nScrapy\nSelenium\n\
SimpleTest\nwatir\nWget\nWireshark\nWSO2 Mashup Server\nYahoo! Query Language (Y\
QL)\n\n"

scraping_wiki %>%
        html_nodes("#mw-content-text > div:nth-child(22)") %>%
        html_text() %>%
        strsplit(split = "\n") %>%
        unlist() %>%
        .[. != ""]
##  [1] "Apache Camel"                 "Archive.is"
##  [3] "Automation Anywhere"          "Convertigo"
##  [5] "cURL"                         "Data Toolbar"
##  [7] "Diffbot"                      "Firebug"
##  [9] "Greasemonkey"                 "Heritrix"
## [11] "HtmlUnit"                     "HTTrack"
## [13] "iMacros"                      "Import.io"
## [15] "Jaxer"                        "Node.js"
## [17] "nokogiri"                     "PhantomJS"
## [19] "ScraperWiki"                  "Scrapy"
## [21] "Selenium"                     "SimpleTest"
## [23] "watir"                        "Wget"
## [25] "Wireshark"                    "WSO2 Mashup Server"
## [27] "Yahoo! Query Language (YQL)"
```

Similarly, as we saw in our example above with scraping the main body content (`body_text`), there are extra characters (i.e. `\n`, `\`, `^`) in the text that we may not want. Using a little regex we can clean

[123] https://en.wikipedia.org/wiki/Web_scraping#Notable_tools

this up so that our character string consists of only text that we see on the screen and no additional HTML code embedded throughout the text.

```
library(stringr)

# read the last 700 characters
substr(body_text, start = nchar(body_text)-700, stop = nchar(body_text))
## [1] " 2010). \"Intellectual Property: Website Terms of Use\". Issue 26: June \
2010. LK Shields Solicitors Update. p. 03. Retrieved 2012-04-19. \n^ National Of\
fice for the Information Economy (February 2004). \"Spam Act 2003: An overview f\
or business\" (PDF). Australian Communications Authority. p. 6. Retrieved 2009-0\
3-09. \n^ National Office for the Information Economy (February 2004). \"Spam Ac\
t 2003: A practical guide for business\" (PDF). Australian Communications Author\
ity. p. 20. Retrieved 2009-03-09. \n^ \"Web Scraping: Everything You Wanted to K\
now (but were afraid to ask)\". Distil Networks. 2015-07-22. Retrieved 2015-11-0\
4. \n\n\nSee also[edit]\n\nData scraping\nData wrangling\nKnowledge extraction\n\
\n\n\n\n\n\n\n\n"

# clean up text
body_text %>%
        str_replace_all(pattern = "\n", replacement = " ") %>%
        str_replace_all(pattern = "[\\^]", replacement = " ") %>%
        str_replace_all(pattern = "\"", replacement = " ") %>%
        str_replace_all(pattern = "\\s+", replacement = " ") %>%
        str_trim(side = "both") %>%
        substr(start = nchar(body_text)-700, stop = nchar(body_text))
## [1] "012-04-19. National Office for the Information Economy (February 2004). \
Spam Act 2003: An overview for business (PDF). Australian Communications Authori\
ty. p. 6. Retrieved 2009-03-09. National Office for the Information Economy (Feb\
ruary 2004). Spam Act 2003: A practical guide for business (PDF). Australian Com\
munications Authority. p. 20. Retrieved 2009-03-09. Web Scraping: Everything You\
 Wanted to Know (but were afraid to ask) . Distil Networks. 2015-07-22. Retrieve\
d 2015-11-04. See also[edit] Data scraping Data wrangling Knowledge extraction"
```

So there we have it, text scraping in a nutshell. Although not all encompassing, this section covered the basics of scraping text from HTML documents. Whether you want to scrape text from all common text-containing nodes such as `<div>`, `<p>`, `<ul>` and the like or you want to scrape from a specific node using the specific ID, this section provides you the basic fundamentals of using `rvest` to scrape the text you need. In the next section we move on to scraping data from HTML tables.

Scraping HTML table data

Another common structure of information storage on the Web is in the form of HTML tables. This section reiterates some of the information from the previous section; however, we focus solely on scraping data from HTML tables. The simplest approach to scraping HTML table data directly into R is by using either the `rvest` package or the `XML` package. To illustrate, I will focus on the BLS employment statistics webpage[124] which contains multiple HTML tables from which we can scrape data.

Scraping HTML tables with rvest

Recall that HTML elements are written with a start tag, an end tag, and with the content in between: `<tagname>content</tagname>`. HTML tables are contained within `<table>` tags; therefore, to extract the tables from the BLS employment statistics webpage we first use the `html_nodes()` function to select the `<table>` nodes. In this case we are interested in all table nodes that exist on the webpage. In this example, `html_nodes` captures 15 HTML tables. This includes data from the 10 data tables seen on the webpage but also includes data from a few additional tables used to format parts of the page (i.e. table of contents, table of figures, advertisements).

```
library(rvest)

webpage <- read_html("http://www.bls.gov/web/empsit/cesbmart.htm")

tbls <- html_nodes(webpage, "table")

head(tbls)
## {xml_nodeset (6)}
## [1] <table id="main-content-table">&#13;\n\t<tr>&#13;\n\t\t<td id="secon ...
## [2] <table id="Table1" class="regular" cellspacing="0" cellpadding="0" x ...
## [3] <table id="Table2" class="regular" cellspacing="0" cellpadding="0" x ...
## [4] <table id="Table3" class="regular" cellspacing="0" cellpadding="0" x ...
## [5] <table id="Table4" class="regular" cellspacing="0" cellpadding="0" x ...
## [6] <table id="Exhibit1" class="regular" cellspacing="0" cellpadding="0" ...
```

Remember that `html_nodes()` does not parse the data; rather, it acts as a CSS selector. To parse the HTML table data we use `html_table()`, which would create a list containing 15 data frames. However, rarely do we need to scrape *every* HTML table from a page, especially since some HTML tables don't catch any information we are likely interested in (i.e. table of contents, table of figures, footers).

More often than not we want to parse specific tables. Lets assume we want to parse the second and third tables on the webpage:

[124] http://www.bls.gov/web/empsit/cesbmart.htm

- Table 2. Nonfarm employment benchmarks by industry, March 2014 (in thousands) and
- Table 3. Net birth/death estimates by industry supersector, April – December 2014 (in thousands)

This can be accomplished two ways. First, we can assess the previous `tbls` list and try to identify the table(s) of interest. In this example it appears that `tbls` list items 3 and 4 correspond with Table 2 and Table 3, respectively. We can then subset the list of table nodes prior to parsing the data with `html_table()`. This results in a list of two data frames containing the data of interest.

```
# subset list of table nodes for items 3 & 4
tbls_ls <- webpage %>%
        html_nodes("table") %>%
        .[3:4] %>%
        html_table(fill = TRUE)

str(tbls_ls)
## List of 2
##  $ :'data.frame':       147 obs. of  6 variables:
##   ..$ CES Industry Code : chr [1:147] "Amount" "00-000000" "05-000000" ...
##   ..$ CES Industry Title: chr [1:147] "Percent" "Total nonfarm" ...
##   ..$ Benchmark         : chr [1:147] NA "137,214" "114,989" "18,675" ...
##   ..$ Estimate          : chr [1:147] NA "137,147" "114,884" "18,558" ...
##   ..$ Differences       : num [1:147] NA 67 105 117 -50 -12 -16 -2.8 ...
##   ..$ NA                : chr [1:147] NA "(1)" "0.1" "0.6" ...
##  $ :'data.frame':       11 obs. of  12 variables:
##   ..$ CES Industry Code : chr [1:11] "10-000000" "20-000000" "30-000000" ...
##   ..$ CES Industry Title: chr [1:11] "Mining and logging" "Construction" ...
##   ..$ Apr               : int [1:11] 2 35 0 21 0 8 81 22 82 12 ...
##   ..$ May               : int [1:11] 2 37 6 24 5 8 22 13 81 6 ...
##   ..$ Jun               : int [1:11] 2 24 4 12 0 4 5 -14 86 6 ...
##   ..$ Jul               : int [1:11] 2 12 -3 7 -1 3 35 7 62 -2 ...
##   ..$ Aug               : int [1:11] 1 12 4 14 3 4 19 21 23 3 ...
##   ..$ Sep               : int [1:11] 1 7 1 9 -1 -1 -12 12 -33 -2 ...
##   ..$ Oct               : int [1:11] 1 12 3 28 6 16 76 35 -17 4 ...
##   ..$ Nov               : int [1:11] 1 -10 2 10 3 3 14 14 -22 1 ...
##   ..$ Dec               : int [1:11] 0 -21 0 4 0 10 -10 -3 4 1 ...
##   ..$ CumulativeTotal   : int [1:11] 12 108 17 129 15 55 230 107 266 29 ...
```

An alternative approach, which is more explicit, is to use the element selector process described in the previous section to call the table ID name.

```
# empty list to add table data to
tbls2_ls <- list()

# scrape Table 2. Nonfarm employment...
tbls2_ls$Table1 <- webpage %>%
        html_nodes("#Table2") %>%
        html_table(fill = TRUE) %>%
        .[[1]]

# Table 3. Net birth/death...
tbls2_ls$Table2 <- webpage %>%
        html_nodes("#Table3") %>%
        html_table() %>%
        .[[1]]

str(tbls2_ls)
## List of 2
##  $ Table1:'data.frame':        147 obs. of  6 variables:
##   ..$ CES Industry Code : chr [1:147] "Amount" "00-000000" "05-000000" ...
##   ..$ CES Industry Title: chr [1:147] "Percent" "Total nonfarm" ...
##   ..$ Benchmark         : chr [1:147] NA "137,214" "114,989" "18,675" ...
##   ..$ Estimate          : chr [1:147] NA "137,147" "114,884" "18,558" ...
##   ..$ Differences       : num [1:147] NA 67 105 117 -50 -12 -16 -2.8 ...
##   ..$ NA                : chr [1:147] NA "(1)" "0.1" "0.6" ...
##  $ Table2:'data.frame':        11 obs. of  12 variables:
##   ..$ CES Industry Code : chr [1:11] "10-000000" "20-000000" "30-000000" ...
##   ..$ CES Industry Title: chr [1:11] "Mining and logging" "Construction" ...
##   ..$ Apr               : int [1:11] 2 35 0 21 0 8 81 22 82 12 ...
##   ..$ May               : int [1:11] 2 37 6 24 5 8 22 13 81 6 ...
##   ..$ Jun               : int [1:11] 2 24 4 12 0 4 5 -14 86 6 ...
##   ..$ Jul               : int [1:11] 2 12 -3 7 -1 3 35 7 62 -2 ...
##   ..$ Aug               : int [1:11] 1 12 4 14 3 4 19 21 23 3 ...
##   ..$ Sep               : int [1:11] 1 7 1 9 -1 -1 -12 12 -33 -2 ...
##   ..$ Oct               : int [1:11] 1 12 3 28 6 16 76 35 -17 4 ...
##   ..$ Nov               : int [1:11] 1 -10 2 10 3 3 14 14 -22 1 ...
##   ..$ Dec               : int [1:11] 0 -21 0 4 0 10 -10 -3 4 1 ...
##   ..$ CumulativeTotal   : int [1:11] 12 108 17 129 15 55 230 107 266 29 ...
```

One issue to note is when using `rvest`'s `html_table()` to read a table with split column headings as in *Table 2. Nonfarm employment…*. `html_table` will cause split headings to be included and can cause the first row to include parts of the headings. We can see this with Table 2. This requires a little clean up.

```
head(tbls2_ls[[1]], 4)
##   CES Industry Code CES Industry Title Benchmark Estimate Differences   NA
## 1            Amount            Percent      <NA>     <NA>          NA <NA>
## 2        00-000000      Total nonfarm   137,214  137,147          67  (1)
## 3        05-000000      Total private   114,989  114,884         105  0.1
## 4        06-000000    Goods-producing    18,675   18,558         117  0.6

# remove row 1 that includes part of the headings
tbls2_ls[[1]] <- tbls2_ls[[1]][-1,]

# rename table headings
colnames(tbls2_ls[[1]]) <- c("CES_Code", "Ind_Title", "Benchmark",
                             "Estimate", "Amt_Diff", "Pct_Diff")

head(tbls2_ls[[1]], 4)
##    CES_Code         Ind_Title Benchmark Estimate Amt_Diff Pct_Diff
## 2 00-000000     Total nonfarm   137,214  137,147       67      (1)
## 3 05-000000     Total private   114,989  114,884      105      0.1
## 4 06-000000   Goods-producing    18,675   18,558      117      0.6
## 5 07-000000 Service-providing   118,539  118,589      -50      (1)
```

Scraping HTML tables with XML

An alternative to `rvest` for table scraping is to use the `XML`[125] package. The XML package provides a convenient `readHTMLTable()` function to extract data from HTML tables in HTML documents. By passing the URL to `readHTMLTable()`, the data in each table is read and stored as a data frame. In a situation like our running example where multiple tables exists, the data frames will be stored in a list similar to `rvest`'s `html_table`.

```
library(XML)

url <- "http://www.bls.gov/web/empsit/cesbmart.htm"

# read in HTML data
tbls_xml <- readHTMLTable(url)

typeof(tbls_xml)
## [1] "list"

length(tbls_xml)
## [1] 15
```

[125] https://cran.r-project.org/web/packages/XML/index.html

You can see that `tbls_xml` captures the same 15 `<table>` nodes that `html_nodes` captured. To capture the same tables of interest we previously discussed (*Table 2. Nonfarm employment…* and *Table 3. Net birth/death…*) we can use a couple approaches. First, we can assess `str(tbls_xml)` to identify the tables of interest and perform normal list subsetting. In our example list items 3 and 4 correspond with our tables of interest.

```
head(tbls_xml[[3]])
##          V1                         V2      V3      V4  V5   V6
## 1 00-000000              Total nonfarm 137,214 137,147  67  (1)
## 2 05-000000              Total private 114,989 114,884 105  0.1
## 3 06-000000            Goods-producing  18,675  18,558 117  0.6
## 4 07-000000          Service-providing 118,539 118,589 -50  (1)
## 5 08-000000 Private service-providing  96,314  96,326 -12  (1)
## 6 10-000000          Mining and logging     868     884 -16 -1.8

head(tbls_xml[[4]], 3)
##   CES Industry Code CES Industry Title Apr May Jun Jul Aug Sep Oct Nov Dec
## 1         10-000000 Mining and logging   2   2   2   2   1   1   1   1   0
## 2         20-000000       Construction  35  37  24  12  12   7  12 -10 -21
## 3         30-000000      Manufacturing   0   6   4  -3   4   1   3   2   0
##   CumulativeTotal
## 1              12
## 2             108
## 3              17
```

Second, we can use the `which` argument in `readHTMLTable()` which restricts the data importing to only those tables specified numerically.

```
# only parse the 3rd and 4th tables
emp_ls <- readHTMLTable(url, which = c(3, 4))

str(emp_ls)
## List of 2
##  $ Table2:'data.frame':        145 obs. of  6 variables:
##   ..$ V1: Factor w/ 145 levels "00-000000","05-000000",..: 1 2 3 4 5 6 7 8 ...
##   ..$ V2: Factor w/ 143 levels "Accommodation",..: 130 131 52 116 102 74 ...
##   ..$ V3: Factor w/ 145 levels "1,010.3","1,048.3",..: 40 35 48 37 145 140 ...
##   ..$ V4: Factor w/ 145 levels "1,008.4","1,052.3",..: 41 34 48 36 144 142 ...
##   ..$ V5: Factor w/ 123 levels "-0.3","-0.4",..: 113 68 71 48 9 19 29 11 ...
##   ..$ V6: Factor w/ 56 levels "-0.1","-0.2",..: 30 31 36 30 30 16 28 14 29 ...
##  $ Table3:'data.frame':        11 obs. of  12 variables:
##   ..$ CES Industry Code : Factor w/ 11 levels "10-000000","20-000000",..:1 ...
```

```
##    ..$ CES Industry Title: Factor w/ 11 levels "263","Construction",..: 8 2 ...
##    ..$ Apr               : Factor w/ 10 levels "0","12","2","204",..: 3 7 1 ...
##    ..$ May               : Factor w/ 10 levels "129","13","2",..: 3 6 8 5 7 ...
##    ..$ Jun               : Factor w/ 10 levels "-14","0","12",..: 5 6 7 3 2 ...
##    ..$ Jul               : Factor w/ 10 levels "-1","-2","-3",..: 6 5 3 10 ...
##    ..$ Aug               : Factor w/ 9 levels "-19","1","12",..: 2 3 9 4 8 ...
##    ..$ Sep               : Factor w/ 9 levels "-1","-12","-2",..: 5 8 5 9 1 ...
##    ..$ Oct               : Factor w/ 10 levels "-17","1","12",..: 2 3 6 5 9 ...
##    ..$ Nov               : Factor w/ 8 levels "-10","-15","-22",..: 4 1 7 5 ...
##    ..$ Dec               : Factor w/ 8 levels "-10","-21","-3",..: 4 2 4 7  ...
##    ..$ CumulativeTotal   : Factor w/ 10 levels "107","108","12",..: 3 2 6 4 ...
```

The third option involves explicitly naming the tables to parse. This process uses the element selector process described in the previous section to call the table by name. We use `getNodeSet()` to select the specified tables of interest. However, a key difference here is rather than copying the table ID names you want to copy the XPath. You can do this with the following: After you've highlighted the table element of interest with the element selector, right click the highlighted element in the developer tools window and select Copy XPath. From here we just use `readHTMLTable()` to convert to data frames and we have our desired tables.

```
library(RCurl)

# parse url
url_parsed <- htmlParse(getURL(url), asText = TRUE)

# select table nodes of interest
tableNodes <- getNodeSet(url_parsed, c('//*[@id="Table2"]', '//*[@id="Table3"]'))

# convert HTML tables to data frames
bls_table2 <- readHTMLTable(tableNodes[[1]])
bls_table3 <- readHTMLTable(tableNodes[[2]])

head(bls_table2)
##          V1                         V2      V3      V4  V5   V6
## 1 00-000000              Total nonfarm 137,214 137,147  67  (1)
## 2 05-000000              Total private 114,989 114,884 105  0.1
## 3 06-000000            Goods-producing  18,675  18,558 117  0.6
## 4 07-000000          Service-providing 118,539 118,589 -50  (1)
## 5 08-000000 Private service-providing  96,314  96,326 -12  (1)
## 6 10-000000          Mining and logging     868     884 -16 -1.8

head(bls_table3, 3)
```

```
##   CES Industry Code CES Industry Title Apr May Jun Jul Aug Sep Oct Nov Dec
## 1         10-000000 Mining and logging   2   2   2   2   1   1   1   1   0
## 2         20-000000       Construction  35  37  24  12  12   7  12 -10 -21
## 3         30-000000      Manufacturing   0   6   4  -3   4   1   3   2   0
##   CumulativeTotal
## 1              12
## 2             108
## 3              17
```

A few benefits of `XML`'s `readHTMLTable` that are routinely handy include:

- We can specify names for the column headings
- We can specify the classes for each column
- We can specify rows to skip

For instance, if you look at `bls_table2` above notice that because of the split column headings on *Table 2. Nonfarm employment…* `readHTMLTable` stripped and replaced the headings with generic names because R does not know which variable names should align with each column. We can correct for this with the following:

```
bls_table2 <- readHTMLTable(tableNodes[[1]],
                            header = c("CES_Code", "Ind_Title", "Benchmark",
                            "Estimate", "Amt_Diff", "Pct_Diff"))

head(bls_table2)
##    CES_Code                 Ind_Title Benchmark Estimate Amt_Diff Pct_Diff
## 1 00-000000             Total nonfarm   137,214  137,147       67      (1)
## 2 05-000000             Total private   114,989  114,884      105      0.1
## 3 06-000000           Goods-producing    18,675   18,558      117      0.6
## 4 07-000000         Service-providing   118,539  118,589      -50      (1)
## 5 08-000000 Private service-providing    96,314   96,326      -12      (1)
## 6 10-000000        Mining and logging       868      884      -16     -1.8
```

Also, for `bls_table3` note that the net birth/death values parsed have been converted to factor levels. We can use the `colClasses` argument to correct this.

```
str(bls_table3)
## 'data.frame':        11 obs. of  12 variables:
##  $ CES Industry Code : Factor w/ 11 levels "10-000000","20-000000",..: 1 2 ...
##  $ CES Industry Title: Factor w/ 11 levels "263","Construction",..: 8 2 7 ...
##  $ Apr               : Factor w/ 10 levels "0","12","2","204",..: 3 7 1 5 ...
##  $ May               : Factor w/ 10 levels "129","13","2",..: 3 6 8 5 7 9 ...
##  $ Jun               : Factor w/ 10 levels "-14","0","12",..: 5 6 7 3 2 7 ...
##  $ Jul               : Factor w/ 10 levels "-1","-2","-3",..: 6 5 3 10 1 7 ...
##  $ Aug               : Factor w/ 9 levels "-19","1","12",..: 2 3 9 4 8 9 5 ...
##  $ Sep               : Factor w/ 9 levels "-1","-12","-2",..: 5 8 5 9 1 1 ...
##  $ Oct               : Factor w/ 10 levels "-17","1","12",..: 2 3 6 5 9 4 ...
##  $ Nov               : Factor w/ 8 levels "-10","-15","-22",..: 4 1 7 5 8 ...
##  $ Dec               : Factor w/ 8 levels "-10","-21","-3",..: 4 2 4 7 4 6 ...
##  $ CumulativeTotal   : Factor w/ 10 levels "107","108","12",..: 3 2 6 4 5 ...

bls_table3 <- readHTMLTable(tableNodes[[2]],
                            colClasses = c("character","character",
                                           rep("integer", 10)))

str(bls_table3)
## 'data.frame':        11 obs. of  12 variables:
##  $ CES Industry Code : Factor w/ 11 levels "10-000000","20-000000",..: 1 2 ...
##  $ CES Industry Title: Factor w/ 11 levels "263","Construction",..: 8 2 7 ...
##  $ Apr               : int  2 35 0 21 0 8 81 22 82 12 ...
##  $ May               : int  2 37 6 24 5 8 22 13 81 6 ...
##  $ Jun               : int  2 24 4 12 0 4 5 -14 86 6 ...
##  $ Jul               : int  2 12 -3 7 -1 3 35 7 62 -2 ...
##  $ Aug               : int  1 12 4 14 3 4 19 21 23 3 ...
##  $ Sep               : int  1 7 1 9 -1 -1 -12 12 -33 -2 ...
##  $ Oct               : int  1 12 3 28 6 16 76 35 -17 4 ...
##  $ Nov               : int  1 -10 2 10 3 3 14 14 -22 1 ...
##  $ Dec               : int  0 -21 0 4 0 10 -10 -3 4 1 ...
##  $ CumulativeTotal   : int  12 108 17 129 15 55 230 107 266 29 ...
```

Between `rvest` and `XML`, scraping HTML tables is relatively easy once you get fluent with the syntax and the available options. This section covers just the basics of both these packages to get you moving forward with scraping tables. In the next section we move on to working with application program interfaces (APIs) to get data from the web.

Working with APIs

An application-programming interface (API) in a nutshell is a method of communication between software programs. APIs allow programs to interact and use each other's functions by acting as a middle man. Why is this useful? Lets say you want to pull weather data from the NOAA[126]. You have a few options:

- You could query the data and download the spreadsheet or manually cut-n-paste the desired data and then import into R. Doesn't get you any coolness points.
- You could use some webscraping techniques previously covered to parse the desired data. Golf clap. The downfall of this strategy is if NOAA changes their website structure down the road your code will need to be adjusted.
- Or, you can use the `rnoaa`[127] package which allows you to send specific instructions to the NOAA API via R, the API will then perform the action requested and return the desired information. The benefit of this strategy is if the NOAA changes its website structure it won't impact the API data retreival structure which means no impact to your code. Standing ovation!

Consequently, APIs provide consistency in data retrieval processes which can be essential for recurring analyses. Luckily, the use of APIs by organizations that collect data are growing exponentially[128]. This is great for you and I as more and more data continues to be at our finger tips. So what do you need to get started?

Prerequisites?

Each API is unique; however, there are a few fundamental pieces of information you'll need to work with an API. First, the reason you're using an API is to request specific types of data from a specific data set from a specific organization. You at least need to know a little something about each one of these:

1. The URL for the organization and data you are pulling. Most pre-built API packages already have this connection established but when using `httr` you'll need to specify.
2. The data set you are trying to pull from. Most organizations have numerous data sets to peruse so you need to make yourself familiar with the names of the available data sets.
3. The data content. You'll need to specify the specific data variables you want the API to retrieve so you'll need to be familiar with, or have access to, the data library.

In addition to these key components you will also, typically, need to provide a form of identification and/or authorization. This is done via:

[126] http://www.ncdc.noaa.gov/cdo-web/webservices
[127] https://ropensci.org/tutorials/rnoaa_tutorial.html
[128] http://www.programmableweb.com/api-research

1. API key (aka token). A key is used to identify the user along with track and control how the API is being used (guard against malicious use). A key is often obtained by supplying basic information (i.e. name, email) to the organization and in return they give you a multi-digit key.
2. OAuth[129]. OAuth is an authorization framework that provides credentials as proof for access to certain information. Multiple forms of credentials exist and OAuth can actually be a fairly confusing topic; however, the `httr` package has simplified this greatly which we demonstrate at the end of this section.

Rather than dwell on these components, they'll likely become clearer as we progress through examples. So, let's move on to the fun stuff. ### Existing API Packages Like everything else you do in R, when looking to work with an API your first question should be "Is there a package for that?" R has an extensive list of packages in which API data feeds have been hooked into R. You can find a slew of them scattered throughout the CRAN Task View: Web Technologies and Services[130] web page, on the rOpenSci[131] web page, and some more here[132].

To give you a taste for how these packages typically work, I'll quickly cover three packages:

- `blsAPI` for pulling U.S. Bureau of Labor Statistics data
- `rnoaa` for pulling NOAA climate data
- `rtimes` for pulling data from multiple APIs offered by the New York Times

blsAPI

The `blsAPI`[133] allows users to request data for one or multiple series through the U.S. Bureau of Labor Statistics API. To use the `blsAPI` app you only need knowledge on the data; no key or OAuth are required. I lllustrate by pulling Mass Layoff Statistics[134] data but you will find all the available data sets and their series code information here[135].

The key information you will be concerned about is contained in the series identifier. For the Mass Layoff data the the series ID code is MLUMS00NN0001003. Each component of this series code has meaning and can be adjusted to get specific Mass Layoff data. The BLS provides this breakdown[136] for what each component means along with the available list of codes for this data set. For instance, the **S00** (MLUM**S00**NN0001003) component represents the division/state[137]. S00 will pull for all states but I could change to D30 to pull data for the Midwest or S39 to pull for Ohio. The **N0001**

[129]http://oauth.net/
[130]https://cran.r-project.org/web/views/WebTechnologies.html
[131]https://ropensci.org/packages/
[132]http://stats.stackexchange.com/questions/12670/data-apis-feeds-available-as-packages-in-r
[133]https://cran.r-project.org/web/packages/blsAPI/index.html
[134]http://www.bls.gov/mls/mlsover.htm
[135]http://www.bls.gov/help/hlpforma.htm
[136]http://www.bls.gov/help/hlpforma.htm#ML
[137]http://download.bls.gov/pub/time.series/ml/ml.srd

(MLUMS00N**N0001**003) component represents the industry/demographics[138]. N0001 pulls data for all industries but I could change to N0008 to pull data for the food industry or C00A2 for all persons age 30-44.

I simply call the series identifier in the `blsAPI()` function which pulls the JSON data object. We can then use the `fromJSON()` function from the `rjson` package to convert to an R data object (a list in this case). You can see that the raw data pull provides a list of 4 items. The first three provide some metadata info (status, response time, and message if applicable). The data we are concerned about is in the 4th (Results$series$data) list item which contains 31 observations.

```
library(rjson)
library(blsAPI)

# supply series identifier to pull data (initial pull is in JSON data)
layoffs_json <- blsAPI('MLUMS00NN0001003')

# convert from JSON into R object
layoffs <- fromJSON(layoffs_json)
```

```
List of 4
 $ status      : chr "REQUEST_SUCCEEDED"
 $ responseTime: num 38
 $ message     : list()
 $ Results     :List of 1
  ..$ series:List of 1
  .. ..$ :List of 2
  .. .. ..$ seriesID: chr "MLUMS00NN0001003"
  .. .. ..$ data    :List of 31
  .. .. .. ..$ :List of 5
  .. .. .. .. ..$ year      : chr "2013"
  .. .. .. .. ..$ period    : chr "M05"
  .. .. .. .. ..$ periodName: chr "May"
  .. .. .. .. ..$ value     : chr "1383"
```

One of the inconveniences of an API is we do not get to specify how the data we receive is formatted. This is a minor price to pay considering all the other benefits APIs provide. Once we understand the received data format we can typically re-format using a little list subsetting which we previously covered and looping which we'll cover in a future chapter.

[138] http://download.bls.gov/pub/time.series/ml/ml.irc

```
# create empty data frame to fill
layoff_df <- data.frame(NULL)

# extract data of interest from each nested year-month list
for(i in seq_along(layoffs$Results$series[[1]]$data)) {
        df <- data.frame(layoffs$Results$series[[1]]$data[i][[1]][1:4])
        layoff_df <- rbind(layoff_df, df)
}

head(layoff_df)
##   year period periodName value
## 1 2013    M05        May  1383
## 2 2013    M04      April  1174
## 3 2013    M03      March  1132
## 4 2013    M02   February   960
## 5 2013    M01    January  1528
## 6 2012    M13     Annual 17080
```

`blsAPI` also allows you to pull multiple data series and has optional arguments (i.e. start year, end year, etc.). You can see other options at `help(package = blsAPI)`.

rnoaa

The `rnoaa`[139] package allows users to request climate data from multiple data sets through the National Climatic Data Center API[140]. Unlike `blsAPI`, the `rnoaa` app requires you to have an API key. To request a key go here[141] and provide your email; a key will immediately be emailed to you.

```
key <- "vXTdwNoAVx..." # truncated
```

With the key in hand, we can begin pulling data. The NOAA provides a comprehensive metadata library[142] to familiarize yourself with the data available. Let's start by pulling all the available NOAA climate stations near my residence. I live in Montgomery county Ohio so we can find all the stations in this county by inserting the FIPS code[143]. Furthermore, I'm interested in stations that provide data for the `GHCND` data set[144] which contains records on numerous daily variables such as "maximum and minimum temperature, total daily precipitation, snowfall, and snow depth; however, about two thirds of the stations report precipitation only." See `?ncdc_stations` for other data sets available via `rnoaa`.

139 https://ropensci.org/tutorials/rnoaa_tutorial.html
140 http://www.ncdc.noaa.gov/cdo-web/webservices/v2
141 http://www.ncdc.noaa.gov/cdo-web/token
142 http://www.ncdc.noaa.gov/homr/reports
143 http://www.census.gov/geo/reference/codes/cou.html
144 https://www.ncdc.noaa.gov/oa/climate/ghcn-daily/

```
library(rnoaa)

stations <- ncdc_stations(datasetid='GHCND',
              locationid='FIPS:39113',
              token = key)

stations$data
## Source: local data frame [23 x 9]
##
##    elevation    mindate    maxdate latitude
##        (dbl)      (chr)      (chr)    (dbl)
## 1      294.1 2009-02-09 2014-06-25  39.6314
## 2      251.8 2009-03-01 2016-01-16  39.6807
## 3      295.7 2009-03-25 2012-09-08  39.6252
## 4      298.1 2009-08-24 2012-07-20  39.8070
## 5      304.5 2010-04-02 2016-01-12  39.6949
## 6      283.5 2012-07-01 2016-01-16  39.7373
## 7      301.4 2012-07-29 2016-01-16  39.8795
## 8      317.3 2012-09-08 2016-01-12  39.8329
## 9      298.1 2012-09-07 2016-01-15  39.6247
## 10     250.5 2012-09-11 2016-01-08  39.7180
## ..       ...        ...        ...      ...
## Variables not shown: name (chr), datacoverage (dbl), id (chr),
##   elevationUnit (chr), longitude (dbl)
```

So we see that several stations are available from which to pull data. To actually pull data from one of these stations we need the station ID. The station I want to pull data from is the Dayton International Airport station. We can see that this station provides data from 1948-present and I can get the station ID as illustrated. Note that I use some `dplyr` for data manipulation here; we will cover `dplyr` in a later chapter but this just illustrates the fact that we received the data via the API.

```
library(dplyr)

stations$data %>%
        filter(name == "DAYTON INTERNATIONAL AIRPORT, OH US") %>%
        select(mindate, maxdate, id)
## Source: local data frame [1 x 3]
##
##      mindate    maxdate                id
##        (chr)      (chr)             (chr)
## 1 1948-01-01 2016-01-15 GHCND:USW00093815
```

To pull all available GHCND data from this station we'll use `ncdc()`. We simply supply the data to pull, the start and end dates (`ncdc` restricts you to a one year limit), station ID, and your key. We can see that this station provides a full range of data types.

```
climate <- ncdc(datasetid='GHCND',
            startdate = '2015-01-01',
            enddate = '2016-01-01',
            stationid='GHCND:USW00093815',
            token = key)

climate$data
## Source: local data frame [25 x 8]
##
##                   date datatype           station value  fl_m  fl_q
##                  (chr)    (chr)             (chr) (int) (chr) (chr)
## 1  2015-01-01T00:00:00     AWND GHCND:USW00093815    72
## 2  2015-01-01T00:00:00     PRCP GHCND:USW00093815     0
## 3  2015-01-01T00:00:00     SNOW GHCND:USW00093815     0
## 4  2015-01-01T00:00:00     SNWD GHCND:USW00093815     0
## 5  2015-01-01T00:00:00     TAVG GHCND:USW00093815   -38     H
## 6  2015-01-01T00:00:00     TMAX GHCND:USW00093815    28
## 7  2015-01-01T00:00:00     TMIN GHCND:USW00093815   -71
## 8  2015-01-01T00:00:00     WDF2 GHCND:USW00093815   240
## 9  2015-01-01T00:00:00     WDF5 GHCND:USW00093815   240
## 10 2015-01-01T00:00:00     WSF2 GHCND:USW00093815   130
## ..                 ...      ...               ...   ...   ...   ...
## Variables not shown: fl_so (chr), fl_t (chr)
```

Since we recently had some snow here let's pull data on snow fall for 2015. We adjust the limit argument (by default `ncdc` limits results to 25) and identify the data type we want. By sorting we see what days experienced the greatest snowfall (don't worry, the results are reported in mm!).

```
snow <- ncdc(datasetid='GHCND',
            startdate = '2015-01-01',
            enddate = '2015-12-31',
            limit = 365,
            stationid='GHCND:USW00093815',
            datatypeid = 'SNOW',
            token = key)

snow$data %>%
        arrange(desc(value))
```

```
## Source: local data frame [365 x 8]
##
##                    date datatype            station value  fl_m  fl_q
##                   (chr)    (chr)              (chr) (int) (chr) (chr)
## 1  2015-03-01T00:00:00     SNOW GHCND:USW00093815   114
## 2  2015-02-21T00:00:00     SNOW GHCND:USW00093815   109
## 3  2015-01-25T00:00:00     SNOW GHCND:USW00093815    71
## 4  2015-01-06T00:00:00     SNOW GHCND:USW00093815    66
## 5  2015-02-16T00:00:00     SNOW GHCND:USW00093815    30
## 6  2015-02-18T00:00:00     SNOW GHCND:USW00093815    25
## 7  2015-02-14T00:00:00     SNOW GHCND:USW00093815    23
## 8  2015-01-26T00:00:00     SNOW GHCND:USW00093815    20
## 9  2015-02-04T00:00:00     SNOW GHCND:USW00093815    20
## 10 2015-02-12T00:00:00     SNOW GHCND:USW00093815    20
## ..                  ...      ...                ...   ...   ...   ...
## Variables not shown: fl_so (chr), fl_t (chr)
```

This is just an intro to `rnoaa` as the package offers a slew of data sets to pull from and functions to apply. It even offers built in plotting functions. Use `help(package = "rnoaa")` to see all that `rnoaa` has to offer.

rtimes

The `rtimes`[145] package provides an interface to Congress, Campaign Finance, Article Search, and Geographic APIs offered by the New York Times. The data libraries and documentation for the several APIs available can be found here[146]. To use the Times' API you'll need to get an API key here[147].

```
article_key <- "4f23572d8..."     # truncated
cfinance_key <- "ee0b7cef..."     # truncated
congress_key <- "57b3e8a3..."     # truncated
```

Lets start by searching NY Times articles. With the presendential elections upon us, we can illustrate by searching the least controversial candidate…Donald Trump. We can see that there are 4,566 article hits for the term "Trump". We can get more information on a particular article by subsetting.

[145]https://cran.r-project.org/web/packages/rtimes/index.html
[146]http://developer.nytimes.com/docs/
[147]http://developer.nytimes.com/apps/register

```
library(rtimes)

# article search for the term 'Trump'
articles <- as_search(q = "Trump",
                      begin_date = "20150101",
                      end_date = '20160101',
                      key = article_key)

# summary
articles$meta
##   hits time offset
## 1 4565   28      0

# pull info on 3rd article
articles$data[3]
## [[1]]
## <NYTimes article>Donald Trumpâs Strongest Supporters: A Certain Kind of Democ\
rat
##   Type: News
##   Published: 2015-12-31T00:00:00Z
##   Word count: 1469
##   URL: http://www.nytimes.com/2015/12/31/upshot/donald-trumps-strongest-suppo\
rters-a-certain-kind-of-democrat.html
##   Snippet: In a survey, he also excels among low-turnout voters and among the\
 less affluent and the less educated, so the question is: Will they show up to v\
ote?
```

We can use the campaign finance API and functions to gain some insight into Trumps compaign income and expenditures. The only special data you need is the FEC ID[148] for the candidate of interest.

```
trump <- cf_candidate_details(campaign_cycle = 2016,
                              fec_id = 'P80001571',
                              key = cfinance_key)

# pull summary data
trump$meta
##          id             name party
## 1 P80001571 TRUMP, DONALD J   REP
##                                                 fec_uri
## 1 http://docquery.fec.gov/cgi-bin/fecimg/?P80001571
```

[148] http://www.fec.gov/finance/disclosure/candcmte_info.shtml?tabIndex=2

```
##                    committee  mailing_address mailing_city
## 1 /committees/C00580100.json 725 FIFTH AVENUE     NEW YORK
##   mailing_state mailing_zip status total_receipts
## 1            NY       10022      O     1902410.45
##   total_from_individuals total_from_pacs total_contributions
## 1               92249.33               0            96298.97
##   candidate_loans total_disbursements begin_cash  end_cash
## 1      1804747.23          1414674.29          0 487736.16
##   total_refunds debts_owed date_coverage_from date_coverage_to
## 1             0 1804747.23         2015-04-02       2015-06-30
##   independent_expenditures coordinated_expenditures
## 1                1644396.8                        0
```

`rtimes` also allows us to gain some insight into what our locally elected officials are up to with the Congress API. First, I can get some informaton on my Senator and then use that information to see if he's supporting my interest. For instance, I can pull the most recent bills that he is co-sponsoring.

```
# pull info on OH senator
senator <- cg_memberbystatedistrict(chamber = "senate",
                                    state = "OH",
                                    key = congress_key)
senator$meta
##        id           name               role gender party
## 1 B000944 Sherrod  Brown Senator, 1st Class      M     D
##   times_topics_url      twitter_id       youtube_id seniority
## 1                   SenSherrodBrown SherrodBrownOhio         9
##   next_election
## 1          2018
##                                                                            \
 api_url
## 1 http://api.nytimes.com/svc/politics/v3/us/legislative/congress/members/B000\
944.json

# use member ID to pull recent bill sponsorship
bills <- cg_billscosponsor(memberid = "B000944",
                           type = "cosponsored",
                           key = congress_key)
head(bills$data)
## Source: local data frame [6 x 11]
##
##   congress    number
##      (chr)     (chr)
```

```
## 1      114    S.2098
## 2      114    S.2096
## 3      114    S.2100
## 4      114    S.2090
## 5      114 S.RES.267
## 6      114 S.RES.269
## Variables not shown: bill_uri (chr), title (chr), cosponsored_date
##   (chr), sponsor_id (chr), introduced_date (chr), cosponsors (chr),
##   committees (chr), latest_major_action_date (chr),
##   latest_major_action (chr)
```

It looks like the most recent bill Sherrod is co-sponsoring is S.2098 - Student Right to Know Before You Go Act. Maybe I'll do a NY Times article search with `as_search()` to find out more about this bill…an exercise for another time.

So this gives you some flavor of how these API packages work. You typically need to know the data sets and variables requested along with an API key. But once you get these basics its pretty straight forward on requesting the data. Your next question may be, what if the API that I want to get data from does not yet have an R package developed for it?

httr for All Things Else

Although numerous R API packages are available, and cover a wide range of data, you may eventually run into a situation where you want to leverage an organization's API but an R package does not exist. Enter `httr`[149]. `httr` was developed by Hadley Wickham to easily work with web APIs. It offers multiple functions (i.e. `HEAD()`, `POST()`, `PATCH()`, `PUT()` and `DELETE()`); however, the function we are most concerned with today is `Get()`. We use the `Get()` function to access an API, provide it some request parameters, and receive an output.

To give you a taste for how the `httr` package works, I'll quickly cover how to use it for a basic key-only API and an OAuth-required API:

- `Key-only API` is illustrated by pulling U.S. Department of Education data available on data.gov[150]
- `OAuth-required API` is illustrated by pulling tweets from my personal Twitter feed

Key-only API

To demonstrate how to use the `httr` package for accessing a key-only API, I'll illustrate with the College Scorecard API[151] provided by the Department of Education. First, you'll need to request your API key[152].

[149] https://cran.r-project.org/web/packages/httr/index.html
[150] https://api.data.gov/docs/
[151] https://api.data.gov/docs/ed/
[152] https://api.data.gov/signup/

```
edu_key <- "fd783wmS3Z..."     # truncated
```

We can now proceed to use `httr` to request data from the API with the `GET()` function. I went to North Dakota State University (NDSU) for my undergrad so I'm interested in pulling some data for this school. I can use the provided data library[153] and query explanation[154] to determine the parameters required. In this example, the `URL` includes the primary path ("https://api.data.gov/ed/collegescorecard/"), the API version ("v1"), and the endpoint ("schools"). The question mark ("?") at the end of the URL is included to begin the list of query parameters, which only includes my API key and the school of interest.

```
library(httr)

URL <- "https://api.data.gov/ed/collegescorecard/v1/schools?"

# import all available data for NDSU
ndsu_req <- GET(URL, query = list(api_key = edu_key,
                                  school.name = "North Dakota State University"))
```

This request provides me with every piece of information collected by the U.S. Department of Education for NDSU. To retrieve the contents of this request I use the `content()` function which will output the data as an R object (a list in this case). The data is segmented into two main components: *metadata* and *results*. I'm primarily interested in the results.

The results branch of this list provides information on lat-long location, school identifier codes, some basic info on the school (city, number of branches, school website, accreditor, etc.), and then student data for the years 1997-2013.

```
ndsu_data <- content(ndsu_req)

names(ndsu_data)
## [1] "metadata" "results"

names(ndsu_data$results[[1]])
##  [1] "2008"     "2009"     "2006"     "ope6_id"  "2007"     "2004"
##  [7] "2013"     "2005"     "location" "2002"     "2003"     "id"
## [13] "1996"     "1997"     "school"   "1998"     "2012"     "2011"
## [19] "2010"     "ope8_id"  "1999"     "2001"     "2000"
```

To see what kind of student data categories are offered we can assess a single year. You can see that available data includes earnings, academics, student info/demographics, admissions, costs, etc. With such a large data set, which includes many embedded lists, sometimes the easiest way to learn the data structure is to peruse names at different levels.

[153] https://collegescorecard.ed.gov/data/documentation/
[154] https://github.com/18F/open-data-maker/blob/api-docs/API.md

```
# student data categories available by year
names(ndsu_data$results[[1]]$`2013`)
## [1] "earnings"   "academics"  "student"    "admissions" "repayment"
## [6] "aid"        "cost"       "completion"

# cost categories available by year
names(ndsu_data$results[[1]]$`2013`$cost)
## [1] "title_iv"      "avg_net_price" "attendance"    "tuition"
## [5] "net_price"

# Avg net price cost categories available by year
names(ndsu_data$results[[1]]$`2013`$cost$avg_net_price)
## [1] "other_academic_year" "overall"             "program_year"
## [4] "public"              "private"
```

So if I'm interested in comparing the rise in cost versus the rise in student debt I can simply subset for this data once I've identified its location and naming structure. Note that for this subsetting we use the `magrittr` package and the 'sapply function; both we cover in later chapters but this is just meant to illustrate the types of data available through this API.

```
library(magrittr)

# subset list for annual student data only
ndsu_yr <- ndsu_data$results[[1]][c(as.character(1996:2013))]

# extract median debt data for each year
ndsu_yr %>%
        sapply(function(x) x$aid$median_debt$completers$overall) %>%
        unlist()
##    1997    1998    1999    2000    2001    2002    2003    2004
## 13388.0 13856.0 14500.0 15125.0 15507.0 15639.0 16251.0 16642.5
##    2005    2006    2007    2008    2009    2010    2011    2012
## 17125.0 17125.0 17125.0 17250.0 19125.0 21500.0 23000.0 24954.5
##    2013
## 25050.0

# extract net price for each year
ndsu_yr %>%
        sapply(function(x) x$cost$avg_net_price$overall) %>%
        unlist()
##  2009  2010  2011  2012  2013
## 13474 12989 13808 15113 14404
```

Quite simple isn't it…at least once you've learned how the query requests are formatted for a particular API.

OAuth-required API

At the outset I mentioned how OAuth is an authorization framework that provides credentials as proof for access. Many APIs are open to the public and only require an API key; however, some APIs require authorization to account data (think personal Facebook & Twitter accounts). To access these accounts we must provide proper credentials and OAuth authentication allows us to do this. This section is not meant to explain the details of OAuth (for that see this[155], this[156], and this[157]) but, rather, how to use `httr` in times when OAuth is required.

I'll demonstrate by accessing the Twitter API using my Twitter account. The first thing we need to do is identify the OAuth endpoints used to request access and authorization. To do this we can use `oauth_endpoint()` which typically requires a *request* URL, *authorization* URL, and *access* URL. `httr` also included some baked-in endpoints to include LinkedIn, Twitter, Vimeo, Google, Facebook, and GitHub. We can see the Twitter endpoints using the following:

```
twitter_endpts <- oauth_endpoints("twitter")
twitter_endpts
## <oauth_endpoint>
##  request:   https://api.twitter.com/oauth/request_token
##  authorize: https://api.twitter.com/oauth/authenticate
##  access:    https://api.twitter.com/oauth/access_token
```

Next, I register my application at https://apps.twitter.com/[158]. One thing to note is during the registration process, it will ask you for the *callback url*; be sure to use "http://127.0.0.1:1410". Once registered, Twitter will provide you with keys and access tokens. The two we are concerned about are the API key and API Secret.

```
twitter_key <- "BZgukbCol..."   # truncated
twitter_secret <- "YpB8Xy..."   # truncated
```

We can then bundle the consumer key and secret into one object with `oauth_app()`. The first argument, `appname` is simply used as a local identifier; it does not need to match the name you gave the Twitter app you developed at https://apps.twitter.com/.

We are now ready to ask for access credentials. Since Twitter uses OAuth 1.0 we use `oauth1.0_token()` function and incorporate the endpoints identified and the `oauth_app` object we previously named `twitter_app`.

[155] http://hueniverse.com/2007/09/05/explaining-oauth/
[156] https://en.wikipedia.org/wiki/OAuth
[157] http://hueniverse.com/oauth/
[158] https://apps.twitter.com/

```
twitter_token <- oauth1.0_token(endpoint = twitter_endpts, twitter_app)

Waiting for authentication in browser...
Press Esc/Ctrl + C to abort
Authentication complete.
```

Once authentication is complete we can now use the API. I can pull all the tweets that show up on my personal timeline using the `GET()` function and the access cridentials I stored in `twitter_token`. I then use `content()` to convert to a list and I can start to analyze the data.

In this case each tweet is saved as an individual list item and a full range of data are provided for each tweet (i.e. id, text, user, geo location, favorite count, etc). For instance, we can see that the first tweet was by FiveThirtyEight[159] concerning American politics and, at the time of this analysis, has been favorited by 3 people.

```
# request Twitter data
req <- GET("https://api.twitter.com/1.1/statuses/home_timeline.json",
           config(token = twitter_token))

# convert to R object
tweets <- content(req)

# available data for first tweet on my timeline
names(tweets[[1]])
 [1] "created_at"                    "id"
 [3] "id_str"                        "text"
 [5] "source"                        "truncated"
 [7] "in_reply_to_status_id"         "in_reply_to_status_id_str"
 [9] "in_reply_to_user_id"           "in_reply_to_user_id_str"
[11] "in_reply_to_screen_name"       "user"
[13] "geo"                           "coordinates"
[15] "place"                         "contributors"
[17] "is_quote_status"               "retweet_count"
[19] "favorite_count"                "entities"
[21] "extended_entities"             "favorited"
[23] "retweeted"                     "possibly_sensitive"
[25] "possibly_sensitive_appealable" "lang"

# further analysis of first tweet on my timeline
tweets[[1]]$user$name
[1] "FiveThirtyEight"
```

[159] http://fivethirtyeight.com/

```
tweets[[1]]$text
[1] "\U0001f3a7 A History Of Data In American Politics (Part 1): William Jenning\
s Bryan to Barack Obama https://t.co/oCKzrXuRHf  https://t.co/6CvKKToxoH"

tweets[[1]]$favorite_count
[1] 3
```

This provides a fairly simple example of incorporating OAuth authorization. The `httr` provides several examples of accessing common social network APIs that require OAuth. I recommend you go through several of these examples to get familiar with using OAuth authorization; see them at `demo(package = "httr")`. The most difficult aspect of creating your own connections with APIs is gaining an understanding of the API and the arguments they leverage. This obviously requires time and energy devoted to digging into the API documentation and data library. Next its just a matter of trial and error (likely more the latter than the former) to learn how to translate these arguments into `httr` function calls to pull the data of interest.

Also, note that `httr` provides several other useful functions not covered here for communicating with APIs (i.e. `POST()`, `BROWSE()`). For more on these other `httr` capabilities see this quickstart vignette[160].

Additional Resources

As I stated in the outset, this chapter is meant to provide an introduction to basic web scraping capabilities in R. This area is vast and complex and this chapter will far from provide you expertise level insight. To advance your knowledge in webscraping with R *Automated Data Collection with R*[161] and *XML and Web Technologies for Data Sciences with R*[162] offer the most detailed resources available. But this chapter should be enough to get your curiousity piqued and to start pulling data from the tangled masses of online data.

[160] https://cran.r-project.org/web/packages/httr/vignettes/quickstart.html

[161] http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX"

[162] http://www.amazon.com/XML-Web-Technologies-Data-Sciences/dp/1461478995

Exporting Data

Although getting data into R is essential, getting data out of R can be just as important. Whether you need to export data or analytic results simply to store, share, or feed into another system it is generally a straight forward process. This section will cover how to export data to text files, Excel files (along with some additional formatting capabilities), and save to R data objects. In addition to the the commonly used base R functions to perform data importing, I will also cover functions from the popular `readr` and `xlsx` packages along with a lesser known but useful `r2excel` package for Excel formatting.

Writing data to text files

As mentioned in the importing data section, text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Consequently, exporting data to a text file is a pretty standard operation. Plus, since you've already learned how to import text files you pretty much have the basics required to write to text files…we just use a slightly different naming convention.

Similar to the examples provided in the importing text files section, the two main groups of functions that I will demonstrate to write to text files include base R functions and `readr` package functions.

Base R functions

`write.table()` is the multipurpose work-horse function in base R for exporting data. The functions `write.csv()` and `write.delim()` are special cases of `write.table()` in which the defaults have been adjusted for efficiency. To illustrate these functions let's work with a data frame that we wish to export to a CSV file in our working directory.

```
df <- data.frame(var1 = c(10, 25, 8),
                 var2 = c("beer", "wine", "cheese"),
                          var3 = c(TRUE, TRUE, FALSE),
                 row.names = c("billy", "bob", "thornton"))
```

```
df
##          var1   var2  var3
## billy      10   beer  TRUE
## bob        25   wine  TRUE
## thornton    8 cheese FALSE
```

To export `df` to a CSV file we can use `write.csv()`. Additional arguments allow you to exclude row and column names, specify what to use for missing values, add or remove quotations around character strings, etc.

```
# write to a csv file
write.csv(df, file = "export_csv")

# write to a csv and save in a different directory
write.csv(df, file = "/folder/subfolder/subsubfolder/export_csv")

# write to a csv file with added arguments
write.csv(df, file = "export_csv", row.names = FALSE, na = "MISSING!")
```

In addition to CSV files, we can also write to other text files using `write.table` and `write.delim()`.

```
# write to a tab delimited text files
write.delim(df, file = "export_txt")

# provides same results as read.delim
write.table(df, file = "export_txt", sep="\t")
```

readr package

The `readr` package uses write functions similar to base R. However, `readr` write functions are about twice as fast and they do not write row names. One thing to note, where base R write functions use the `file =` argument, `readr` write functions use `path =`.

```
library(readr)

# write to a csv file
write_csv(df, path = "export_csv2")

# write to a csv and save in a different directory
write_csv(df, path = "/folder/subfolder/subsubfolder/export_csv2")

# write to a csv file without column names
write_csv(df, path = "export_csv2", col_names = FALSE)

# write to a txt file without column names
write_delim(df, path = "export_txt2", col_names = FALSE)
```

Writing data to Excel files

As previously mentioned, many organizations still rely on Excel to hold and share data so exporting to Excel is a useful bit of knowledge. And rather than saving to a .csv file to send to a co-worker who wants to work in Excel, its more efficient to just save R outputs directly to an Excel workbook. Since I covered importing data with the `xlsx` package, I'll also cover exporting data with this package. However, the `readxl` package which I demonstrated in the importing data section does not have a function to export to Excel. But there is a lesser known package called `r2excel` that provides exporting and formatting functions for Excel which I will cover.

xlsx package

Saving a data frame to a .xlsx file is as easy as saving to a .csv file:

```
library(xlsx)

# write to a .xlsx file
write.xlsx(df, file = "output_example.xlsx")

# write to a .xlsx file without row names
write.xlsx(df, file = "output_example.xlsx", row.names = FALSE)
```

In some cases you may wish to create a .xlsx file that contains multiple data frames. In this you can just create an empty workbook and save the data frames on seperate worksheets within the same workbook:

```
# create empty workbook
multiple_df <- createWorkbook()

# create worksheets within workbook
car_df <- createSheet(wb = multiple_df, sheetName = "Cars")
iris_df <- createSheet(wb = multiple_df, sheetName = "Iris")

# add data frames to worksheets; for this example I use the
# built in mtcars and iris data frames
addDataFrame(x = mtcars, sheet = car_df)
addDataFrame(x = iris, sheet = iris_df)

# save as a .xlsx file
saveWorkbook(multiple_df, file = "output_example_2.xlsx")
```

By default this saves the row and column names but this can be adjusted by adding `col.names = FALSE` and/or `row.names = FALSE` to the `addDataFrame()` function. There is also the ability to do some formatting with the `xlsx` package. The following provides several examples of how you can edit titles, subtitles, borders, column width, etc.[163] Although at first glance this can appear tedious for simple Excel editing, the real benefits present themselves when you integrate this editing into automated analyses.

```
# create new workbook
wb <- createWorkbook()

#--------------------
# DEFINE CELL STYLES
#--------------------
# title and subtitle styles
title_style <- CellStyle(wb) +
               Font(wb, heightInPoints = 16,
                    color = "blue",
                    isBold = TRUE,
                    underline = 1)

subtitle_style <- CellStyle(wb) +
                  Font(wb, heightInPoints = 14,
                       isItalic = TRUE,
                       isBold = FALSE)

# data table styles
rowname_style <- CellStyle(wb) +
                 Font(wb, isBold = TRUE)

colname_style <- CellStyle(wb) +
                 Font(wb, isBold = TRUE) +
                 Alignment(wrapText = TRUE, horizontal = "ALIGN_CENTER") +
                 Border(color = "black",
                        position = c("TOP", "BOTTOM"),
                        pen = c("BORDER_THIN", "BORDER_THICK"))

#-------------------------
# CREATE & EDIT WORKSHEET
#-------------------------
# create worksheet
```

[163] This example was derived from STHDA. Additional options, such as adding plot outputs can be found at STHDA and also in the *XML and Web Technologies for Data Sciences with R* book.

```
Cars <- createSheet(wb, sheetName = "Cars")

# helper function to add titles
xlsx.addTitle <- function(sheet, rowIndex, title, titleStyle) {
        rows <- createRow(sheet, rowIndex = rowIndex)
        sheetTitle <- createCell(rows, colIndex = 1)
        setCellValue(sheetTitle[[1,1]], title)
        setCellStyle(sheetTitle[[1,1]], titleStyle)
}

# add title and sub title to worksheet
xlsx.addTitle(sheet = Cars, rowIndex = 1,
              title = "1974 Motor Trend Car Data",
              titleStyle = title_style)

xlsx.addTitle(sheet = Cars, rowIndex = 2,
              title = "Performance and design attributes of 32 automobiles",
              titleStyle = subtitle_style)

# add data frame to worksheet
addDataFrame(mtcars, sheet = Cars, startRow = 3, startColumn = 1,
             colnamesStyle = colname_style,
             rownamesStyle = rowname_style)

# change row name column width
setColumnWidth(sheet = Cars, colIndex = 1, colWidth = 18)

# save workbook
saveWorkbook(wb, file = "output_example_3.xlsx")
```

	A	B	C	D	E	F	G	H	I	J	K	L
1	1974 Motor Trend Car Data											
2	*Performance and design attributes of 32 automobiles*											
3		mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
4	**Mazda RX4**	21	6	160	110	3.9	2.62	16.46	0	1	4	4
5	**Mazda RX4 Wag**	21	6	160	110	3.9	2.875	17.02	0	1	4	4
6	**Datsun 710**	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
7	**Hornet 4 Drive**	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
8	**Hornet Sportabout**	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
9	**Valiant**	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
10	**Duster 360**	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
11	**Merc 240D**	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
12	**Merc 230**	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
13	**Merc 280**	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
14	**Merc 280C**	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4
15	**Merc 450SE**	16.4	8	275.8	180	3.07	4.07	17.4	0	0	3	3
16	**Merc 450SL**	17.3	8	275.8	180	3.07	3.73	17.6	0	0	3	3
17	**Merc 450SLC**	15.2	8	275.8	180	3.07	3.78	18	0	0	3	3
18	**Cadillac Fleetwood**	10.4	8	472	205	2.93	5.25	17.98	0	0	3	4
19	**Lincoln Continental**	10.4	8	460	215	3	5.424	17.82	0	0	3	4
20	**Chrysler Imperial**	14.7	8	440	230	3.23	5.345	17.42	0	0	3	4
21	**Fiat 128**	32.4	4	78.7	66	4.08	2.2	19.47	1	1	4	1

Formatted Excel Output Example 1

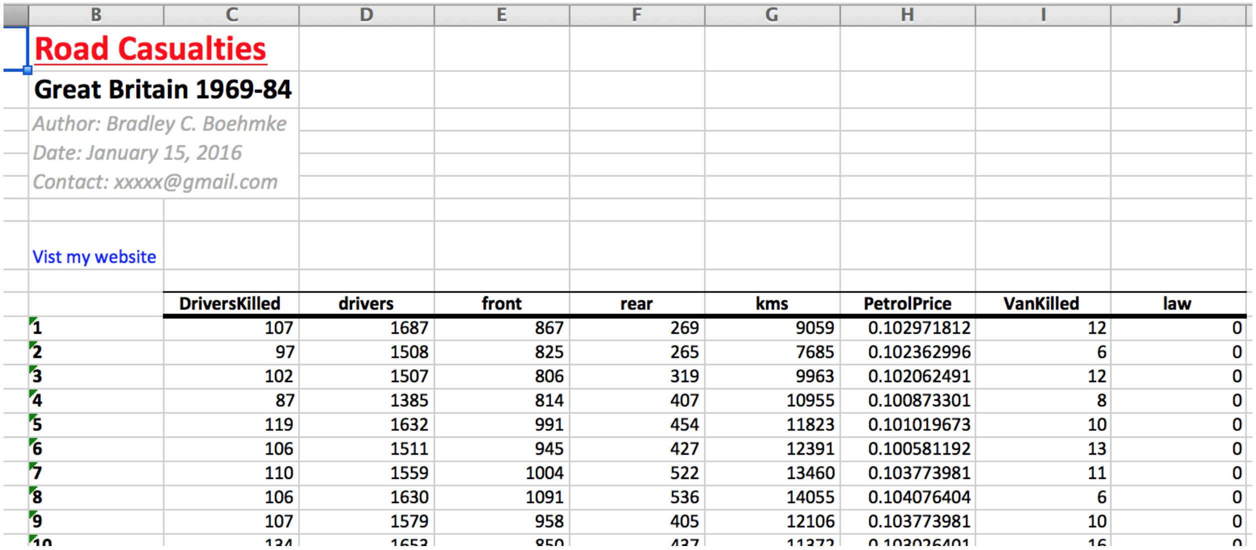
r2excel package

Although Formatting Excel files using the `xlsx` package is possible, the last section illustrated that it is a bit cumbersome. For this reason, A. Kassambara[164] created the `r2excel` package which depends on the `xlsx` package but provides easy to use functions for Excel formatting. The following provides a simple example but you can find many additional formatting functions here[165]

```
# install.packages("devtools")
devtools::install_github("kassambara/r2excel")
library(r2excel)

# create new workbook
wb <- createWorkbook()

# create worksheet
Casualties <- createSheet(wb, sheetName = "Casualties")

# add title
xlsx.addHeader(wb, sheet = Casualties,
               value = "Road Casualties",
               level = 1,
               color = "red",
               underline = 1)
```

[164] https://github.com/kassambara
[165] http://www.sthda.com/english/wiki/r2excel-read-write-and-format-easily-excel-files-using-r-software

```
# add subtitle
xlsx.addHeader(wb, sheet = Casualties,
               value = "Great Britain 1969-84",
               level = 2,
               color = "black")

# add author information
author = paste("Author: Bradley C. Boehmke \n",
             "Date: January 15, 2016 \n",
             "Contact: xxxxx@gmail.com", sep = "")

xlsx.addParagraph(wb, sheet = Casualties,
                  value = author,
                  isItalic = TRUE,
                  colSpan = 2,
                  rowSpan = 4,
                  fontColor = "darkgray",
                  fontSize = 14)

# add hyperlink
xlsx.addHyperlink(wb, sheet = Casualties,
                  address = "http://bradleyboehmke.github.io/",
                  friendlyName = "Vist my website", fontSize = 12)

xlsx.addLineBreak(sheet = Casualties, 1)

# add data frame to worksheet, I'm using the built in
# Seatbelt data which you can view at data(Seatbelt)
xlsx.addTable(wb, sheet = Casualties, data = Seatbelts, startCol = 2)

# save the workbook to an Excel file
saveWorkbook(wb, file = "output_example_4.xlsx")
```



B	C	D	E	F	G	H	I	J
Road Casualties								
Great Britain 1969-84								
Author: Bradley C. Boehmke								
Date: January 15, 2016								
Contact: xxxxx@gmail.com								
Vist my website								
	DriversKilled	drivers	front	rear	kms	PetrolPrice	VanKilled	law
1	107	1687	867	269	9059	0.102971812	12	0
2	97	1508	825	265	7685	0.102362996	6	0
3	102	1507	806	319	9963	0.102062491	12	0
4	87	1385	814	407	10955	0.100873301	8	0
5	119	1632	991	454	11823	0.101019673	10	0
6	106	1511	945	427	12391	0.100581192	13	0
7	110	1559	1004	522	13460	0.103773981	11	0
8	106	1630	1091	536	14055	0.104076404	6	0
9	107	1579	958	405	12106	0.103773981	10	0
10	134	1653	850	437	11372	0.103026401	16	0

Formatted Excel Output Example 2

Saving data as an R object file

Sometimes you may need to save data or other R objects outside of your workspace. You may want to share R data/objects with co-workers, transfer between projects or computers, or simply archive them. There are three primary ways that people tend to save R data/objects: as .RData, .rda, or as .rds files.

.rda is just short for .RData, therefore, these file extensions represent the same underlying object type. You use the .rda or .RData file types when you want to save several, or all, objects and functions that exist in your global environment. On the other hand, if you only want to save a single R object such as a data frame, function, or statistical model results its best to use .rds file type. You can use .rda or .RData to save a single object but the benefit of .rds is it only saves a representation of the object and not the name whereas .rda and .RData save the both the object and its name. As a result, with .rds the saved object can be loaded into a named object within R that is different from the name it had when originally saved. The following illustrates how you save R objects with each type.

```
# save() can be used to save multiple objects in you global environment,
# in this case I save two objects to a .RData file
x <- stats::runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.RData")

# save.image() is just a short-cut for â€˜save my current workspaceâ€™,
# i.e. all objects in your global environment
save.image()
```

```
# save a single object to file
saveRDS(x, "x.rds")

# restore it under a different name
x2 <- readRDS("x.rds")
identical(x, x2)
[1] TRUE
```

Additional resources

The following provides additional resources for exporting data:

- R data import/export manual[166]
- `WriteXLS` package[167]
- `XLConnect` package[168]

---

166 https://cran.r-project.org/doc/manuals/R-data.html
167 https://cran.r-project.org/web/packages/WriteXLS/WriteXLS.pdf
168 https://cran.r-project.org/web/packages/XLConnect/vignettes/XLConnect.pdf

Creating Efficient & Readable Code in R

> *"To iterate is human, to recurse divine."* - L. Peter Deutsch

Don't repeat yourself (DRY) is a software development principle aimed at reducing repetition. Formulated by Andy Hunt and Dave Thomas in their book The Pragmatic Programmer[169], the DRY principle states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system." This principle has been widely adopted to imply that you should not duplicate code. Although the principle was meant to be far grander than that[170], there's plenty of merit behind this slight misinterpretation.

Removing duplication is an important part of writing efficient code and reducing potential errors. First, reduced duplication of code can improve computing time and reduces the amount of code writing required. Second, less duplication results in less creation and saving of unnecessary objects. Inefficient code invariably creates copies of objects you have little interest in other than to feed into some future line of code; this wrecks havoc on properly managing your objects as it basically results in a global environment charlie foxtrot! Less duplication also results in less editing. When changes to code are required, duplicated code becomes tedious to edit and invariably mistakes or fat-fingering occur in the cut-and-paste editing process which just lengthens the editing that much more.

Furthermore, its important to have readable code. Clarity in your code creates clarity in your data analysis process. This is important as data analysis is a collaborative process so your code will likely need to be read and interpreted by others. Plus, invariably there will come a time where you will need to go back to an old analysis so your code also needs to be clear to your future-self.

This section covers the process of creating efficient and readable code. First, I cover the basics of [writing your own functions(#functions) so that you can reduce code duplication and automate generalized tasks to be applied recursively. I then cover loop control statements which allow you to perform repetititve code processes with different intentions and allow these automated expressions to naturally respond to features of your data. Lastly, I demonstrate how you can simplify your code to make it more readable and clear. Combined, these tools will move you forward in writing efficient, simple, *and* readable code.

[169] http://www.amazon.com/Pragmatic-Programmer-Journeyman-Master/dp/020161622X/ref=sr_1_1?s=books&ie=UTF8&qid=1456066112&sr=1-1&keywords=the+pragmatic+programmer

[170] According to Dave Thomas, "DRY says that every piece of system knowledge should have one authoritative, unambiguous representation. Every piece of knowledge in the development of something should have a single representation. A system's knowledge is far broader than just its code. It refers to database schemas, test plans, the build system, even documentation."

Functions

R is a functional programming language, meaning that everything you do is basically built on functions. However, moving beyond simply *using* pre-built functions to *writing* your own functions is when your capabilities really start to take off and your code development/writing takes on a new level of efficiency. Functions allow you to reduce code duplication by automating a generalized task to be applied recursively. Whenever you catch yourself repeating a function or copy and pasting code there is a good change that you should write a function to eliminate the redundancies.

Unfortunately, due to their abstractness, grasping the idea of writing functions (let alone writing them well) can take some time. However, in this chapter I will provide you with the basic knowledge of how functions operate in R to get you started on the right path. To do this, I cover the general components of functions, specifying function arguments, scoping and evaluation rules, managing function outputs, handling invalid parameters, and saving & sourcing functions for reuse. This will provide you the with the required knowledge to start building your own functions. Lastly, I offer some additional resources that will help you learn more about functions in R.

Function Components

With the exception of primitive functions[171] all R functions have three parts:

- `body()`: the code inside the function
- `formals()`: the list of arguments used to call the function
- `environment()`: the mapping of the location(s) of the function's variables

For example, let's build a function that calculates the present value (PV) of a single future sum. The equation for a single sum PV is: $PV = FV/(1+r)^n$ where FV is future value, r is the interest rate, and n is the number of periods. In the function that follows the `body` of the function includes the equation $FV/(1+r)^n$ and then rounding the output to two decimals. The `formals` (or arguments) required for the function include `FV`, `r`, and `n`. And the `environment` shows that function operates in the global environment.

[171] https://cran.r-project.org/doc/manuals/r-release/R-ints.html#g_t_002eInternal-vs-_002ePrimitive

```
PV <- function(FV, r, n) {
        PV <- FV/(1+r)^n
        round(PV, 2)
}

body(PV)
## {
##     PV <- FV/(1 + r)^n
##     round(PV, 2)
## }

formals(PV)
## $FV
##
##
## $r
##
##
## $n

environment(PV)
## <environment: R_GlobalEnv>
```

Arguments

To perform the `PV()` function we can call the arguments in different ways.

```
# using argument names
PV(FV = 1000, r = .08, n = 5)
## [1] 680.58

# same as above but without using names (aka "positional matching")
PV(1000, .08, 5)
## [1] 680.58

# if using names you can change the order
PV(r = .08, FV = 1000, n = 5)
## [1] 680.58

# if not using names you must insert arguments in proper order
# in this e.g. the function assumes FV = .08, r = 1000, and n = 5
```

```
PV(.08, 1000, 5)
## [1] 0
```

Note that when building a function you can also set default values for arguments. In our original `PV()` we did not provide any default values so if we do not supply all the argument parameters an error will be returned. However, if we set default values then the function will use the stated default if any parameters are missing:

```
# missing the n argument
PV(1000, .08)
## Error in PV(1000, 0.08): argument "n" is missing, with no default

# creating default argument values
PV <- function(FV = 1000, r = .08, n = 5) {
        PV <- FV/(1+r)^n
        round(PV, 2)
}

# function will use default n value
PV(1000, .08)
## [1] 680.58

# specifying a different n value
PV(1000, .08, 3)
## [1] 793.83
```

Scoping Rules

Scoping refers to the set of rules a programming language uses to lookup the value to variables and/or symbols. The following illustrates the basic concept behind the lexical scoping rules that R follows.

A function will first look inside the function to identify all the variables being called. If all variables exist then their is no additional search required to identify variables.

```
PV1 <- function() {
        FV <- 1000
        r <- .08
        n <- 5
        FV/(1+r)^n
}

PV1()
## [1] 680.5832
```

However, if a variable does not exist within the function, R will look one level up to see if the variable exists.

```
# the FV variable is outside the function environment
FV <- 1000

PV2 <- function() {
        r <- .08
        n <- 5
        FV/(1+r)^n
}

PV2()
## [1] 680.5832
```

This same concept applies if you have functions embeded within functions:

```
FV <- 1000

PV3 <- function() {
        r <- .08
        n <- 5
        denominator <- function() {
                (1+r)^n
        }
        FV/denominator()
}

PV3()
## [1] 680.5832
```

This also applies for functions in which some arguments are called but not all variables used in the body are identified as arguments:

```
# n is specified within the function
PV4 <- function(FV, r) {
        n <- 5
        FV/(1+r)^n
}

PV4(1000, .08)
## [1] 680.5832

# n is specified within the function and
# r is specified outside the function
r <- 0.08

PV5 <- function(FV) {
        n <- 5
        FV/(1+r)^n
}

PV5(1000)
## [1] 680.5832
```

Lazy Evaluation

R functions perform "lazy" evaluation in which arguments are only evaluated if required in the body of the function.

```
# the y argument is not used so not including it causes
# no harm
lazy <- function(x, y){
        x*2
}
lazy(4)
## [1] 8

# however, if both arguments are required in the body
# an error will result if an argument is missing
lazy2 <- function(x, y){
        (x+y)*2
}
lazy2(4)
## Error in lazy2(4): argument "y" is missing, with no default
```

Returning Multiple Outputs from a Function

If a function performs multiple tasks and therefore has multiple results to report then we have to include the `c()` function inside the function to display all the results. If you do not include the `c()` function then the function output will only return the last expression:

```
bad <- function(x, y) {
        2*x + y
        x + 2*y
        2*x + 2*y
        x/y
}
bad(1, 2)
## [1] 0.5

good <- function(x, y) {
        output1 <- 2*x + y
        output2 <- x + 2*y
        output3 <- 2*x + 2*y
        output4 <- x/y
        c(output1, output2, output3, output4)
}
good(1, 2)
## [1] 4.0 5.0 6.0 0.5
```

Furthermore, when we have a function which performs multiple tasks (i.e. computes multiple computations) then it is often useful to save the results in a list.

```
good_list <- function(x, y) {
        output1 <- 2*x + y
        output2 <- x + 2*y
        output3 <- 2*x + 2*y
        output4 <- x/y
        c(list(Output1 = output1, Output2 = output2,
               Output3 = output3, Output4 = output4))
}
good_list(1, 2)
## $Output1
## [1] 4
##
## $Output2
## [1] 5
```

```
##
## $Output3
## [1] 6
##
## $Output4
## [1] 0.5
```

Dealing with Invalid Parameters

For functions that will be used again, and especially for those used by someone other than the creator of the function, it is good to check the validity of arguments within the function. One way to do this is to use the `stop()` function. The following uses an `if()` statement to check if the class of each argument is numeric. If one or more arguments are not numeric then the `stop()` function will be triggered to provide a meaningful message to the user.

```
PV <- function(FV, r, n) {
        if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){
                stop('This function only works for numeric inputs!\n',
                     'You have provided objects of the following classes:\n',
                     'FV: ', class(FV), '\n',
                     'r: ', class(r), '\n',
                     'n: ', class(n))
        }

        PV <- FV/(1+r)^n
        round(PV, 2)
}

PV("1000", 0.08, "5")
## Error in PV("1000", 0.08, "5"): This function only works for numeric inputs!
## You have provided objects of the following classes:
## FV: character
## r: numeric
## n: character
```

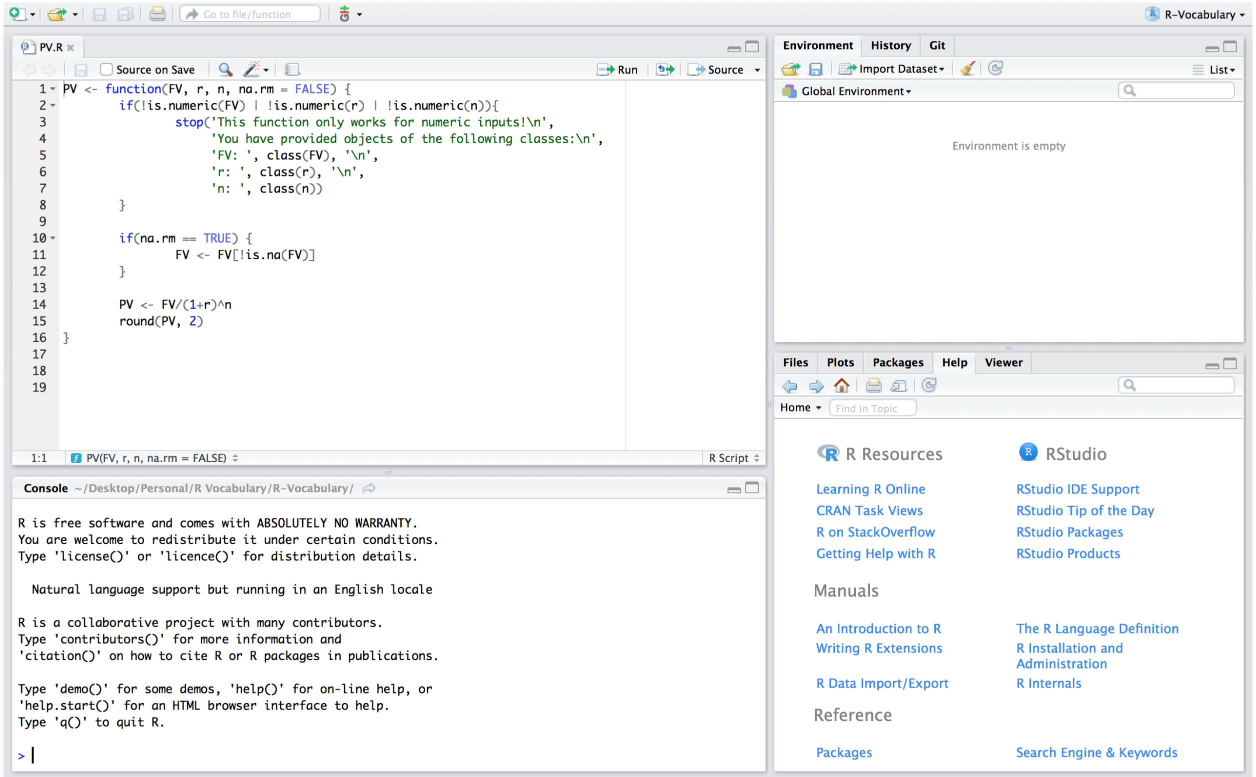
Another concern is dealing with missing or `NA` values. Lets say you wanted to perform the `PV()` function on a vector of potential future values. The function as is will output `NA` in place of any missing values in the FV input vector. If you want to remove the missing values then you can incorporate the `na.rm` parameter in the function arguments along with an `if` statement to remove missing values if `na.rm = TRUE`.

```
# vector of future value inputs
fv <- c(800, 900, NA, 1100, NA)

# original PV() function will return NAs
PV(fv, .08, 5)
## [1] 544.47 612.52     NA 748.64     NA

# add na.rm argument
PV <- function(FV, r, n, na.rm = FALSE) {
        if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){
                stop('This function only works for numeric inputs!\n',
                     'You have provided objects of the following classes:\n',
                     'FV: ', class(FV), '\n',
                     'r: ', class(r), '\n',
                     'n: ', class(n))
        }

        if(na.rm == TRUE) {
                FV <- FV[!is.na(FV)]
        }

        PV <- FV/(1+r)^n
        round(PV, 2)
}

# setting na.rm = TRUE argument eliminates NA outputs
PV(fv, 0.08, 5, na.rm = TRUE)
## [1] 544.47 612.52 748.64
```

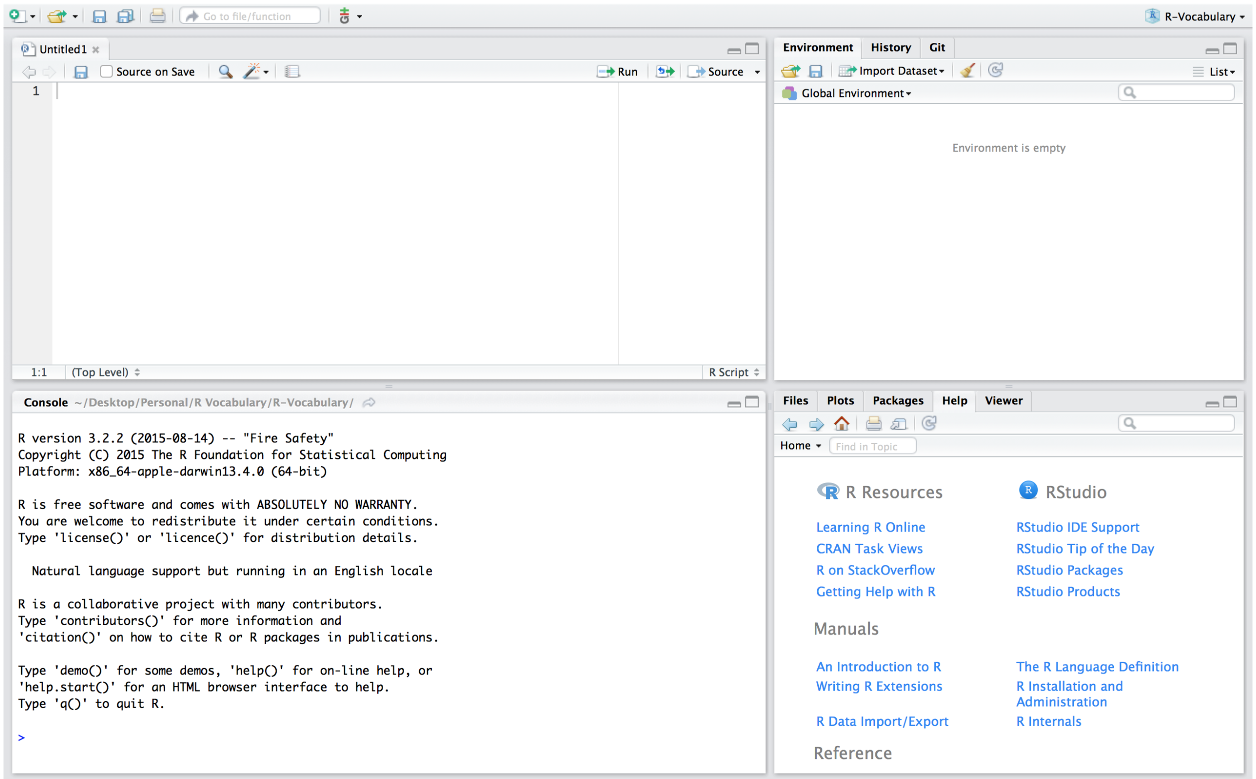
Saving and Sourcing Functions

If you want to save a function to be used at other times and within other scripts there are two main ways to do this. One way is to build a package which I do not cover in this book but is discussed in more details here[172]. Another option, and the one discussed here, is to save the function in a script. For example, we can save a script that contains the `PV()` function and save this script as `PV.R`.

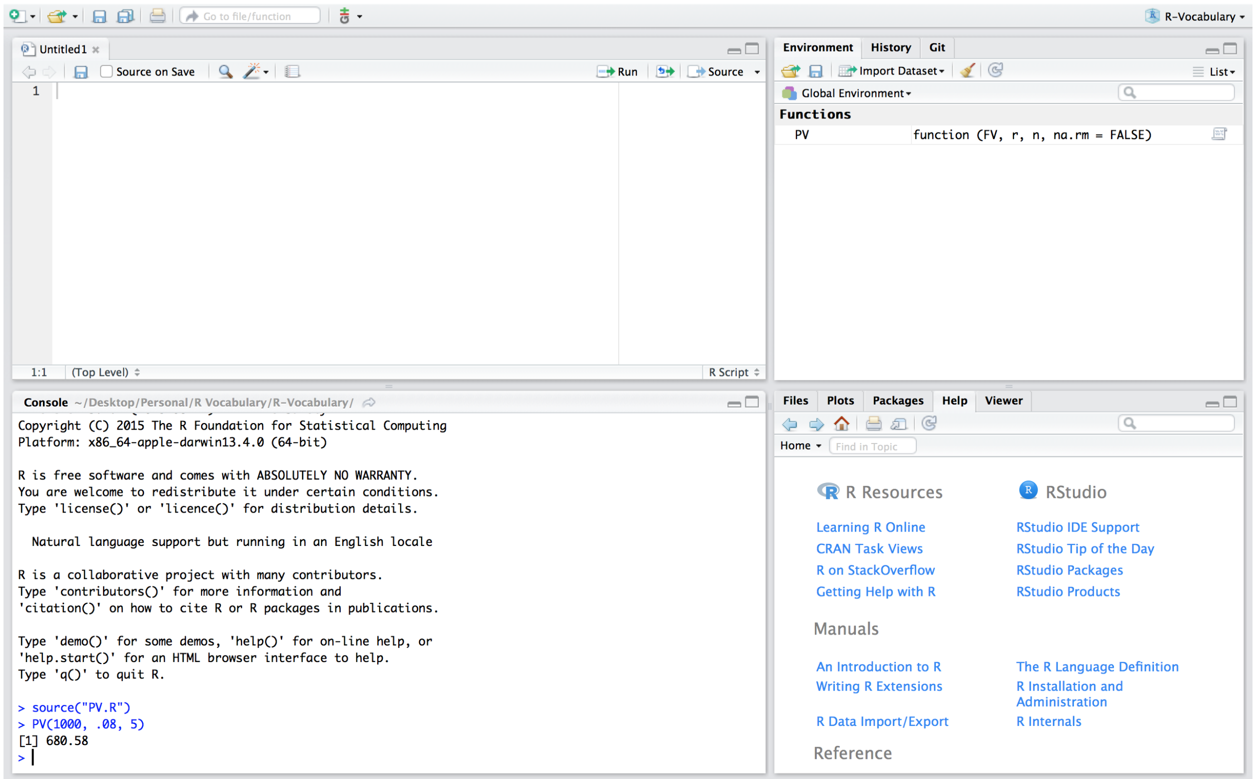
[172] http://r-pkgs.had.co.nz/



Now, if we are working in a fresh script you'll see that we have no objects and functions in our working environment:



If we want to use the PV function in this new script we can simply read in the function by sourcing the script using `source("PV.R")`. Now, you'll notice that we have the `PV()` function in our global environment and can use it as normal. Note that if you are working in a different directory then where the `PV.R` file is located you'll need to include the proper command to access the relevant directory.



Additional Resources

Functions are a fundamental building block of R and writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. As a result, its important to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how they work. A few additional resources that can help you get to the next step of understanding functions include:

- Hadley Wickham's Advanced R book[173]
- Roger Peng's R Programming for Data Science book[174]
- DataCamp's Intermediate R course[175]
- Coursera's R Programming course[176]

---

[173] http://adv-r.had.co.nz/Functions.html

[174] https://leanpub.com/rprogramming

[175] https://www.datacamp.com/courses/intermediate-r?utm_source=functions_r_tutorial_post&utm_medium=blog&utm_campaign=functions_r_tutorial_post

[176] https://www.coursera.org/course/rprog

Loop Control Statements

Looping is similiar to creating functions in that they are merely a means to automate a certain multi step process by organizing sequences of R expressions. R consists of several loop control statements which allow you to perform repetititve code processes with different intentions and allow these automated expressions to naturally respond to features of your data. Consequently, learning these loop control statements will go a long ways in reducing code redundancy and becoming a more efficient data wrangler.

This chapter starts by covering the basic control statements in R, which includes `if`, `else`, along with the `for`, `while`, and `repeat` loop control structures. In addition, I cover `break` and `next` which allow you to further control flow within the aforementioned control statements. Next I cover a set of vectorized functions known as the apply family of functions which minimize your need to explicitly create loops. I then provide some additional "loop-like" functions that are helpful in everyday data analysis followed by a list of additional resources to learn more about control structures in R.

Basic control statements (i.e. `if`, `for`, `while`, etc.)

if Statement

The conditional `if` statement is used to test an expression. If the `test_expression` is `TRUE`, the `statement` gets executed. But if it's `FALSE`, nothing happens.

```
# syntax of if statement
if (test_expression) {
        statement
}
```

The following is an example that tests if any values in a vector are negative. Notice there are two ways to write this `if` statement; since the body of the statement is only one line you can write it with or without curly braces. I recommend getting in the habit of using curly braces, that way if you build onto if statements with additional functions in the body or add an `else` statement later you will not run into issues with unexpected code procedures.

```
x <- c(8, 3, -2, 5)

# without curly braces
if(any(x < 0)) print("x contains negative numbers")
## [1] "x contains negative numbers"

# with curly braces produces same result
if(any(x < 0)){
        print("x contains negative numbers")
}
## [1] "x contains negative numbers"

# an if statement in which the test expression is FALSE
# does not produce any output
y <- c(8, 3, 2, 5)

if(any(y < 0)){
        print("y contains negative numbers")
}
```

if...else Statement

The conditional `if...else` statement is used to test an expression similar to the `if` statement. However, rather than nothing happening if the `test_expression` is `FALSE`, the `else` part of the function will be evaluated.

```
# syntax of if...else statement
if (test_expression) {
        statement 1
} else {
        statement 2
}
```

The following extends the previous example illustrated for the `if` statement in which the `if` statement tests if any values in a vector are negative; if `TRUE` it produces one output and if `FALSE` it produces the `else` output.

```
# this test results in statement 1 being executed
x <- c(8, 3, -2, 5)

if(any(x < 0)){
        print("x contains negative numbers")
} else{
        print("x contains all positive numbers")
}
## [1] "x contains negative numbers"

# this test results in statement 2 (or the else statement) being executed
y <- c(8, 3, 2, 5)

if(any(y < 0)){
        print("y contains negative numbers")
} else{
        print("y contains all positive numbers")
}
## [1] "y contains all positive numbers"
```

Simple `if...else` statements, as above, in which only one line of code is being executed in the statements can be written in a simplified alternative manner. These alternatives are only recommended for very short `if...else` code:

```
x <- c(8, 3, 2, 5)

# alternative 1
if(any(x < 0)) print("x contains negative numbers") else print("x contains all p\
ositive numbers")
## [1] "x contains all positive numbers"

# alternative 2 using the ifelse function
ifelse(any(x < 0), "x contains negative numbers", "x contains all positive numbe\
rs")
## [1] "x contains all positive numbers"
```

We can also nest as many `if...else` statements as required (or desired). For example:

```
# this test results in statement 1 being executed
x <- 7

if(x >= 10){
        print("x exceeds acceptable tolerance levels")
} else if(x >= 0 & x < 10){
        print("x is within acceptable tolerance levels")
} else {
         print("x is negative")
}
## [1] "x is within acceptable tolerance levels"
```

for Loop

The `for` loop is used to execute repetitive code statements for a particular number of times. The general syntax is provided below where `i` is the counter and as `i` assumes each sequential value defined (1 through 100 in this example) the code in the body will be performed for that ith value.

```
# syntax of for loop
for(i in 1:100) {
        <do stuff here with i>
}
```

An important lesson to learn is that R is not efficient at *growing* data objects. As a result, it is more efficient to create an empty data object and *fill* it with the `for` loop outputs. For example, if you want to create a vector in which 5 values are randomly drawn from a poisson distribution with mean 5, it is less efficient to perform the first example in the following code chunk than to perform the second example. Although this inefficiency is not noticed in this small example, when you perform larger repetitions it will become noticable so you might as well get in the habit of *filling* rather than *growing*.

```
# not advised
for(i in 5){
        x <- rpois(5, lambda = 5)
        print(x)
}
## [1] 11  5  8  8  7

# advised
x <- vector(mode = "numeric", length = 5)

for(i in 5){
```

```
        x <- rpois(5, lambda = 5)
        print(x)
}
## [1] 5 8 9 5 4
```

Another example in which we create an empty matrix with 5 rows and 5 columns. The `for` loop then iterates over each column (note how *i* takes on the values 1 through the number of columns in the `my.mat` matrix) and takes a random draw of 5 values from a poisson distribution with mean *i* in column *i*:

```
my.mat <- matrix(NA, nrow = 5, ncol = 5)

for(i in 1:ncol(my.mat)){
        my.mat[, i] <- rpois(5, lambda = i)
}
my.mat
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    2    1    7    1
## [2,]    1    2    2    3    9
## [3,]    2    1    5    6    6
## [4,]    2    1    5    2   10
## [5,]    0    2    2    2    4
```

while Loop

While loops begin by testing a condition. If it is true, then they execute the statement. Once the statement is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits. It's considered a best practice to include a counter object to keep track of total iterations

```
# syntax of while loop
counter <- 1

while(test_expression) {
        statement
        counter <- counter + 1
}
```

`while` loops can potentially result in infinite loops if not written properly; therefore, you must use them with care. To provide a simple example to illustrate how similiar `for` and `while` loops are:

```
counter <- 1

while(counter <= 10) {
        print(counter)
        counter <- counter + 1
}

# this for loop provides the same output
counter <- vector(mode = "numeric", length = 10)

for(i in 1:length(counter)) {
        print(i)
}
```

The primary difference between a `for` loop and a `while` loop is: a `for` loop is used when the number of iterations a code should be run is known where a `while` loop is used when the number of iterations is not known. For instance, the following takes value `x` and adds or subtracts 1 from the value randomly until `x` exceeds the values in the test expression. The output illustrates that the code runs 14 times until x exceeded the threshold with the value 9.

```
counter <- 1
x <- 5
set.seed(3)

while(x >= 3 && x <= 8 ) {
        coin <- rbinom(1, 1, 0.5)

        if(coin == 1) { ## random walk
                x <- x + 1
        } else {
                x <- x - 1
        }
        cat("On iteration", counter, ", x =", x, '\n')
        counter <- counter + 1
}
## On iteration 1 , x = 4
## On iteration 2 , x = 5
## On iteration 3 , x = 4
## On iteration 4 , x = 3
## On iteration 5 , x = 4
## On iteration 6 , x = 5
## On iteration 7 , x = 4
```

```
## On iteration 8 , x = 3
## On iteration 9 , x = 4
## On iteration 10 , x = 5
## On iteration 11 , x = 6
## On iteration 12 , x = 7
## On iteration 13 , x = 8
## On iteration 14 , x = 9
```

repeat Loop

A `repeat` loop is used to iterate over a block of code multiple number of times. There is test expression in a repeat loop to end or exit the loop. Rather, we must put a condition statement explicitly inside the body of the loop and use the `break` function to exit the loop. Failing to do so will result into an infinite loop.

```
# syntax of repeat loop
counter <- 1

repeat {
        statement

        if(test_expression){
                break
        }
        counter <- counter + 1
}
```

For example ,say we want to randomly draw values from a uniform distribution between 1 and 25. Furthermore, we want to continue to draw values randomly until our sample contains at least each integer value between 1 and 25; however, we do not care if we've drawn a particular value multiple times. The following code repeats the random draws of values between 1 and 25 (in which we round). We then include an `if` statement to check if all values between 1 and 25 are present in our sample. If so, we use the `break` statement to exit the loop. If not, we add to our counter and let the loop repeat until the conditional `if` statement is found to be true. We can then check the `counter` object to assess how many iterations were required to reach our conditional requirement.

```
counter <- 1
x <- NULL

repeat {
        x <- c(x, round(runif(1, min = 1, max = 25)))

        if(all(1:25 %in% x)){
                break
        }

        counter <- counter + 1
}

counter
## [1] 75
```

break Function to Exit a Loop

The `break` function is used to exit a loop immediately, regardless of what iteration the loop may be on. `break` functions are typically embedded in an `if` statement in which a condition is assessed, if TRUE `break` out of the loop, if FALSE continue on with the loop. In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

```
x <- 1:5

for (i in x) {
        if (i == 3){
                break
                }
        print(i)
}
## [1] 1
## [1] 2
```

next Function to Skip an Iteration in a Loop

The `next` statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

```
x <- 1:5

for (i in x) {
        if (i == 3){
                next
                }
        print(i)
}
## [1] 1
## [1] 2
## [1] 4
## [1] 5
```

Apply family

The apply family consists of vectorized functions which minimize your need to explicitly create loops. These functions will apply a specified function to a data object and there primary difference is in the object class in which the function is applied to (list vs. matrix, etc) and the object class that will be returned from the function. The following presents the most common forms of apply functions that I use for data analysis but realize that additional functions exist (`mapply`, `rapply`, & `vapply`) which are not covered here.

apply() for Matrices and Data Frames

The `apply()` function is most often used to apply a function to the rows or columns (margins) of matrices or data frames. However, it can be used with general arrays, for example, to take the average of an array of matrices. Using `apply()` is not faster than using a loop function, but it is highly compact and can be written in one line.

The syntax for `apply()` is as follows where

- `x` is the matrix, dataframe or array
- `MARGIN` is a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns.
- `FUN` is the function to be applied
- `...` is for any other arguments to be passed to the function

```
# syntax of apply function
apply(x, MARGIN, FUN, ...)
```

To provide examples let's use the `mtcars` data set provided in R:

```
# show first few rows of mtcars
head(mtcars)
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

# get the mean of each column
apply(mtcars, 2, mean)
##        mpg        cyl       disp         hp       drat         wt
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##       qsec         vs         am       gear       carb
##  17.848750   0.437500   0.406250   3.687500   2.812500

# get the sum of each row (not really relevant for this data
# but it illustrates the capability)
apply(mtcars, 1, sum)
##           Mazda RX4       Mazda RX4 Wag          Datsun 710
##             328.980             329.795             259.580
##      Hornet 4 Drive   Hornet Sportabout             Valiant
##             426.135             590.310             385.540
##          Duster 360           Merc 240D            Merc 230
##             656.920             270.980             299.570
##            Merc 280           Merc 280C          Merc 450SE
##             350.460             349.660             510.740
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood
##             511.500             509.850             728.560
## Lincoln Continental   Chrysler Imperial            Fiat 128
##             726.644             725.695             213.850
##         Honda Civic      Toyota Corolla       Toyota Corona
##             195.165             206.955             273.775
##    Dodge Challenger         AMC Javelin          Camaro Z28
##             519.650             506.085             646.280
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2
##             631.175             208.215             272.570
##        Lotus Europa      Ford Pantera L        Ferrari Dino
##             273.683             670.690             379.590
##       Maserati Bora          Volvo 142E
##             694.710             288.890
```

```
# get column quantiles (notice the quantile percents as row names)
apply(mtcars, 2, quantile, probs = c(0.10, 0.25, 0.50, 0.75, 0.90))
##        mpg cyl    disp    hp  drat      wt    qsec vs am gear carb
## 10% 14.340   4  80.610  66.0 3.007 1.95550 15.5340  0  0    3    1
## 25% 15.425   4 120.825  96.5 3.080 2.58125 16.8925  0  0    3    2
## 50% 19.200   6 196.300 123.0 3.695 3.32500 17.7100  0  0    4    2
## 75% 22.800   8 326.000 180.0 3.920 3.61000 18.9000  1  1    4    4
## 90% 30.090   8 396.000 243.5 4.209 4.04750 19.9900  1  1    5    4
```

lapply() for Lists…Output as a List

The `lapply()` function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a function to each element of the list (a function that you specify)
3. and returns a list (the l is for "list").

The syntax for `lapply()` is as follows where

- `x` is the list
- `FUN` is the function to be applied
- `...` is for any other arguments to be passed to the function

```
# syntax of lapply function
lapply(x, FUN, ...)
```

To provide examples we'll generate a list of four items:

```
data <- list(item1 = 1:4, item2 = rnorm(10),
             item3 = rnorm(20, 1), item4 = rnorm(100, 5))

# get the mean of each list item
lapply(data, mean)
## $item1
## [1] 2.5
##
## $item2
## [1] 0.5529324
##
```

```
## $item3
## [1] 1.193884
##
## $item4
## [1] 5.013019
```

The above provides a simple example where each list item is simply a vector of numeric values. However, consider the case where you have a list that contains data frames and you would like to loop through each list item and perform a function to the data frame. In this case we can embed an `apply` function within an `lapply` function.

For example, the following creates a list for R's built in beaver data sets. The `lapply` function loops through each of the two list items and uses `apply` to calculate the mean of the columns in both list items. Note that I wrap the apply function with `round` to provide an easier to read output.

```
# list of R's built in beaver data
beaver_data <- list(beaver1 = beaver1, beaver2 = beaver2)

# get the mean of each list item
lapply(beaver_data, function(x) round(apply(x, 2, mean), 2))
## $beaver1
##     day    time    temp   activ
##  346.20 1312.02   36.86    0.05
##
## $beaver2
##     day    time    temp   activ
##  307.13 1446.20   37.60    0.62
```

sapply() for Lists…Output Simplified

The `sapply()` function behaves similarly to `lapply()`; the only real difference is in the return value. `sapply()` will try to simplify the result of `lapply()` if possible. Essentially, `sapply()` calls `lapply()` on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If neither of the above simplifications can be performed then a list is returned

To illustrate the differences we can use the previous example using a list with the beaver data and compare the `sapply` and `lapply` outputs:

```
# list of R's built in beaver data
beaver_data <- list(beaver1 = beaver1, beaver2 = beaver2)

# get the mean of each list item and return as a list
lapply(beaver_data, function(x) round(apply(x, 2, mean), 2))
## $beaver1
##     day    time    temp   activ
##  346.20 1312.02   36.86    0.05
##
## $beaver2
##     day    time    temp   activ
##  307.13 1446.20   37.60    0.62

# get the mean of each list item and simply the output
sapply(beaver_data, function(x) round(apply(x, 2, mean), 2))
##       beaver1 beaver2
## day    346.20  307.13
## time  1312.02 1446.20
## temp    36.86   37.60
## activ    0.05    0.62
```

tapply() for Vectors

`tapply()` is used to apply a function over subsets of a vector. It is primarily used when we have the following circumstances:

1. A dataset that can be broken up into groups (via categorical variables - aka factors)
2. We desire to break the dataset up into groups
3. Within each group, we want to apply a function

The arguments to `tapply()` are as follows:

- `x` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed FUN
- `simplify`, should we simplify the result?

```
# syntax of tapply function
tapply(x, INDEX, FUN, ..., simplify = TRUE)
```

To provide an example we'll use the built in mtcars dataset and calculate the mean of the `mpg` variable grouped by the `cyl` variable.

```
# show first few rows of mtcars
head(mtcars)
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

# get the mean of the mpg column grouped by cylinders
tapply(mtcars$mpg, mtcars$cyl, mean)
##        4        6        8
## 26.66364 19.74286 15.10000
```

Now let's say you want to calculate the mean for *each* column in the mtcars dataset grouped by the cylinder categorical variable. To do this you can embed the `tapply` function within the `apply` function.

```
# get the mean of all columns grouped by cylinders
apply(mtcars, 2, function(x) tapply(x, mtcars$cyl, mean))
##        mpg cyl     disp        hp     drat       wt     qsec        vs
## 4 26.66364   4 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909
## 6 19.74286   6 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286
## 8 15.10000   8 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000
##          am     gear     carb
## 4 0.7272727 4.090909 1.545455
## 6 0.4285714 3.857143 3.428571
## 8 0.1428571 3.285714 3.500000
```

Note that this type of summarization can also be done using the `dplyr` package with clearer syntax. This is covered in the dplyr section*

Other useful "loop-like" functions

In addition to the `apply` family which provide vectorized functions that minimize your need to explicitly create loops, there are also a few commonly applied `apply` functions that have been further simplified. These include the calculation of column and row sums, means, medians, standard deviations, variances, and summary quantiles across the entire data set.

The most common apply functions that have been include calculating the sums and means of columns and rows. For instance, to calculate the sum of columns across a data frame or matrix you could do the following:

```
apply(mtcars, 2, sum)
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  642.900  198.000 7383.100 4694.000  115.090  102.952  571.160   14.000
##       am     gear     carb
##   13.000  118.000   90.000
```

However, you can perform the same function with the shorter `colSums()` function and it performs faster:

```
colSums(mtcars)
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  642.900  198.000 7383.100 4694.000  115.090  102.952  571.160   14.000
##       am     gear     carb
##   13.000  118.000   90.000
```

To illustrate the speed difference we can compare the performance of using the `apply()` function versus the `colSums()` function on a matrix with 100 million values (10K x 10K). You can see that the speed of `colSums()` is significantly faster.

```
# develop a 10,000 x 10,000 matrix
mat = matrix(sample(1:10, size=100000000, replace=TRUE), nrow=10000)

system.time(apply(mat, 2, sum))
##    user  system elapsed
##   1.544   0.329   1.879

system.time(colSums(mat))
##    user  system elapsed
##   0.126   0.000   0.127
```

Base R provides the following simplified `apply` functions:

- colSums (x, na.rm = FALSE)
- rowSums (x, na.rm = FALSE)
- colMeans(x, na.rm = FALSE)
- rowMeans(x, na.rm = FALSE)

In addition, the following functions are provided through the specified packages:

- miscTools package[177] (note that these functions will work on data frames)
  - colMedians()
  - rowMedians()
- matrixStats package[178] (note that these functions only operate on matrices)
  - colMedians() & rowMedians()
  - colSds() & rowSds()
  - colVar() & rowVar()
  - colRanges() & rowRanges()
  - colQuantiles() & rowQuantiles()
  - along with several additional summary statistic functions

In addition, the summary() function will provide relevant summary statistics over each column of data frames and matrices. Note in the the example that follows that for the first four columns of the iris data set the summary statistics include min, med, mean, max, and 1st & 3rd quantiles. Whereas the last column (Species) only provides the total count since this is a factor variable.

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

[177] https://cran.r-project.org/web/packages/mixtools/index.html

[178] https://cran.r-project.org/web/packages/matrixStats/index.html

Additional Resources

This provides an introduction to control statements in R. However, the following provides additional resources to learn more:

- Tutorial on loops by DataCamp[179]
- Roger Peng's R Programming for Data Science[180]
- Hadley Wickham's Advanced R[181]

[179]https://www.datacamp.com/community/tutorials/tutorial-on-loops-in-r
[180]https://leanpub.com/rprogramming
[181]http://adv-r.had.co.nz/

Simplify Your Code with `%>%`

Removing duplication is an important principle to keep in mind with your code; however, equally important is to keep your code efficient and readable. Efficiency is often accomplished by leveraging functions and control statements in your code. However, efficiency also includes eliminating the creation and saving of unnecessary objects that often result when you are trying to make your code more readable, clear, and explicit. Consequently, writing code that is simple, readable, *and* efficient is often considered contradictory. For this reason, the `magrittr` package is a powerful tool to have in your data wrangling toolkit.

The `magrittr`[182] package was created by Stefan Milton Bache[183] and, in Stefan's words, has two primary aims: "to decrease development time and to improve readability and maintainability of code." Hence, it aims to increase efficiency and improve readability; and in the process it greatly simplifies your code. The following covers the basics of the `magrittr` toolkit.

Pipe (%>%) Operator

The principal function provided by the `magrittr` package is `%>%`, or what's called the "pipe" operator. This operator will forward a value, or the result of an expression, into the next function call/expression. For instance a function to filter data can be written as:

```
filter(data, variable == numeric_value)
```

or

```
data %>% filter(variable == numeric_value)
```

Both functions complete the same task and the benefit of using `%>%` may not be immediately evident; however, when you desire to perform multiple functions its advantage becomes obvious. For instance, if we want to filter some data, group it by categories, summarize it, and then order the summarized results we could write it out three different ways. Don't worry, you'll learn how to operate these specific functions in the next section.

Nested Option:

[182] https://cran.r-project.org/web/packages/magrittr/index.html
[183] https://twitter.com/stefanbache

```r
library(magrittr)
library(dplyr)

arrange(
   summarize(
       group_by(
           filter(mtcars, carb > 1),
           cyl
          ),
       Avg_mpg = mean(mpg)
      ),
   desc(Avg_mpg)
 )
## Source: local data frame [3 x 2]
##
##     cyl Avg_mpg
##   (dbl)   (dbl)
## 1     4   25.90
## 2     6   19.74
## 3     8   15.10
```

This first option is considered a “nested” option such that functions are nested within one another. Historically, this has been the traditional way of integrating code; however, it becomes extremely difficult to read what exactly the code is doing and it also becomes easier to make mistakes when making updates to your code. Although not in violation of the DRY principle, it definitely violates the basic principle of readability and clarity, which makes communication of your analysis more difficult. To make things more readable, people often move to the following approach…

Multiple Object Option:

```r
a <- filter(mtcars, carb > 1)
b <- group_by(a, cyl)
c <- summarise(b, Avg_mpg = mean(mpg))
d <- arrange(c, desc(Avg_mpg))
print(d)
## Source: local data frame [3 x 2]
##
##     cyl Avg_mpg
##   (dbl)   (dbl)
## 1     4   25.90
## 2     6   19.74
## 3     8   15.10
```

This second option helps in making the data wrangling steps more explicit and obvious but definitely violates the DRY principle. By sequencing multiple functions in this way you are likely saving multiple outputs that are not very informative to you or others; rather, the only reason you save them is to insert them into the next function to eventually get the final output you desire. This inevitably creates unnecessary copies and wrecks havoc on properly managing your objects…basically it results in a global environment charlie foxtrot! To provide the same readability (or even better), we can use `%>%` to string these arguments together without unnecessary object creation…

%>% Option:

```
mtcars %>%
        filter(carb > 1) %>%
        group_by(cyl) %>%
        summarise(Avg_mpg = mean(mpg)) %>%
        arrange(desc(Avg_mpg))
## Source: local data frame [3 x 2]
##
##      cyl Avg_mpg
##    (dbl)   (dbl)
## 1      4   25.90
## 2      6   19.74
## 3      8   15.10
```

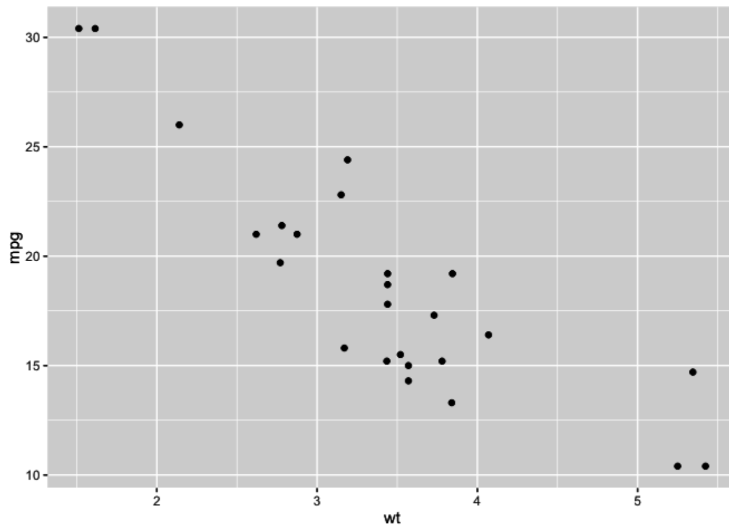
This final option which integrates `%>%` operators makes for more efficient *and* legible code. Its efficient in that it doesn't save unncessary objects (as in option 2) and performs as effectively (as both option 1 & 2) but makes your code more readable in the process. Its legible in that you can read this as you would read normal prose (we read the `%>%` as *"and then"*)- "take `mtcars` *and then* `filter` *and then* `group by` *and then* `summarize` *and then* `arrange`."

And since R is a functional programming language, meaning that everything you do is basically built on functions, you can use the pipe operator to feed into just about any argument call. For example, we can pipe into a linear regression function and then get the summary of the regression parameters. Note in this case I insert "`data = .`" into the `lm()` function. When using the `%>%` operator the default is the argument that you are forwarding will go in as the **first** argument of the function that follows the `%>%`. However, in some functions the argument you are forwarding does not go into the default first position. In these cases, you place "." to signal which argument you want the forwarded expression to go to.

```
mtcars %>%
        filter(carb > 1) %>%
        lm(mpg ~ cyl + hp, data = .) %>%
        summary()
##
## Call:
## lm(formula = mpg ~ cyl + hp, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.6163 -1.4162 -0.1506  1.6181  5.2021
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 35.67647    2.28382  15.621 2.16e-13 ***
## cyl         -2.22014    0.52619  -4.219 0.000353 ***
## hp          -0.01414    0.01323  -1.069 0.296633
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.689 on 22 degrees of freedom
## Multiple R-squared:  0.7601,	Adjusted R-squared:  0.7383
## F-statistic: 34.85 on 2 and 22 DF,  p-value: 1.516e-07
```

You can also use %>% to feed into plots:

```
library(ggplot2)

mtcars %>%
        filter(carb > 1) %>%
        qplot(x = wt, y = mpg, data = .)
```



Piping into a Plot

You will also find that the `%>%` operator is now being built into packages to make programming much easier. For instance, in the section that follows where I illustrate how to reshape and transform your data with the `dplyr` and `tidyr` packages, you will see that the `%>%` operator is already built into these packages. It is also built into the `ggvis` and `dygraphs` packages (visualization packages), the `httr` package (which we covered in the data scraping chapter), and a growing number of newer packages.

Additional Functions

In addition to the `%>%` operator, `magrittr` provides several additional functions which make operations such as addition, multiplication, logical operators, re-naming, etc more pleasant when composing chains using the `%>%` operator. Some examples follow but you can see the current list of the available aliased functions by typing `?magrittr::add` in your console.

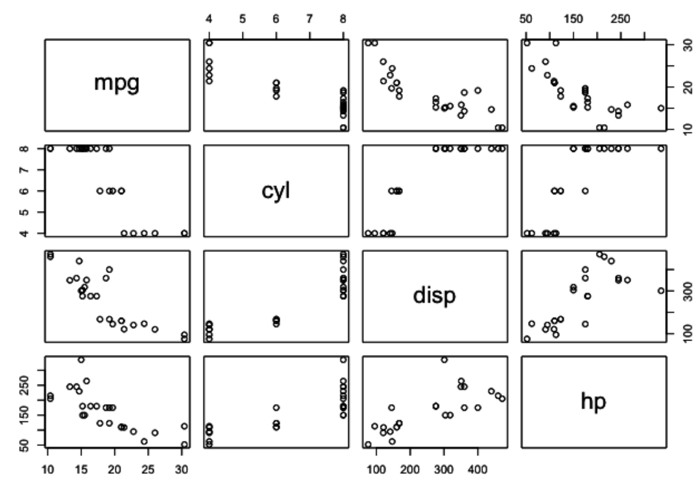
```
# subset with extract
mtcars %>%
        extract(, 1:4) %>%
        head
##                    mpg cyl disp  hp
## Mazda RX4         21.0   6  160 110
## Mazda RX4 Wag     21.0   6  160 110
## Datsun 710        22.8   4  108  93
## Hornet 4 Drive    21.4   6  258 110
## Hornet Sportabout 18.7   8  360 175
## Valiant           18.1   6  225 105
```

```
# add, subtract, multiply, divide and other operations are available
mtcars %>%
        extract(, "mpg") %>%
        multiply_by(5)
##  [1] 105.0 105.0 114.0 107.0  93.5  90.5  71.5 122.0 114.0  96.0  89.0
## [12]  82.0  86.5  76.0  52.0  52.0  73.5 162.0 152.0 169.5 107.5  77.5
## [23]  76.0  66.5  96.0 136.5 130.0 152.0  79.0  98.5  75.0 107.0

# logical assessments and filters are available
mtcars %>%
        extract(, "cyl") %>%
        equals(4)
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE

# renaming columns and rows is available
mtcars %>%
        head %>%
        set_colnames(paste("Col", 1:11, sep = ""))
##                   Col1 Col2 Col3 Col4 Col5  Col6  Col7 Col8 Col9 Col10
## Mazda RX4         21.0    6  160  110 3.90 2.620 16.46    0    1     4
## Mazda RX4 Wag     21.0    6  160  110 3.90 2.875 17.02    0    1     4
## Datsun 710        22.8    4  108   93 3.85 2.320 18.61    1    1     4
## Hornet 4 Drive    21.4    6  258  110 3.08 3.215 19.44    1    0     3
## Hornet Sportabout 18.7    8  360  175 3.15 3.440 17.02    0    0     3
## Valiant           18.1    6  225  105 2.76 3.460 20.22    1    0     3
##                   Col11
## Mazda RX4             4
## Mazda RX4 Wag         4
## Datsun 710            1
## Hornet 4 Drive        1
## Hornet Sportabout     2
## Valiant               1
```

Additional Pipe Operators

`magrittr` also offers some alternative pipe operators. Some functions, such as plotting functions, will cause the string of piped arguments to terminate. The tee (`%T>%`) operator allows you to continue piping functions that normally cause termination.

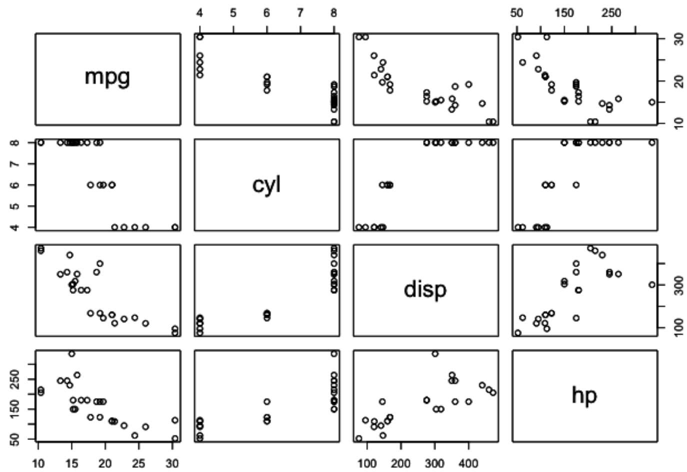
```r
# normal piping terminates with the plot() function resulting in
# NULL results for the summary() function
mtcars %>%
        filter(carb > 1) %>%
        extract(, 1:4) %>%
        plot() %>%
        summary()
```



Regular Pipe Operator Terminates String of Functions at a Plot

```
## Length  Class   Mode
##      0   NULL   NULL
```

```r
# inserting %T>% allows you to plot and perform the functions that
# follow the plotting function
mtcars %>%
        filter(carb > 1) %>%
        extract(, 1:4) %T>%
        plot() %>%
        summary()
```



Tee Operator Allows You to Pipe Through a Plot

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.00   Min.   : 75.7   Min.   : 52.0  
##  1st Qu.:15.20   1st Qu.:6.00   1st Qu.:146.7   1st Qu.:110.0  
##  Median :17.80   Median :8.00   Median :275.8   Median :175.0  
##  Mean   :18.62   Mean   :6.64   Mean   :257.7   Mean   :163.7  
##  3rd Qu.:21.00   3rd Qu.:8.00   3rd Qu.:351.0   3rd Qu.:205.0  
##  Max.   :30.40   Max.   :8.00   Max.   :472.0   Max.   :335.0
```

The compound assignment `%<>%` operator is used to update a value by first piping it into one or more expressions, and then assigning the result. For instance, let's say you want to transform the `mpg` variable in the `mtcars` data frame to a square root measurement. Using `%<>%` will perform the functions to the right of `%<>%` and save the changes these functions perform to the variable or data frame called to the left of `%<>%`.

```
# note that mpg is in its typical measurement
head(mtcars)
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

# we can square root mpg and save this change using %<>%
mtcars$mpg %<>% sqrt
```

```
head(mtcars)
##                        mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         4.582576   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     4.582576   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        4.774935   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    4.626013   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 4.324350   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           4.254409   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Some functions (e.g. lm, aggregate, cor) have a data argument, which allows the direct use of names inside the data as part of the call. The exposition (`%$%`) operator is useful when you want to pipe a dataframe, which may contain many columns, into a function that is only applied to some of the columns. For example, the correlation (`cor`) function only requires an `x` and `y` argument so if you pipe the `mtcars` data into the `cor` function using `%>%` you will get an error because `cor` doesn't know how to handle `mtcars`. However, using `%$%` allows you to say "take this dataframe and then perform `cor()` on these specified columns within `mtcars`."

```
# regular piping results in an error
mtcars %>%
        subset(vs == 0) %>%
        cor(mpg, wt)
## Error in pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs", : \
object 'wt' not found

# using %$% allows you to specify variables of interest
mtcars %>%
        subset(vs == 0) %$%
        cor(mpg, wt)
## [1] -0.830671
```

Additional Resources

The `magrittr` package and its pipe operators are a great tool for making your code simple, efficient, and readable. There are limitations, or at least suggestions, on when and how you should use the operators. Garrett Grolemund and Hadley Wickham offer some advice on the proper use of pipe operators in their R for Data Science[184] book. However, the `%>%` has greatly transformed our ability to write "simplified" code in R. As the pipe gains in popularity you will likely find it in more future packages and being familiar will likely result in better communication of your code.

Some additional resources regarding `magrittr` and the pipe operators you may find useful:

[184] http://r4ds.had.co.nz/

- The `magrittr` vignette (`vignette("magrittr")`) in your console) provides additional examples of using pipe operators and functions provided by `magrittr`.
- A blog post[185] by Stefan Milton Bache regarding the past, present and future of `magrittr`
- `magrittr` questions[186] on Stack Overflow
- The `ensurer`[187] package, also written by Stefan Milton Bache[188], provides a useful way of verifying and validating data outputs in a sequence of pipe operators.

---

[185] http://www.r-bloggers.com/simpler-r-coding-with-pipes-the-present-and-future-of-the-magrittr-package/
[186] http://stackoverflow.com/questions/tagged/magrittr
[187] https://cran.r-project.org/web/packages/ensurer/vignettes/ensurer.html
[188] https://twitter.com/stefanbache

Shaping & Transforming Your Data with R

> *Up to 80% of data analysis is spent on the process of cleaning and preparing data.* - cf. Wickham, 2014[189] and Dasu and Johnson, 2003[190]

A tremendous amount of time is spent on fundamental preprocessing tasks to get your data into the right form in order to feed it into the visualization and modeling stages. This typically requires a large amount of reshaping and transformation of your data. Although many fundamental data processing functions exist in R, they have been a bit convoluted to date and have lacked consistent coding and the ability to easily flow together. The RStudio team[191] has been driving a lot of new packages to collate data management tasks and better integrate them with other analysis activities. As a result, a lot of data processing tasks are becoming packaged in more cohesive and consistent ways which leads to more efficient code and easier to read syntax. This section covers two of these packages: `tidyr` and `dplyr`.

In this section, I start by providing a fundamental understanding of tidy data followed by demonstrating how to to use `tidyr` to turn wide data to long, long data to wide, splitting and combining variables, along with illustrating some lesser-known functions. Subsequently, I provide an introduction to the `dplyr` package by covering seven primary functions `dplyr` provides for simplified data transformation and manipulation. This includes tasks such as filtering, summarizing, ordering, joining, and much more. Understanding and using these two packages will help to significantly reduce the time you spend on the data wrangling process.

[189] https://www.jstatsoft.org/article/view/v059i10

[190] http://onlinelibrary.wiley.com/doi/10.1002/0471448354.ch4/summary

[191] https://www.rstudio.com/home/

Reshaping Your Data with `tidyr`

> *"Cannot emphasize enough how much time you save by putting analysis efforts into tidying data first."* - Hilary Parker

Jenny Bryan[192] stated that "classroom data are like teddy bears and real data are like a grizzley bear with salmon blood dripping out its mouth." In essence, she was getting to the point that often when we learn how to perform a modeling approach in the classroom, the data used is provided in a format that appropriately feeds into the modeling tool of choice. In reality, datasets are messy and "every messy dataset is messy in its own way."[193] The concept of "tidy data" was established by Hadley Wickham and represents "standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning)."[194] The objective should always to be to get a dataset into a tidy form which consists of:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

To create tidy data you need to be able to reshape your data; preferably via efficient and simple code. To help with this process Hadley created the `tidyr`[195] package. This chapter covers the basics of `tidyr` to help you reshape your data as necessary. I demonstrate how to turn wide data to long, long data to wide, splitting and combining variables, and finally I will cover some lesser known functions in `tidyr` that are useful. Note that throughout I use the `%>%` operator we covered in the last chapter. Although not required, the `tidyr` package has the `%>%` operator baked in to its functionality, which allows you to sequence multiple tidy functions together.

Making wide data long

There are times when our data is considered "wide" or "unstacked" and a common attribute/variable of concern is spread out across columns. To reformat the data such that these common attributes are *gathered* together as a single variable, the `gather()` function will take multiple columns and collapse them into key-value pairs, duplicating all other columns as needed.

For example, let's say we have the given data frame.

[192] https://twitter.com/JennyBryan
[193] Wickham, H. (2014). "Tidy data." Journal of Statistical Software, 59(10). [document]
[194] Ibid
[195] https://cran.r-project.org/web/packages/tidyr/index.html

```
library(dplyr) # I'm using dplyr just to create the data frame with tbl_df()

wide <- tbl_df(read.table(header = TRUE, text = "
   Group    Year   Qtr.1  Qtr.2  Qtr.3  Qtr.4
    1       2006   15     16     19     17
    1       2007   12     13     27     23
    1       2008   22     22     24     20
    1       2009   10     14     20     16
    2       2006   12     13     25     18
    2       2007   16     14     21     19
    2       2008   13     11     29     15
    2       2009   23     20     26     20
    3       2006   11     12     22     16
    3       2007   13     11     27     21
    3       2008   17     12     23     19
    3       2009   14     9      31     24
"))
```

This data is considered wide since the *time* variable (represented as quarters) is structured such that each quarter represents a variable. To re-structure the time component as an individual variable, we can *gather* each quarter within one column variable and also *gather* the values associated with each quarter in a second column variable.

```
library(tidyr)

long <- wide %>% gather(Quarter, Revenue, Qtr.1:Qtr.4)

head(long, 15) # note, for brevity, I only show the first 15 observations
## Source: local data frame [15 x 4]
##
##    Group  Year Quarter Revenue
##    (int) (int)  (fctr)   (int)
## 1      1  2006   Qtr.1      15
## 2      1  2007   Qtr.1      12
## 3      1  2008   Qtr.1      22
## 4      1  2009   Qtr.1      10
## 5      2  2006   Qtr.1      12
## 6      2  2007   Qtr.1      16
## 7      2  2008   Qtr.1      13
## 8      2  2009   Qtr.1      23
## 9      3  2006   Qtr.1      11
## 10     3  2007   Qtr.1      13
```

```
## 11      3  2008   Qtr.1      17
## 12      3  2009   Qtr.1      14
## 13      1  2006   Qtr.2      16
## 14      1  2007   Qtr.2      13
## 15      1  2008   Qtr.2      22
```

It's important to note that there is flexibility in how you specify the columns you would like to gather. These all produce the same results:

```
wide %>% gather(Quarter, Revenue, Qtr.1:Qtr.4)
wide %>% gather(Quarter, Revenue, -Group, -Year)
wide %>% gather(Quarter, Revenue, 3:6)
wide %>% gather(Quarter, Revenue, Qtr.1, Qtr.2, Qtr.3, Qtr.4)
```

Making long data wide

There are also times when we are required to turn long formatted data into wide formatted data. As a complement to `gather()`, the `spread()` function spreads a key-value pair across multiple columns. So now let's take our `long` data frame from above and and turn the `Quarter` variable into column headings and spread the `Revenue` values across the quarters they are related to.

```
back2wide <- long %>% spread(Quarter, Revenue)
back2wide
## Source: local data frame [12 x 6]
##
##    Group  Year Qtr.1 Qtr.2 Qtr.3 Qtr.4
##    (int) (int) (int) (int) (int) (int)
## 1      1  2006    15    16    19    17
## 2      1  2007    12    13    27    23
## 3      1  2008    22    22    24    20
## 4      1  2009    10    14    20    16
## 5      2  2006    12    13    25    18
## 6      2  2007    16    14    21    19
## 7      2  2008    13    11    29    15
## 8      2  2009    23    20    26    20
## 9      3  2006    11    12    22    16
## 10     3  2007    13    11    27    21
## 11     3  2008    17    12    23    19
## 12     3  2009    14     9    31    24
```

Splitting a single column into multiple columns

Many times a single column variable will capture multiple variables, or even parts of a variable you just don't care about. This is exemplified in the following `messy_df` data frame. Here, the `Grp_Ind` variable combines an individual variable (a, b, c) with the group variable (1, 2, 3), the `Yr_Mo` variable combines a year variable with a month variable, etc. In each case there may be a purpose for separating parts of these columns into *separate* variables.

```
messy_df
##   Grp_Ind    Yr_Mo       City_State Extra_variable
## 1     1.a 2006_Jan      Dayton (OH)   XX01person_1
## 2     1.b 2006_Feb Grand Forks (ND)   XX02person_2
## 3     1.c 2006_Mar       Fargo (ND)   XX03person_3
## 4     2.a 2007_Jan   Rochester (MN)   XX04person_4
```

This can be accomplished using the `separate()` function which turns a single character column into multiple columns. Additional arguments provide some flexibility with separating columns.

```
# separate Grp_Ind column into two variables named "Grp" & "Ind"
messy_df %>% separate(col = Grp_Ind, into = c("Grp", "Ind"))
##   Grp Ind    Yr_Mo       City_State Extra_variable
## 1   1   a 2006_Jan      Dayton (OH)   XX01person_1
## 2   1   b 2006_Feb Grand Forks (ND)   XX02person_2
## 3   1   c 2006_Mar       Fargo (ND)   XX03person_3
## 4   2   a 2007_Jan   Rochester (MN)   XX04person_4

# default separater is any non alpha-numeric character but you can
# specify the specific character to separate at
messy_df %>% separate(col = Extra_variable, into = c("X", "Y"), sep = "_")
##   Grp_Ind    Yr_Mo       City_State          X Y
## 1     1.a 2006_Jan      Dayton (OH) XX01person 1
## 2     1.b 2006_Feb Grand Forks (ND) XX02person 2
## 3     1.c 2006_Mar       Fargo (ND) XX03person 3
## 4     2.a 2007_Jan   Rochester (MN) XX04person 4

# you can keep the original column that you are separating
messy_df %>% separate(col = Grp_Ind, into = c("Grp", "Ind"), remove = FALSE)
##   Grp_Ind Grp Ind    Yr_Mo       City_State Extra_variable
## 1     1.a   1   a 2006_Jan      Dayton (OH)   XX01person_1
## 2     1.b   1   b 2006_Feb Grand Forks (ND)   XX02person_2
## 3     1.c   1   c 2006_Mar       Fargo (ND)   XX03person_3
## 4     2.a   2   a 2007_Jan   Rochester (MN)   XX04person_4
```

Combining multiple columns into a single column

Similarly, there are times when we would like to combine the values of two variables. As a compliment to `separate()`, the `unite()` function is a convenient function to paste together multiple variable values into one. Consider the following data frame that has separate date variables. To perform time series analysis or for visualizations we may desire to have a single date column.

```
expenses <- tbl_df(read.table(header = TRUE, text = "
        Year    Month    Day    Expense
        2015       01     01        500
        2015       02     05         90
        2015       02     22        250
        2015       03     10        325
"))
```

To perform time series analysis or for visualizations we may desire to have a single date column. We can accomplish this by *uniting* these columns into one variable with `unite()`.

```
# default separator when uniting is "_"
expenses %>% unite(col = "Date", c(Year, Month, Day))
## Source: local data frame [4 x 2]
##
##          Date Expense
##         (chr)   (int)
## 1   2015_1_1     500
## 2   2015_2_5      90
## 3 2015_2_22     250
## 4 2015_3_10     325

# specify sep argument to change separater
expenses %>% unite(col = "Date", c(Year, Month, Day), sep = "-")
## Source: local data frame [4 x 2]
##
##          Date Expense
##         (chr)   (int)
## 1   2015-1-1     500
## 2   2015-2-5      90
## 3 2015-2-22     250
## 4 2015-3-10     325
```

Additional tidyr functions

The previous four functions (gather, spread, separate and unite) are the primary functions you will find yourself using on a continuous basis; however, there are some handy functions that are lesser known with the tidyr package.

```
expenses <- tbl_df(read.table(header = TRUE, text = "
        Dept     Year   Month   Day         Cost
           A     2015      01    01      $500.00
          NA       NA      02    05       $90.00
          NA       NA      02    22    $1,250.45
          NA       NA      03    NA      $325.10
           B       NA      01    02      $260.00
          NA       NA      02    05       $90.00
", stringsAsFactors = FALSE))
```

Often Excel reports will not repeat certain variables. When we read these reports in, the empty cells are typically filled in with NA such as in the Dept and Year columns of our expense data frame. We can fill these values in with the previous entry using fill().

```
expenses %>% fill(Dept, Year)
## Source: local data frame [6 x 5]
##
##    Dept  Year Month   Day      Cost
##   (chr) (int) (int) (int)     (chr)
## 1     A  2015     1     1   $500.00
## 2     A  2015     2     5    $90.00
## 3     A  2015     2    22 $1,250.45
## 4     A  2015     3    NA   $325.10
## 5     B  2015     1     2   $260.00
## 6     B  2015     2     5    $90.00
```

Also, sometimes accounting values in Excel spreadsheet get read in as a character value, which is the case for the Cost variable. We may wish to extract only the numeric part of this regular expression, which can be done with extract_numeric(). Note that extract_numeric works on a single variable so when you pipe the expense data frame into the function you need to use %$% operator as discussed in the last chapter.

```
library(magrittr)

expenses %$% extract_numeric(Cost)
## [1]  500.00   90.00 1250.45  325.10  260.00   90.00

# you can use this to convert and save the Cost column to a
# numeric variable
expenses$Cost <- expenses %$% extract_numeric(Cost)
expenses
## Source: local data frame [6 x 5]
##
##    Dept  Year Month   Day    Cost
##   (chr) (int) (int) (int)   (dbl)
## 1     A  2015     1     1  500.00
## 2    NA    NA     2     5   90.00
## 3    NA    NA     2    22 1250.45
## 4    NA    NA     3    NA  325.10
## 5     B    NA     1     2  260.00
## 6    NA    NA     2     5   90.00
```

You can also easily replace missing (or `NA`) values with a specified value:

```
library(magrittr)

# replace the missing Day value
expenses %>% replace_na(replace = list(Day = "unknown"))
## Source: local data frame [6 x 5]
##
##    Dept  Year Month     Day    Cost
##   (chr) (int) (int)   (chr)   (dbl)
## 1     A  2015     1       1  500.00
## 2    NA    NA     2       5   90.00
## 3    NA    NA     2      22 1250.45
## 4    NA    NA     3 unknown  325.10
## 5     B    NA     1       2  260.00
## 6    NA    NA     2       5   90.00

# replace both the missing Day and Year values
expenses %>% replace_na(replace = list(Year = 2015, Day = "unknown"))
## Source: local data frame [6 x 5]
##
##    Dept  Year Month     Day    Cost
```

```
##    (chr) (dbl) (int)    (chr)   (dbl)
## 1      A  2015     1        1  500.00
## 2     NA  2015     2        5   90.00
## 3     NA  2015     2       22 1250.45
## 4     NA  2015     3  unknown  325.10
## 5      B  2015     1        2  260.00
## 6     NA  2015     2        5   90.00
```

Sequencing your tidyr operations

Since the %>% operator is embedded in tidyr, we can string multiple operations together to efficiently tidy data *and* make the process easy to read and follow. To illustrate, let's use the following data, which has multiple *messy* attributes.

```
a_mess <- tbl_df(read.table(header = TRUE, text = "
   Dep_Unt    Year      Q1      Q2      Q3      Q4
    A.1       2006      15      NA      19      17
    B.1         NA      12      13      27      23
    A.2         NA      22      22      24      20
    B.2         NA      12      13      25      18
    A.1       2007      16      14      21      19
    B.2         NA      13      11      16      15
    A.2         NA      23      20      26      20
    B.2         NA      11      12      22      16
"))
```

In this case, a tidy dataset should result in columns of Dept, Unit, Year, Quarter, and Cost. Furthermore, we want to fill in the year column where NAs currently exist. And we'll assume that we know the missing value that exists in the Q2 column, and we'd like to update it.

```
a_mess %>%
        fill(Year) %>%
        gather(Quarter, Cost, Q1:Q4) %>%
        separate(Dep_Unt, into = c("Dept", "Unit")) %>%
        replace_na(replace = list(Cost = 17))
## Source: local data frame [32 x 5]
##
##      Dept  Unit  Year Quarter  Cost
##     (chr) (chr) (int)  (fctr) (dbl)
## 1       A     1  2006      Q1    15
## 2       B     1  2006      Q1    12
```

```
## 3      A     2  2006     Q1    22
## 4      B     2  2006     Q1    12
## 5      A     1  2007     Q1    16
## 6      B     2  2007     Q1    13
## 7      A     2  2007     Q1    23
## 8      B     2  2007     Q1    11
## 9      A     1  2006     Q2    17
## 10     B     1  2006     Q2    13
## ..   ...   ...   ...    ...   ...
```

Additional resources

This chapter covers most, but not all, of what `tidyr` provides. There are several other resources you can check out to learn more.

- Data wrangling presentation[196] I gave at Miami University
- Hadley Wickham's tidy data paper[197]
- `tidyr` reference manual[198]
- R Studio's Data wrangling with R and RStudio webinar[199]
- R Studio's Data wrangling GitHub repository[200]
- R Studio's Data wrangling cheat sheet[201]

[196] http://bradleyboehmke.github.io/2015/10/data-wrangling-presentation.html
[197] http://jstatsoft.org/v59/i10
[198] https://cran.r-project.org/web/packages/tidyr/tidyr.pdf
[199] http://www.rstudio.com/resources/webinars/
[200] https://github.com/rstudio/webinars/blob/master/2015-01/wrangling-webinar.pdf
[201] http://www.rstudio.com/resources/cheatsheets/

Transforming Your Data with `dplyr`

Transforming your data is a basic part of data wrangling. This can include filtering, summarizing, and ordering your data by different means. This also includes combining disperate data sets, creating new variables, and many other manipulation tasks. Although many fundamental data transformation and manipulation functions exist in R, historically they have been a bit convoluted and lacked a consistent and cohesive code structure. Consequently, Hadley Wickham developed the very popular `dplyr` package to make these data processing tasks more efficient along with a syntax that is consistent and easier to remember and read.

`dplyr`'s roots originate in the popular `plyr`[202] package, also produced by Hadley Wickham. `plyr` covers data transformation and manipulation for a range of data structures (data frames, lists, arrays) whereas `dplyr` is focused on transformation and manipulation of data frames. And since the bulk of data analysis leverages data frames I am going to focus on `dplyr`. Even so, `dplyr` offers far more functionality than I can cover in one chapter. Consequently, I'm going to cover the seven primary functions `dplyr` provides for data transformation and manipulation. Throughout, I also mention additional, useful functions that can be integrated with these functions. The full list of capabilities can be found in the `dplyr` reference manual[203]; I highly recommend going through it as there are many great functions provided by `dplyr` that I will not cover here. Also, similar to `tidyr`, `dplyr` has the `%>%` operator baked in to its functionality.

For most of these examples we'll use the following census data[204] which includes the K-12 public school expenditures by state. This dataframe currently is 50x16 and includes expenditure data for 14 unique years (50 states and has data through year 2011). Here I only show you a subset of the data.

```
##   Division      State   X1980    X1990    X2000    X2001    X2002    X2003
## 1        6    Alabama 1146713  2275233  4176082  4354794  4444390  4657643
## 2        9     Alaska  377947   828051  1183499  1229036  1284854  1326226
## 3        8    Arizona  949753  2258660  4288739  4846105  5395814  5892227
## 4        7   Arkansas  666949  1404545  2380331  2505179  2822877  2923401
## 5        9 California 9172158 21485782 38129479 42908787 46265544 47983402
## 6        8   Colorado 1243049  2451833  4401010  4758173  5151003  5551506
##      X2004    X2005    X2006    X2007    X2008    X2009    X2010    X2011
## 1  4812479  5164406  5699076  6245031  6832439  6683843  6670517  6592925
## 2  1354846  1442269  1529645  1634316  1918375  2007319  2084019  2201270
## 3  6071785  6579957  7130341  7815720  8403221  8726755  8482552  8340211
## 4  3109644  3546999  3808011  3997701  4156368  4240839  4459910  4578136
```

[202] https://cran.r-project.org/web/packages/plyr/index.html

[203] https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

[204] http://www.census.gov/en.html

```
## 5 49215866 50918654 53436103 57352599 61570555 60080929 58248662 57526835
## 6  5666191  5994440  6368289  6579053  7338766  7187267  7429302  7409462
```

Selecting variables of interest

When working with a sizable dataframe, often we desire to only assess specific variables. The `select()` function allows you to select and/or rename variables. Let's say our goal is to only assess the 5 most recent years worth of expenditure data. Applying the `select()` function we can *select* only the variables of concern.

```
sub_exp <- expenditures %>% select(Division, State, X2007:X2011)
head(sub_exp)  # for brevity only display first 6 rows
##   Division      State    X2007    X2008    X2009    X2010    X2011
## 1        6    Alabama  6245031  6832439  6683843  6670517  6592925
## 2        9     Alaska  1634316  1918375  2007319  2084019  2201270
## 3        8    Arizona  7815720  8403221  8726755  8482552  8340211
## 4        7   Arkansas  3997701  4156368  4240839  4459910  4578136
## 5        9 California 57352599 61570555 60080929 58248662 57526835
## 6        8   Colorado  6579053  7338766  7187267  7429302  7409462
```

We can also apply some of the special functions within `select()`. For instance we can select all variables that start with 'X' (`?select` to see the available functions):

```
expenditures %>%
        select(starts_with("X")) %>%
        head
##     X1980    X1990    X2000    X2001    X2002    X2003    X2004    X2005
## 1 1146713  2275233  4176082  4354794  4444390  4657643  4812479  5164406
## 2  377947   828051  1183499  1229036  1284854  1326226  1354846  1442269
## 3  949753  2258660  4288739  4846105  5395814  5892227  6071785  6579957
## 4  666949  1404545  2380331  2505179  2822877  2923401  3109644  3546999
## 5 9172158 21485782 38129479 42908787 46265544 47983402 49215866 50918654
## 6 1243049  2451833  4401010  4758173  5151003  5551506  5666191  5994440
##      X2006    X2007    X2008    X2009    X2010    X2011
## 1  5699076  6245031  6832439  6683843  6670517  6592925
## 2  1529645  1634316  1918375  2007319  2084019  2201270
## 3  7130341  7815720  8403221  8726755  8482552  8340211
## 4  3808011  3997701  4156368  4240839  4459910  4578136
## 5 53436103 57352599 61570555 60080929 58248662 57526835
## 6  6368289  6579053  7338766  7187267  7429302  7409462
```

You can also de-select variables by using "-" prior to name or function. The following produces the inverse of functions above:

```
expenditures %>% select(-X1980:-X2006)
expenditures %>% select(-starts_with("X"))
```

And for convenience, you can rename selected variables with two options:

```
# select and rename a single column
expenditures %>% select(Yr_1980 = X1980)

# Select and rename the multiple variables with an "X" prefix:
expenditures %>% select(Yr_ = starts_with("X"))

# keep all variables and rename a single variable
expenditures %>% rename(`2011` = X2011)
```

Filtering rows

Filtering data is a common task to identify/select observations in which a particular variable matches a specific value/condition. The `filter()` function provides this capability. Continuing with our `sub_exp` dataframe which includes only the recent 5 years worth of expenditures, we can filter by `Division`:

```
sub_exp %>% filter(Division == 3)
##   Division     State    X2007    X2008    X2009    X2010    X2011
## 1        3  Illinois 20326591 21874484 23495271 24695773 24554467
## 2        3   Indiana  9497077  9281709  9680895  9921243  9687949
## 3        3  Michigan 17013259 17053521 17217584 17227515 16786444
## 4        3      Ohio 18251361 18892374 19387318 19801670 19988921
## 5        3 Wisconsin  9029660  9366134  9696228  9966244 10333016
```

We can apply multiple logic rules in the `filter()` function such as:

```
<    Less than                    !=      Not equal to
>    Greater than                 %in%    Group membership
==   Equal to                     is.na   is NA
<=   Less than or equal to        !is.na  is not NA
>=   Greater than or equal to     &,|,!   Boolean operators
```

For instance, we can filter for Division 3 and expenditures in 2011 that were greater than \$10B. This results in Indiana being excluded since it falls within division 3 and its expenditures were < \$10B*(FYI - the raw census data are reported in units of \$1,000)*.

```
# Raw census data are in units of $1,000
sub_exp %>% filter(Division == 3, X2011 > 10000000)
##   Division     State    X2007    X2008    X2009    X2010    X2011
## 1        3  Illinois 20326591 21874484 23495271 24695773 24554467
## 2        3  Michigan 17013259 17053521 17217584 17227515 16786444
## 3        3      Ohio 18251361 18892374 19387318 19801670 19988921
## 4        3 Wisconsin  9029660  9366134  9696228  9966244 10333016
```

There are additional filtering and subsetting functions that are quite useful:

```
# remove duplicate rows
sub_exp %>% distinct()

# random sample, 50% sample size without replacement
sub_exp %>% sample_frac(size = 0.5, replace = FALSE)

# random sample of 10 rows with replacement
sub_exp %>% sample_n(size = 10, replace = TRUE)

# select rows 3-5
sub_exp %>% slice(3:5)

# select top n entries - in this case ranks variable X2011 and selects
# the rows with the top 5 values
sub_exp %>% top_n(n = 5, wt = X2011)
```

Grouping data by categorical variables

Often, observations are nested within groups or categories and our goal is to perform statistical analysis both at the observation level and also at the group level. The `group_by()` function allows us to create these categorical groupings.

The `group_by()` function is a *silent* function in which no observable manipulation of the data is performed as a result of applying the function. Rather, the only change you'll notice is, when you print the dataframe you will notice underneath the *Source* information and prior to the actual dataframe, an indicator of what variable the data is grouped by will be provided. In the example that follows you'll notice that we grouped by `Division` and there are nine categories for this variable. The real magic of the `group_by()` function comes when we perform summary statistics which we will cover shortly.

```
group.exp <- sub_exp %>% group_by(Division)

group.exp
## Source: local data frame [50 x 7]
## Groups: Division [9]
##
##    Division       State    X2007    X2008    X2009    X2010    X2011
##       (int)       (chr)    (int)    (int)    (int)    (int)    (int)
## 1         6     Alabama  6245031  6832439  6683843  6670517  6592925
## 2         9      Alaska  1634316  1918375  2007319  2084019  2201270
## 3         8     Arizona  7815720  8403221  8726755  8482552  8340211
## 4         7    Arkansas  3997701  4156368  4240839  4459910  4578136
## 5         9  California 57352599 61570555 60080929 58248662 57526835
## 6         8    Colorado  6579053  7338766  7187267  7429302  7409462
## 7         1 Connecticut  7855459  8336789  8708294  8853337  9094036
## 8         5    Delaware  1437707  1489594  1518786  1549812  1613304
## 9         5     Florida 22887024 24224114 23328028 23349314 23870090
## 10        5     Georgia 14828715 16030039 15976945 15730409 15527907
## ..      ...         ...      ...      ...      ...      ...      ...

# we can ungroup our data with
ungroup(group.exp)
## Source: local data frame [50 x 7]
##
##    Division       State    X2007    X2008    X2009    X2010    X2011
##       (int)       (chr)    (int)    (int)    (int)    (int)    (int)
## 1         6     Alabama  6245031  6832439  6683843  6670517  6592925
## 2         9      Alaska  1634316  1918375  2007319  2084019  2201270
## 3         8     Arizona  7815720  8403221  8726755  8482552  8340211
## 4         7    Arkansas  3997701  4156368  4240839  4459910  4578136
## 5         9  California 57352599 61570555 60080929 58248662 57526835
## 6         8    Colorado  6579053  7338766  7187267  7429302  7409462
## 7         1 Connecticut  7855459  8336789  8708294  8853337  9094036
## 8         5    Delaware  1437707  1489594  1518786  1549812  1613304
## 9         5     Florida 22887024 24224114 23328028 23349314 23870090
## 10        5     Georgia 14828715 16030039 15976945 15730409 15527907
## ..      ...         ...      ...      ...      ...      ...      ...
```

Performing summary statistics on variables

Obviously the goal of all this data *wrangling* is to be able to perform statistical analysis on our data. The `summarise()` function allows us to perform the majority of summary statistics when performing exploratory data analysis.

Lets get the mean expenditure value across all states in 2011:

```
sub_exp %>% summarise(Mean_2011 = mean(X2011))
##   Mean_2011
## 1  10513678
```

Not too bad, lets get some more summary stats:

```
sub_exp %>% summarise(Min = min(X2011, na.rm = TRUE),
                      Median = median(X2011, na.rm = TRUE),
                      Mean = mean(X2011, na.rm = TRUE),
                      Var = var(X2011, na.rm = TRUE),
                      SD = sd(X2011, na.rm = TRUE),
                      Max = max(X2011, na.rm = TRUE),
                      N = n())
##       Min  Median     Mean         Var       SD      Max  N
## 1 1049772 6527404 10513678 1.48619e+14 12190938 57526835 50
```

This information is useful, but being able to compare summary statistics at multiple levels is when you really start to gather some insights. This is where the `group_by()` function comes in. First, let's group by `Division` and see how the different regions compared in by 2010 and 2011.

```
sub_exp %>%
        group_by(Division)%>%
        summarise(Mean_2010 = mean(X2010, na.rm = TRUE),
                  Mean_2011 = mean(X2011, na.rm = TRUE))
## Source: local data frame [9 x 3]
##
##   Division Mean_2010 Mean_2011
##      (int)     (dbl)     (dbl)
## 1        1   5121003   5222277
## 2        2  32415457  32877923
## 3        3  16322489  16270159
## 4        4   4672332   4672687
## 5        5  10975194  11023526
## 6        6   6161967   6267490
## 7        7  14916843  15000139
## 8        8   3894003   3882159
## 9        9  15540681  15468173
```

Now we're starting to see some differences pop out. How about we compare states within a Division? We can start to apply multiple functions we've learned so far to get the 5 year average for each state within Division 3.

```
library(tidyr)

sub_exp %>%
        gather(Year, Expenditure, X2007:X2011) %>%   # turn wide data to long
        filter(Division == 3) %>%                    # only assess Division 3
        group_by(State) %>%                          # summarize data by state
        summarise(Mean = mean(Expenditure),          # calculate mean & SD
                  SD = sd(Expenditure))
## Source: local data frame [5 x 3]
##
##       State     Mean        SD
##       (chr)    (dbl)     (dbl)
## 1  Illinois 22989317 1867527.7
## 2   Indiana  9613775  238971.6
## 3  Michigan 17059665  180245.0
## 4      Ohio 19264329  705930.2
## 5 Wisconsin  9678256  507461.2
```

There are several built-in summary functions in dplyr as displayed below. You can also build in your own functions as well.

first()	First value of a vector	min()	Min value in vector
last()	Last value of a vector	max()	Max value in vector
nth()	Nth value of a vector	mean()	Mean value of vector
n()	# of values in a vector	median()	Median value of vector
n_distinct()	# of distinct values	var()	Variance of vector
IQR()	IQR of a vector	sd()	St. dev. of vector

Built-in Summary Functions

Arranging variables by value

Sometimes we wish to view observations in rank order for a particular variable(s). The arrange() function allows us to order data by variables in accending or descending order. Let's say we want to assess the average expenditures by division. We could apply the arrange() function at the end to order the divisions from lowest to highest expenditure for 2011. This makes it easier to see the significant differences between Divisions 8,4,1 & 6 as compared to Divisions 5,7,9,3 & 2.

```
sub_exp %>%
        group_by(Division)%>%
        summarise(Mean_2010 = mean(X2010, na.rm = TRUE),
                  Mean_2011 = mean(X2011, na.rm = TRUE)) %>%
        arrange(Mean_2011)
## Source: local data frame [9 x 3]
##
##   Division Mean_2010 Mean_2011
##      (int)     (dbl)     (dbl)
## 1        8   3894003   3882159
## 2        4   4672332   4672687
## 3        1   5121003   5222277
## 4        6   6161967   6267490
## 5        5  10975194  11023526
## 6        7  14916843  15000139
## 7        9  15540681  15468173
## 8        3  16322489  16270159
## 9        2  32415457  32877923
```

We can also apply a *descending* argument to rank-order from highest to lowest. The following shows the same data but in descending order by applying `desc()` within the `arrange()` function.

```
sub_exp %>%
        group_by(Division)%>%
        summarise(Mean_2010 = mean(X2010, na.rm = TRUE),
                  Mean_2011 = mean(X2011, na.rm = TRUE)) %>%
        arrange(desc(Mean_2011))
## Source: local data frame [9 x 3]
##
##   Division Mean_2010 Mean_2011
##      (int)     (dbl)     (dbl)
## 1        2  32415457  32877923
## 2        3  16322489  16270159
## 3        9  15540681  15468173
## 4        7  14916843  15000139
## 5        5  10975194  11023526
## 6        6   6161967   6267490
## 7        1   5121003   5222277
## 8        4   4672332   4672687
## 9        8   3894003   3882159
```

Joining datasets

Often we have separate dataframes that can have common and differing variables for similar observations and we wish to *join* these dataframes together. `dplyr` offers multiple joining functions (`xxx_join()`) that provide alternative ways to join data frames:

- inner_join()
- left_join()
- right_join()
- full_join()
- semi_join()
- anti_join()

Our public education expenditure data represents then-year dollars. To make any accurate assessments of longitudinal trends and comparison we need to adjust for inflation. I have the following data frame which provides inflation adjustment factors for base-year 2012 dollars *(obviously I should use 2015 values but I had these easily accessable and it only serves for illustrative purposes)*.

```
##    Year  Annual Inflation
## 28 2007 207.342 0.9030811
## 29 2008 215.303 0.9377553
## 30 2009 214.537 0.9344190
## 31 2010 218.056 0.9497461
## 32 2011 224.939 0.9797251
## 33 2012 229.594 1.0000000
```

To join to my expenditure data I obviously need to get my expenditure data in the proper form that allows me to join these two data frames. I can apply the following functions to accomplish this:

```
long_exp <- sub_exp %>%
        gather(Year, Expenditure, X2007:X2011) %>%
        separate(Year, into=c("x", "Year"), sep = "X") %>%
        select(-x) %>%
        mutate(Year = as.numeric(Year))

head(long_exp)
##   Division      State Year Expenditure
## 1        6    Alabama 2007     6245031
## 2        9     Alaska 2007     1634316
## 3        8    Arizona 2007     7815720
## 4        7   Arkansas 2007     3997701
## 5        9 California 2007    57352599
## 6        8   Colorado 2007     6579053
```

I can now apply the `left_join()` function to join the inflation data to the expenditure data. This aligns the data in both dataframes by the *Year* variable and then joins the remaining inflation data to the expenditure data frame as new variables.

```
join_exp <- long_exp %>% left_join(inflation)

head(join_exp)
##   Division      State Year Expenditure  Annual Inflation
## 1        6    Alabama 2007     6245031 207.342 0.9030811
## 2        9     Alaska 2007     1634316 207.342 0.9030811
## 3        8    Arizona 2007     7815720 207.342 0.9030811
## 4        7   Arkansas 2007     3997701 207.342 0.9030811
## 5        9 California 2007    57352599 207.342 0.9030811
## 6        8   Colorado 2007     6579053 207.342 0.9030811
```

To illustrate the other joining methods we can use the `a` and `b` data frames from the `EDAWR` package:

```
library(EDAWR)

a
##   x1 x2
## 1  A  1
## 2  B  2
## 3  C  3

b
##   x1    x2
## 1  A  TRUE
## 2  B FALSE
## 3  D  TRUE


# include all of a, and join matching rows of b
left_join(a, b, by = "x1")
##   x1 x2.x  x2.y
## 1  A    1  TRUE
## 2  B    2 FALSE
## 3  C    3    NA

# include all of b, and join matching rows of a
right_join(a, b, by = "x1")
##   x1 x2.x  x2.y
```

```
## 1  A    1  TRUE
## 2  B    2 FALSE
## 3  D   NA  TRUE

# join data, retain only matching rows in both data frames
inner_join(a, b, by = "x1")
##   x1 x2.x  x2.y
## 1  A    1  TRUE
## 2  B    2 FALSE

# join data, retain all values, all rows
full_join(a, b, by = "x1")
##   x1 x2.x  x2.y
## 1  A    1  TRUE
## 2  B    2 FALSE
## 3  C    3    NA
## 4  D   NA  TRUE

# keep all rows in a that have a match in b
semi_join(a, b, by = "x1")
##   x1 x2
## 1  A  1
## 2  B  2

# keep all rows in a that do not have a match in b
anti_join(a, b, by = "x1")
##   x1 x2
## 1  C  3
```

There are additional dplyr functions for merging data sets worth exploring:

```
intersect(y, z)    # Rows that appear in both y and z
union(y, z)        # Rows that appear in either or both y and z
setdiff(y, z)      # Rows that appear in y but not z
bind_rows(y, z)    # Append z to y as new rows
bind_cols(y, z)    # Append z to y as new columns
```

Creating new variables

Often we want to create a new variable that is a function of the current variables in our data frame or even just add a new variable. The `mutate()` function allows us to add new variables while preserving the existing variables. If we go back to our previous `join_exp` dataframe, remember that we joined inflation rates to our non-inflation adjusted expenditures for public schools. The dataframe looks like:

```
##   Division      State Year Expenditure  Annual Inflation
## 1        6    Alabama 2007     6245031 207.342 0.9030811
## 2        9     Alaska 2007     1634316 207.342 0.9030811
## 3        8    Arizona 2007     7815720 207.342 0.9030811
## 4        7   Arkansas 2007     3997701 207.342 0.9030811
## 5        9 California 2007    57352599 207.342 0.9030811
## 6        8   Colorado 2007     6579053 207.342 0.9030811
```

If we wanted to adjust our annual expenditures for inflation we can use `mutate()` to create a new inflation adjusted cost variable which we'll name `Adj_Exp`:

```
inflation_adj <- join_exp %>% mutate(Adj_Exp = Expenditure / Inflation)

head(inflation_adj)
##   Division      State Year Expenditure  Annual Inflation  Adj_Exp
## 1        6    Alabama 2007     6245031 207.342 0.9030811  6915249
## 2        9     Alaska 2007     1634316 207.342 0.9030811  1809711
## 3        8    Arizona 2007     7815720 207.342 0.9030811  8654505
## 4        7   Arkansas 2007     3997701 207.342 0.9030811  4426735
## 5        9 California 2007    57352599 207.342 0.9030811 63507696
## 6        8   Colorado 2007     6579053 207.342 0.9030811  7285119
```

Lets say we wanted to create a variable that rank-orders state-level expenditures (inflation adjusted) for the year 2010 from the highest level of expenditures to the lowest.

```
rank_exp <- inflation_adj %>%
        filter(Year == 2010) %>%
        arrange(desc(Adj_Exp)) %>%
        mutate(Rank = 1:length(Adj_Exp))

head(rank_exp)
##   Division      State Year Expenditure  Annual Inflation  Adj_Exp Rank
## 1        9 California 2010    58248662 218.056 0.9497461 61330774    1
## 2        2   New York 2010    50251461 218.056 0.9497461 52910417    2
## 3        7      Texas 2010    42621886 218.056 0.9497461 44877138    3
## 4        3   Illinois 2010    24695773 218.056 0.9497461 26002501    4
## 5        2 New Jersey 2010    24261392 218.056 0.9497461 25545135    5
## 6        5    Florida 2010    23349314 218.056 0.9497461 24584797    6
```

If you wanted to assess the percent change in cost for a particular state you can use the `lag()` function within the `mutate()` function:

```
inflation_adj %>%
        filter(State == "Ohio") %>%
        mutate(Perc_Chg = (Adj_Exp - lag(Adj_Exp)) / lag(Adj_Exp))
##   Division State Year Expenditure  Annual Inflation  Adj_Exp     Perc_Chg
## 1        3  Ohio 2007    18251361 207.342 0.9030811 20210102           NA
## 2        3  Ohio 2008    18892374 215.303 0.9377553 20146378 -0.003153057
## 3        3  Ohio 2009    19387318 214.537 0.9344190 20747992  0.029862103
## 4        3  Ohio 2010    19801670 218.056 0.9497461 20849436  0.004889357
## 5        3  Ohio 2011    19988921 224.939 0.9797251 20402582 -0.021432441
```

You could also look at what percent of all US expenditures each state made up in 2011. In this case we use `mutate()` to take each state's inflation adjusted expenditure and divide by the sum of the entire inflation adjusted expenditure column. We also apply a second function within `mutate()` that provides the cummalative percent in rank-order. This shows that in 2011, the top 8 states with the highest expenditures represented over 50% of the total U.S. expenditures in K-12 public schools. *(I remove the non-inflation adjusted Expenditure, Annual & Inflation columns so that the columns don't wrap on the screen view)*

```
cum_pct <- inflation_adj %>%
        filter(Year == 2011) %>%
        arrange(desc(Adj_Exp)) %>%
        mutate(Pct_of_Total = Adj_Exp/sum(Adj_Exp),
               Cum_Perc = cumsum(Pct_of_Total)) %>%
        select(-Expenditure, -Annual, -Inflation)

head(cum_pct, 8)
##   Division        State Year  Adj_Exp Pct_of_Total  Cum_Perc
## 1        9   California 2011 58717324   0.10943237 0.1094324
## 2        2     New York 2011 52575244   0.09798528 0.2074177
## 3        7        Texas 2011 43751346   0.08154005 0.2889577
## 4        3     Illinois 2011 25062609   0.04670957 0.3356673
## 5        5      Florida 2011 24364070   0.04540769 0.3810750
## 6        2   New Jersey 2011 24128484   0.04496862 0.4260436
## 7        2 Pennsylvania 2011 23971218   0.04467552 0.4707191
## 8        3         Ohio 2011 20402582   0.03802460 0.5087437
```

An alternative to `mutate()` is `transmute()` which creates a new variable and then drops the other variables. In essence, it allows you to create a new data frame with only the new variables created. We can perform the same string of functions as above but this time use transmute to only keep the newly created variables.

```
inflation_adj %>%
        filter(Year == 2011) %>%
        arrange(desc(Adj_Exp)) %>%
        transmute(Pct_of_Total = Adj_Exp/sum(Adj_Exp),
               Cum_Perc = cumsum(Pct_of_Total)) %>%
        head()
##   Pct_of_Total  Cum_Perc
## 1   0.10943237 0.1094324
## 2   0.09798528 0.2074177
## 3   0.08154005 0.2889577
## 4   0.04670957 0.3356673
## 5   0.04540769 0.3810750
## 6   0.04496862 0.4260436
```

Lastly, you can easily also apply the `summarise` and `mutate` functions to multiple columns by using `summarise_each()` and `mutate_each()` respectively.

```
# calculate the mean for each division with summarise_each
# call the function of interest with the `funs()` argument
sub_exp %>%
        select(-State) %>%
        group_by(Division) %>%
        summarise_each(funs(mean)) %>%
        head()
## Source: local data frame [6 x 6]
##
##   Division    X2007    X2008    X2009    X2010    X2011
##      (int)    (dbl)    (dbl)    (dbl)    (dbl)    (dbl)
## 1        1  4680691  4952992  5173184  5121003  5222277
## 2        2 28844158 30652645 31304697 32415457 32877923
## 3        3 14823590 15293644 15895459 16322489 16270159
## 4        4  4175766  4425739  4658533  4672332  4672687
## 5        5 10230416 10857410 11018102 10975194 11023526
## 6        6  5584277  6023424  6076507  6161967  6267490

# for each division calculate the percent of total
# expenditures for each state across each year
sub_exp %>%
        select(-State) %>%
        group_by(Division) %>%
        mutate_each(funs(. / sum(.))) %>%
        head()
```

```
## Source: local data frame [6 x 6]
## Groups: Division [4]
##
##   Division     X2007      X2008      X2009      X2010      X2011
##      (int)     (dbl)      (dbl)      (dbl)      (dbl)      (dbl)
## 1        6 0.27958099 0.28357787 0.27498705 0.27063262 0.26298109
## 2        9 0.02184221 0.02387438 0.02515947 0.02682018 0.02846193
## 3        8 0.28093187 0.27793321 0.28144201 0.27229536 0.26854292
## 4        7 0.07854895 0.07565703 0.07402700 0.07474621 0.07630156
## 5        9 0.76650258 0.76625202 0.75304632 0.74962818 0.74380904
## 6        8 0.23648054 0.24272678 0.23179279 0.23848536 0.23857413
```

Similar to the summary function, `dplyr` allows you to build in your own functions to be applied within `mutate_each()` and also has the following built in functions that can be applied.

`lead()`	`ntile()`	`cumsum()`
`lag()`	`between()`	`cummax()`
`dense_rank()`	`cume_dist()`	`cummin()`
`min_rank()`	`cumall()`	`cumprod()`
`percent_rank()`	`cumany()`	`pmax()`
`row_number()`	`cumean()`	`pmin()`

Built-in Functions for `mutate_each()`

Additional resources

This chapter introduced you to `dplyr`'s basic set of tools and demonstrated how to use them on data frames. Additional resources are available that go into more detail or provide additional examples of how to use `dpyr`. In addition, there are other resouces that illustrate how `dplyr` can perform tasks not mentioned in this chapter such as connecting to remote databases and translating your R code into SQL code for data pulls.

- Data wrangling presentation[205] I gave at Miami University
- `dplyr` reference manual[206]
- R Studio's Data wrangling with R and RStudio webinar[207]

[205] http://bradleyboehmke.github.io/2015/10/data-wrangling-presentation.html
[206] https://cran.r-project.org/web/packages/dplyr/dplyr.pdf
[207] http://www.rstudio.com/resources/webinars/

- R Studio's Data wrangling GitHub repository[208]
- R Studio's Data wrangling cheat sheet[209]
- Hadley Wickhamâ€™s dplyr tutorial at useR! 2014, Part 1[210]
- Hadley Wickhamâ€™s dplyr tutorial at useR! 2014, Part 2[211]

[208] https://github.com/rstudio/webinars/blob/master/2015-01/wrangling-webinar.pdf
[209] http://www.rstudio.com/resources/cheatsheets/
[210] http://www.r-bloggers.com/hadley-wickhams-dplyr-tutorial-at-user-2014-part-1/
[211] http://www.r-bloggers.com/hadley-wickhams-dplyr-tutorial-at-user-2014-part-2/