R for HR:
An Introduction to Human Resource Analytics Using R

David E. Caughlin

2020-07-30

Contents

Preface

The *Preface* offers an orientation to this book, foundational human resource analytics concepts, and the book's raison d'être.

0.1 Growth of HR Analytics

The term *human resource analytics* can mean different things to different people and to different organizations. Further, human resource analytics sometimes goes by other names like people analytics, talent analytics, workforce analytics, and human capital analytics. While some may argue distinctions between these different names, for this book, I will treat them as interchangeable synonyms. Moreover, for the purposes of this book, **human resource (HR) analytics** refers to the "process of collecting, analyzing, interpreting, and reporting people-related data for the purpose of improving decision making, achieving strategic objectives, and sustaining a competitive advantage" (Bauer et al., 2020, p. 34).

The foundation of HR analytics was solidified over a century ago with the emergence of disciplines like industrial and organizational (I/O) psychology. In recent decades, advances in information technology and systems have reduced the time HR professionals spend on transactional and administrative activities, thereby creating more time and space for transformational activities that facilitate the realization of strategic objectives. HR analytics can play a critical role in such transformational activities, as it can inform HR system design (e.g., selection tool selection, validation, and process) and high-stakes decision making regarding people within the organization.

0.2 Skills Gap

Although HR analytics is now widely regarded as strategically important for organizational success, an HR analytics skills gap has emerged. Historically, data analytics, data literacy, and numeracy were not major focal points of academic

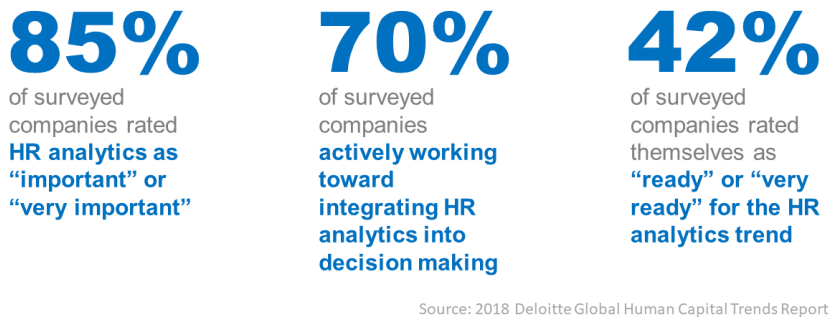


Figure 1: A 2018 survey of companies highlighted the perceived importance of HR analytics but a relative lack of readiness to adopt and integrate HR analytics (Deloitte, 2018).

and industry training for HR professionals, which has left organizations scrambling to hire new talent or to upskill existing HR professionals. For some organizations, attempting to fill the skills gap by hiring a data scientist or statistician may prove fruitful if the individual works closely with HR professionals who possess expertise in HR systems, polices, and procedures, as well as knowledge of legal and ethical considerations that are specific to HR. I contend, however, that a better alternative is to upskill existing HR professionals. Presumably, they already possess rich knowledge and skills related to the HR domain, which can facilitate their ability to acquire, manage, and analyze data ethically and in line with prevailing legal guidelines and to interpret and tell a story about the process that can lead to deployment and implementation of data-informed system design and practices – and ultimately to strategic transformation of the organization and its workforce.

0.3 HR Analytics Project Life Cycle

I developed the HR Analytics Project Life Cycle (HRAPLC) as a way to conceptualize the prototypical phases of a generic project life cycle. These phases include: (a) Question Formulation, Data Acquisition, Data Management, Data Analysis, Data Interpretation and Storytelling, and Deployment and Implementation. This book provides hands-on learning opportunities for topics and tools related to the Data Acquisition, Data Management, Data Analysis, and Data Interpretation and Storytelling phases.

The phases of the HRAPLC generally align with the generic scientific process steps of formulating a hypothesis, designing a study, collecting data, analyzing data, and reporting findings. This highlights how HR analytics represents a

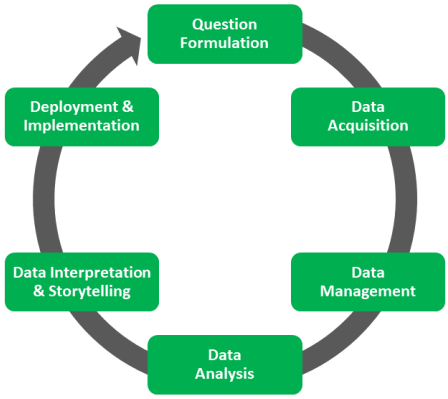


Figure 2: The Human Resource Analytics Project Life Cycle (HRAPLC) offers a way to conceptualize the prototypical phases of a generic HR analytics project life cycle.

scientific approach to HR management.

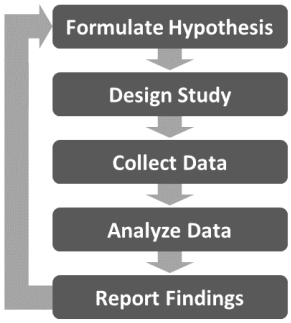


Figure 3: The phases of the Human Resource Analytics Project Life Cycle (HRAPLC) generally align with steps of the scientific process.

0.3.1 Question Formulation

Question formulation refers to the process of posing strategy-inspired research questions and hypotheses that can be answered or tested using data. Effective question formulation results in (a) greater data acquisition, management, and analysis efficiency and (b) findings that are meaningful to stakeholders.

0.3.2 Data Acquisition

Data acquisition refers to the process of collecting, retrieving, gathering, and sourcing data that can be used to answer questions and test hypotheses. Different tools can be used for data acquisition, such as employee surveys, (performance) rating forms, surveillance and monitoring, database queries, and scraping or crawling. In some instances, the required data may already reside in an HR information system (HRIS) or enterprise resource planning (ERP) platform.

0.3.3 Data Management

Data management refers to the process of wrangling, cleaning, manipulating, and structuring data. Different tools can be used for data management, such as database management systems and data analysis software programs. The general rule of thumb is that you can expect to spend 80% of your time managing data and about 20% of your time analyzing data.

0.3.4 Data Analysis

Data analysis refers to the process of applying mathematical, statistical, and/or computational techniques to data to identify associations, differences or changes, or classes (categories), as well as to predict the likelihood of future events, values, or differences or changes. Various tools used in data analysis, such as mathematics, statistics, simulations, and computational modeling.

0.3.5 Data Interpretion & Storytelling

Data interpretation and storytelling refers to the process of making sense of data analysis findings and evaluating questions and hypotheses, as well as disseminating the findings to different stakeholders. To support interpretation and storytelling, data visualization is frequently used (e.g., graphs, charts, plots).

0.3.6 Deployment & Implementation

Deployment and implementation refers to the process of prescribing or taking action based on interpretation of data-analysis findings. This phase requires an (a) understanding of stakeholder needs, (b) an understanding of the business context, and (c) knowledge of change management theories and practices.

0.4 My Philosophy for This Book

Working with data does not need to be scary or intimidating; yet, over the years, I have interacted with students and professionals who carry with them what I refer to as a *numerical phobia* or *quantitative trauma*. Unfortunately, at some point in their lives, some people are made to believe that they are not suited for mathematics, statistics, and/or generally working with data. Given these mental barriers, a primary objective of this book is to make data analytics – and HR analytics specifically – relevant, accessible, and maybe even a little fun. In early chapters, my intention is to ease the reader into foundational concepts, applications, and tools in order to incrementally build self-efficacy in HR analytics. Each chapter is grounded in a what I hope will be a meaningful context for those who work in HR or who, at the very least, have some familiarity with the function and how it relates to the business. As the book progresses, more challenging statistical concepts and data-analytic techniques are introduced. Reading this book and following along with the in-chapter tutorials will not lead to expert-level knowledge and skill; however, my hope is that completing all or portions of this book will do the following:

1. Build excitement for working with data to inform decision making.
2. Instill a sense of intellectual curiosity about data and a hunger to expand boundaries of expertise.
3. Inspire further in-depth training, education, and learning in areas and topics introduced in this book.
4. Enhance data literacy, including knowledge and skills related to (a) critical thinking and logic, (b) mathematics, statistics, and data analysis, and (c) data visualization and storytelling with data.

0.4.1 Rationale for Using R

Today, we have the potential to access and use a remarkable number of statistical and data-analytic programs. Examples of programs and languages with such capabilities include (in no particular order) R, Python, SPSS, SAS, Stata, MatLab, Mplus, Tableau, PowerBI, and Microsoft Excel. Some of these programs can be quite expensive when it comes to lifetime or annual user licensing costs, which can be a barrier to access for many.

Programming languages like R and Python have several desirable qualities when it comes to managing, analyzing, and visualizing data. Namely, both are free and both have an ever-growing number of free (add-on) packages with domain- or area-specific functions (e.g., data visualizations). It is beyond the scope of this *Preface* to provide an exhaustive comparison of the relative merits of R versus Python; however, when it comes to the statistical analysis of data, specifically, I argue that R provides a more user-friendly entry point for beginners as well as more advanced capabilities desired by expert users. Moreover, the integrated

development environment program called RStudio (which "sits on top of"" base R) offers useful workflow tools and generally makes for an inviting environment within which the R engine can be run.

With all that said, Python has been catching up in these regards, and I wouldn't be surprised if Python closes these gaps relative to R in the next few years. I would be remiss if I didn't mention that the Python language is powerful and has capabilities that extend far beyond the management, analysis, and visualization of data. Fortunately, learning R makes learning Python easier (and vice versa), which means that this book can serve as springboard for learning Python or other programming languages. Finally, I believe it to be unlikely that one program or language will emerge that is ideal for every task, and thus, I encourage people to build familiarity with multiple tools so that the best (or at least better) tool can be used for each task.

0.4.2 Audience

I have written this book with current or soon-to-be HR professionals in mind, particularly those who have an interest in upskilling their data-analytic knowledge and skills. With that said, I believe this book can provide a meaningful context for learning key data-analytic concepts, applications, and tools that are applicable beyond the HR context. Relatedly, this book may serve as a gateway to a user-friendly introduction to the programming language called R.

0.4.3 Structure

This book consists of six parts:

1. Introduction
2. Data Acquisition
3. Data Management
4. Data Analysis & Visualization
5. References
6. Additional Topics

0.4.3.1 Introduction

The *Introduction* (Part 1) introduces the reader to area of HR analytics and the R programming language. This part also focuses on how to install and get started with R and RStudio, including a gentle introduction to foundational concepts and operations associated with the R language.

0.4.3.2 Data Acquisition

Data Acquisition (Part 2) focuses on how to bring data into the R environment that have been acquired – and how to export data outside of the R environment. Data Acquisition is a key phase of the HR Analytics Project Life Cycle.

0.4.3.3 Data Management

Data Management (Part 3) provides an overview to foundational data management concepts and techniques, such as arranging (sorting), joining (merging), manipulating (wrangling), aggregating, and cleaning. Data Management is a key phase of the HR Analytics Project Life Cycle.

0.4.3.4 Data Analysis & Visualization

Data Analysis & Visualization (Part 4) acts as the heart of this book, as it introduces various mathematical and statistical concepts as they relate to specific functional areas of HR (e.g., selection, training). To facilitate the interpretation and communication of data-analysis findings, various data visualization displays are showcased. This part integrates the Data Analysis and Data Interpretation and Storytelling phases of the HR Analytics Project Life Cycle.

0.4.3.5 References

The *References* (Part 5) lists the references for the sources that are cited throughout the book.

0.4.3.6 Additional Topics

The *Additional Topics* (Part 6) provides a home to "else" – or rather, topics that would not fit neatly into Parts 1-4 of the book. Examples of such topics include question formulation and HR information systems.

0.5 About the Author

David Caughlin works for Portland State University's School of Business where he engages in research and teaching on topics related to organizational behavior, human resource management, and data analytics. David received his B.S. in psychology and B.A. in Spanish from Indiana University, his M.S. in industrial & organizational psychology from Indiana University - Purdue University at Indianapolis, and his Ph.D. in industrial & organizational psychology from

Portland State University with concentrations in quantitative methodology and occupational health psychology. His research interests are generally focused on supervisor support, work motivation, and occupational safety and health. His research has been published in peer-reviewed outlets such as *Journal of Applied Psychology*, *Human Resource Management*, *Journal of Occupational Health Psychology*, and *Psychology, Public Policy, and the Law*. He co-authored the textbooks *Human Resource Management: People, Data, and Analytics* and *Fundamentals of Human Resource Management: People, Data, and Analytics*. In the School of Business, David teaches undergraduate and graduate courses on topics related to human resource management, information systems, and data analytics. In his HR analytics courses, David teaches students how to apply the statistical programming language R to manage, analyze, and visualize HR data to improve strategic decision making; in the process, students build their data literacy and develop their critical-thinking and reasoning skills. He has received the following teaching awards from the School of Business: Teaching Innovation Award (2018), "Extra Mile" Teaching Excellence Award (2019), and Teaching Innovation Award (2020). In his free time, David enjoys outdoor activities like trail running, skiing, mountain biking, and paddle boarding.

0.6 Acknowledgements

My inspiration for writing and compiling the contents of this book stems from interactions with countless colleagues, professional acquaintances, and undergraduate and graduate students, and a broad "thank you" is in order for anyone with whom I have taught or had a conversation about HR analytics specifically or data analytics in general. Finally, I created this book using the following programs and packages: R (R Core Team, 2020), RStudio (RStudio Team, 2020), `rmarkdown` (Xie et al., 2018; Allaire et al., 2020), `knitr` (Xie, 2015, 2014, 2020b), and `bookdown` (Xie, 2016, 2020a).

Part I

Introduction

Chapter 1

Installing R & RStudio

If you have a Windows, Mac, or Linux operating system, you have several ways in which you can begin working in R. Commonly, users install R on their computer along with an integrated development environment (IDE) software application like RStudio. Recently, RStudio Cloud (https://rstudio.cloud) has emerged as an alternative to installing R and RStudio by allowing users to use R and RStudio via the cloud, which has the advantage of allowing access to R and RStudio when using the Chrome operating system.

1.0.0.1 Video Tutorial

Link to video tutorial: https://youtu.be/b18IHQERT4A.

1.1 Downloading & Installing R

In the following sections, you will learn how to download and install the R program for Windows and Mac operating systems. The base R program must be installed prior to installing the RStudio program. R is open-source software and free to download.

1.1.1 For Windows Operation Systems

R can currently run under operating systems as old as Windows Vista (circa 2007). To download R for your Windows operating system for the first time, click on this link: https://cran.r-project.org/bin/windows/base/. Once you are on the R download page, click on the hyperlink to download the current version of R for Windows. Once the file has downloaded, follow the installation prompts.

1.1.2 For Mac Operating Systems

The current version of R works with Mac OS X (release 10.6 and higher). To download R for Mac OS X operating system for the first time, click on this link: https://cran.r-project.org/bin/macosx/. If you have Mac OS X 10.11 or higher, click on the hyperlink (with .pkg extension) under the "Latest release" section to begin your download. If you have Mac OS X 10.10 or lower, click on the appropriate hyperlink (with .pkg extension) under the "Binaries for legacy OS X systems" section. Once the file has downloaded, follow the installation prompts.

I don't advise using a Mac operating system that is older than Mac OS X 10.6 (which came out in 2009), as you may run into issues when using certain R packages for data analysis and visualization.

1.2 Downloading & Installing RStudio

RStudio is not required to use R; however, RStudio offers a number of helpful features and a user-friendly interface. More specifically, RStudio is an integrated development environment (IDE) for R. The open-source edition of RStudio is free to download. I recommend downloading the RStudio Desktop Open-Source License edition. To do so, click on this link: https://www.rstudio.com/products/rstudio/download/.

1. Click on the download button below the name "RStudio Desktop Open Source License". Again, this version is free.

2. Under the heading "Installers for Supported Platforms", click on the link that corresponds to your operating system.

3. Once the file has downloaded, follow the installation prompts.

1.3 Summary

In this chapter, we learned how to install R and RStudio for our Windows or Mac operating system.

Chapter 2

Getting Started with R

XXXXX

2.1 Orientation to RStudio

XXXXX

2.1.0.1 Video Tutorial

Link to Video Tutorial: XXXXX

2.2 Setting a Working Directory

A **working directory** is the location of a folder within a hierarchical file system. For our purposes, a working directory contains data files for a particular task/project. Ideally, a single working directory contains all of the data files you need for a particular task/project, but in some instances, it might make sense to have multiple working directories for a single project. From our designated working directory, we can read in data files (i.e., import files) to the R environment. Further, anytime you save a plot, data frame, or other object created in R, the default will be to save it to the folder you have set as your working directory (i.e., export files).

2.2.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/oSqOqvMkhSE

2.2.0.2 Functions & Packages Introduced

Function	Package
`getwd`	base
`setwd`	base

2.2.1 Identifying the Current Working Directory

To determine if a working directory has already been set, and if so, what that working directory is, use the `getwd` (get working directory) function from base R. For this function, you don't need any arguments within the parentheses; in other words, leave the function parentheses empty. Alternatively, if you are using RStudio, you will see your current working directory next to the word "Console" in your Console window.

```
# Find your current working directory
getwd()
```

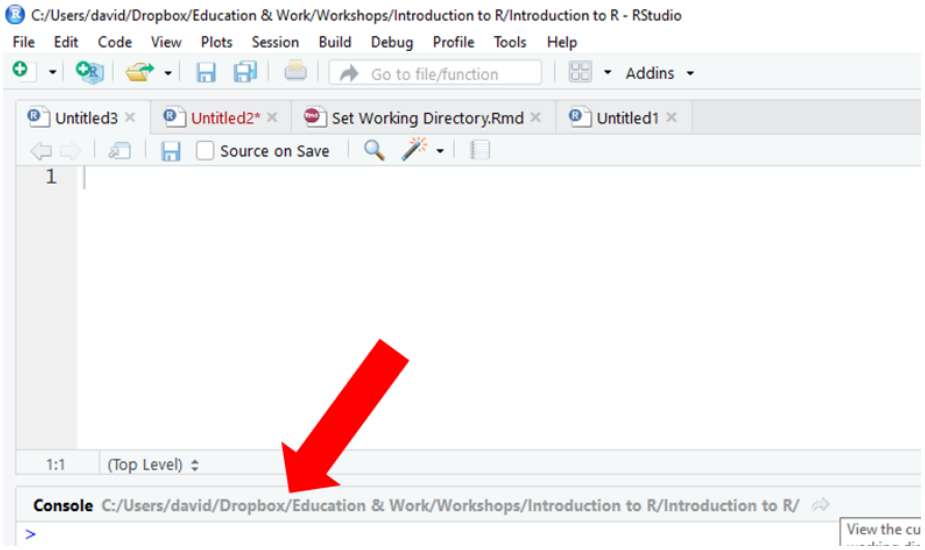


Figure 2.1:

2.2.2 Setting a New Working Directory

Let's assume that the current working directory is *not* what we want; meaning, we need to set a new or different working directory. If you need to set a new

working directory, you can use the `setwd` function from base R. Within the parentheses, your only argument will be the working directory in quotation marks. I recommend typing your `setwd` function into an R Script (.R) file so that it can be saved for future sessions. I also recommend using the `#` to annotate your script so that you can remind yourself (and others) what you are doing.

When it comes to working directories, R likes the forward slash (`/`) (as opposed to backslash). Remember, the working directory is the location of the data files you wish to access and bring into the R environment. You can access any folder you would like and set it as your working directory.

```
# Set your working directory
setwd("H:/RWorkshop")
```

Alternatively, you may use the drop-down menus to select a working directory folder. To do so, go to *Session > Set Working Directory > Choose Directory…*, and select the folder where your files live. Upon doing so, your working directory will appear in the Console. You can copy and paste the working directory into your `setwd` function.

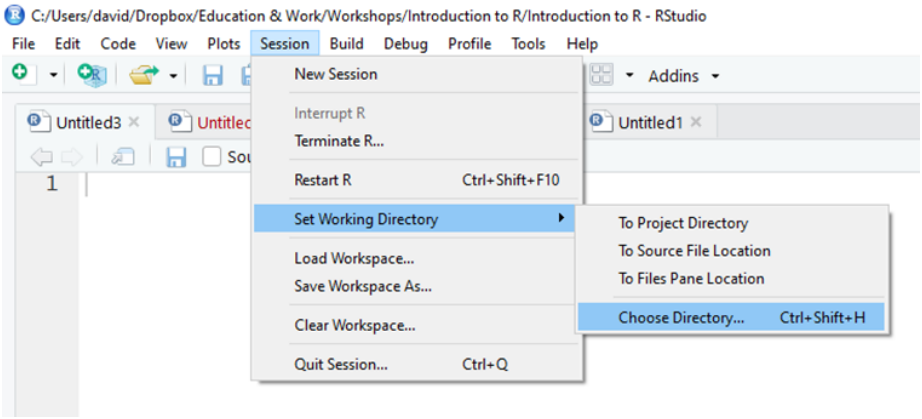


Figure 2.2:

Once you have set your working directory, you can verify that it was set to the correct folder by (a) typing `getwd()` into your console or (b) looking at the working directory listed next to the word "Console" in your Console window.

2.3 Creating & Saving an R Script

An **R Script** is a text editor file in which you can create, edit, and save your R code for a particular task or project. An R Script file has the .R file extension.

It is advisable that you type code directly into an R Script file if you wish to use the code again in the future or if you wish to save the code for another session. In general, try to avoid writing code directly into the Console using the command line if you wish to later reproduce your work. An R Script also allows you to make and save annotations (using the `#` symbol) to explain your code and decision making. Once you typed code (and annotations) into an R Script, you can highlight all of it (or chunks of it) and then click the Run button (or CTRL+Enter for Windows users or Command+Enter for Mac users), which is located in the upper right hand corner of the R Script editor window.

In essence, an R Script allows you to save your code and to tell a story about what you have done. As much as you believe you'll never forget what you were doing in a particular R session, you will likely forget important details as time passes. Or, imagine a scenario in which someone else inherits your data project; a well-written and -documented R Script file will help them retrace your footsteps and onboard them onto the project.

2.3.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/6_CFx5-KmMI

2.3.1 Creating a New R Script

To create a new R Script in RStudio, in the drop-down menu, select *File* > *New File* > *R Script* (as shown below).

2.3.2 Using an R Script

To use an R Script, simply type into the script interface. To illustrate how to do this, let's type `# Adding 2 plus 3` on the first line; note that I began the line with the `#` symbol, which tells R that any text written to the right is annotation and thus won't be interpreted by R when you select it and click Run. On the next line, let's type `2 + 3`. Highlight both lines of script you typed and click the Run button (or CTRL+Enter for Windows users or Command+Enter for Mac users) (as shown below).

```
# Adding 2 plus 3
2 + 3
```

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

Your Console window should show your output (as shown above).

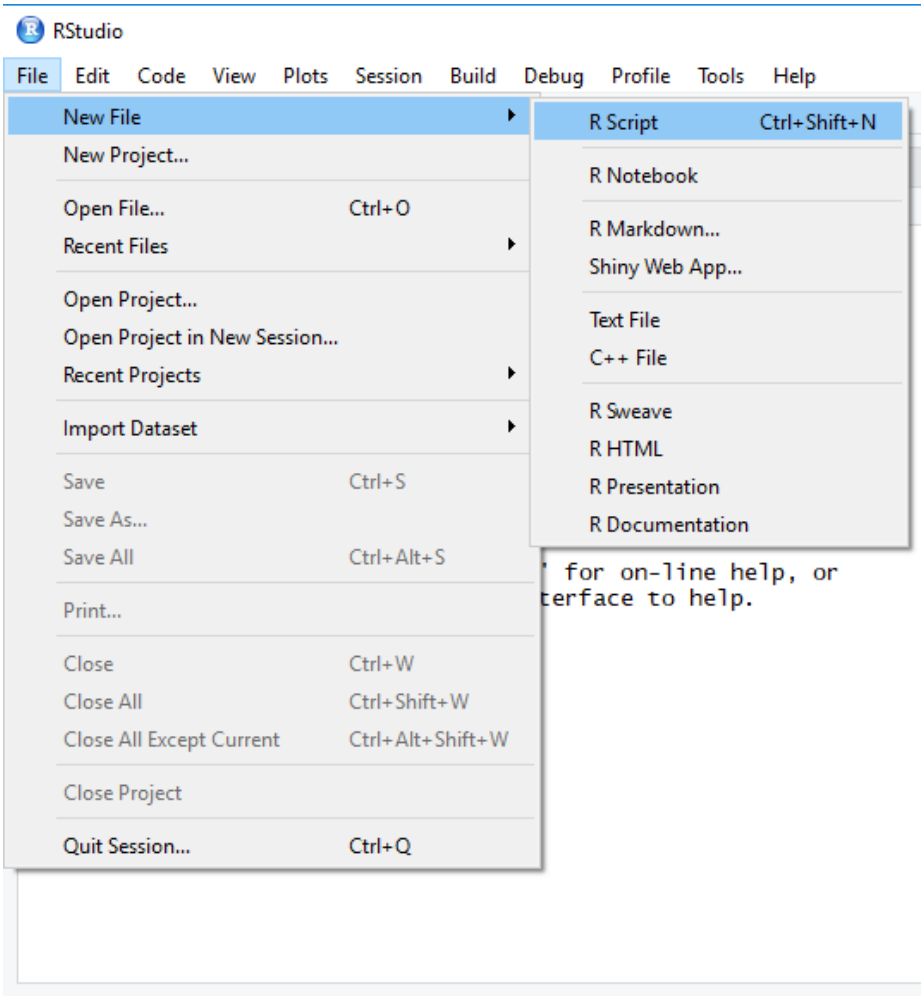

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
New File
New Project...
Open File... Ctrl+O
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Save All Ctrl+Alt+S
Print...
Close Ctrl+W
Close All Ctrl+Shift+W
Close All Except Current Ctrl+Alt+Shift+W
Close Project
Quit Session... Ctrl+Q
R Script Ctrl+Shift+N
R Notebook
R Markdown...
Shiny Web App...
Text File
C++ File
R Sweave
R HTML
R Presentation
R Documentation
for on-line help, or
terface to help.

Figure 2.3:

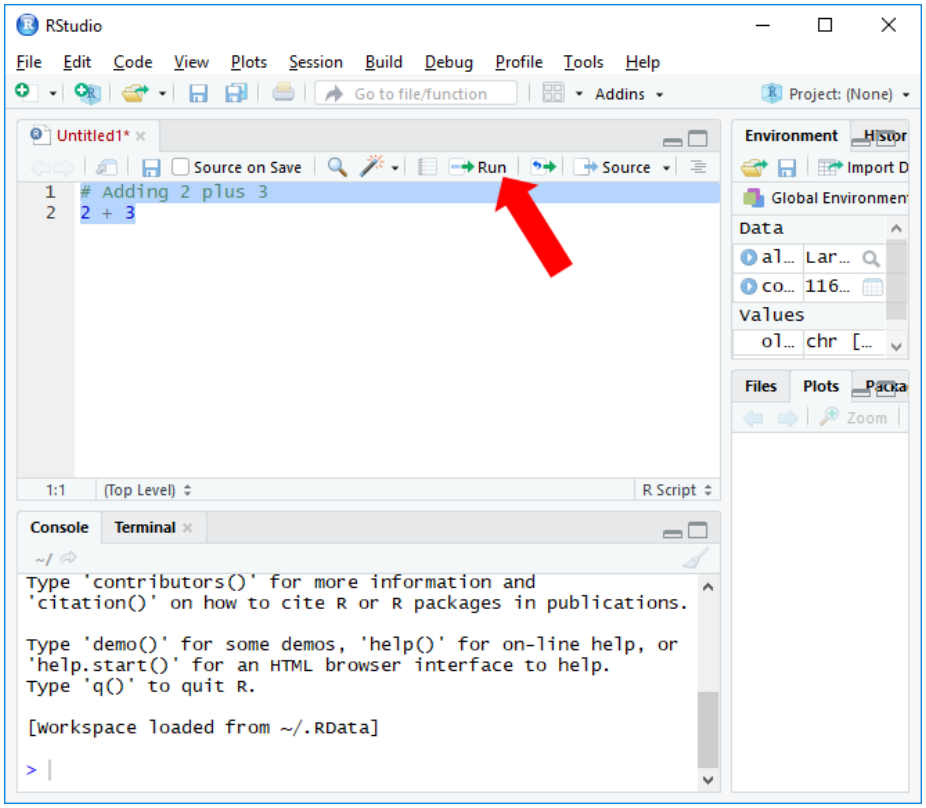

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Untitled1*
Source on Save
Run
Source
1 # Adding 2 plus 3
2 2 + 3
1:1
(Top Level)
R Script
Console
Terminal
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/.RData]
>
Environment
Import D
Global Environmen
Data
Values
Files
Plots
Zoom

Figure 2.4:

2.3.3 Saving an R Script

Always remember to save your R Script, and do so frequently. To save an R Script in RStudio, in the drop-down menu, select *File > Save As* (as shown below). After that, a window will open, and you can save the R Script file in a location of your choosing and with a name of your choosing.

2.3.4 Opening a Saved R Script

To open a saved R Script in RStudio, in the drop-down menu, select *File > Open File…* (as shown below). After that, a window will open, and you can select the R Script file to open.

2.4 Creating an RStudio Project

An **RStudio project** (or **R project**) file (.Rproj) is specific to RStudio and allows one to cluster associated scripts and data files into into a single workflow. For example, if you were evaluating a new onboarding program for your company, you could create an RStudio project with a common working directory that ties together any data files and R scripts that are relevant for evaluating the program. Creating an R project is not required for data management, analysis, and visualization work in RStudio, but it can be helpful. For more information on the value of RStudio projects, check out Wickham and Grolemund's (2017) section on RStudio projects: https://r4ds.had.co.nz/workflow-projects.html#rstudio-projects.

2.4.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/WyrJmJWgPiU

2.4.1 Creating a New RStudio Project

First, to create a new project in RStudio, in the drop-down menu, select *File > New Project…*.

Second, when the "Create Project" window pops up, select the "New Directory" option if you have not yet created a working directory that can be used for your project (see Figure 2). [Alternatively, select the "Existing Directory" option if already have a working directory in place that can be used for your project.]

Third, in the "Project Type" window, select "New Project".

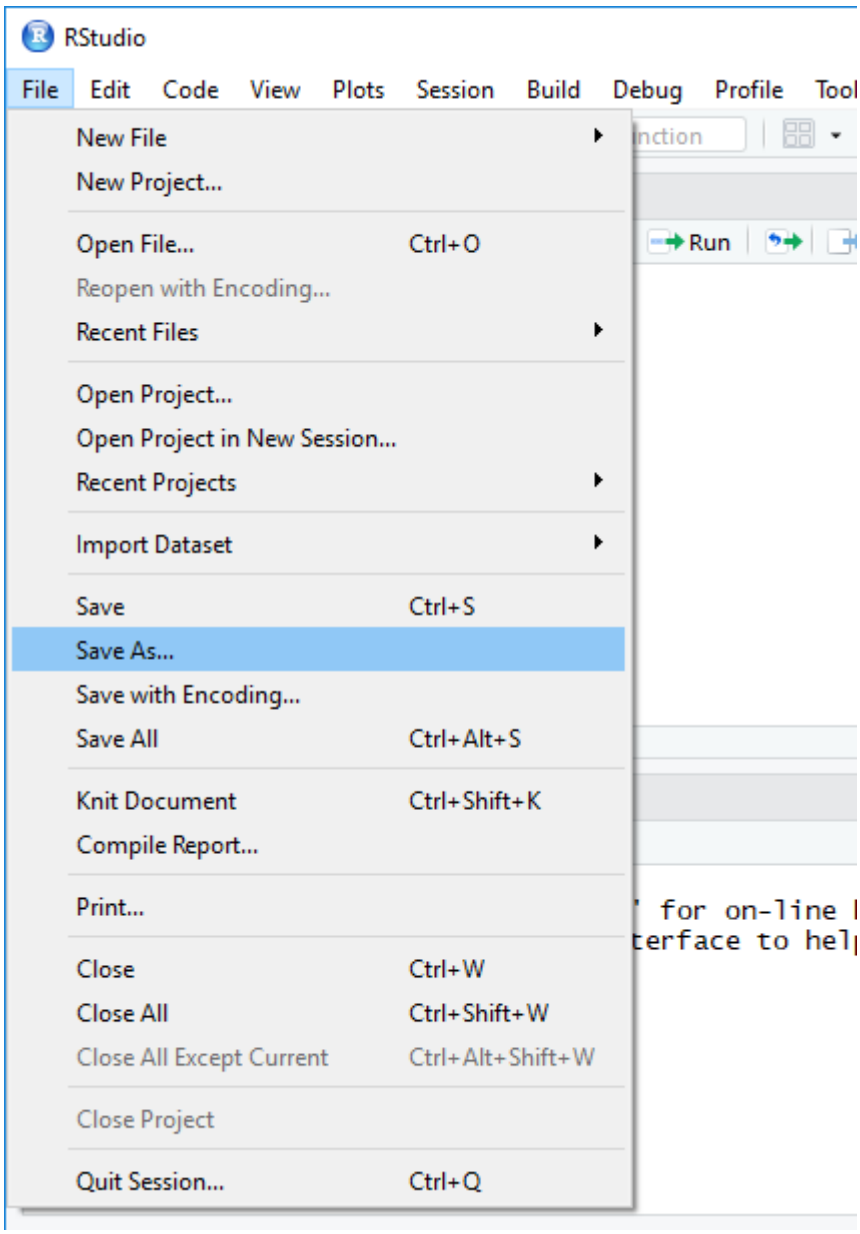

RStudio
File
Edit
Code
View
Plots
Session
Build
Debug
Profile
Tool
New File
New Project...
Open File... Ctrl+O
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Save with Encoding...
Save All Ctrl+Alt+S
Knit Document Ctrl+Shift+K
Compile Report...
Print...
Close Ctrl+W
Close All Ctrl+Shift+W
Close All Except Current Ctrl+Alt+Shift+W
Close Project
Quit Session... Ctrl+Q
Run
for on-line
terface to hel

Figure 2.5:

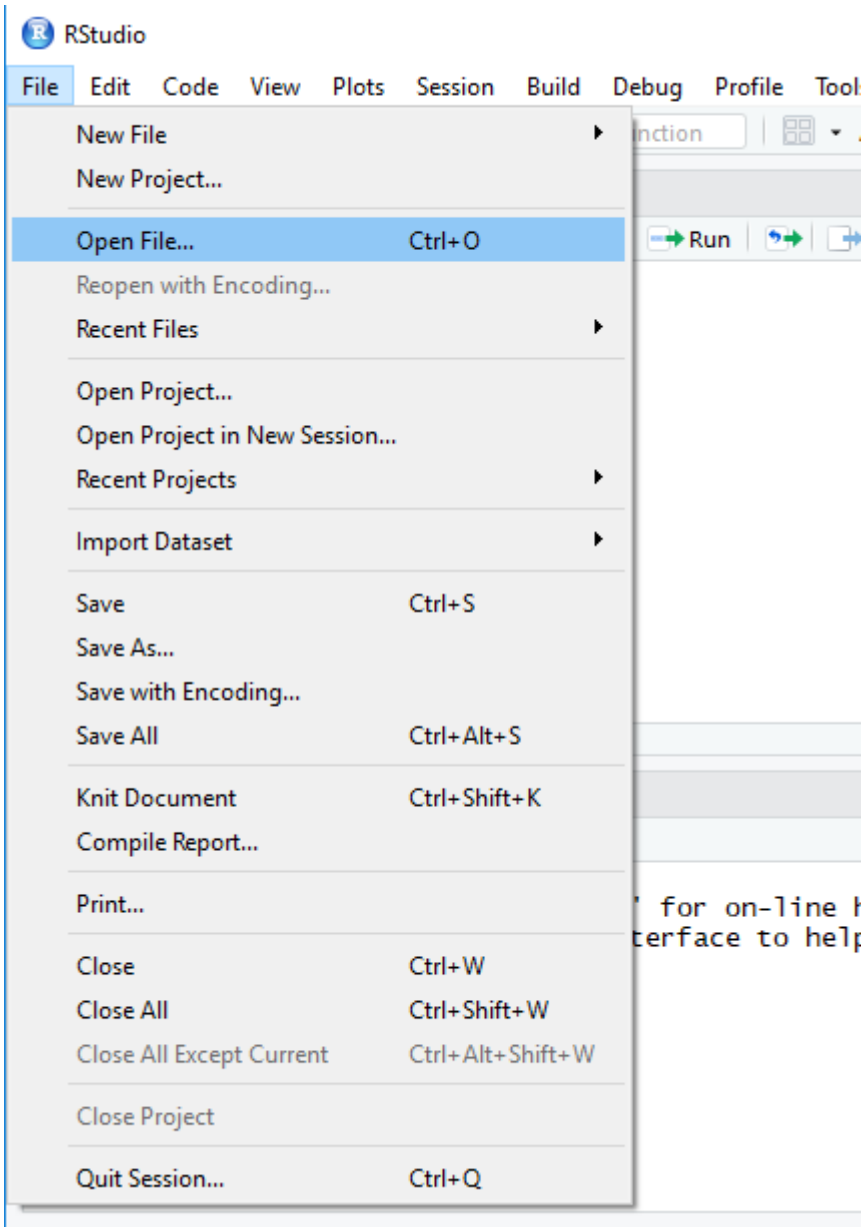

RStudio
File
Edit
Code
View
Plots
Session
Build
Debug
Profile
Tool
New File
New Project...
Open File... Ctrl+O
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Save with Encoding...
Save All Ctrl+Alt+S
Knit Document Ctrl+Shift+K
Compile Report...
Print...
Close Ctrl+W
Close All Ctrl+Shift+W
Close All Except Current Ctrl+Alt+Shift+W
Close Project
Quit Session... Ctrl+Q
Run
for on-line
terface to help

Figure 2.6:

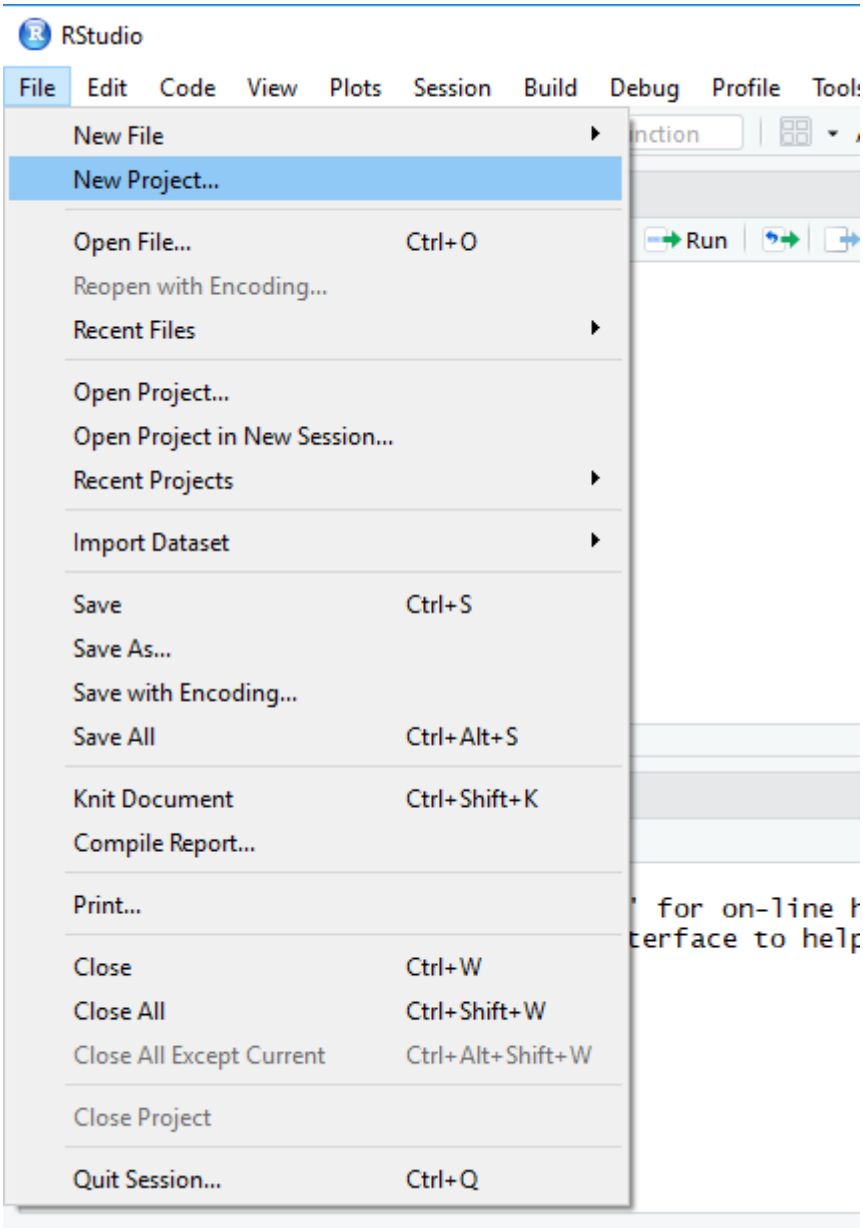

RStudio
File
Edit
Code
View
Plots
Session
Build
Debug
Profile
Tools
New File
New Project...
Open File... Ctrl+O
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Save with Encoding...
Save All Ctrl+Alt+S
Knit Document Ctrl+Shift+K
Compile Report...
Print...
Close Ctrl+W
Close All Ctrl+Shift+W
Close All Except Current Ctrl+Alt+Shift+W
Close Project
Quit Session... Ctrl+Q
Run
for on-line h
terface to help

Figure 2.7:

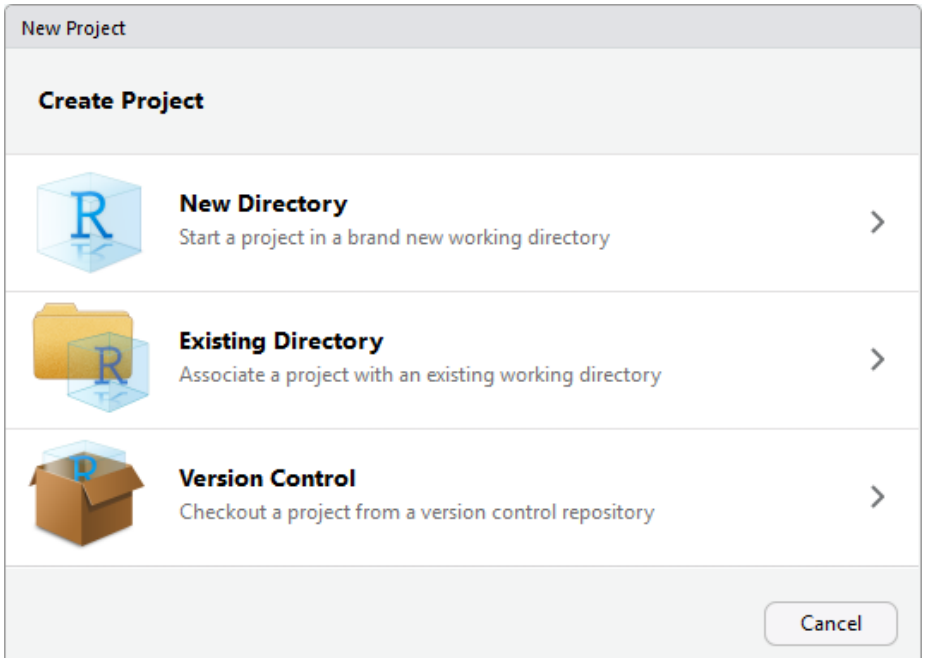


Figure 2.8:

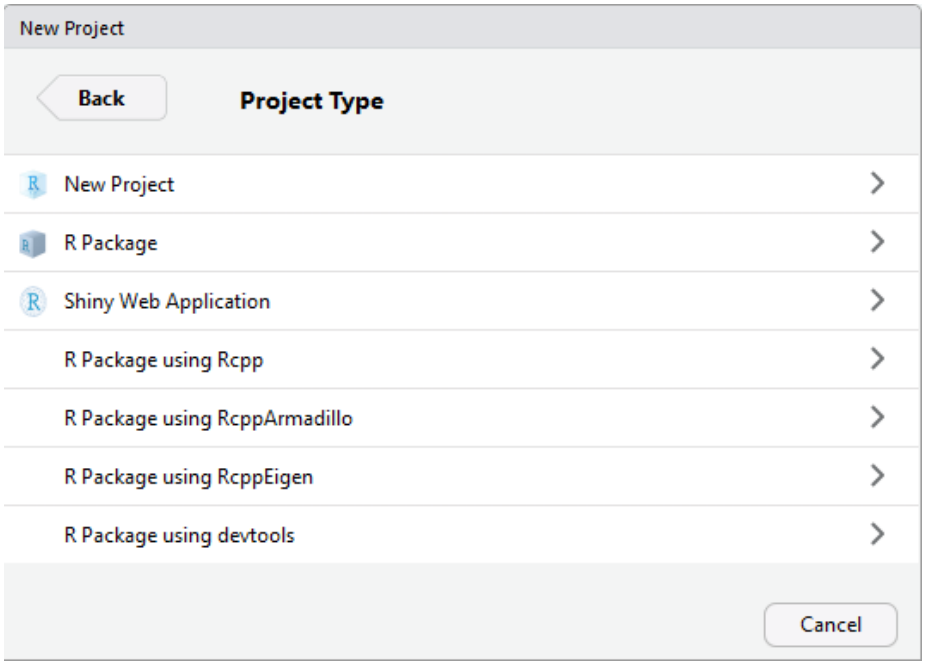


Figure 2.9:

Fourth, in the "Create New Project" window, input what you would like to name the new project (in the field under "Directory name") and select the location of your working directory. Finally, click the "Create Project" button.

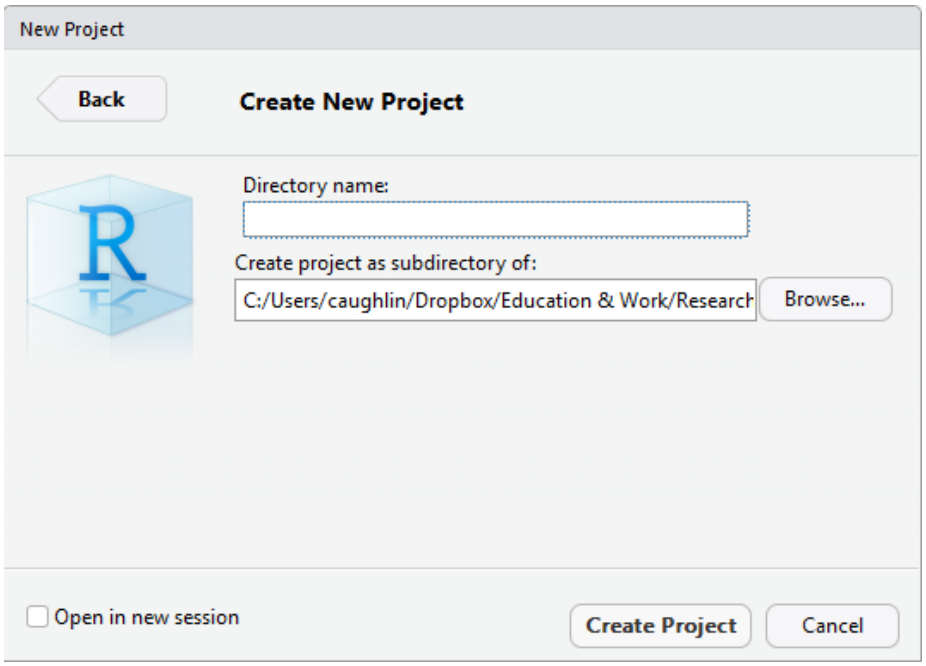


Figure 2.10:

2.4.2 Opening an Existing RStudio Project

To open an existing RStudio project, in the drop-down menu, select *File* > *Open Project...*.

2.5 Orientation to Written Tutorials

Throughout this book, I have included example R code, which I did so using RMarkdown. This approach to demonstrating R tools and techniques is common, and thus it's good to orient yourself to written tutorials in this format. The following video provides an orientation to written R tutorials.

2.5.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/1Wh6eUYAoZc

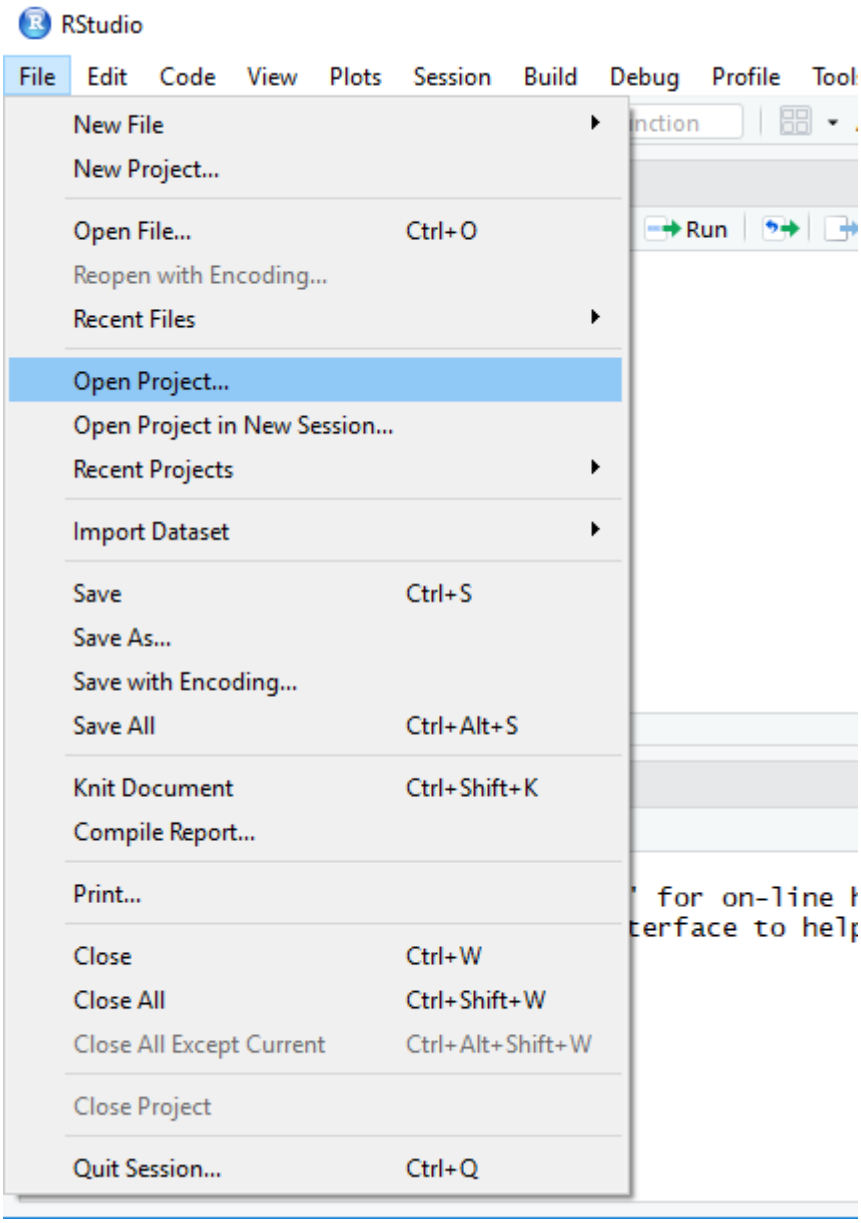

RStudio
File
Edit
Code
View
Plots
Session
Build
Debug
Profile
Tool
New File
New Project...
Open File... Ctrl+O
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Save with Encoding...
Save All Ctrl+Alt+S
Knit Document Ctrl+Shift+K
Compile Report...
Print...
Close Ctrl+W
Close All Ctrl+Shift+W
Close All Except Current Ctrl+Alt+Shift+W
Close Project
Quit Session... Ctrl+Q
Run
for on-line
terface to help

Figure 2.11:

2.6 Summary

In this chapter, you learned how to set a working directory, create an R script, create an RStudio project, and orient yourself to written R tutorials. First, setting the working directory is often an important step when reading (importing) and writing (exporting objects) in R. You can use the `getwd` function to check where your current working directory is, whereas the `setwd` can be used to set a new working directory. Second, writing and saving your R code in an R Script file (.R) is an important step towards reproducible data management, analysis, and visualization. Third, creating an RStudio project can streamline data-analytic projects and provides some user-friendly features. Finally, written R tutorials are common in both printed and web-based formats, and thus it's worthwhile to familiarize yourself with how to follow along with these types of tutorials.

Chapter 3

Basic Features and Operations of the R Language

In this chapter, you will learn about basic features of the R language along with key bits of terminology. Think of this chapter as a "gentle introduction to R."

3.0.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/yHbVbHEjhLQ

3.0.0.2 Functions & Packages Introduced

Function	Package
`print`	base
`class`	base
`str`	base
`install.packages`	base
`library`	base
`is.numeric`	base
`is.integer`	base
`is.character`	base
`is.logical`	base
`as.Date`	base
`as.POSIXct`	base
`c`	base

Function	Package
`data.frame`	base
`names`	base

3.1 R as a Calculator

In its simplest form, R is a calculator. You can use R to carry out basic arithmetic, algebra, and other mathematical operations. The arithmetic operators in R are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `^` (exponent), and `sqrt` (square root). Below, you will find an example of these different arithmetic operators in action. In this book, lines of output are preceded by double hashtags (`##`); however, in your own R Console, you will not see the double hashtags before your output – unless, that is, you use double hashtags before your lines of script annotations.

```
3 + 2
```

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

```
3 - 2
```

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

```
3 * 2
```

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

```
3 / 2
```

```
## [1] 1.5
```

```
3 ^ 2
```

```
## [1] 9
```

```
sqrt(3)
```

```
## [1] 1.732051
```

Note how the six lines of output we generated (see above) appear in the same order in your Console; relatedly, remember that in R (like many other languages) the order of operations is important.

In R it doesn't matter whether there are spaces between the numeric values and the arithmetic operators. As such, we can write our code as follows and arrive at the same output.

```
3+2
```

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

```
3-2
```

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

```
3*2
```

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

```
3/2
```

```
## [1] 1.5
```

```
3^2
```

```
## [1] 9
```

```
sqrt(3)
```

```
## [1] 1.732051
```

3.2 Functions

A **function** refers to an integrated set of instructions that can be applied consistently. Some functions also accept arguments, where an **argument** is used to further refine the instructions and resulting operations of the function. In R we can use functions that come standard from base R or functions that come from downloadable packages. Let's take a look at the `print` function that comes standard with base R, which means that we don't need a special package to access the function. This won't be terribly exciting, but we can enter `3` as an argument within the `print` function parentheses; in general, arguments will appear within the inclusive parentheses.

```
print(3)
```

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

Note how the `print` function simply "printed" the numeric value `3` that we entered.

We can also do the classic - yet super cliche - "Hello world!" example to illustrate how R and the `print` function handle text/character/string data; except, let's change it to `"Hello HR Analytics!"`.

```
print("Hello HR Analytics!")
```

```
## [1] "Hello HR Analytics!"
```

Note how we have to put text/character/string data in quotation marks. We can use double (`" "`) or single quotes (`' '`). Some people prefer double quotes and some prefer single quotes. I happen to prefer double quotes.

Now, let's play around with the `class` function. The `class` function is used for determining the data type represented by a datum or by multiple data that are contained in a vector or variable. By entering `3` as an argument in the `class` variable, we find that the data type is `numeric`.

```
class(3)
```

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

If you would like to learn more about a function and the types of arguments that can be used within the function, you can access the help feature in R to access documentation on the function. The easiest way to do this is to enter `?` before the name of the function. Upon doing so, a help window will open; if you're using RStudio, a specific window pane dedicated to Help will open.

```
?class
```

3.3 Packages

A **package** is a collection of functions with a common theme or that can be applied to address a similar set of problems. R packages go through a rigorous and laborious development and vetting process before being posted on the CRAN website (https://cran.r-project.org/).

There are two functions that are important when it comes to installing and using packages. First, the `install.packages` function is used to install a package. The name of the package you wish to install should be surrounded with quotation marks (`" "` or `' '`) and entered as an argument in the function. For example, if we wish to install the `lessR` package (Gerbing, 2020), we type `install.packages("lessR")`, as shown below. Please note that the names of packages (and functions, arguments, and objects) are case sensitive in R.

```
install.packages("lessR")
```

Once you have installed a package, you use the `library` function to "check out" the package from your "library" of functions. To use the function, enter the exact name of the function *without* quotation marks.

```
library(lessR)
```

3.4 Variable Assignment

Variable assignment is the process of assigning a value or multiple values to a variable. There are two assignment operators that can be used for variable assignment as well as for (re)naming objects such as tables and data frames: `<-` and `=`. Both work the same way. I prefer to use `<-`, but others prefer `=`. In the example below, we assign the value `3` to a variable (i.e., object) we are naming `x`.

```
x <- 3
```

```
x = 3
```

Both functions achieved the same end, and the function that was run most recently overrides the previous attempt at assigning `3` to `x`. Using the `print` function we check with this worked.

```
print(x)
```

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

Or, instead of using the `print` function , we can simply run `x` by itself.

```
x
```

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

3.5 Types of Data

In general, there are four different types of data in R: `numeric`, `character`, `Date`, and `logical`.

3.5.1 `numeric` Data

`numeric` data are numbers or numeric values. This data type is ready-made for quantitative analysis. We can apply the `is.numeric` function to determine whether a value or variable is `numeric`; if the value or variable entered as an argument is `numeric`, R will return `TRUE`, and if it is not `numeric`, R will return `FALSE`. [Note that `TRUE` and `FALSE` statements don't require quotation marks like text/character/string data, as they are handled differently in R.] Finally, let's see if that `"Hello data science!"` phrase is numeric.

```
is.numeric(3)
```

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

```
is.numeric(TRUE)
```

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

```
is.numeric("Hello data science!")
```

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

An `integer` is a special type of `numeric` data. An `integer` does not have any decimals, and thus is a whole number. To specify that `numeric` data are of type `integer`, `L` must be appended to the value. For example, to specify that `3` is an `integer`, it should be written as `3L`. To verify that a value is in fact of type `integer`, we can apply the `as.integer` function.

```
is.integer(3L)
```

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

```
is.integer(3)
```

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

Alternatively, we can use the `class` or `str` functions to determine whether a value or variable is `integer` or `numeric`. The function `str` is used to identify the structure of an object (e.g., data frame, variable, value).

```
class(3L)
```

```
## [1] "integer"
```

```
str(3L)
```

```
##  int 3
```

```
class(3)
```

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

```
str(3)
```

```
##  num 3
```

Finally, if we assign a `numeric` or `integer` value to a variable, the resulting variable will take on the `numeric` or `integer` data type (respectively).

```
x <- 3
class(x)
```

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

```
x <- 3L
class(x)
```

```
## [1] "integer"
```

3.5.2 `character` Data

Data of type `character` do not explicitly or innately have quantitative properties. Sometimes this type of data is called "string" or "text" data. Data of type `factor` is similar to `character` but handled differently by R; this distinction becomes more important when working with vectors and analyses. That said, many analysis functions automatically convert `character` to `factor` for analyses, but when it comes to working with and manipulating data frames, this

`character` versus `factor` distinction becomes more important. When data are of type `character`, we place quotation marks (`" "` or `' '`) around the text. For example, if the `character` of interest is `old`, then we place quotation marks around text like this `"old"`. Also note that `character` data are case sensitive, which means that `"old"` is not the same as `"Old"`. Using the function `is.character`, we can determine whether data are in fact of type `character`.

```
is.character("old")
```

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

Note how omitting the `" "` results in an error message.

```
is.character(old)
```

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

Finally, if we assign a `numeric` or `integer` value to a variable, the resulting variable will take on the `numeric` or `integer` data types.

```
y <- "old"
class(y)
```

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

3.5.3 `Date` Data

When working with dates in R, there are two different types: `Date` and `POSIXct`. `Date` captures just the date, whereas `POSIXct` captures the date and time. Behind the scenes, R treats `Date` numerically as the number of days since January 1, 1970, and `POSIXct` as the number of seconds since January 1, 1970. To specify a value as a date, we can use the `as.Date` function.

```
z <- as.Date("1970-03-01")
class(z)
```

```
## [1] "Date"
```

If we convert a variable of type `Date` to `numeric` using the `as.numeric` function, the result is the number of days since January 1, 1970.

```
z <- as.Date("1970-03-01")
as.numeric(z)
```

```
## [1] 59
```

Now we can use the `as.POSIXct` function to specify a value as a date and time. Note the very specific format in which the data and time are to be written.

```
z <- as.POSIXct("1970-03-01 13:10")
class(z)
```

```
## [1] "POSIXct" "POSIXt"
```

If we convert a variable of type `POSIXct` to `numeric` using the `as.numeric` function, the result is the number of seconds since January 1, 1970.

```
z <- as.POSIXct("1970-03-01 13:10")
as.numeric(z)
```

```
## [1] 5173800
```

3.5.4 `logical` Data

Data that are of type `logical` can take on values of either `TRUE` or `FALSE`, which correspond to the integers `1` and `0`, respectively. As mentioned above, although `TRUE` and `FALSE` appear to be `character` or `factor` data, they are actually `logical` data, which means they do not require quotation marks (`" "` or `' '`).

```
w <- FALSE
class(w)
```

```
## [1] "logical"
```

```
is.logical(w)
```

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

3.6 Vectors

A **vector** is a group of data elements in a particular order that are all the same data type. To create a vector, we can use the `c` function, which stands for "combine." Within the `c` function parentheses, we can list the data elements and separate them by commas, as commas separate arguments within a function's parentheses. We can also assign a vector to a variable using either the `<-` or `=` operator. We can create vectors for all of the data types: `numeric`, `character`, `Date`, and `logical`.

As an example, let's create a vector of `numeric` values, and let's call it `a`.

```
a <- c(1, 4, 7, 11, 19)
```

Using the `class` and `print` functions, we can determine the class of our new `a` object and print its values, respectively.

```
class(a)
```

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

```
print(a)
```

```
## [1]  1  4  7 11 19
```

Let's repeat this process by creating vectors containing `integer`, `character`, `Date`, and `logical` values.

```
b <- c(3L, 10L, 2L, 5L, 5L)
class(b)
```

```
## [1] "integer"
```

```
print(b)
```

```
## [1]  3 10  2  5  5
```

```
c <- c("old", "young", "young", "old", "young")
class(c)
```

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

```
print(c)
```

```
## [1] "old"   "young" "young" "old"   "young"
```

```
d <- as.Date(c("2018-06-01", "2018-06-01", "2018-10-31", "2018-01-01", "2018-06-01"))
class(d)
```

```
## [1] "Date"
```

```
print(d)
```

```
## [1] "2018-06-01" "2018-06-01" "2018-10-31" "2018-01-01" "2018-06-01"
```

```
e <- c(TRUE, TRUE, TRUE, FALSE, FALSE)
class(e)
```

```
## [1] "logical"
```

```
print(e)
```

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

We can also perform mathematical operations on vectors. For instance, we can multiply vector `a` (which we created above) by a numeric value, and as a result each vector value will be multiplied by that value. This is an important type of operation to remember when it comes time to transform a variable.

```
a * 11
```

```
## [1]  11  44  77 121 209
```

Note that performing mathematical operations on a vector does not automatically change the properties of the vector itself. If you inspect the `a` vector, you will see that the original data (e.g., `1, 4, 7, 11, 19`) remain.

```
print(a)
```

```
## [1]  1  4  7 11 19
```

If we want to overwrite a vector with new values based on our operations, we can use `<-` or `=` to name the new vector (which, if named the same thing as the old vector, will override the old vector) and, ultimately, to create a vector with the operations applied to the original values.

```
a <- a * 11
print(a)
```

```
## [1]  11  44  77 121 209
```

To revert back to the original vector values for object `a`, we can simply specify the original values using the `c` function once more.

```
a <- c(1, 4, 7, 11, 19)
```

Let's now apply subtraction, addition, and division operators to the vector. Note that R adheres to the standard mathematical orders of operation.

```
(3 + a) / 2 - 1
```

```
## [1]  1.0  2.5  4.0  6.0 10.0
```

We can also perform mathematical operations on vectors of the same length (i.e., with the same number of data elements). In order, the mathematical operator will be applied to each pair of vector values from the respective vectors. Let's begin by creating a new vector called `f`.

```
f <- c(3, 1, 3, 5, 3)
```

Both `a` and `f` are the same length, which means we can multiply, add, divide, subtract, and exponentiate

```
a * f
```

```
## [1]  3  4 21 55 57
```

```
a + f
```

```
## [1]  4  5 10 16 22
```

```
a / f
```

```
## [1] 0.3333333 4.0000000 2.3333333 2.2000000 6.3333333
```

```
a - f
```

```
## [1] -2  3  4  6 16
```

```
a ^ f
```

```
## [1]      1      4    343 161051   6859
```

3.7 Lists

If we wish to combine data elements into a single list that with different data types, we can use the `list` function. The `list` function orders each data element and retains its value.

```
g <- list(1, "dog", TRUE, "2018-05-30")
print(g)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "dog"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] "2018-05-30"
```

```
class(g)
```

```
## [1] "list"
```

3.8 Data Frames

A **data frame** is a specific type of table in which columns represent variables (i.e., fields) and rows represent cases (i.e., observations). We can create a simple data frame object by combining vectors of the same length. Let's begin by creating six vector objects, which we will label `a` through `f`.

```
a <- c(1, 4, 7, 11, 19)
b <- c(3L, 10L, 2L, 5L, 5L)
c <- c("old", "young", "young", "old", "young")
d <- as.Date(c("2018-06-01", "2018-06-01", "2018-10-31", "2018-01-01", "2018-06-01"))
e <- c(TRUE, TRUE, TRUE, FALSE, FALSE)
f <- c(3, 1, 3, 5, 3)
```

Using the `data.frame` function from base R we can combine the six vectors to create a data frame object. All we need to do is enter the names of the six vectors as separate arguments in the function parentheses. Just as we did with the vectors, we can name the data frame object using the `<-` operator (or `=` operator). Let's name this data frame object `r`.

```
r <- data.frame(a, b, c, d, e, f)
```

Using the `print` function, we can view the contents of our new data frame object called `r`.

```
print(r)
```

```
##    a  b     c          d     e f
## 1  1  3   old 2018-06-01  TRUE 3
## 2  4 10 young 2018-06-01  TRUE 1
## 3  7  2 young 2018-10-31  TRUE 3
## 4 11  5   old 2018-01-01 FALSE 5
## 5 19  5 young 2018-06-01 FALSE 3
```

We can also rename the columns (i.e., variables) of the data frame object by using the `names` function from base R along with the `c` function from base R.

```
names(r) <- c("TenureSup", "TenureOrg", "Age", "HireDate", "FTE", "NumEmp")
```

To view the changes to our data frame object, use the `print` function once more.

```
print(r)
```

```
##   TenureSup TenureOrg   Age   HireDate   FTE NumEmp
## 1         1         3   old 2018-06-01  TRUE      3
## 2         4        10 young 2018-06-01  TRUE      1
## 3         7         2 young 2018-10-31  TRUE      3
## 4        11         5   old 2018-01-01 FALSE      5
## 5        19         5 young 2018-06-01 FALSE      3
```

Finally, we can use the `class` function to verify that the object is in fact a data frame.

```
class(r)
```

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

3.9 Annotations

Part of the value of using a code/script-based program like R is that you can leave notes and explain your decisions and operations. When preceding text, the # symbol indicates that all text that follows on that line is a comment or annotation; as a result, R knows not to interpret or analyze the text that follows. To illustrate annotations, let's repeat the steps from the previous section; however, this time, let's include annotations.

```
# Create six vectors
a <- c(1, 4, 7, 11, 19) # Vector a
b <- c(3L, 10L, 2L, 5L, 5L) # Vector b
c <- c("old", "young", "young", "old", "young") # Vector c
d <- as.Date(c("2018-06-01", "2018-06-01", "2018-10-31", "2018-01-01", "2018-06-01")) # Vector d
e <- c(TRUE, TRUE, TRUE, FALSE, FALSE) # Vector e
f <- c(3, 1, 3, 5, 3) # Vector f

# Combine vectors into data frame
r <- data.frame(a, b, c, d, e, f)

# Print data frame
print(r)
```

```
##    a  b     c          d     e f
## 1  1  3   old 2018-06-01  TRUE 3
## 2  4 10 young 2018-06-01  TRUE 1
## 3  7  2 young 2018-10-31  TRUE 3
## 4 11  5   old 2018-01-01 FALSE 5
## 5 19  5 young 2018-06-01 FALSE 3
```

```
# Rename columns in data frame
names(r) <- c("TenureSup", "TenureOrg", "Age", "HireDate", "FTE", "NumEmp")

# Print data frame
print(r)
```

```
##   TenureSup TenureOrg   Age   HireDate   FTE NumEmp
## 1         1         3   old 2018-06-01  TRUE      3
## 2         4        10 young 2018-06-01  TRUE      1
## 3         7         2 young 2018-10-31  TRUE      3
## 4        11         5   old 2018-01-01 FALSE      5
## 5        19         5 young 2018-06-01 FALSE      3
```

```
# Determine class of object
class(r)
```

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

Can you start to envision how annotated code might help to tell a story about data-related decision-making processes?

3.10 Summary

This chapter provided you with a gentle introduction to R. This chapter is by no means comprehensive, but hopefully it provided you with an understanding of the basic operations and building blocks of R.

Part II

Data Acquisition

Chapter 4

Reading Data into R

Reading data refers to the process of importing data from a working directory or website into the R environment. When we read a data file into R, we often read it in as a **data frame (df)**, where a data frame is a tabular display with columns representing variables and rows representing cases. Many different data file formats can be read into R as data frames, such as .csv, .xls/x, .txt, .sas7bdat (SAS), and .sav (SPSS). Finally, as you will learn in this tutorial, different functions can be used to read data into R.

4.0.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/smWjqhaxHY8

4.0.0.2 Functions & Packages Introduced

Function	Package
`read.csv`	base
`read_csv`	`readr`
`Read`	`lessR`
`excel_sheets`	`readxl`
`read_excel`	`readxl`
`View`	base
`print`	base
`head`	base
`tail`	base
`names`	base
`colnames`	base
`install.packages`	base

Function	Package
`library`	base
`list.files`	base

4.0.0.3 Initial Steps

Please note, that any function that appears in the *Initial Steps* section has been covered in a previous chapter. If you need a refresher, please view the relevant chapter. In addition, a previous chapter may show you how to perform the same action using different functions or packages.

To get started, please save the following data files into a folder on your computer that you will set as your working directory: **"PersData.csv"** and **"PersData_Excel.xlsx"**. As a reminder, you can access all of the data files referenced in this book by downloading them as a compressed (zipped) folder from the my GitHub site: https://github.com/davidcaughlin/R-Tutorial-Data-Files; once you've followed the link to GitHub, just click "Code" (or "Download") followed by "Download ZIP", which will download all of the data files referenced in this book. For the sake of parsimony, I recommend downloading all of the data files into the same folder on your computer, which will allow you to set that same folder as your working directory for each of the chapters in this book.

Next, set your working directory by using the `setwd` function (see below) or by doing it using drop-down menus. Your working directory folder will likely be different than the one shown below; "H:/RWorkshop" just happens to be the name of the folder that I save my data files to and that I set as my working directory. You can manually set your working directory folder in your drop-down menus by going to *Session* > *Set Working Directory* > *Choose Directory…*. If you need a refresher on how to set a working directory, please refer to Setting a Working Directory.

```
# Set your working directory to the folder containing your data file
setwd("H:/RWorkshop")
```

Finally, I highly recommend that you create a new R Script file (.R), which will allow you to edit and save your script and annotations. To learn more, please refer to Creating & Saving an R Script.

4.1 Read Data

One of the easiest data file formats to work with when reading data into R is the .csv (comma-separated values) format. The .csv (comma-separated values) format is commonly used among R users, and such files can be created in

Microsoft Excel and Google Sheets (as well as other programs). For example, many survey, data analysis, and data-acquisition platforms allow data to be exported to .csv files. When getting started in R, the way in which the .csv file is formatted can make your life easier. Specifically, the most straightforward .csv file format to read in is one in which the first row contains the name of each variable in each column, and in which the second row contains the first row of observed values (i.e., data) for the cases (i.e., observations, entities, people, units). Later in the chapter, I will show you how to read in .csv files in which the observed values do not begin until the third row or later; in addition, I will demonstrate how to read in other file formats. however, as mentioned above, other file formats can be read into R as well.

In this tutorial, you will learn how to read data into R using four different functions. If there are any missing values in your data, each function we cover will replace those missing values with `NA` by default. *I personally recommend that you get comfortable with Option 2 (`read_csv` function from `readr` package), as this function has some advantages when it comes to reading in .csv files specifically.*

4.1.1 Option 1: `read.csv` Function from Base R

The `read.csv` file comes standard with base R, which means that you don't need to install a package to access the function. As the function name implies, this function is used when the source data file is in .csv format. Typically, the `read.csv` function requires only a single argument within the parentheses, which will be the *exact* name of the data file enclosed with quotation marks; the file should be located your working directory folder. Remember, R is a language where case and space sensitivity matters when it comes to names; meaning, if there are spaces in your file name, there needs to be spaces when the file name appears in your R script, and if some letters are upper case in your file name, there needs to be corresponding upper-case letters in your R script. Let's practice reading in a file called **"PersData.csv"** by entering the exact name of the file followed by the .csv extension, all within in quotation marks. Remember, the file called **"PersData.csv"** should already be saved in your working directory folder (see *Initial Steps*.

```
# Read data from working directory
read.csv("PersData.csv")
```

```
##    id   lastname firstname startdate gender
## 1 153    Sanchez Alejandro  1/1/2016   male
## 2 154   McDonald    Ronald  1/9/2016   male
## 3 155      Smith      John  1/9/2016   male
## 4 165        Doe      Jane  1/4/2016 female
## 5 125   Franklin  Benjamin  1/5/2016   male
```

```
## 6 111     Newton     Isaac  1/9/2016   male
## 7 198    Morales     Linda  1/7/2016 female
## 8 201 Providence     Cindy  1/9/2016 female
## 9 282     Legend      John  1/9/2016   male
```

As you can see, the data that appear in your Console contains only a handful of rows and columns; nonetheless, this gives you an idea of how the read.csv function works.

Often, you will want to create a data frame object that is stored in your Global Environment for subsequent use. By creating a data frame object, you can manipulate and/or analyze the data within the object using a variety of functions (and without changing the data in the source file). To create a data frame object, we simply (a) use the same `read.csv` function from above, (b) add either a `<-` or `=` to the left of the `read.csv` function, and (c) create a name of our choosing for the data frame object by entering that name to the left of the `<-` or `=`. You can name your data frame object whatever you would like as long as it doesn't include spaces, doesn't start with a numeral, and doesn't include special characters like `*` or `-` (to name a few). I recommend choosing a name that is relatively short but descriptive, and that is not the same as another R function or variable name that you plan to use. Below, I name the new data frame object `personaldata`.

```
# Read in data and name data frame object
personaldata <- read.csv("PersData.csv")
```

If your data file resides in a folder other than your specified working directory, then you can simply add the path directory followed by a forward slash (`/`) before the file name. Please note that your working directory will almost certainly be different than the one I show below.

```
# Read data and name data frame object
personaldata <- read.csv("H:/RWorkshop/PersData.csv")
```

If you are working in RStudio, you will see the data frame object appear in your Global Environment window, as shown below. If you click on the name of the data frame object in your Global Environment window, a new tab will open up, allowing you to view the data.

Alternatively, you can use the `View` function from base R with the name of the data frame object we just created as the parenthetical argument. Note that the `View` function begins with an upper-case `V`. Remember, R is case and space sensitive when it comes to function names. Further, the name of the data frame object you enter into the parentheses of the function must be *exactly* the same as what you originally named the data frame object when you created it (e.g., read it into R and named it). That is, R won't recognize the data frame object if you

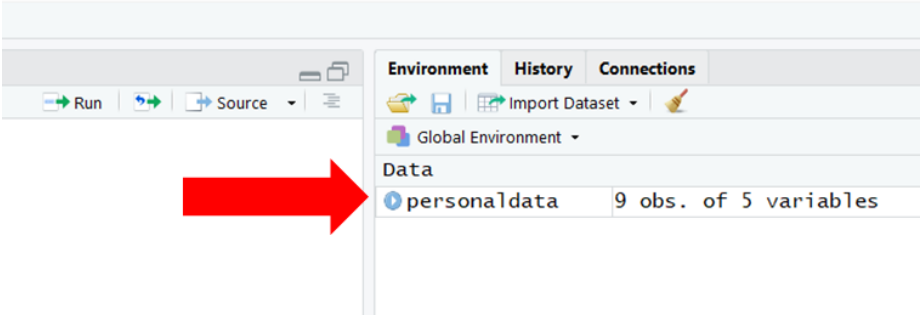


Figure 4.1:

type it as `PersonalData`, but R will recognize it if you type it as `personaldata`. Sometimes it helps to copy and paste the exact names of functions and variables into the function parentheses.

```
# View data within data frame object
View(personaldata)
```

Instead of using the `View` function, you could just "run" the name of the data frame object by highlighting `personaldata` in your R Script and clicking "Run" (or you can enter the name of the data frame object directly into your Console command line and click Enter). Another option is to use the `print` function (from base R) with the name of the data frame object as the sole argument in the parentheses. Similarly, if you have many rows of data, you can use the `head` function from base R to see just the first 6 rows of data, or you can use the `tail` function from base R to see the last 6 rows of data.

```
# Highlight the name of data frame object and click Run to view data in Console
personaldata
```

```
##    id   lastname firstname startdate gender
## 1 153    Sanchez Alejandro  1/1/2016   male
## 2 154   McDonald    Ronald  1/9/2016   male
## 3 155      Smith      John  1/9/2016   male
## 4 165        Doe      Jane  1/4/2016 female
## 5 125   Franklin  Benjamin  1/5/2016   male
## 6 111     Newton     Isaac  1/9/2016   male
## 7 198    Morales     Linda  1/7/2016 female
## 8 201 Providence     Cindy  1/9/2016 female
## 9 282     Legend      John  1/9/2016   male
```

```
# Use print function with the name of the data frame object to view data in Console
print(personaldata)
```

```
##    id   lastname firstname startdate gender
## 1 153    Sanchez Alejandro  1/1/2016   male
## 2 154   McDonald    Ronald  1/9/2016   male
## 3 155      Smith      John  1/9/2016   male
## 4 165        Doe      Jane  1/4/2016 female
## 5 125   Franklin  Benjamin  1/5/2016   male
## 6 111     Newton     Isaac  1/9/2016   male
## 7 198    Morales     Linda  1/7/2016 female
## 8 201 Providence     Cindy  1/9/2016 female
## 9 282     Legend      John  1/9/2016   male
```

```
# View just the first 6 rows of the data frame object in Console
head(personaldata)
```

```
##    id lastname firstname startdate gender
## 1 153  Sanchez Alejandro  1/1/2016   male
## 2 154 McDonald    Ronald  1/9/2016   male
## 3 155    Smith      John  1/9/2016   male
## 4 165      Doe      Jane  1/4/2016 female
## 5 125 Franklin  Benjamin  1/5/2016   male
## 6 111   Newton     Isaac  1/9/2016   male
```

```
# View just the last 6 rows of the data frame object in Console
tail(personaldata)
```

```
##    id   lastname firstname startdate gender
## 4 165        Doe      Jane  1/4/2016 female
## 5 125   Franklin  Benjamin  1/5/2016   male
## 6 111     Newton     Isaac  1/9/2016   male
## 7 198    Morales     Linda  1/7/2016 female
## 8 201 Providence     Cindy  1/9/2016 female
## 9 282     Legend      John  1/9/2016   male
```

As a final note, where available, you can use the `read.csv` function to read in .csv data from a website. For example, rather than save the .csv file to a folder on your computer, you can read in the raw data directly from my GitHub site. Within the quotation marks (`" "`), simply paste in the following URL: https://raw.githubusercontent.com/davidcaughlin/R-Tutorial-Data-Files/master/PersData.csv.

```
# Read data using URL
personaldata <- read.csv("https://raw.githubusercontent.com/davidcaughlin/R-Tutorial-Data-Files/m
```

Note that by naming the data frame object `personaldata` we have overwritten the previous version of the object with that same name.

4.1.2 Option 2: `read_csv` Function from `readr` Package

As part of the `tidyverse` of R packages (Wickham, 2019; Wickham et al., 2019), the `readr` package (Wickham et al., 2018) and its functions can be used to read in a few different data file formats (as long as they are rectangular), including .csv files. We will use the `read_csv` function from the package, which as the name implies is used to read in .csv files. Among other advantages over the `read.csv` function we learned in Option 1, the `read_csv` function is notably faster. Further, `read_csv` creates a tibble (as opposed to a data frame), which behaves like a data frame for most purposes; for more information on tibbles, check out Wickham and Grolemund's (2017) chapter on tibbles: http://r4ds.had.co.nz/tibbles.html.

To use the `read_csv` function, the `readr` package must be installed and accessed using the `install.packages` and `library` functions, respectively. Type `"readr"` (note the quotation marks) into the parentheses of the `install.packages` function. Next, type `readr` (without quotation marks) into the parentheses of the `library` function.

Just like with the `read.csv` function, enter the *exact* name of the data file (as named in your working directory), followed by .csv – and all within quotation marks (`" "`). Further, either the `<-` or `=` operator can be used to name the data frame object. Below, I name the data frame object `personaldata2` to distinguish it from the data frame object we previously read in and named using the `read.csv` function.

```
# Install readr package
install.packages("readr")
```

```
# Access readr package
library(readr)

# Read data and name data frame object
personaldata2 <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
```

```
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
# View just the first 6 rows of the data frame in Console
head(personaldata2)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   153 Sanchez  Alejandro 1/1/2016  male
## 2   154 McDonald Ronald    1/9/2016  male
## 3   155 Smith    John      1/9/2016  male
## 4   165 Doe      Jane      1/4/2016  female
## 5   125 Franklin Benjamin  1/5/2016  male
## 6   111 Newton   Isaac     1/9/2016  male
```

Where available, you can also use the `read_csv` function to read in .csv data from a website. For example, rather than save the .csv file to a folder on your computer, you can read in the raw data directly from my GitHub site. Within the quotation marks (" "), simply paste in the following URL: https://raw.githubusercontent.com/davidcaughlin/R-Tutorial-Data-Files/master/PersData.csv.

```
# Read data using URL
personaldata2 <- read_csv("https://raw.githubusercontent.com/davidcaughlin/R-Tutorial-
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

Note that by naming the data frame object `personaldata2` we have overwritten the previous version of the object with that same name.

4.1.3 Option 3: `Read` Function from `lessR` Package

Just like the `read.csv` and `read_csv` functions, the `Read` function from the `lessR` package (Gerbing, 2020) can read in .csv files; however, it can also read

in other file formats like .xls/x, .sas7bdat (SAS), and .sav (SPSS). When reading in a .csv file using the `Read` function, the *exact* name of your data file from your working directory needs to be entered as an argument (followed by .csv and surrounded by quotation marks). Further, either the `<-` or `=` operator can be used to name the data frame object. To use the `Read` function, the `lessR` package needs to be installed and accessed using the `install.packages` and `library` functions, respectively.

```
# Install lessR package
install.packages("lessR")
```

```
# Access lessR package
library(lessR)

# Read data and name data frame object
personaldata3 <- Read("PersData.csv")
```

```
##
## >>> Suggestions
## To read a csv or Excel file of variable labels, var_labels=TRUE
##    Each row of the file:  Variable Name, Variable Label
## Details about your data, Enter:  details()  for d, or  details(name)
##
## Data Types
## ------------------------------------------------------------
## character: Non-numeric data values
## integer: Numeric data values, integers only
## ------------------------------------------------------------
##
##      Variable                   Missing  Unique
##          Name      Type  Values  Values  Values   First and last values
## ---------------------------------------------------------------------------------------
##  1         id   integer      9       0       9   153  154  155 ... 198  201  282
##  2   lastname character      9       0       9   Sanchez  McDonald ... Providence  Legend
##  3  firstname character      9       0       8   Alejandro  Ronald ... Cindy  John
##  4  startdate character      9       0       5   1/1/2016  1/9/2016 ... 1/9/2016  1/9/2016
##  5     gender character      9       0       2   male  male  male ... female  female  male
## ---------------------------------------------------------------------------------------
```

```
# View just the first 6 rows of the data frame object in Console
head(personaldata3)
```

```
##    id lastname firstname startdate gender
## 1 153  Sanchez Alejandro  1/1/2016   male
```

```
## 2 154 McDonald    Ronald  1/9/2016   male
## 3 155    Smith      John  1/9/2016   male
## 4 165      Doe      Jane  1/4/2016 female
## 5 125 Franklin  Benjamin  1/5/2016   male
## 6 111   Newton     Isaac  1/9/2016   male
```

Where available, you can also use the `Read` function to read in data from a website. For example, rather than save the .csv file to a folder on your computer, you can read in the raw data directly from my GitHub site. Within the quotation marks (" "), simply paste in the following URL: https://raw.githubusercontent.com/davidcaughlin/R-Tutorial-Data-Files/master/PersData.csv.

```
# Read data using URL
personaldata3 <- Read("https://raw.githubusercontent.com/davidcaughlin/R-Tutorial-Data-
```

```
##
## >>> Suggestions
## To read a csv or Excel file of variable labels, var_labels=TRUE
##    Each row of the file:  Variable Name, Variable Label
## Details about your data, Enter:  details()  for d, or  details(name)
##
## Data Types
## ------------------------------------------------------------
## character: Non-numeric data values
## integer: Numeric data values, integers only
## ------------------------------------------------------------
##
##      Variable                   Missing  Unique
##          Name     Type  Values  Values  Values   First and last values
## ------------------------------------------------------------------------------
##  1         id   integer      9       0       9   153  154  155 ... 198  201  282
##  2   lastname character      9       0       9   Sanchez  McDonald ... Providence
##  3  firstname character      9       0       8   Alejandro  Ronald ... Cindy  John
##  4  startdate character      9       0       5   1/1/2016  1/9/2016 ... 1/9/2016  1/
##  5     gender character      9       0       2   male  male  male ... female  female
## ------------------------------------------------------------------------------
```

Note that by naming the data frame object `personaldata3` we have overwritten the previous version of the object with that same name.

For more information on the `Read` function from the `lessR` package, check out David Gerbing's website for the package and specifically the section with links to video tutorials: http://www.lessrstats.com/videos.html.

4.1.4 Option 4: `read_excel` Function from `readxl` Package

Reading in Excel workbook files with more than one worksheet requires a bit more work. To read in a .xlsx file with two worksheets, we will use the `excel_sheets` and `read_excel` functions from the `readxl` package (Wickham and Bryan, 2019).

```
# Install readxl package
install.packages("readxl")
```

```
# Access readxl package
library(readxl)
```

To view the worksheets within an Excel workbook file, simply type the name of the `excel_sheets` function, and as the sole parenthetical argument type the exact name of the data file with the .xlsx extension – all within quotation marks (i.e., "PersData_Excel.xlsx").

```
# View Excel file worksheets
excel_sheets("PersData_Excel.xlsx")
```

```
## [1] "Year1" "Year2"
```

Note that the .xlsx file contains two worksheets called "Year1" and "Year2". We can now reference each of these worksheets when reading in the data from the Excel workbook file. To do so, we will use the `read_excel` function. As the first argument, enter the *exact* name of the data file (as named in your working directory), followed by .xlsx – and all within quotation marks (`" "`). As the second argument, type `sheets=` followed by the name of the worksheet containing the data you wish to read in; let's read in the data from the worksheet called "Year1". Finally, either the `<-` or `=` operator can be used to name the data frame object. Below, I name the data frame object `personaldata4`. *Remember to type a comma (`,`) before the second argument, as this is how we separate arguments from one another when there are more than one.*

```
# Read data from sheet called "Year1" and name data frame object
personaldata4 <- read_excel("H:/RWorkshop/PersData_Excel.xlsx", sheet="Year1")
```

```
# View the data frame object in Console
print(personaldata4)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate           gender
```

```
##   <dbl> <chr>      <chr>     <dttm>              <chr>
## 1   153 Sanchez    Alejandro 2016-01-01 00:00:00 male
## 2   154 McDonald   Ronald    2016-01-09 00:00:00 male
## 3   155 Smith      John      2016-01-09 00:00:00 male
## 4   165 Doe        Jane      2016-01-04 00:00:00 female
## 5   125 Franklin   Benjamin  2016-01-05 00:00:00 male
## 6   111 Newton     Isaac     2016-01-09 00:00:00 male
## 7   198 Morales    Linda     2016-01-07 00:00:00 female
## 8   201 Providence Cindy     2016-01-09 00:00:00 female
## 9   282 Legend     John      2016-01-09 00:00:00 male
```

Let's repeat the process for the worksheet called "Year2".

```
# Read data from sheet called "Year2" and name data frame object
personaldata5 <- read_excel("H:/RWorkshop/PersData_Excel.xlsx", sheet="Year2")
```

```
# View the data frame object in Console
print(personaldata5)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate           gender
##   <dbl> <chr>      <chr>     <dttm>              <chr>
## 1   153 Sanchez    Alejandro 2016-01-01 00:00:00 male
## 2   155 Smith      John      2016-01-09 00:00:00 male
## 3   165 Doe        Jane      2016-01-04 00:00:00 female
## 4   125 Franklin   Benjamin  2016-01-05 00:00:00 male
## 5   111 Newton     Isaac     2016-01-09 00:00:00 male
## 6   201 Providence Cindy     2016-01-09 00:00:00 female
## 7   282 Legend     John      2016-01-09 00:00:00 male
## 8   312 Ramos      Jorge     2017-03-01 00:00:00 male
## 9   395 Lucas      Nadia     2017-03-04 00:00:00 female
```

4.2 Special Topics

Thus far in this chapter, I have showcased some of the most common approaches to reading in data files, with an emphasis on reading in .csv files with the first row corresponding to the column (variable) names and the remaining rows containing the substantive data for cases. There are, however, other challenges and considerations you might encounter along the way, and this section, I cover some special topics related to reading data into R.

4.2.1 List Data File Names in Working Directory

If you would like to obtain the exact names of files located in a (working) directory, the `list.files` function from base R comes in handy. This function will return a list of all file names within a particular directory or file names that meet a particular pattern. For our purposes, let's identify all of the .csv data file names contained within our current working directory. As the first argument, type `path=` followed by the path associated with your working directory. Second, because we are only pulling the file names associated with .csv files, enter the argument `all.files=FALSE`. Third, type the argument `full.names=FALSE` to indicate that we do not want the path to precede the file names. Finally, type the argument `pattern=".csv"` to request the names of only those file names that match the regular expression of ".csv" will be returned.

```
# List data file names in working directory
list.files(path="H:/RWorkshop",
           all.files=FALSE,
           full.names=FALSE,
           pattern=".csv")
```

```
##  [1] "Aft_Monday_Week1.csv"
##  [2] "Aft_Tuesday_Week1.csv"
##  [3] "Aft_Wednesday_Week1.csv"
##  [4] "ANOVA_PerformanceMGMT.csv"
##  [5] "Baseline.csv"
##  [6] "CData.csv"
##  [7] "ChiSquareTurnover.csv"
##  [8] "DataCleaningExample.csv"
##  [9] "Descriptive Statistics.csv"
## [10] "DiffPred.csv"
## [11] "Edited PersData.csv"
## [12] "EmployeeDemographics.csv"
## [13] "EmployeeSurveyData.csv"
## [14] "EmployeeSurveyExample.csv"
## [15] "Eve_Monday_Week1.csv"
## [16] "Eve_Tuesday_Week1.csv"
## [17] "Eve_Wednesday_Week1.csv"
## [18] "Headcount.csv"
## [19] "KruskalWallis.csv"
## [20] "lasso.csv"
## [21] "ManipulatingData.csv"
## [22] "MarketPayLine.csv"
## [23] "MarketSurveyData.csv"
## [24] "MMR.csv"
## [25] "Morn_Monday_Week1.csv"
```

```
## [26] "Morn_Tuesday_Week1.csv"
## [27] "Morn_Wednesday_Week1.csv"
## [28] "Nonlinear.csv"
## [29] "PayDeterminants.csv"
## [30] "PayEquity.csv"
## [31] "PData.csv"
## [32] "PerfData.csv"
## [33] "PerfMgmtRewardSystemsExample.csv"
## [34] "PersData.csv"
## [35] "PlannedBehavior.csv"
## [36] "Practice Table.csv"
## [37] "PredictiveAnalytics.csv"
## [38] "PulseSurvey.csv"
## [39] "Regression.csv"
## [40] "Sample1.csv"
## [41] "Sample2.csv"
## [42] "SelectionExercise.csv"
## [43] "Spearman.csv"
## [44] "Survival.csv"
## [45] "TrainingEval_inclass.csv"
## [46] "TrainingEvaluation_PrePostControl.csv"
## [47] "TrainingEvaluation_PrePostOnly.csv"
## [48] "TrainingEvaluation_ThreeGroupPost.csv"
## [49] "Turnover.csv"
## [50] "WilcoxonRankSum.csv"
## [51] "WilcoxonSignedRank.csv"
```

In your Console, you should see the list of file names you requested. You could then copy specific file names that you wish to read into R.

4.2.2 Skip Rows of Data When Reading

Some survey platforms like Qualtrics allow for data to be downloaded in .csv format; however, sometimes these platforms include variable name and label information in the second and even third rows of data as opposed to in just the first row. Fortunately, we can skip rows when reading in such data files.

Let's pretend that the first row of the **"PersData.csv"** data file contains variable names, and the second and third rows contain variable label information and explanations. We can nest the `read_csv` function (see above) within the `names` function, which will result in a vector of names from the first row of the data file. Using the `<-` operator, let's name this vector `var_names` so that we can reference it in the subsequent step.

```
# Read variable names from first row of data
var_names <- names(read_csv("PersData.csv"))
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

Next, using the `read_csv` function, we will read in the data file, skip the first three rows, and add the variable names we pulled in the previous step. As usual, as the first argument of the `read_csv` function, type the exact name of the data file you wish to read in within quotation marks (`" "`). As the second argument, type `skip=3` to indicate that you wish to skip the first three rows when reading in the data. As the third argument, type `col_names=` followed by the name of the `var_names` vector object we created in the previous step. Using the `<-` operator, let's name this data frame object `test`.

```
# Read data file (but skip rows 1-3) & introduce variable names
test <- read_csv("PersData.csv",
                 skip=3,
                 col_names=var_names)
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

Finally, let's see the fruits of our labor by printing the contents of the `test` data frame object to our Console.

```
# Print data frame object
print(test)
```

```
## # A tibble: 7 x 5
##      id lastname   firstname startdate gender
```

```
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   155 Smith      John      1/9/2016  male
## 2   165 Doe        Jane      1/4/2016  female
## 3   125 Franklin   Benjamin  1/5/2016  male
## 4   111 Newton     Isaac     1/9/2016  male
## 5   198 Morales    Linda     1/7/2016  female
## 6   201 Providence Cindy     1/9/2016  female
## 7   282 Legend     John      1/9/2016  male
```

4.3 Summary

Reading data into R is an important first step, and often, it is the step that causes the most problems for new R users. The `read.csv`, `read_csv`, and `Read` functions can all be used to read data into R. The `read_csv` has the advantage of being fast, which can be helpful when reading in large data files. The `Read` function has the advantage of being able to read in data file formats other than .csv. With all that said, if you're working with smaller data files in the .csv format, the `read.csv` format typically works just fine. In all subsequent tutorials, I use the `read_csv` function from the `readr` package. Finally, I also demonstrated how to read in Excel workbooks (.xlsx) files using the `read_excel` function as well as introduced some special topics related to reading data into R.

Chapter 5

Removing and Adding Variable Names

When working with a data frame object, you may encounter situations in which it makes sense to remove the variable names (and not the variable data) or to add or replace variables names. In this chapter, you will learn simple techniques for removing the variable names (i.e., column names) from a data frame object and adding (or replacing) variable names in a data frame object.

5.0.0.1 Video Tutorial

Link to Video Tutorial: XXXX

5.0.0.2 Functions & Packages Introduced

Function	Package
`names`	base
`colnames`	base
`c`	base
`head`	base

5.0.0.3 Initial Steps

If you haven't already, save the file called **"PersData.csv"** into a folder that you will subsequently set as your working directory. Your working directory will likely be different than the one shown below (i.e., `"H:/RWorkshop"`). As a reminder, you can access all of the data files referenced in this book by

downloading them as a compressed (zipped) folder from the my GitHub site: https://github.com/davidcaughlin/R-Tutorial-Data-Files; once you've followed the link to GitHub, just click "Code" (or "Download") followed by "Download ZIP", which will download all of the data files referenced in this book. For the sake of parsimony, I recommend downloading all of the data files into the same folder on your computer, which will allow you to set that same folder as your working directory for each of the chapters in this book.

Next, using the `setwd` function, set your working directory to the folder in which you saved the data file for this chapter. Alternatively, you can manually set your working directory folder in your drop-down menus by going to *Session > Set Working Directory > Choose Directory…*. Be sure to create a new R script file (.R) or update an existing R script file so that you can save your script and annotations. If you need refreshers on how to set your working directory and how to create and save an R script, please refer to Setting a Working Directory and Creating & Saving an R Script.

```
# Set your working directory
setwd("H:/RWorkshop")
```

Next, read in the .csv data file called **"PersData.csv"** using your choice of read function. In this example, I use the `read_csv` function from the `readr` package (Wickham et al., 2018). If you choose to use the `read_csv` function, be sure that you have installed and accessed the `readr` package using the `install.packages` and `library` functions. *Note: You don't need to install a package every time you wish to access it; in general, I would recommend updating a package installation once ever 1-3 months.* For refreshers on installing packages and reading data into R, please refer to Packages and Reading Data into R.

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)

# Read data and name data frame (tibble) object
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
```

```
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
# View the names of the variables in the data frame (tibble) object
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
# View data frame (tibble) object
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

As you can see from the output generated in your console, the `personaldata` data frame object contains basic employee demographic information. The variable names include: `id`, `lastname`, `firstname`, `startdate`, and `gender`. *Technically, the `read_csv` function reads in what is called a "tibble" object (as opposed to a data frame object), but for our purposes a tibble will behave similarly to a data frame. For more information on tibbles, check out Wickham and Grolemund's (2017) chapter on tibbles: http://r4ds.had.co.nz/tibbles.html.*

5.1 Remove Variable Names from a Data Frame Object

In some instances, you may wish to remove the variable names from a data frame. For example, I sometimes write (i.e., export) a data frame object I've been cleaning in R so that I may use the data file with the statistical software program called Mplus (Muthén and Muthén, 2018). Because Mplus does accept variable names within its data files, I may drop the variable names from the data frame object prior to writing to my working directory.

To remove variable names, just apply the `names` function with the data frame name as the argument, and then use either the `<-` operator with `NULL` to remove the variable names.

```
# Remove variable names
names(personaldata) <- NULL

# View just the first 6 rows of the data frame object in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##   <dbl> <chr>    <chr>     <chr>    <chr>
## 1   153 Sanchez  Alejandro 1/1/2016 male
## 2   154 McDonald Ronald    1/9/2016 male
## 3   155 Smith    John      1/9/2016 male
## 4   165 Doe      Jane      1/4/2016 female
## 5   125 Franklin Benjamin  1/5/2016 male
## 6   111 Newton   Isaac     1/9/2016 male
```

As you can see, the variable names do not appear in the overwritten `personaldata` data frame object.

5.2 Add Variable Names from a Data Frame Object

In other instances, you might find yourself with a dataset that lacks variable names (or has variable names that need to be replaced), which means that you will need to add those variable names to the data frame.

Let's work with the `personaldata` data frame object from the previous section for practice. To add variable names, we can use the `colnames` function from base R, and enter the name of the data frame as the argument. Using the `<-` operator, we can specify the variable names using the `c` (combine) function that contains a vector of variable names in quotation marks (`" "`) as the arguments. *Remember to type a comma (`,`) between the function arguments, as commas are used to separate arguments from one another when there are more than one.* Please note that the it's important that the vector of variable names contains the same number of names as the data frame object has columns.

```
# Add (or replace) variable names to data frame object
colnames(personaldata) <- c("id", "lastname", "firstname", "startdate", "gender")

# View just the first 6 rows of data in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   153 Sanchez  Alejandro 1/1/2016  male
## 2   154 McDonald Ronald    1/9/2016  male
## 3   155 Smith    John      1/9/2016  male
## 4   165 Doe      Jane      1/4/2016  female
## 5   125 Franklin Benjamin  1/5/2016  male
## 6   111 Newton   Isaac     1/9/2016  male
```

Now the data frame object has variable names!

5.3 Summary

In this chapter, we reviewed how to remove and add variable names in a data frame object.

Chapter 6

Writing Data from R

Writing data refers to the process of exporting data from the R environment to a (working directory) folder. If you collaborate with others who do not work in R, writing data will allow them to use the data you cleaned, managed, or manipulated in the R environment in other software programs. In this chapter, we will focus on how to write a data frame and a table to our working directory folder as .csv files.

6.0.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/ORTe8vE7nzU

6.0.0.2 Functions & Packages Introduced

Function	Package
`write.csv`	base
`write.table`	base
`table`	base

6.0.0.3 Initial Steps

If you haven't already, save the file called **"PersData.csv"** into a folder that you will subsequently set as your working directory. Your working directory will likely be different than the one shown below (i.e., `"H:/RWorkshop"`). As a reminder, you can access all of the data files referenced in this book by downloading them as a compressed (zipped) folder from the my GitHub site: https://github.com/davidcaughlin/R-Tutorial-Data-Files; once you've followed

the link to GitHub, just click "Code" (or "Download") followed by "Download ZIP", which will download all of the data files referenced in this book. For the sake of parsimony, I recommend downloading all of the data files into the same folder on your computer, which will allow you to set that same folder as your working directory for each of the chapters in this book.

Next, using the `setwd` function, set your working directory to the folder in which you saved the data file for this chapter. Alternatively, you can manually set your working directory folder in your drop-down menus by going to *Session > Set Working Directory > Choose Directory…*. Be sure to create a new R script file (.R) or update an existing R script file so that you can save your script and annotations. If you need refreshers on how to set your working directory and how to create and save an R script, please refer to Setting a Working Directory and Creating & Saving an R Script.

```
# Set your working directory
setwd("H:/RWorkshop")
```

Next, read in the .csv data file called **"PersData.csv"** using your choice of read function. In this example, I use the `read_csv` function from the `readr` package (Wickham et al., 2018). If you choose to use the `read_csv` function, be sure that you have installed and accessed the `readr` package using the `install.packages` and `library` functions. *Note: You don't need to install a package every time you wish to access it; in general, I would recommend updating a package installation once ever 1-3 months.* For refreshers on installing packages and reading data into R, please refer to Packages and Reading Data into R.

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)

# Read data and name data frame (tibble) object
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
# View the names of the variables in the data frame (tibble) object
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
# View data frame (tibble) object
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

As you can see from the output generated in your console, the `personaldata` data frame object contains basic employee demographic information. The variable names include: `id`, `lastname`, `firstname`, `startdate`, and `gender`. *Technically, the `read_csv` function reads in what is called a "tibble" object (as opposed to a data frame object), but for our purposes a tibble will behave similarly to a data frame. For more information on tibbles, check out Wickham and Grolemund's (2017) chapter on tibbles: http://r4ds.had.co.nz/tibbles.html.*

6.1 Write Data Frame to Working Directory

The `write.csv` function from base R can be used to write a data frame object to your working directory or to a folder of your choosing. Let's write the `personaldata` data frame (that we read in and named above) to our working directory. Before doing so, however, let's make a minor change to the data frame to illustrate a scenario in which you clean your data in R and then write the data to a .csv file so that a colleague can work with the data in another program. Specifically, let's remove the `lastname` variable from the data frame. To do so, type the name of the data frame (`personaldata`), followed by the `$` symbol and then the name of the variable in question (`lastname`). Next, type the `<-` operator followed by `NULL`. This code will remove the variable from the data frame.

```
# Remove variable from data frame
personaldata$lastname <- NULL

# View data frame object
personaldata
```

```
## # A tibble: 9 x 4
##      id firstname startdate gender
##   <dbl> <chr>     <chr>     <chr>
## 1   153 Alejandro 1/1/2016  male
## 2   154 Ronald    1/9/2016  male
## 3   155 John      1/9/2016  male
## 4   165 Jane      1/4/2016  female
## 5   125 Benjamin  1/5/2016  male
## 6   111 Isaac     1/9/2016  male
## 7   198 Linda     1/7/2016  female
## 8   201 Cindy     1/9/2016  female
## 9   282 John      1/9/2016  male
```

As you can see in your Console output, the variable called `lastname` is no longer present in the data frame object.

To write our "cleaned"" data frame (`personaldata`) to our working directory, we use the `write.csv` function from base R. As the first argument in the parentheses, type the name of the data frame (`personaldata`). *Remember to type a comma (`,`) before the second argument, as this is how we separate arguments from one another when there are more than one.* As the second argument, let's type what we want to name the file that we will create in our working directory. Make sure that the name of the new .csv file is in quotation marks (`" "`). Here, I name the new file **"Cleaned PersData.csv"**; it is important that you keep the .csv extension at the end of the name you provide.

```
# Write data frame to working directory
write.csv(personaldata, "Cleaned PersData.csv")
```

If you go to your working directory folder, you will find the file called **"Cleaned PersData.csv"** saved there.

We can also specify which folder that we want to write our data to using the full path extension and what we would like to name the new .csv file.

```
# Write data frame to folder
write.csv(personaldata, "H:/RWorkshop/Cleaned PersData2.csv")
```

If you go to your working directory folder, you will find the file called **"Cleaned PersData2.csv*"**.

6.2 Write Table to Working Directory

Sometimes we work with table objects in R. If we wish to write a table to our working directory, we can use the `write.table` function from base R. Before doing so, we need to create a data table object as an example, which we can do using the `table` function from base R.

To create a table, first, come up with a name for your new table object; in this example, I name the table `table_example` (because I'm so creative). Second, type the `<-` operator to the right of your new table name to tell R that you are creating a new object. Third, type the name of the table-creation function, which is `table`. Fourth, in the function's parentheses, as the first argument, enter the name of first variable you wish to use to make the table, and use the `$` symbol to indicate that the variable (`gender`) belongs to the data frame in question (`personaldata`), which should look like this: `personaldata$gender`. Fifth, as the second argument, enter the name of the second variable you wish to use to make the table, and use the `$` symbol to indicate that the variable (`startdate`) belongs to the data frame in question (`personaldata`), which should look like this: `personaldata$startdate`.

```
# Create table from gender and startdate variables from personaldata data frame
table_example <- table(personaldata$gender, personaldata$startdate)

# View table in Console
table_example
```

```
##
##          1/1/2016 1/4/2016 1/5/2016 1/7/2016 1/9/2016
##   female        0        1        0        1        1
##   male          1        0        1        0        4
```

The table above shows how how many female versus male employees started working on a given date.

Now we are ready to write the table called `table_example` to our working directory using the `write.table` function. As the first argument, type the name of the table object (`table_example`). Second, type what we would like to call the file when it is saved in our working directory (`**"Practice Table.csv"**`); be sure to include the .csv extension in the name and wrap it all in quotation marks. Third, use the `sep=","` argument to specify that the values in the table are separated by commas, as this will be a comma separated values file. Fourth, add the argument `col.names=NA` to format the table such that the column names will be aligned with their respective values. The reason for this fourth argument is that in our table the first column will contain the row names of one of the variables; if we don't include this argument, the function will by default enter the name of the first column name associated with one of the levels of

the variables in the first column, and because the first column actually contains the row names for the table, the row names will be off by one column. The `col.names=NA` argument simply leaves the first cell in the top row blank so that in the next column to the right, the first column name for one of the variables will appear. [To understand what the table would look like *without* this fourth argument, simply omit it, and open the resulting file in your working directory to see what happens.]

```
# Write table to working directory
write.table(table_example, "Practice Table.csv", sep=",", col.names=NA)
```

If you go to your working directory, you will find the file called **"Practice Table.csv"**.

6.3 Summary

Writing data from the R environment to your working directory or another folder can be useful, especially when collaborating with those who do not use R. The `write.csv` function writes a data frame object to a .csv file, whereas the `write.table` function writes a data table object to a .csv file.

Part III

Data Management

Chapter 7

Arranging (Sorting) Data

Arranging (sorting) data refers to the process of ordering rows numerically or alphabetically in a data frame by the values of one or more variables. When sorting data in R, the underlying source data file does not change; rather, the data frame object in the R Global Environment changes. Sorting can make it easier to visually scan raw data, particularly when used in conjunction with the `View`, `head`, or `tail` functions from base R.

7.0.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/wVwJQsLNbmw

7.0.0.2 Functions & Packages Introduced

Function	Package
`arrange`	`dplyr`
`desc`	`dplyr`
`order`	base
`c`	base

7.0.0.3 Initial Steps

Please note, that any function that appears in the *Initial Steps* section has been covered in a previous chapter. If you need a refresher, please view the relevant chapter. In addition, a previous chapter may show you how to perform the same action using different functions or packages.

If you haven't already, save the file called **"PersData.csv"** into a folder that

you will subsequently set as your working directory. Your working directory will likely be different than the one shown below (i.e., `"H:/RWorkshop"`). As a reminder, you can access all of the data files referenced in this book by downloading them as a compressed (zipped) folder from the my GitHub site: https://github.com/davidcaughlin/R-Tutorial-Data-Files; once you've followed the link to GitHub, just click "Code" (or "Download") followed by "Download ZIP", which will download all of the data files referenced in this book. For the sake of parsimony, I recommend downloading all of the data files into the same folder on your computer, which will allow you to set that same folder as your working directory for each of the chapters in this book.

Next, using the `setwd` function, set your working directory to the folder in which you saved the data file for this chapter. Alternatively, you can manually set your working directory folder in your drop-down menus by going to *Session > Set Working Directory > Choose Directory…*. Be sure to create a new R script file (.R) or update an existing R script file so that you can save your script and annotations. If you need refreshers on how to set your working directory and how to create and save an R script, please refer to Setting a Working Directory and Creating & Saving an R Script.

```
# Set your working directory
setwd("H:/RWorkshop")
```

Next, read in the .csv data file called **"PersData.csv"** using your choice of read function. In this example, I use the `read_csv` function from the `readr` package (Wickham et al., 2018). If you choose to use the `read_csv` function, be sure that you have installed and accessed the `readr` package using the `install.packages` and `library` functions. *Note: You don't need to install a package every time you wish to access it; in general, I would recommend updating a package installation once ever 1-3 months.* For refreshers on installing packages and reading data into R, please refer to Packages and Reading Data into R.

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)

# Read data and name data frame (tibble) object
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
```

```
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
# View the names of the variables in the data frame (tibble) object
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
# View data frame (tibble) object
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

As you can see from the output generated in your console, the `personaldata` data frame object contains basic employee demographic information. The variable names include: `id`, `lastname`, `firstname`, `startdate`, and `gender`. Technically, the `read_csv` function reads in what is called a "tibble" object (as opposed to a data frame object), but for our purposes a tibble will behave similarly to a data frame. For more information on tibbles, check out Wickham and Grolemund's (2017) chapter on tibbles: http://r4ds.had.co.nz/tibbles.html.

7.1 Arrange (Sort) Data

There are different functions we could use to arrange (sort) the data in the data frame, and in this chapter, we will focus on the `arrange` function from the `dplyr` package (Wickham et al., 2020). Please note that there are other functions we could use to sort data, and if you're interested, in the Arranging (Sorting) Data:

Chapter Supplement, I demonstrate how to use the `order` function from base R to carry out the same operations we will cover below.

Because the `arrange` function comes from the `dplyr` package, which is part of the `tidyverse` of R packages (Wickham, 2019; Wickham et al., 2019). If you haven't already, install and access the `dplyr` package using the `install.packages` and `library` functions, respectively.

```
# Install dplyr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("dplyr")
```

```
# Access dplyr package
library(dplyr)
```

Before diving into arranging the data, as a disclaimer, I will demonstrate two techniques for arranging (sorting) data using the `arrange` function.

The *first technique* uses a "pipe" which in R is represented by the `%>%` operator. The pipe operator comes from a package called `magrittr` (Bache and Wickham, 2014), on which the `dplyr` is partially dependent. In short, a pipe allows a person to more efficiently write code and to improve the readability of the code and overall script. Specifically, a **pipe** forwards the result or value of one object or expression to a subsequent function. In doing so, one can avoid writing functions in which other functions are nested parenthetically. For more information on the pipe operator, check out Wickham and Grolemund's (2017) chapter on pipes: https://r4ds.had.co.nz/pipes.html.

This brings us to the *second technique* for arranging (sorting) data using the `arrange` function. The second technique uses a more traditional approach that some may argue lacks the efficiency and readability of the pipe. Conversely, others may argue against the use of pipes altogether. I'm not here to settle any "pipes versus no pipes" debate, and you're welcome to use either technique. If you don't want to learn how to use pipes (or would like to learn how to use them at a later date), feel free to skip to the section below called Without Pipe.

7.1.1 *With* Pipe

To use the "with pipe" technique, first, type the name of our data frame object, which we previously named `personaldata`, followed by the pipe (`%>%`) operator. This will "pipe" our data frame into the subsequent function. Second, either on the same line or on the next line, type the name of the `arrange` function, and within the parentheses, enter the variable name `startdate` as the argument to indicate that we want to arrange (sort) the data by the start date of the employees. The default operation of the `arrange` function is to arrange (sort)

the data in *ascending* order. If you're wondering where I found the exact names of the variables in the data frame, revisit the use of the `names` function, which I demonstrated previously in this chapter in the Initial Steps section.

```
# Arrange (sort) data by variable in ascending order (single line) (with pipe)
personaldata %>% arrange(startdate)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   165 Doe        Jane      1/4/2016  female
## 3   125 Franklin   Benjamin  1/5/2016  male
## 4   198 Morales    Linda     1/7/2016  female
## 5   154 McDonald   Ronald    1/9/2016  male
## 6   155 Smith      John      1/9/2016  male
## 7   111 Newton     Isaac     1/9/2016  male
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

Alternatively, we can write this script over two lines and achieve the same output in our Console.

```
# Arrange (sort) data by variable in ascending order (two lines) (with pipe)
personaldata %>%
  arrange(startdate)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   165 Doe        Jane      1/4/2016  female
## 3   125 Franklin   Benjamin  1/5/2016  male
## 4   198 Morales    Linda     1/7/2016  female
## 5   154 McDonald   Ronald    1/9/2016  male
## 6   155 Smith      John      1/9/2016  male
## 7   111 Newton     Isaac     1/9/2016  male
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

Please note that the operations we have performed thus far have not changed anything in the `personaldata` data frame object itself; rather, the output in the Console simply shows what it *looks like* if the data are sorted by the variable in question. We can verify this by viewing the first six rows of data in our data

frame object using the `head` function. As you can see below, nothing changed in the data frame itself.

```
# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   153 Sanchez  Alejandro 1/1/2016  male
## 2   154 McDonald Ronald    1/9/2016  male
## 3   155 Smith    John      1/9/2016  male
## 4   165 Doe      Jane      1/4/2016  female
## 5   125 Franklin Benjamin  1/5/2016  male
## 6   111 Newton   Isaac     1/9/2016  male
```

To change the ordering of data in the `personaldata` data frame object itself, we will need to (re)name the data frame object using the `<-` variable assignment operator. In this example, I will demonstrate how to overwrite the existing data frame object, and thus I give the data frame object the *exact* same name as it had originally (i.e., `personaldata`). To do so, to the *left* of the `<-` operator, type what you would like to name the new (updated) sorted data frame object (`personaldata`). Next, to the *right* of the `<-` operator, copy and paste the same code we wrote above. Finally, use the `head` function from base R to view the first six rows of the new data frame object.

```
# Arrange (sort) data by variable in ascending order and
# overwrite existing data frame object (with pipe)
personaldata <- personaldata %>% arrange(startdate)

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   153 Sanchez  Alejandro 1/1/2016  male
## 2   165 Doe      Jane      1/4/2016  female
## 3   125 Franklin Benjamin  1/5/2016  male
## 4   198 Morales  Linda     1/7/2016  female
## 5   154 McDonald Ronald    1/9/2016  male
## 6   155 Smith    John      1/9/2016  male
```

As you can see in the Console output, now the `personaldata` data frame object has been changed such that the data are arranged (sorted) by the `startdate` variable.

To arrange the data in *descending* order, just use the `desc` function from `dplyr` within the `arrange` function as shown below.

```
# Arrange (sort) data by variable in ascending order and
# overwrite existing data frame object (with pipe)
personaldata <- personaldata %>% arrange(desc(startdate))

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   111 Newton     Isaac     1/9/2016  male
## 4   201 Providence Cindy     1/9/2016  female
## 5   282 Legend     John      1/9/2016  male
## 6   198 Morales    Linda     1/7/2016  female
```

To arrange (sort) data by values/levels of two variables, we simply enter the names of two variables as consecutive arguments. Let's enter the `gender` variable first, followed by the `startdate` variable. The ordering of the two variables matters; the function sorts initially by the values/levels of the first variable listed and sorts subsequently by the values/levels of the second variable listed, but does so *within* the values/levels of the first variable listed. As shown below, `startdate` is sorted within the sorted levels of the `gender` variable. As a reminder, the default operation of the `arrange` function is to arrange (sort) the data in *ascending* order. *Remember, we use commas to separate arguments used in a function (if there are more than one arguments).*

```
# Arrange (sort) data by two variables in ascending order (with pipe)
personaldata %>% arrange(gender, startdate)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   165 Doe        Jane      1/4/2016  female
## 2   198 Morales    Linda     1/7/2016  female
## 3   201 Providence Cindy     1/9/2016  female
## 4   153 Sanchez    Alejandro 1/1/2016  male
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   154 McDonald   Ronald    1/9/2016  male
## 7   155 Smith      John      1/9/2016  male
```

```
## 8   111 Newton     Isaac     1/9/2016  male
## 9   282 Legend     John      1/9/2016  male
```

Watch what happens when we switch the order of the two variables we are using to sort the data.

```
# Arrange (sort) data by two variables in ascending order (with pipe)
personaldata %>% arrange(startdate, gender)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   165 Doe        Jane      1/4/2016  female
## 3   125 Franklin   Benjamin  1/5/2016  male
## 4   198 Morales    Linda     1/7/2016  female
## 5   201 Providence Cindy     1/9/2016  female
## 6   154 McDonald   Ronald    1/9/2016  male
## 7   155 Smith      John      1/9/2016  male
## 8   111 Newton     Isaac     1/9/2016  male
## 9   282 Legend     John      1/9/2016  male
```

As you can see, the order of the two sorting variables matters.

To arrange the data in *descending* order, just use the `desc` function from `dplyr` within the `arrange` function.

```
# Arrange (sort) data by variable in descending order (with pipe)
personaldata %>% arrange(desc(gender,startdate))
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   111 Newton     Isaac     1/9/2016  male
## 4   282 Legend     John      1/9/2016  male
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   153 Sanchez    Alejandro 1/1/2016  male
## 7   201 Providence Cindy     1/9/2016  female
## 8   198 Morales    Linda     1/7/2016  female
## 9   165 Doe        Jane      1/4/2016  female
```

Or, we can sort one variable in the default ascending order and the other in descending order.

```
# Arrange (sort) data by two variables in ascending & descending order (with pipe)
personaldata %>% arrange(gender, desc(startdate))
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   201 Providence Cindy     1/9/2016  female
## 2   198 Morales    Linda     1/7/2016  female
## 3   165 Doe        Jane      1/4/2016  female
## 4   154 McDonald   Ronald    1/9/2016  male
## 5   155 Smith      John      1/9/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   282 Legend     John      1/9/2016  male
## 8   125 Franklin   Benjamin  1/5/2016  male
## 9   153 Sanchez    Alejandro 1/1/2016  male
```

7.1.2 *Without* Pipe

We can achieve the same output *without* using the pipe (`%>%`) operator as *with* the pipe operator; again, your choice of using or not using the pipe operator is up to you.

To use the `arrange` function without the pipe operator, type the name of the `arrange` function, and within the parentheses, as the first argument, type the name of the `personaldata` data frame object, and as the second argument, type the `startdate` variable, where the latter indicates that we want to arrange (sort) the data frame object by the start date of the employees. The default operation of the `arrange` function is to arrange (sort) the data in *ascending* order. *Remember, we use commas to separate arguments used in a function (if there are more than one arguments).* If you're wondering where I found the exact names of the variables in the data frame, revisit the use of the `names` function, which I demonstrated previously in this chapter in the Initial Steps section.

```
# Arrange (sort) data by variable in ascending order without pipe
arrange(personaldata, startdate)
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   165 Doe        Jane      1/4/2016  female
## 3   125 Franklin   Benjamin  1/5/2016  male
## 4   198 Morales    Linda     1/7/2016  female
```

```
## 5   154 McDonald   Ronald    1/9/2016  male
## 6   155 Smith      John      1/9/2016  male
## 7   111 Newton     Isaac     1/9/2016  male
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

To change the ordering of data in the `personaldata` data frame object itself, we will need to (re)name the data frame object using the `<-` variable assignment operator. In this example, I will demonstrate how to overwrite the existing data frame object, and thus I give the data frame object the *exact* same name as it had originally (i.e., `personaldata`). To do so, to the *left* of the `<-` operator, type what you would like to name the new (updated) sorted data frame object (`personaldata`). Next, to the *right* of the `<-` operator, copy and paste the same code we wrote above. Finally, use the `head` function from base R to view the first six rows of the new data frame object.

```
# Arrange (sort) data by variable in ascending order and
# overwrite existing data frame object without pipe
personaldata <- arrange(personaldata, startdate)

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   153 Sanchez  Alejandro 1/1/2016  male
## 2   165 Doe      Jane      1/4/2016  female
## 3   125 Franklin Benjamin  1/5/2016  male
## 4   198 Morales  Linda     1/7/2016  female
## 5   154 McDonald Ronald    1/9/2016  male
## 6   155 Smith    John      1/9/2016  male
```

To arrange the data in *descending* order, just use the `desc` function from `dplyr` within the `arrange` function as shown below.

```
# Arrange (sort) data by variable in descending order and
# overwrite existing data frame object without pipe
personaldata <- arrange(personaldata, desc(startdate))

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
```

```
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   111 Newton     Isaac     1/9/2016  male
## 4   201 Providence Cindy     1/9/2016  female
## 5   282 Legend     John      1/9/2016  male
## 6   198 Morales    Linda     1/7/2016  female
```

To arrange (sort) data by values/levels of two variables, we simply enter the names of two variables as consecutive arguments (after the name of the data frame, which is the first argument). Let's enter the `gender` variable first, followed by the `startdate` variable. The ordering of the two variables matters; the function sorts initially by the values/levels of the first variable listed and sorts subsequently by the values/levels of the second variable listed, but does so *within* the values/levels of the first variable listed.

```
# Arrange (sort) data by variable in ascending order without pipe
personaldata <- arrange(personaldata, gender, startdate)
```

As shown in the output above, `startdate` is sorted within the sorted levels of the `gender` variable. This also verifies that the default operation of the `arrange` function is to arrange (sort) the data in *ascending* order.

To arrange the data in *descending* order, just use the `desc` function from `dplyr` within the `arrange` function as shown below. You can use the `desc` function on one or both sorting variables.

```
# Arrange (sort) data by one variable in ascending order and
# the other in descending order without pipe
personaldata <- arrange(personaldata, gender, desc(startdate))
```

Or we can apply the `desc` function to both variables.

```
# Arrange (sort) data by both variables descending order without pipe
personaldata <- arrange(personaldata, desc(gender, startdate))
```

7.2 Summary

In this chapter, we learned how to arrange (sort) data by one or more variables using the `arrange` and `desc` functions from the `dplyr` package. This chapter also introduced the pipe (`%>%`) operator, which can help make code easier to read in some contexts.

Chapter 8

Joining (Merging) Data

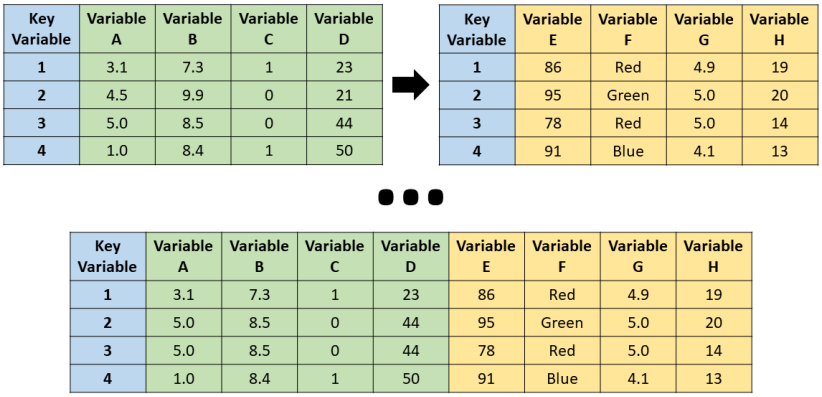
Joining refers to the process of matching two data frames by either one or more key variables (i.e., horizontal join) or by variable names or columns (i.e., vertical join). Sometimes a **join** is referred to as a **merge**, and thus I will use these terms interchangeably. Broadly speaking, there are two types of joins (merges): horizontal and vertical.

8.0.0.1 Joining Data Horizontally

A **horizontal join (merge)** refers to the process of matching cases (i.e., rows, observations) between two data frames using a *key variable* (*matching variable*), which results in distinct sets of variables (i.e., fields, columns) being combined horizontally (laterally) across two data frames. The resulting joined data frame will be wider (in terms of the number of variables) than either of the original data frames in isolation. For example, imagine that we pull data from separate information systems, each with different variables (i.e., fields) but at least some employees (i.e., cases) in common; to combine these two data frames, we can perform a horizontal join. This is often a necessary step when creating a data frame that contains all of the variables we will need in subsequent data analyses. For instance, if we wish to estimate the criterion-related validities (see Selection Tool Validation) using the selection tool scores from one data frame with criterion (e.g., job performance) scores from another data frame, then we could perform a horizontal join.

We will focus on four different types of horizontal joins:

1. Inner join
2. Full join
3. Left join



Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50

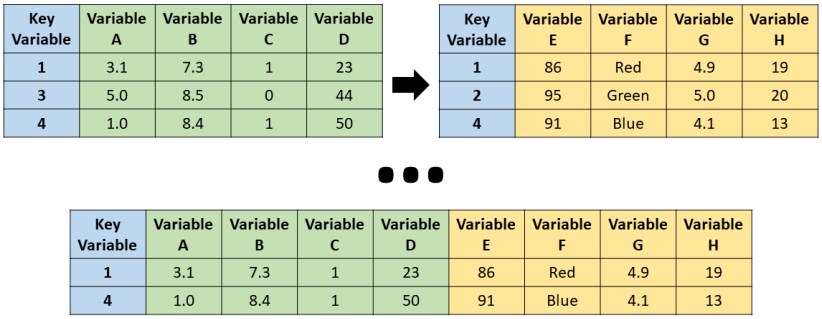
Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
3	78	Red	5.0	14
4	91	Blue	4.1	13

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
2	5.0	8.5	0	44	95	Green	5.0	20
3	5.0	8.5	0	44	78	Red	5.0	14
4	1.0	8.4	1	50	91	Blue	4.1	13

Figure 8.1: In a *horizontal join*, cases (or observations) are matched between two data frames using one or more key variables.

4. Right join

5. **Inner join**: All unmatched cases (or observations) are dropped, thereby retaining only those cases that are present in both the left (x, first) and right (y, second) data frames. In other words, a case is only included in the merged data frame if it appears in both of the original data data frames.



Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
3	5.0	8.5	0	44
4	1.0	8.4	1	50

Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
4	91	Blue	4.1	13

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
4	1.0	8.4	1	50	91	Blue	4.1	13

Figure 8.2: In an *inner join*, all unmatched cases (or observations) are dropped, thereby retaining only those cases that are present in both the left (x, first) and right (y, second) data frames.

2. **Full join**: All cases (or observations) are retained, including those cases that do not have a match in the other data data frame. In other words, a case is included in the merged data frame even if it only appears in one of the original data data frames. These type of join leads to the highest

number of retained cases under conditions in which both data frames contain unique cases.

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
3	5.0	8.5	0	44
4	1.0	8.4	1	50

Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
4	91	Blue	4.1	13

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
2	*NA*	*NA*	*NA*	*NA*	95	Green	5.0	20
3	5.0	8.5	0	44	*NA*	*NA*	*NA*	*NA*
4	1.0	8.4	1	50	91	Blue	4.1	13

Figure 8.3: In a *full join*, all cases (or observations) are retained, including those cases that do not have a match in the other data data frame.

3. **Left join**: All cases (or observations) that appear in the left (x, first) data frame are retained, even if they lack a match in the right (y, second) data frame. Consequently, cases from the right data frame that lack a match in the left data frame are dropped in the merged data frame.

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
3	5.0	8.5	0	44
4	1.0	8.4	1	50

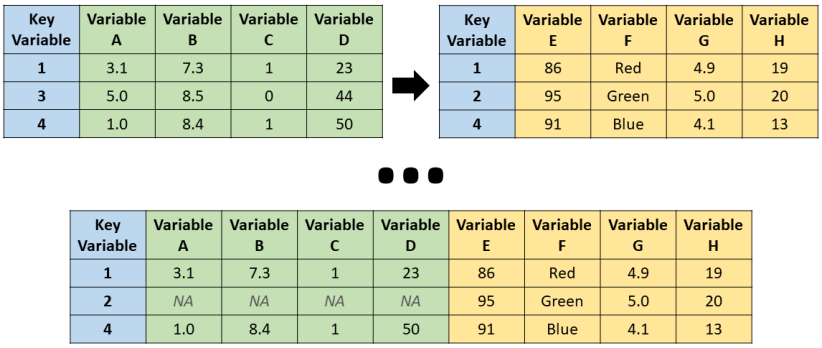
Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
4	91	Blue	4.1	13

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
3	5.0	8.5	0	44	*NA*	*NA*	*NA*	*NA*
4	1.0	8.4	1	50	91	Blue	4.1	13

Figure 8.4: In a *left join*, all cases (or observations) that appear in the left (x, first) data frame are retained, even if they lack a match in the right (y, second) data frame.

4. **Right join**: All cases (or observations) that appear in the right (y, second) data frame are retained, even if they lack a match in the left (x, first) data

frame. Consequently, cases from the left data frame that lack a match in the right data frame are dropped in the merged data frame.



Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
3	5.0	8.5	0	44
4	1.0	8.4	1	50

Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
4	91	Blue	4.1	13

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
2	*NA*	*NA*	*NA*	*NA*	95	Green	5.0	20
4	1.0	8.4	1	50	91	Blue	4.1	13

Figure 8.5: In a *left join*, only cases (or observations) that appear in the left (x, first) data frame are retained, even if they lack a match in the right (y, second) data frame.

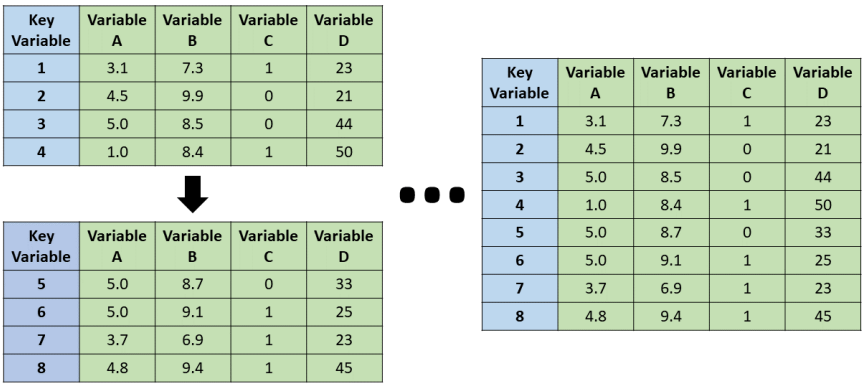
Please note that I have illustrated different types of horizontal joins using a single key variable. It is entirely possible to perform horizontal joins using two or more key variables. For example, imagine that each morning we administered a pulse survey to employees and each afternoon we afternoon we administered a different pulse survey to the same employees, and that we repeated this process for five consecutive workdays. In this instance, we would likely need to horizontally join the data frames using both a unique employee identifier variable and a unique day-of-week variable.

8.0.0.2 Joining Data Vertically

A **vertical join (merge)** refers to the process of matching identical variables from two data frames, which results in distinct sets of cases or observations being combined vertically. The resulting joined data frame will be longer (in terms of the number of cases) than either of the original data frames in isolation. For example, imagine an organization administered the same survey to two facilities (i.e., independent groups) each with unique employees; we could combine the two resulting data frames by performing a vertical join.

8.0.0.3 Video Tutorial

Link to Video Tutorial: https://youtu.be/wVwJQsLNbmw



Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50

Key Variable	Variable A	Variable B	Variable C	Variable D
5	5.0	8.7	0	33
6	5.0	9.1	1	25
7	3.7	6.9	1	23
8	4.8	9.4	1	45

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50
5	5.0	8.7	0	33
6	5.0	9.1	1	25
7	3.7	6.9	1	23
8	4.8	9.4	1	45

Figure 8.6: In a *vertical join*, identical variables are matched between two data frames, each with distinct sets of cases or observations.

8.0.0.4 Functions & Packages Introduced

Function	Package
`merge`	base
`right_join`	`dplyr`
`left_join`	`dplyr`
`inner_join`	`dplyr`
`full_join`	`dplyr`
`data.frame`	base
`c`	base
`rep`	base
`rbind`	base

8.0.0.5 Initial Steps

If you haven't already, save the files called **"PersData.csv"** and **"PerfData.csv"** into a folder that you will subsequently set as your working directory. Your working directory will likely be different than the one shown below (i.e., `"H:/RWorkshop"`). As a reminder, you can access all of the data files referenced in this book by downloading them as a compressed (zipped) folder from the my GitHub site: https://github.com/davidcaughlin/R-Tutorial-Data-Files; once you've followed the link to GitHub, just click "Code" (or "Download") followed by "Download ZIP", which will download all of the data files referenced in this book. For the sake of parsimony, I recommend downloading all of the data files into the same folder on your computer, which will allow you to set that same folder as your working directory for each of the chapters in this book.

Next, using the `setwd` function, set your working directory to the folder in which you saved the data file for this chapter. Alternatively, you can manually set your working directory folder in your drop-down menus by going to *Session > Set Working Directory > Choose Directory…*. Be sure to create a new R script file (.R) or update an existing R script file so that you can save your script and annotations. If you need refreshers on how to set your working directory and how to create and save an R script, please refer to Setting a Working Directory and Creating & Saving an R Script.

```
# Set your working directory
setwd("H:/RWorkshop")
```

Next, read in the .csv data files called **"PersData.csv"** and **"PerfData.csv"** using your choice of read function. In this example, I use the `read_csv` function from the `readr` package (Wickham et al., 2018). If you choose to use the `read_csv` function, be sure that you have installed and accessed the `readr` package using the `install.packages` and `library` functions. *Note: You don't need to install a package every time you wish to access it; in general, I would recommend updating a package installation once ever 1-3 months.* For refreshers

on installing packages and reading data into R, please refer to Packages and Reading Data into R.

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)

# Read data and name data frame (tibble) objects
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
performancedata <- read_csv("PerfData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   perf_q1 = col_double(),
##   perf_q2 = col_double(),
##   perf_q3 = col_double(),
##   perf_q4 = col_double()
## )
```

```
# View the names of the variables in the data frame (tibble) objects
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
names(performancedata)
```

```
## [1] "id"      "perf_q1" "perf_q2" "perf_q3" "perf_q4"
```

```
# View data frame (tibble) objects
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

```
performancedata
```

```
## # A tibble: 6 x 5
##      id perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   153     3.9     4.8     4.9     5
## 2   125     2.1     1.9     2.1     2.3
## 3   111     3.3     3.3     3.4     3.3
## 4   198     4.9     4.5     4.4     4.8
## 5   201     1.2     1.1     1       1
## 6   282     2.2     2.3     2.4     2.5
```

As you can see from the output generated in your console, on the one hand, the `personaldata` data frame object contains basic employee demographic information. The variable names include: `id`, `lastname`, `firstname`, `startdate`, and `gender`. On the other hand, the `personaldata` data frame object contains the same `id` unique identifer variable as the `personaldata` data frame object, but instead of employee demographic information, this data frame object includes varuables associated with quarterly employee performance: `perf_q1`, `perf_q2`, `perf_q3`, and `perf_q4`.

In order to better illustrate certain join functions later on in this chapter, we'll begin by removing the case (i.e., employee) associated with the `id` variable value of 153 (i.e., Alejandro Sanchez); in terms of a rationale for doing so, let's imagine that Alejandro no longer works for the organization, and thus we would like to remove him from the `personaldata` data frame. If you don't completely understand the following process for removing this individual from the data frame, no need to worry, as you will learn more in the subsequent chapter on filtering data.

1. Type the name of the data frame object (`personaldata`) followed by the `<-` operator to overwrite the existing data frame object.
2. Type the name of the original data frame object (`personaldata`) followed by brackets (`[ ]`).
3. Within the brackets (`[ ]`), type the name of the data frame object (`personaldata`) again, followed by the `$` operator and the name of the variable we wish to use to select the case that will be removed, which in this instance is the `id` unique identifier variable. The `$` operator indicates to R that the `id` variable belongs to the `personaldata` data frame.
4. Type the "not equal to" operator, which is `!=` (the `!` means "not"), followed by the `id` variable value we wish to use to remove the case (i.e., 153).
5. Type a comma (`,`) to indicate that we are removing a row, not a column. *When referencing rows and columns in R, as we are doing in the brackets ([ ]), rows are entered first (before a comma), and columns are entered second (after a comma).* In doing so, we are telling R to retain all rows of data in `personaldata` except for the one corresponding to `id` equal to 153.

```
# Remove case with id variable equal to 153
personaldata <- personaldata[personaldata$id != 153,]
```

Check out the first 6 rows of the updated data frame for `personaldata`, and note that the data corresponding to the case associated with `id` equal to 153 is gone.

```
# View first 6 rows of first data frame object once more
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   154 McDonald Ronald    1/9/2016  male
## 2   155 Smith    John      1/9/2016  male
## 3   165 Doe      Jane      1/4/2016  female
## 4   125 Franklin Benjamin  1/5/2016  male
## 5   111 Newton   Isaac     1/9/2016  male
## 6   198 Morales  Linda     1/7/2016  female
```

8.1 Horizontal Join (Merge)

Recall that a horizontal join (merge) means that cases are matched using one more more key variables, and as a result, variables (i.e., columns, fields) are

combined across two data frames. We will review two options for performing horizontal joins.

To perform horizontal joins, we will learn how to use the `join` functions from the the `dplyr` package (Wickham et al., 2020), which include: `right_join`, `left_join`, `inner_join`, and `full_join`. Please note that there are other functions we could use to perform horizontal joins, and if you're interested, in the Joining (Merging) Data: Chapter Supplement, I demonstrate how to use the `merge` function from base R to carry out the same operations we will cover below.

Using the aformentioned `join` functions, we will match cases from the `personaldata` and `performancedata` data frames using the `id` unique identifer variable as a key variable. So how can we verify that `id` is an appropriate key variable? Well, let's use the `names` function from base R to retrieve the list of variable names from the two data frames, which we already did above. Nevertheless, let's call up those variable names once more. Simply enter the name of the data frame as a parenthetical argument in the `names` function.

```
# Retrieve variable names from first data frame
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
# Retrieve variable names from second data frame
names(performancedata)
```

```
## [1] "id"      "perf_q1" "perf_q2" "perf_q3" "perf_q4"
```

As you can see in the variable names listed above, the `id` variable is common to both data frames, and thus it will serve as our key variable.

Now we are almost ready to begin joining the two data frames using the `id` unique identifer as a key variable. Before doing so, however, we should make sure that we have installed and accessed the `dplyr` package (if we haven't already), as the `join` functions come from that package.

```
# Install dplyr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("dplyr")
```

```
# Access dplyr package
library(dplyr)
```

I will demonstrate two techniques for applying the `join` function.

The *first technique* uses the pipe operator (`%>%`). The pipe operator comes from a package called `magrittr` (Bache and Wickham, 2014), on which the `dplyr` is partially dependent. In short, a pipe allows a person to more efficiently write code and to improve the readability of the code and overall script. Specifically, a **pipe** forwards the result or value of one object or expression to a subsequent function. In doing so, one can avoid writing functions in which other functions are nested parenthetically. For more information on the pipe operator, check out Wickham and Grolemund's (2017) chapter on pipes: https://r4ds.had.co.nz/pipes.html.

The *second technique* for applying the `join` function takes a more traditional approach in that it involves nested functions being nested parenthetically. If you don't want to learn how to use pipes (or would like to learn how to use them at a later date), feel free to skip to the section below called Without Pipe.

8.1.1 *With* Pipe

Using the pipe (`%>%`) operator technique, let's begin with what is referred to as an **inner join** by doing the following:

1. Use the `<-` symbol to name the joined (merged) data frame that we will create using the one of the `dplyr` join functions. For this example, I name the new joined data frame `mergeddf`, which is completely arbitrary; you could name it whatever you would like. Make sure you put the name of the new data frame object to the *left* of the `<-` operator.
2. To the *right* of the `<-` operator, type the name of the first data frame, which we named `personaldata`, followed by the pipe (`%>%`) operator. This will "pipe" our data frame into the subsequent function.
3. On the same line or on the next line, type the `inner_join` function, and within the parentheses as the first argument, type the name of the second data frame, which we called `performancedata`. As the second argument, use the `by=` argument to indicate the name of the key variable, which in this example is `id`; make sure the key variable is in quotation marks (`" "`), and remember, object and variable names in R are case and space sensitive.

```
# Inner join (with pipe)
mergeddf <- personaldata %>% inner_join(performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 5 x 9
```

```
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 2   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 3   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 4   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 5   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Now, let's revisit the original data frame objects that we read in initially.

```
# View the first original data frame
personaldata
```

```
## # A tibble: 8 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   165 Doe        Jane      1/4/2016  female
## 4   125 Franklin   Benjamin  1/5/2016  male
## 5   111 Newton     Isaac     1/9/2016  male
## 6   198 Morales    Linda     1/7/2016  female
## 7   201 Providence Cindy     1/9/2016  female
## 8   282 Legend     John      1/9/2016  male
```

```
# View the second original data frame
performancedata
```

```
## # A tibble: 6 x 5
##      id perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   153     3.9     4.8     4.9     5
## 2   125     2.1     1.9     2.1     2.3
## 3   111     3.3     3.3     3.4     3.3
## 4   198     4.9     4.5     4.4     4.8
## 5   201     1.2     1.1     1       1
## 6   282     2.2     2.3     2.4     2.5
```

In the output, first, note how all of the variables from the original data frames (i.e., `personaldata`, `performancedata`) are represented in the merged data frame (i.e., `mergeddf`). Second, note how the cases are matched by the `id` key variable. Third, note that the `personaldata` data frame has 8 cases, the `performancedata` data frame has 6 cases, and the `mergeddf` data frame has 6 cases. By default, the `merge` function performs an *inner join* and retains

only those matched cases that have data in *both* data frames. Because cases whose `id` values were `154`, `155`, and `165` had data in `personaldata` but not `performancedata` and because the case with an `id` value equal to 153 was in `performancedata` but not `personaldata`, only the 5 cases that had available data in both data frames were retained.

To perform what is referred to as a **full join** in which we retain all cases and available data, we simply swap out the `inner_join` function from our previous code with the `full_join` function.

```
# Full join (with pipe)
mergeddf <- personaldata %>% full_join(performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 9 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   154 McDonald   Ronald    1/9/2016  male      NA      NA      NA      NA
## 2   155 Smith      John      1/9/2016  male      NA      NA      NA      NA
## 3   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 4   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 5   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 6   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 7   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 8   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
## 9   153 <NA>       <NA>      <NA>      <NA>       3.9     4.8     4.9     5
```

Note how the `full_join` function retains all available cases that had available data in at least one of the data frames, which in this example is 9 cases. *When in doubt, I recommend using the `full_join` function to retain all available data.*

To perform what is referred to as a **left join** in which we retain only those cases with data available in the first (left, x) data frame (`personaldata`), we use the `left_join` function instead, while keeping the rest of the previous code the same.

```
# Left join (with pipe)
mergeddf <- personaldata %>% left_join(performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 8 x 9
```

```
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   154 McDonald   Ronald    1/9/2016  male      NA      NA      NA      NA
## 2   155 Smith      John      1/9/2016  male      NA      NA      NA      NA
## 3   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 4   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 5   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 6   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 7   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 8   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Note how the `left_join` function retains only those cases for which the first (left, x) data frame (i.e., `personaldata`) has complete data, which in this case happens to be 8 cases. Notably absent is the case associated with `id` equal to 153 because the first (left, x) data frame (i.e., `personaldata`) lacked that case. An `NA` appears for each case from the second (right, y) data frame that contained missing values on variables from that data frame.

To perform what is referred to as a **right join** in which we retain only those cases with data available in the second (right, y) data frame (`performancedata`), we use the `right_join` function instead, while keeping the rest of the previous code the same.

```
# Right join (with pipe)
mergeddf <- personaldata %>% right_join(performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 6 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 2   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 3   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 4   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 5   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
## 6   153 <NA>       <NA>      <NA>      <NA>       3.9     4.8     4.9     5
```

Note how the `right_join` function retains only those cases for which the joined (second, right, y) data frame (i.e., `performancedata`) has complete data. Because the first (left, x) data frame lacks data for the case in which `id` is equal to 153, an `NA` appears for each case from the first data frame that contained missing values on variables from that data frame.

8.1.2 *Without* Pipe

In this section, I demonstrate the same `dplyr` join functions as above, except here I demonstrate how to specify the functions *without* the use of a pipe (`%>%`) operator.

Let's begin with what is referred to as an **inner join** by doing the following:

1. Use the `<-` operator to name the joined (merged) data frame that we will create using the one of the `dplyr` join functions. For this example, I name the new joined data frame `mergeddf`, which is completely arbitrary; you could name it whatever you would like. Make sure you put the name of the new data frame object to the *left* of the `<-` operator.
2. To the *right* of the `<-` operator, type the name of the `inner_join` function. As the first argument within the parentheses, type the name of the first data frame, which we named `personaldata`. As the second argument, type the name of the second data frame we named `performancedata`. As the third argument, use the `by=` argument to indicate the name of the key variable, which in this example is `id`; make sure the key variable is in quotation marks (`" "`), and remember, object and variable names in R are case and space sensitive.

```
# Inner join (without pipe)
mergeddf <- inner_join(personaldata, performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 5 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 2   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 3   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 4   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 5   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Now, let's revisit the original data frame objects that we read in initially.

```
# View the first original data frame
personaldata
```

```
## # A tibble: 8 x 5
##      id lastname   firstname startdate gender
```

```
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   165 Doe        Jane      1/4/2016  female
## 4   125 Franklin   Benjamin  1/5/2016  male
## 5   111 Newton     Isaac     1/9/2016  male
## 6   198 Morales    Linda     1/7/2016  female
## 7   201 Providence Cindy     1/9/2016  female
## 8   282 Legend     John      1/9/2016  male
```

```
# View the second original data frame
performancedata
```

```
## # A tibble: 6 x 5
##      id perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   153     3.9     4.8     4.9     5
## 2   125     2.1     1.9     2.1     2.3
## 3   111     3.3     3.3     3.4     3.3
## 4   198     4.9     4.5     4.4     4.8
## 5   201     1.2     1.1     1       1
## 6   282     2.2     2.3     2.4     2.5
```

In the output, first, note how all of the variables from the original data frames (i.e., `personaldata`, `performancedata`) are represented in the merged data frame (i.e., `mergeddf`). Second, note how the cases are matched by the `id` key variable. Third, note that the `personaldata` data frame has 8 cases, the `performancedata` data frame has 6 cases, and the `mergeddf` data frame has 6 cases. By default, the `merge` function performs an *inner join* and retains only those matched cases that have data in *both* data frames. Because cases whose `id` values were `154`, `155`, and `165` had data in `personaldata` but not `performancedata` and because the case with an `id` value equal to 153 was in `performancedata` but not `personaldata`, only the 5 cases that had available data in both data frames were retained.

To perform what is referred to as a **full join** in which we retain all cases and available data, we simply swap out the `inner_join` function from our previous code with the `full_join` function.

```
# Full join (without pipe)
mergeddf <- full_join(personaldata, performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 9 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   154 McDonald   Ronald    1/9/2016  male      NA      NA      NA      NA
## 2   155 Smith      John      1/9/2016  male      NA      NA      NA      NA
## 3   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 4   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 5   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 6   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 7   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 8   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
## 9   153 <NA>       <NA>      <NA>      <NA>       3.9     4.8     4.9     5
```

Note how the `full_join` function retains all available cases that had available data in at least one of the data frames, which in this example is 9 cases. *When in doubt, I recommend using the `full_join` function to retain all available data.*

To perform what is referred to as a **left join** in which we retain only those cases with data available in the first (left, x) data frame (`personaldata`), we use the `left_join` function instead, while keeping the rest of the previous code the same.

```
# Left join (without pipe)
mergeddf <- left_join(personaldata, performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 8 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   154 McDonald   Ronald    1/9/2016  male      NA      NA      NA      NA
## 2   155 Smith      John      1/9/2016  male      NA      NA      NA      NA
## 3   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 4   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 5   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 6   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 7   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 8   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Note how the `left_join` function retains only those cases for which the first (left, x) data frame (i.e., `personaldata`) has complete data, which in this case happens to be 8 cases. Notably absent is the case associated with `id` equal to 153 because the first (left, x) data frame (i.e., `personaldata`) lacked that case. An `NA` appears for each case from the second (right, y) data frame that contained missing values on variables from that data frame.

To perform what is referred to as a **right join** in which we retain only those cases with data available in the second (right, y) data frame (`performancedata`), we use the `right_join` function instead, while keeping the rest of the previous code the same.

```
# Right join (without pipe)
mergeddf <- right_join(personaldata, performancedata, by="id")

# View the joined data frame
mergeddf
```

```
## # A tibble: 6 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 2   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 3   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 4   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 5   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
## 6   153 <NA>       <NA>      <NA>      <NA>       3.9     4.8     4.9     5
```

Note how the `right_join` function retains only those cases for which the joined (second, right, y) data frame (i.e., `performancedata`) has complete data. Because the first (left, x) data frame lacks data for the case in which `id` is equal to 153, an `NA` appears for each case from the first data frame that contained missing values on variables from that data frame.

8.2 Vertical Join (Merge)

To perform a vertical join (merge), we will use the `rbind` function from base R, which stands for "row bind." As a reminder, with a horizontal join, our focus is on joining variables (i.e., columns, fields) from two data frames containing overlapping cases (i.e., rows). In contrast, with a vertical join, our focus is on joining cases from data frames with the same variables.

To illustrate how to perform a vertical join, we take a slightly different approach than what we did with horizontal joins. Instead of reading in data files, we will create two "toy" employee demographic data frames with the exact same variables but different cases. We will use the `data.frame` function from base R to indicate that we wish to create a data frame object; we use the `c` (combine) function from base R to combine values into a vector; and we use the `rep` (replicate) function from base R to replicate the same value a specified number of times. Also note that the `:` operator, when used between two numbers, creates a vector of consecutive values, beginning with the first value and ending

with the second. Please note, that using and understanding the `data.frame`, `c`, and `rep` functions is not consequential for understanding how to do a vertical merge; rather, I merely use these functions in this tutorial to create quick toy data frames that we can use to illustrate how to do a vertical join. For more information on the `data.frame` function and the `c` function, please refer to the chapter called Basic Features and Operations of the R Language.

```
# Create data frames with same variables but arbitrary values
df1 <- data.frame(id=c(1:6), age=c(21:26), sex=c(rep("male", 6)))
df2 <- data.frame(id=c(7:10), age=c(27:30), sex=c(rep("female", 4)))
```

```
# View first data frame
df1
```

```
##   id age  sex
## 1  1  21 male
## 2  2  22 male
## 3  3  23 male
## 4  4  24 male
## 5  5  25 male
## 6  6  26 male
```

```
# View second data frame
df2
```

```
##   id age    sex
## 1  7  27 female
## 2  8  28 female
## 3  9  29 female
## 4 10  30 female
```

Given that these two data frames (i.e., `df1`, `df2`) have the exact same variable names (`id`, `age`, and `sex`), we can easily perform a vertical join using the `rbind` function. To do so, enter the names of the two data frames as arguments, separated by a comma. Use the `<-` symbol to name the merged data frame something, which for this case, I arbitrarily named it `mergeddf2`.

```
# Verticle merge
mergeddf2 <- rbind(df1, df2)

# View the merged data frame
mergeddf2
```

```
##    id age    sex
```

```
## 1   1  21   male
## 2   2  22   male
## 3   3  23   male
## 4   4  24   male
## 5   5  25   male
## 6   6  26   male
## 7   7  27 female
## 8   8  28 female
## 9   9  29 female
## 10 10  30 female
```

Note how the two data frames are now "stacked" on one another. This was possible because they shared the same variables names and variables types (e.g., numeric and character).

8.3 Summary

Joining (merging) data frames in R is a useful practice. In this chapter, we learned how to perform a horizontal join using the `right_join`, `left_join`, `inner_join`, and `full_join` functions from the `dplyr` package. We also learned how to perform a vertical join using the `rbind` function from base R.

Chapter 9

Filtering Data

Applying a **filter** to data (i.e., creating a **subset** of data) is an important aspect of data management. When filter data, we either (a) select a subset of cases based on values/scores on one or more variables or (b) select a subset of variables. In this tutorial, you will learn some fundamental techniques for filtering your data in `R`.

9.0.0.1 Video Tutorial

Link to Video Tutorial: https://youtu.be/izVcbPmu0D0

9.0.0.2 Functions & Packages Introduced

Function	Package
`str`	base
`filter`	`dplyr`
`c`	base
`as.Date`	base
`select`	`dplyr`
`subset`	base

9.0.0.3 Initial Steps

If you haven't already, save the files called **"PersData.csv"** and **"PerfData.csv"** into a folder that you will subsequently set as your working directory. Your working directory will likely be different than the one shown below (i.e., `"H:/RWorkshop"`). As a reminder, you can access all of the data files refer-

enced in this book by downloading them as a compressed (zipped) folder from the my GitHub site: https://github.com/davidcaughlin/R-Tutorial-Data-Files; once you've followed the link to GitHub, just click "Code" (or "Download") followed by "Download ZIP", which will download all of the data files referenced in this book. For the sake of parsimony, I recommend downloading all of the data files into the same folder on your computer, which will allow you to set that same folder as your working directory for each of the chapters in this book.

Next, using the `setwd` function, set your working directory to the folder in which you saved the data file for this chapter. Alternatively, you can manually set your working directory folder in your drop-down menus by going to *Session > Set Working Directory > Choose Directory…*. Be sure to create a new R script file (.R) or update an existing R script file so that you can save your script and annotations. If you need refreshers on how to set your working directory and how to create and save an R script, please refer to Setting a Working Directory and Creating & Saving an R Script.

```
# Set your working directory
setwd("H:/RWorkshop")
```

Next, read in the .csv data files called **"PersData.csv"** and **"PerfData.csv"** using your choice of read function. In this example, I use the `read_csv` function from the `readr` package (Wickham et al., 2018). If you choose to use the `read_csv` function, be sure that you have installed and accessed the `readr` package using the `install.packages` and `library` functions. *Note: You don't need to install a package every time you wish to access it; in general, I would recommend updating a package installation once ever 1-3 months.* For refreshers on installing packages and reading data into R, please refer to Packages and Reading Data into R.

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)

# Read data and name data frame (tibble) objects
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
```

```
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
performancedata <- read_csv("PerfData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   perf_q1 = col_double(),
##   perf_q2 = col_double(),
##   perf_q3 = col_double(),
##   perf_q4 = col_double()
## )
```

```
# View the names of the variables in the data frame (tibble) objects
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
names(performancedata)
```

```
## [1] "id"      "perf_q1" "perf_q2" "perf_q3" "perf_q4"
```

```
# View data frame (tibble) objects
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

```
performancedata
```

```
## # A tibble: 6 x 5
##      id perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   153     3.9     4.8     4.9     5
## 2   125     2.1     1.9     2.1     2.3
## 3   111     3.3     3.3     3.4     3.3
## 4   198     4.9     4.5     4.4     4.8
## 5   201     1.2     1.1     1       1
## 6   282     2.2     2.3     2.4     2.5
```

As you can see from the output generated in your console, on the one hand, the `personaldata` data frame object contains basic employee demographic information. The variable names include: `id`, `lastname`, `firstname`, `startdate`, and `gender`. On the other hand, the `personaldata` data frame object contains the same `id` unique identifer variable as the `personaldata` data frame object, but instead of employee demographic information, this data frame object includes varuables associated with quarterly employee performance: `perf_q1`, `perf_q2`, `perf_q3`, and `perf_q4`.

To make this chapter more interesting (and for the sake of practice), let's use the `full_join` function from `dplyr` to join (merge) the two data frames we just read in (`personaldata`, `performancedata`) using the `id` variable as the key variable. Let's arbitrarily name the new joined (merged) data frame `mergeddf` using the `<-` operator. For more information on joining data, check out the chapter called Joining (Merging) Data.

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("dplyr")
```

```
# Access package
library(dplyr)
```

```
# Full join (without pipe)
mergeddf <- full_join(personaldata, performancedata, by="id")

# View joined (merged) data frame object
mergeddf
```

```
## # A tibble: 9 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
```

```
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez    Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald   Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith      John      1/9/2016  male      NA      NA      NA      NA
## 4   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 5   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 6   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 7   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 8   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 9   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Now we have a joined data frame called `mergeddf`!

9.1 Filter Data by Cases

Sometimes we want to select only a subset of cases from a data frame or table. There are different functions that can achieve this end. For example, the `subset` function filter from base `R` will do the trick. With that said, the `dplyr` package offers the `filter` function which has some advantages (e.g., faster with larger amounts of data), and thus, I recommend that you use the `filter` function, as we will do in this tutorial. In this tutorial, we will first learn how to apply a filter using the `filter` function from `dplyr` and later using the `subset` function from base `R`. You can use either approach, but as I mentioned there are some speed advantages to the `dplyr` version of the function that become more noticeable with larger datasets.

In order to properly filter data by cases, we need to know the respective types (classes) of the variables in the data frame. Perhaps the quickest way to find out the type (class) of each variable in the data frame is to use the `str` (structure) function from base `R`, and the function's parentheses, just enter the name of the data frame (`mergeddf`).

```
# Determine class of variables
str(mergeddf)
```

```
## tibble [9 x 9] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ id       : num [1:9] 153 154 155 165 125 111 198 201 282
##  $ lastname : chr [1:9] "Sanchez" "McDonald" "Smith" "Doe" ...
##  $ firstname: chr [1:9] "Alejandro" "Ronald" "John" "Jane" ...
##  $ startdate: chr [1:9] "1/1/2016" "1/9/2016" "1/9/2016" "1/4/2016" ...
##  $ gender   : chr [1:9] "male" "male" "male" "female" ...
##  $ perf_q1  : num [1:9] 3.9 NA NA NA 2.1 3.3 4.9 1.2 2.2
##  $ perf_q2  : num [1:9] 4.8 NA NA NA 1.9 3.3 4.5 1.1 2.3
##  $ perf_q3  : num [1:9] 4.9 NA NA NA 2.1 3.4 4.4 1 2.4
```

```
##  $ perf_q4  : num [1:9] 5 NA NA NA 2.3 3.3 4.8 1 2.5
##  - attr(*, "spec")=
##   .. cols(
##   ..   id = col_double(),
##   ..   lastname = col_character(),
##   ..   firstname = col_character(),
##   ..   startdate = col_character(),
##   ..   gender = col_character()
##   .. )
```

Note that the `id` variable is of type integer; the `lastname`, `firstname`, `startdate`, and `gender` variables are of type character (string); and the `perf_q4`, `perf_q4`, `perf_q4`, and `perf_q4` variables are of type numeric. The variable type will have important implications for how use use the `filter` function from `dplyr`.

In `R`, we can apply any one of the following logical operators when filtering our data:

Logical Operator	Definition
`<`	"less than"
`>`	"greater than"
`<=`	"less than or equal to"
`>=`	"greater than or equal to"
`==`	"equal to"
`!=`	"not equal to"
`|`	"or"
`&`	"and"
`!`	"not"

9.2 Option 1: Using `filter` function from `dplyr`

To get started, install and access the `dplyr` package.

```
# Install package
install.packages("dplyr")
```

```
# Access package
library(dplyr)
```

I will demonstrate two approaches applying the `filter` function from `dplyr`. The first option uses "pipe(s)," which in `R` is represented by the `%>%` operator. The pipe operator comes from a package called `magrittr`, on which the

dplyr is partially dependent. In short, a pipe allows one to more efficiently code/script and to improve the readability of the code/script under certain conditions. Specifically, a **pipe** forwards the result or value of one object or expression to a subsequent function. In doing so, one can avoid writing functions in which other functions are nested parenthetically. The second option is more traditional and lacks the efficiency and readability of pipes. You can use either approach, and if don't you want to use pipes, skip to the section below called *Filter Data without Pipes*. For more information on the pipe operator, check out this link: https://r4ds.had.co.nz/pipes.html.

9.2.1 *With* Pipes

Using an approach with pipes, first, use the <- symbol to name the filtered data frame that we will create. For this example, I name the new joined data frame filterdf; you could name it whatever you would like. Second, type the name of the first data frame, which we named mergeddf (see above), followed by the pipe (%>%) operator. This will "pipe" our data frame into the subsequent function. Third, either on the same line or on the next line, type the filter function. Fourth, within the function parentheses, type the name of the variable we wish to filter the data frame by, which in this example is gender. Fourth, type a logical operator, which for this example is ==. Fifth, type a value for the filter variable, which in this example is "female"; because the gender variable is of type *character*, we need to put quotation marks (" ") around the value of the variable that we wish to filter by. Remember, object names in R are case and space sensitive; for instance, gender is different from Gender, and "female" is different from "Female".

```
# Filter in by gender with pipe
filterdf <- mergeddf %>% filter(gender=="female")

# View filtered data frame
filterdf
```

```
## # A tibble: 3 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 2   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 3   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
```

Note how the data frame above contains only those cases with "female" as their gender variable designation. The filter worked as expected.

Alternatively, we could filter *out* those cases in which gender is equal to "female" using the != (not equal to) logical operator.

```
# Filter out by gender with pipe
filterdf <- mergeddf %>% filter(gender!="female")

# View filtered data frame
filterdf
```

```
## # A tibble: 6 x 9
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
## 4   125 Franklin Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 5   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 6   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Note how cases with `gender` equal to "female" are no longer in the data frame, while every other case is retained.

Let's now filter by a variable of type numeric (or integer). Specifically, let's select those cases in which the `perf_q2` variable is greater than (`>`) 4.0. Because the `perf_q2` variable is of type *numeric*, we don't use quotation marks (`" "`) around the value we wish to filter by, which in this case is 4.0.

```
# Filter by perf_q2 with pipe
filterdf <- mergeddf %>% filter(perf_q2>4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 2 x 9
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
```

If we wish to filter by two variables, we can apply the logical "or" (`|`) operator or "and" (`&`) operator. First, let's select those cases in which either `gender` is equal to "female" or `perf_q2` is greater than 4.0 using the "or" (`|`) operator.

```
# Filter by gender or perf_q2 with pipe
filterdf <- mergeddf %>% filter(gender=="female" | perf_q2>4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 4 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez    Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 3   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 4   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
```

Watch what happens if we apply the logical "and" (&) operator with the same syntax as above.

```
# Filter by gender and perf_q2 with pipe
filterdf <- mergeddf %>% filter(gender=="female" & perf_q2>4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 1 x 9
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
```

We can also use the logical "or" (|) operator to select two values of the same variable.

```
# Filter by two values of firstname with pipe
filterdf <- mergeddf %>% filter(firstname=="John" | firstname=="Jane")

# View filtered data frame
filterdf
```

```
## # A tibble: 3 x 9
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
## 2   165 Doe      Jane      1/4/2016  female    NA      NA      NA      NA
## 3   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Or we can select two ranges of values from the same variable using the logical "or" (|) operator, assuming the variable is of type numeric, integer, or date.

```
# Filter by two ranges of values of perf_q1 with pipe
filterdf <- mergeddf %>% filter(perf_q1<=2.5 | perf_q1>=4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 4 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 2   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 3   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 4   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

The `filter` function can also be used to remove multiple specific cases (such as from a unique identifier variable), which might be useful when you've identified outliers that need to be removed. As a first step, identify a vector of values that need to be removed. In this example, let's pretend that cases with `id` variable values of 198 and 201 no longer work for this company, so they should be removed from the sample. To create a vector of these two values, use the `c` function like this: `c(198,201)`. Next, because you are now filtering by a vector, you will need to use the `%in%` operator, which is an operator that instructs `R` to go through each value of the filter variable (`id`) and identify instances of 198 and 201 (`c(198,201)`); if the values match, then those cases are retained. However, because we entered `!` in front of the filter variable, this actually reverses our logic and instructs `R` to *remove* those cases in which a value of the filter variable matches a value contained in the vector.

```
# Filter out id of 198 and 201 with pipe
filterdf <- mergeddf %>% filter(!id %in% c(198,201))

# View filtered data frame
filterdf
```

```
## # A tibble: 7 x 9
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
## 4   165 Doe      Jane      1/4/2016  female    NA      NA      NA      NA
## 5   125 Franklin Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 6   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 7   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
```

Note that in the output above cases with `id` variable values equal to 198 and 201 are no longer present.

If you remove the `!` in front of the filter variable, only cases `198` and `201` are retained.

```
# Filter in id of 198 and 201 with pipe
filterdf <- mergeddf %>% filter(id %in% c(198,201))

# View filtered data frame
filterdf
```

```
## # A tibble: 2 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   198 Morales    Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 2   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
```

And if you wanted to remove just a single case, you could use the unique identifier variable (`id`) and the following script/code.

```
# Filter out id of 198 with pipe
filterdf <- mergeddf %>% filter(id!=198)

# View filtered data frame
filterdf
```

```
## # A tibble: 8 x 9
##      id lastname   firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>      <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez    Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald   Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith      John      1/9/2016  male      NA      NA      NA      NA
## 4   165 Doe        Jane      1/4/2016  female    NA      NA      NA      NA
## 5   125 Franklin   Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 6   111 Newton     Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 7   201 Providence Cindy     1/9/2016  female     1.2     1.1     1       1
## 8   282 Legend     John      1/9/2016  male       2.2     2.3     2.4     2.5
```

When working with variables of type *Date*, things can get a bit trickier. When we applied the `str` function from base `R` (see above), we found that the `startdate` variable was read in and joined as a character variable as opposed to a date variable. As such, we need to convert the `startdate` variable using the `as.Date` function from base `R`. First, type the name of the data frame object (`mergeddf`), followed by the `$` operator and the name of whatever you want to call the new variable (`startdate2`); remember, the `$` operator tells `R` that a variable belongs to (or will belong to) a particular data frame. Second, type the `<-` operator. Third, type the name of the `as.Date` function. Fourth, in the function parentheses, as the first argument, enter the `as.character` function with the name of the data frame object (`mergeddf`), followed by the `$` operator

and the name the original variable (`startdate`) as the sole argument. Fifth, as the second argument in the `as.Date` function, type `format="%m/%d/%Y"` to indicate the format for the data variable; note that the capital `Y` in `%Y` implies a 4-digit year, whereas a lower case would imply a 2-digit year.

```
# Convert character startdate variable to the Date type startdate2 variable
mergeddf$startdate2 <- as.Date(as.character(mergeddf$startdate), format="%m/%d/%Y")
```

To verify that the new `startdate2` variable is of type date, use the `str` function from base `R`, and enter the name of the data frame object (`mergeddf`) as the sole argument. As you will see, the new `startdate2` variable is now of type *Date*.

```
# Verify that the startdate2 variable is now a variable of type Date
str(mergeddf)
```

```
## tibble [9 x 10] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ id        : num [1:9] 153 154 155 165 125 111 198 201 282
##  $ lastname  : chr [1:9] "Sanchez" "McDonald" "Smith" "Doe" ...
##  $ firstname : chr [1:9] "Alejandro" "Ronald" "John" "Jane" ...
##  $ startdate : chr [1:9] "1/1/2016" "1/9/2016" "1/9/2016" "1/4/2016" ...
##  $ gender    : chr [1:9] "male" "male" "male" "female" ...
##  $ perf_q1   : num [1:9] 3.9 NA NA NA 2.1 3.3 4.9 1.2 2.2
##  $ perf_q2   : num [1:9] 4.8 NA NA NA 1.9 3.3 4.5 1.1 2.3
##  $ perf_q3   : num [1:9] 4.9 NA NA NA 2.1 3.4 4.4 1 2.4
##  $ perf_q4   : num [1:9] 5 NA NA NA 2.3 3.3 4.8 1 2.5
##  $ startdate2: Date[1:9], format: "2016-01-01" "2016-01-09" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   id = col_double(),
##   ..   lastname = col_character(),
##   ..   firstname = col_character(),
##   ..   startdate = col_character(),
##   ..   gender = col_character()
##   .. )
```

Now we are ready to filter using the new `startdate2` variable. When specify the value of the `startdate2` variable by which you wish to filter by, make sure to use the `as.Date` function once more with the date (formatted as YYYY-MM-DD) in quotation marks (`" "`) as the sole argument. Here, I filter for those cases in which their `startdate2` values are greater than 2016-01-07.

```
# Filter by startdate2 with pipe
filterdf <- mergeddf %>% filter(startdate2 > as.Date("2016-01-07"))
```

```
# View filtered data frame
print(filterdf)
```

```
## # A tibble: 5 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   154 McDonald Ronald    1/9/2016  male      NA      NA      NA      NA
## 2   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
## 3   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 4   201 Provide~ Cindy     1/9/2016  female     1.2     1.1     1       1
## 5   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

9.2.2 *Without* Pipes

We can also filter using the `filter` function from the `dplyr` package without using the pipe (`%>%`) operator. Note how I simply move the name of the data frame object from before the pipe (`%>%`) operator to the first argument in the `filter` function. Everything else remains the same. For simplicity, I don't display the output below as it is the same as the output as above using pipes. *Your decision whether to use a pipe operator is completely up to you.*

Let's filter the `mergeddf` data frame object such that only those cases for which the `gender` variable is equal to "female" are retained. Note how we apply the equal to (`==`) logical operator. A table of logical operators is presented towards the beginning of this tutorial.

```
# Filter in by gender without pipe
filterdf <- filter(mergeddf, gender=="female")

# View filtered data frame
filterdf
```

Now let's filter *out* those cases in which `gender` is *not* equal to "female". We apply the not equal to (`!=`) logical operator to do so.

```
# Filter in by gender without pipe
filterdf <- filter(mergeddf, gender!="female")

# View filtered data frame
filterdf
```

Filter the data frame such that we retain those cases for which the `perf_q2` variable is greater than (`>`) 4.0. Because the `perf_q2` variable is numeric, we *don't* put the value 4.0 in quotation marks.

```
# Filter by perf_q2 without pipe
filterdf <- filter(mergeddf, perf_q2>4.0)

# View filtered data frame
filterdf
```

Using the logical "or" operator (`|`), select those cases for which `gender` is equal to "female" *or* for which `perf_q2` is greater than 4.0.

```
# Filter by gender or perf_q2 without pipe
filterdf <- filter(mergeddf, gender=="female" | perf_q2>4.0)

# View filtered data frame
filterdf
```

Using the logical "and" operator (`&`), select those cases for which `gender` is equal to "female" *and* for which `perf_q2` is greater than 4.0. Note the difference in the resulting filtered data frame.

```
# Filter by gender and perf_q2 without pipe
filterdf <- filter(mergeddf, gender=="female" & perf_q2>4.0)

# View filtered data frame
filterdf
```

Using the logical "or" operator (`|`), select those cases for which `firstname` is equal to "John" *or* for which `firstname` is equal to "Jane". In other words, select those individuals whose names are either "John" or "Jane".

```
# Filter by two values of firstname without pipe
filterdf <- filter(mergeddf, firstname=="John" | firstname=="Jane")

# View filtered data frame
filterdf
```

Using the logical "or" operator (`|`), select the range of cases for which `perf_q1` is less than equal to (`<=`) 2.5 *or* for which `perf_q1` is greater than or equal (`>=`) to 4.0.

```
# Filter by two ranges of values of perf_q1 without pipe
filterdf <- filter(mergeddf, perf_q1<=2.5 | perf_q1>=4.0)

# View filtered data frame
filterdf
```

The `filter` function can also be used to remove multiple specific cases (such as from a unique identifier variable), which might be useful when you've identified outliers that need to be removed. As a first step, identify a vector of values that need to be removed. In this example, let's pretend that cases with `id` variable values of 198 and 201 no longer work for this company, so they should be removed from the sample. To create a vector of these two values, use the `c` function like this: `c(198,201)`. Next, because you are now filtering by a vector, you will need to use the `%in%` operator, which is an operator that instructs `R` to go through each value of the filter variable (`id`) and identify instances of 198 and 201 (`c(198,201)`); if the values match, then those cases are retained. However, because we entered `!` in front of the filter variable, this actually reverses our logic and instructs `R` to *remove* those cases in which a value of the filter variable matches a value contained in the vector.

```
# Filter out id of 198 and 201 without pipe
filterdf <- filter(mergeddf, !id %in% c(198,201))

# View filtered data frame
filterdf
```

Or if you wish to retain only those cases for which the `id` variable is equal to 198 and 201, drop the not operator (`!`) from the previous script.

```
# Filter in id of 198 and 201 without pipe
filterdf <- filter(mergeddf, id %in% c(198,201))

# View filtered data frame
filterdf
```

You can also drop specific cases one by one using the not equal to operator (`!=`) and the a unique identifier value associated with the case you wish to remove. We accomplish the same result as above but use two steps instead. Also, note that in the second step below, the new data frame object (`filterdf`) is used as the first argument because we want to retain the changes we made in the prior step (i.e., dropping case with `id` equal to 198).

```
# Filter in id of 198 without pipe
filterdf <- filter(mergeddf, id!=198)

# Filter in id of 201 without pipe
filterdf <- filter(filterdf, id!=201)

# View filtered data frame
filterdf
```

When working with variables of type *Date*, things can get a bit trickier. When we applied the `str` function from base `R` (see above), we found that the `startdate` variable was read in and joined as a character variable as opposed to a date variable. As such, we need to convert the `startdate` variable using the `as.Date` function from base `R`. First, type the name of the data frame object (`mergeddf`), followed by the `$` operator and the name of whatever you want to call the new variable (`startdate2`); remember, the `$` operator tells `R` that a variable belongs to (or will belong to) a particular data frame. Second, type the `<-` operator. Third, type the name of the `as.Date` function. Fourth, in the function parentheses, as the first argument, enter the `as.character` function with the name of the data frame object (`mergeddf`), followed by the `$` operator and the name the original variable (`startdate`) as the sole argument. Fifth, as the second argument in the `as.Date` function, type `format="%m/%d/%Y"` to indicate the format for the data variable; note that the capital `Y` in `%Y` implies a 4-digit year, whereas a lower case would imply a 2-digit year. To verify that the new `startdate2` variable is of type date, on the next line, use the `str` function from base `R`, and enter the name of the data frame object (`mergeddf`) as the sole argument. As you will see, the new `startdate2` variable is now of type *Date*.

```
# Convert character startdate variable to the date type startdate2 variable
mergeddf$startdate2 <- as.Date(as.character(mergeddf$startdate), format="%m/%d/%Y")

# Verify that the startdate2 variable is now a variable of type date
str(mergeddf)
```

Now we are ready to filter using the new `startdate2` variable. When specify the value of the `startdate2` variable by which you wish to filter by, make sure to use the `as.Date` function once more with the date (formatted as YYYY-MM-DD) in quotation marks (`" "`) as the sole argument. Here, I filter for those cases in which their `startdate2` values are greater than 2016-01-07.

```
# Filter by startdate2 without pipe
filterdf <- filter(mergeddf, startdate2 > as.Date("2016-01-07"))

# View filtered data frame
print(filterdf)
```

9.3 Option 2: Using `subset` function from base `R`

To filter a data frame by cases using the `subset` function from base `R`, use the `<-` symbol to name the filtered data frame that we are about to create. For

this example, I name the new joined data frame `filterdf`; you could name it whatever you would like. To the right of the `<-` symbol, type the name of `subset` function from base `R`. As the first argument in the function, enter the name of the data frame we created above (`mergeddf`). As the second argument, type the name of the variable we wish to filter the data frame by, which in this example is `gender` followed by a logical/conditional argument. For this example, we wish to to retain only those cases in which `gender` is equal to "female", and we do so using this logical arguent `gender=="female"`. Because the `gender` variable is of type *character*, we need to put quotation marks (`" "`) around the value of the variable that we wish to filter by. Remember, object names in `R` are case and space sensitive; for instance, `gender` is different from `Gender`, and "female" is different from "Female".

```
# Filter by gender
filterdf <- subset(mergeddf, gender=="female")

# View filtered data frame
filterdf
```

```
## # A tibble: 3 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   165 Doe      Jane      1/4/2016  female    NA      NA      NA      NA
## 2   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 3   201 Provide~ Cindy     1/9/2016  female     1.2     1.1     1       1
## # ... with 1 more variable: startdate2 <date>
```

Note how the data frame above contains only those cases with "female" as their `gender` variable designation. The filter worked as expected.

Alternatively, we could filter *out* those cases in which `gender` is equal to "female" using the `!=` (not equal to) logical operator.

```
# Filter by gender
filterdf <- subset(mergeddf, gender!="female")

# View filtered data frame
filterdf
```

```
## # A tibble: 6 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
```

```
## 4   125 Franklin Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 5   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 6   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

Note how cases with `gender` equal to "female" are no longer in the data frame, while every other case is retained.

Let's now filter by a variable of type *numeric* (or *integer*). Specifically, let's select those cases in which the `perf_q2` variable is greater than (`>`) 4.0. Because the `perf_q2` variable is of type *numeric*, we don't use quotation marks (`" "`) around the value we wish to filter by, which in this case is 4.0.

```
# Filter by perf_q2
filterdf <- subset(mergeddf, perf_q2>4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 2 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## # ... with 1 more variable: startdate2 <date>
```

If we wish to filter by two variables, we can apply the logical "or" (`|`) operator or "and" (`&`) operator. First, let's select those cases in which either `gender` is equal to "female" or `perf_q2` is greater than 4.0 using the "or" (`|`) operator.

```
# Filter by gender or perf_q2
filterdf <- subset(mergeddf, gender=="female" | perf_q2>4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 4 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   165 Doe      Jane      1/4/2016  female    NA      NA      NA      NA
## 3   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 4   201 Provide~ Cindy     1/9/2016  female     1.2     1.1     1       1
## # ... with 1 more variable: startdate2 <date>
```

Watch what happens if we apply the logical "and" (&) operator with the same syntax as above.

```
# Filter by gender and perf_q2
filterdf <- subset(mergeddf, gender=="female" & perf_q2>4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 1 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## # ... with 1 more variable: startdate2 <date>
```

We can also use the logical "or" (|) operator to select two values of the same variable.

```
# Filter by two values of firstname
filterdf <- subset(mergeddf, firstname=="John" | firstname=="Jane")

# View filtered data frame
filterdf
```

```
## # A tibble: 3 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   155 Smith    John      1/9/2016  male       NA      NA      NA      NA
## 2   165 Doe      Jane      1/4/2016  female     NA      NA      NA      NA
## 3   282 Legend   John      1/9/2016  male      2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

Or we can select two ranges of values from the same variable using the logical "or" (|) operator, assuming the variable is of type numeric, integer, or date.

```
# Filter by two ranges of values of perf_q1
filterdf <- subset(mergeddf, perf_q1<=2.5 | perf_q1>=4.0)

# View filtered data frame
filterdf
```

```
## # A tibble: 4 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
```

```
## 1   125 Franklin Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 2   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 3   201 Provide~ Cindy     1/9/2016  female     1.2     1.1     1       1
## 4   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

The `subset` function can also be used to remove multiple specific cases (such as from a unique identifier variable), which might be useful when you've identified outliers that need to be removed. As a first step, identify a vector of values that need to be removed. In this example, let's pretend that cases with `id` variable values of 198 and 201 no longer work for this company, so they should be removed from the sample. To create a vector of these two values, use the `c` function like this: `c(198,201)`. Next, because you are now filtering by a vector, you will need to use the `%in%` operator, which is an operator that instructs `R` to go through each value of the filter variable (`id`) and identify instances of 198 and 201 (`c(198,201)`); if the values match, then those cases are retained. However, because we entered `!` in front of the filter variable, this actually reverses our logic and instructs `R` to *remove* those cases in which a value of the filter variable matches a value contained in the vector.

```
# Filter out id of 198 and 201
filterdf <- subset(mergeddf, !id %in% c(198,201))

# View filtered data frame
filterdf
```

```
## # A tibble: 7 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
## 4   165 Doe      Jane      1/4/2016  female    NA      NA      NA      NA
## 5   125 Franklin Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 6   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 7   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

Note that in the output above cases with `id` variable values equal to 198 and 201 are no longer present.

If you remove the `!` in front of the filter variable, only cases `198` and `201` are retained.

```
# Filter in id of 198 and 201
filterdf <- subset(mergeddf, id %in% c(198,201))

# View filtered data frame
filterdf
```

```
## # A tibble: 2 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   198 Morales  Linda     1/7/2016  female     4.9     4.5     4.4     4.8
## 2   201 Provide~ Cindy     1/9/2016  female     1.2     1.1     1       1
## # ... with 1 more variable: startdate2 <date>
```

You can also drop specific cases one by one using the not equal to operator (`!=`) and the a unique identifier value associated with the case you wish to remove. We accomplish the same result as above but use two steps instead. Also, note that in the second step below, the new data frame object (`filterdf`) is used as the first argument because we want to retain the changes we made in the prior step (i.e., dropping case with `id` equal to 198).

```
# Filter out id of 198
filterdf <- subset(mergeddf, id!=198)

# Filter out id of 201
filterdf <- subset(filterdf, id!=201)

# View filtered data frame
filterdf
```

```
## # A tibble: 7 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   153 Sanchez  Alejandro 1/1/2016  male       3.9     4.8     4.9     5
## 2   154 McDonald Ronald    1/9/2016  male      NA      NA      NA      NA
## 3   155 Smith    John      1/9/2016  male      NA      NA      NA      NA
## 4   165 Doe      Jane      1/4/2016  female    NA      NA      NA      NA
## 5   125 Franklin Benjamin  1/5/2016  male       2.1     1.9     2.1     2.3
## 6   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 7   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

When working with variables of type *Date*, things can get a bit trickier. When we applied the `str` function from base `R` (see above), we found that the `startdate` variable was read in and joined as a character variable as opposed to

a date variable. As such, we need to convert the `startdate` variable using the `as.Date` function from base `R`. First, type the name of the data frame object (`mergeddf`), followed by the `$` operator and the name of whatever you want to call the new variable (`startdate2`); remember, the `$` operator tells `R` that a variable belongs to (or will belong to) a particular data frame. Second, type the `<-` operator. Third, type the name of the `as.Date` function. Fourth, in the function parentheses, as the first argument, enter the `as.character` function with the name of the data frame object (`mergeddf`), followed by the `$` operator and the name the original variable (`startdate`) as the sole argument. Fifth, as the second argument in the `as.Date` function, type `format="%m/%d/%Y"` to indicate the format for the data variable; note that the capital `Y` in `%Y` implies a 4-digit year, whereas a lower case would imply a 2-digit year.

```
# Convert character startdate variable to the Date type startdate2 variable
mergeddf$startdate2 <- as.Date(as.character(mergeddf$startdate), format="%m/%d/%Y")
```

To verify that the new `startdate2` variable is of type *Date*, use the `str` function from base `R`, and enter the name of the data frame object (`mergeddf`) as the sole argument. As you will see, the new `startdate2` variable is now of type *Date*.

```
# Verify that the startdate2 variable is now a variable of type Date
str(mergeddf)
```

```
## tibble [9 x 10] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ id        : num [1:9] 153 154 155 165 125 111 198 201 282
##  $ lastname  : chr [1:9] "Sanchez" "McDonald" "Smith" "Doe" ...
##  $ firstname : chr [1:9] "Alejandro" "Ronald" "John" "Jane" ...
##  $ startdate : chr [1:9] "1/1/2016" "1/9/2016" "1/9/2016" "1/4/2016" ...
##  $ gender    : chr [1:9] "male" "male" "male" "female" ...
##  $ perf_q1   : num [1:9] 3.9 NA NA NA 2.1 3.3 4.9 1.2 2.2
##  $ perf_q2   : num [1:9] 4.8 NA NA NA 1.9 3.3 4.5 1.1 2.3
##  $ perf_q3   : num [1:9] 4.9 NA NA NA 2.1 3.4 4.4 1 2.4
##  $ perf_q4   : num [1:9] 5 NA NA NA 2.3 3.3 4.8 1 2.5
##  $ startdate2: Date[1:9], format: "2016-01-01" "2016-01-09" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   id = col_double(),
##   ..   lastname = col_character(),
##   ..   firstname = col_character(),
##   ..   startdate = col_character(),
##   ..   gender = col_character()
##   .. )
```

Now we are ready to filter using the new `startdate2` variable. When specify the value of the `startdate2` variable by which you wish to filter by, make sure

to use the `as.Date` function once more with the date (formatted as YYYY-MM-DD) in quotation marks (`" "`) as the sole argument. Here, I filter for those cases in which their `startdate2` values are greater than 2016-01-07.

```
# Filter by startdate2
filterdf <- subset(mergeddf, startdate2 > as.Date("2016-01-07"))

# View filtered data frame
print(filterdf)
```

```
## # A tibble: 5 x 10
##      id lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl> <chr>    <chr>     <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl>
## 1   154 McDonald Ronald    1/9/2016  male        NA      NA      NA      NA
## 2   155 Smith    John      1/9/2016  male        NA      NA      NA      NA
## 3   111 Newton   Isaac     1/9/2016  male       3.3     3.3     3.4     3.3
## 4   201 Provide~ Cindy     1/9/2016  female     1.2     1.1     1       1
## 5   282 Legend   John      1/9/2016  male       2.2     2.3     2.4     2.5
## # ... with 1 more variable: startdate2 <date>
```

9.4 Remove Single Variable from Data Frame

If you just need to remove a single variable from a data frame, using the `NULL` object in `R` in conjunction with the `<-` operator can designate which variable to drop. For example, if we wish to drop the `startdate` variable from the `mergeddf` data frame, we simply note that `startdate` belongs to `mergeddf` by joining them with `$`. Next, we set `<- NULL` adjacent to `mergeddf$startdate` to indicate that we wish to remove that variable from that data frame.

```
# Remove variable
mergeddf$startdate <- NULL

# View updated data frame
mergeddf
```

```
## # A tibble: 9 x 9
##      id lastname   firstname gender perf_q1 perf_q2 perf_q3 perf_q4 startdate2
##   <dbl> <chr>      <chr>     <chr>    <dbl>   <dbl>   <dbl>   <dbl> <date>
## 1   153 Sanchez    Alejandro male       3.9     4.8     4.9     5   2016-01-01
## 2   154 McDonald   Ronald    male        NA      NA      NA      NA   2016-01-09
## 3   155 Smith      John      male        NA      NA      NA      NA   2016-01-09
## 4   165 Doe        Jane      female      NA      NA      NA      NA   2016-01-04
## 5   125 Franklin   Benjamin  male       2.1     1.9     2.1     2.3 2016-01-05
```

```
## 6   111 Newton     Isaac     male       3.3     3.3     3.4     3.3 2016-01-09
## 7   198 Morales    Linda     female     4.9     4.5     4.4     4.8 2016-01-07
## 8   201 Providence Cindy     female     1.2     1.1     1       1   2016-01-09
## 9   282 Legend     John      male       2.2     2.3     2.4     2.5 2016-01-09
```

9.5 Select Multiple Variables from Data Frame

If you wish to *select multiple variables* from a data frame (and remove all others), the `select` function from `dplyr` and `subset` function from base `R` are quite useful. You can use either function, and I demonstrate both below.

9.6 Option 1: Using `select` Function from `dplyr`

We'll begin our quest to select multiple variables by applying the `select` function from the `dplyr` package. In addition, I demonstrate how to do so with and without pipes. If you don't want to use pipes, feel free to skip down to the section called *Without Pipes*.

9.6.1 *With* Pipe

Using the pipe (`%>%`) operator, first, decide whether you want to override an existing data frame or create a new data frame based on our seletion; here, I override the `mergeddf` data frame using the `<-` operator, which results in `mergeddf <-`. Second, type the name of the original data frame (`mergeddf`), followed by the pipe (`%>%`) operator. Third, type the name of the `select` function. Fourth, in the parentheses, list the names of the variables you wish to select as arguments; all variables that are not listed will be dropped. Here, we are selecting (to retain) the `id`, `perf_q1`, `gender`, `lastname`, and `firstname` variables. Note that the updated date frame includes the selected variables in the order in which you listed them.

```
# Select multiple variables with pipe
mergeddf <- mergeddf %>% select(id, perf_q1, gender, lastname, firstname)

# View updated data frame
mergeddf
```

```
## # A tibble: 9 x 5
##      id perf_q1 gender lastname   firstname
##   <dbl>   <dbl> <chr>  <chr>      <chr>
## 1   153     3.9 male   Sanchez    Alejandro
```

```
## 2   154    NA   male   McDonald   Ronald
## 3   155    NA   male   Smith      John
## 4   165    NA   female Doe        Jane
## 5   125     2.1 male   Franklin   Benjamin
## 6   111     3.3 male   Newton     Isaac
## 7   198     4.9 female Morales    Linda
## 8   201     1.2 female Providence Cindy
## 9   282     2.2 male   Legend     John
```

9.6.2 *Without* Pipe

If you decide not to use the pipe (`%>%`) operator, the syntax remains almost the same except the name of the original data frame object (`mergeddf`) is moved from before the pipe (`%>%`) operator to the first argument in the `select` function. Everything else remains the same.

```
# Select multiple variables without pipe
mergeddf <- select(mergeddf, id, gender, lastname, firstname)

# View updated data frame
mergeddf
```

9.7 Option 2: Using `subset` Function from base `R`

As an alternative to the `select` function from `dplyr`, if you wish to *select multiple variables* from a data frame (and remove all others), the `subset` function from base `R` has a special argument that can do this. As before, name the new data frame using the `<-` symbol. As the first argument in the `subset` function, type the name of your original data frame object (`mergeddf`). As the second argument, type `select=` followed by a vector of variable name you wish to select/retain. The order in which you enter the variable names will correspond to the order in which they appear in the new data frame object. Use the `c` (combine) function from base `R` with each variable name you wish to select as arguments separated by commas.

```
# Select multiple variables
mergeddf <- subset(mergeddf, select=c(id, gender, lastname, firstname))

# View updated data frame
mergeddf
```

9.8 Remove Multiple Variables from Data Frame

If you wish to *remove multiple variables* from a data frame, either the `select` function from `dplyr` or the `subset` function from base `R` will work. Choose whichever one is most intuitive to you.

9.9 Option 1: Using `select` Function from `dplyr`

As the first option, let's apply the `select` function from `dplyr`. I demonstrate how to do so with and without pipes. If you don't want to use pipes, feel free to skip down to the section called *Without Pipes*.

9.9.1 *With* Pipe

Using the pipe (`%>%`) operator, first, decide whether you want to override an existing data frame or create a new data frame from the subset; here, I override the `mergeddf` data frame using the `<-` operator, which results in `mergeddf <-`. Second, type the name of the original data frame (`mergeddf`), followed by the pipe (`%>%`) operator. Third, enter the `select` function. Fourth, use the `c` (combine) function with `-` in front of it to note that you want to select all other variables *except* the ones listed in the `c` function.

```
# Remove multiple variables with pipe
mergeddf <- mergeddf %>% select(-c(lastname, firstname))

# View updated data frame
mergeddf
```

```
## # A tibble: 9 x 3
##      id perf_q1 gender
##   <dbl>   <dbl> <chr>
## 1   153     3.9 male
## 2   154    NA   male
## 3   155    NA   male
## 4   165    NA   female
## 5   125     2.1 male
## 6   111     3.3 male
## 7   198     4.9 female
## 8   201     1.2 female
## 9   282     2.2 male
```

Removing a single variable can also be done using the `select` function. To do so, just list a single variable with `-` in front of it (as the sole argument) to indicate that you wish to drop that variable.

```
# Remove single variable with pipe
mergeddf <- mergeddf %>% select(-gender)

# View updated data frame
mergeddf
```

```
## # A tibble: 9 x 2
##      id perf_q1
##   <dbl>   <dbl>
## 1   153     3.9
## 2   154    NA
## 3   155    NA
## 4   165    NA
## 5   125     2.1
## 6   111     3.3
## 7   198     4.9
## 8   201     1.2
## 9   282     2.2
```

9.9.2 *Without* Pipe

If you decide *not* to use the pipe (`%>%`) operator, the syntax remains mostly the same except the name of the original data frame object (`mergeddf`) is moved from before the pipe (`%>%`) operator to the first argument in the `select` function. Everything else remains the same.

```
# Remove multiple variables without pipe
mergeddf <- select(mergeddf, -c(lastname, firstname))

# View updated data frame
mergeddf
```

And here's the non-pipe equivalent to removing a single variable using this approach.

```
# Remove single variable without pipe
mergeddf <- mergeddf %>% select(-gender)

# View updated data frame
mergeddf
```

9.10 Option 2: Using `subset` Function from base `R`

As an alternative to the `select` function from `dplyr`, if our desire is to *remove multiple variables* from a data frame, we could use the `subset` function from base `R` instead. Use the same syntax as you did when selecting multiple variables, except insert a `-` (minus sign) in from the of the `c` function. This tells the function to *not* select those variables. Here, we remove the `id` and `gender` variables.

```
# Remove multiple variables
mergeddf <- subset(mergeddf, select= -c(id, gender))

# View updated data frame
mergeddf
```

9.11 Summary

Applying filters and creating subsets of cases (rows) and variables (columns) from a data frame is an important part of data management. The `dplyr` package has two useful functions that can be used for these purposes: `filter` and `select`. In addition, the `subset` function from base `R` can tackle both types of filters.

Chapter Supplements

Arranging (Sorting) Data: Chapter Supplement

In addition to the `arrange` function from the `dplyr` package covered in Arranging (Sorting) Data, we can use the `order` function from base R to arrange (sort) data by values for one or more variable. Because this function comes from base R, we do not need to install and access an additional package like we do with the `arrange` functions, which some may find advantageous.

Functions & Packages Introduced

Function	Package
`order`	base
`c`	base

Initial Steps

If required, please refer to the Initial Steps section from the chapter for more information on these initial steps.

```
# Set your working directory
setwd("H:/RWorkshop")
```

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)
```

```
# Read data and name data frame (tibble) object
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
# View the names of the variables in the data frame (tibble) object
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
# View data frame (tibble) object
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

`order` Function from Base R

To sort a data frame object in ascending order based on a single variable, we will use the `order` function from base R to do the following:

1. Type the name of the data frame object that you wish to arrange (sort) (`personaldata`).

2. Insert brackets (`[ ]`), which allow us to reference rows or columns depending on how we format the brackets. If we type a function or value before the comma, we are indicating that we wish to apply operations to row(s), and if we type a function or value after the comma, we are indicating that we wish to apply operations to column(s).
3. To sort the data frame into ascending rows by the `startdate` variable, type the name of the `order` function *before* the comma in the brackets. As the sole parenthetical argument of the `order` function, type the name of the `personaldata` data frame object, followed by the `$` operator and the name of the variable by which we wish to sort the data frame, which to reiterate is the `startdate` variable. The `$` operator signals to R that a variable belongs to a particular data frame object. By default, the `order` function sorts in ascending order.

```
# Arrange (sort) data by variable in ascending order
personaldata[order(personaldata$startdate),]
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   165 Doe        Jane      1/4/2016  female
## 3   125 Franklin   Benjamin  1/5/2016  male
## 4   198 Morales    Linda     1/7/2016  female
## 5   154 McDonald   Ronald    1/9/2016  male
## 6   155 Smith      John      1/9/2016  male
## 7   111 Newton     Isaac     1/9/2016  male
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

To change the ordering of data in the `personaldata` data frame object itself, we will need to (re)name the data frame object using the `<-` variable assignment operator. In this example, I will demonstrate how to overwrite the existing data frame object, and thus I give the data frame object the *exact* same name as it had originally (i.e., `personaldata`). To do so, to the *left* of the `<-` operator, type what you would like to name the new (updated) sorted data frame object (`personaldata`). Next, to the *right* of the `<-` operator, copy and paste the same code we wrote above. Finally, use the `head` function from base R to view the first six rows of the new data frame object.

```
# Arrange (sort) data by variable in ascending order
# and overwrite existing data frame object
personaldata <- personaldata[order(personaldata$startdate),]
```

```
# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   153 Sanchez  Alejandro 1/1/2016  male
## 2   165 Doe      Jane      1/4/2016  female
## 3   125 Franklin Benjamin  1/5/2016  male
## 4   198 Morales  Linda     1/7/2016  female
## 5   154 McDonald Ronald    1/9/2016  male
## 6   155 Smith    John      1/9/2016  male
```

To sort in *descending* order, add the argument `decreasing=TRUE` within the `order` function parentheses. *Remember, we use commas to separate arguments used in a function (if there are two or more arguments).*

```
# Arrange (sort) data by variable in descending order
personaldata <- personaldata[order(personaldata$startdate, decreasing=TRUE),]

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   111 Newton     Isaac     1/9/2016  male
## 4   201 Providence Cindy     1/9/2016  female
## 5   282 Legend     John      1/9/2016  male
## 6   198 Morales    Linda     1/7/2016  female
```

If we wish to sort a data frame object by two variables, as the second argument in the `order` function parentheses, simply add the name of the data frame object, followed by the `$` operator and the name of the second second variable. We will sort the data frame in by `gender` and `startdate`. The ordering of the two variables matters; the function sorts initially by the values/levels of the first variable listed and sorts subsequently by the values/levels of the second variable listed, but does so *within* the values/levels of the first variable listed. As shown below, `startdate` is sorted within the sorted levels of the `gender` variable. The default operation of the `arrange` function is to arrange (sort) the data in *ascending* order.

```
# Arrange (sort) data by two variables in ascending order
personaldata <- personaldata[order(personaldata$gender, personaldata$startdate),]

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   165 Doe        Jane      1/4/2016  female
## 2   198 Morales    Linda     1/7/2016  female
## 3   201 Providence Cindy     1/9/2016  female
## 4   153 Sanchez    Alejandro 1/1/2016  male
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   154 McDonald   Ronald    1/9/2016  male
```

To sort by one of the variables in descending order and the other variable by the default ascending order, we need to add the `decreasing=` argument, but because we have two variables, we need to provide a vector containing logical values (`TRUE`, `FALSE`) to indicate which variable we wish to apply a *descending* order. If the logical value is `TRUE` for the `decreasing=` argument, then we sort in descending variable. Using the `c` (combine) function from base R, we create a vector of two logical values whose order corresponds to the order in which we listed the two variables in the `order` function. For example, if the argument is `decreasing=c(FALSE, TRUE)`, then we sort the first variable in the default ascending order and the second variable in descending order, which is what we do below.

```
# Arrange (sort) data by gender in ascending order and
# startdate in descending order
personaldata <- personaldata[order(personaldata$gender, personaldata$startdate, decreasing=c(FALS

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   165 Doe        Jane      1/4/2016  female
## 2   198 Morales    Linda     1/7/2016  female
## 3   201 Providence Cindy     1/9/2016  female
## 4   153 Sanchez    Alejandro 1/1/2016  male
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   154 McDonald   Ronald    1/9/2016  male
```

Or, you could sort by both variables in descending order by change the argument to `decreasing=c(TRUE, TRUE)`.

```
# Arrange (sort) data by gender and id variables descending order
personaldata <- personaldata[order(personaldata$gender, personaldata$startdate, decreas

# View just the first 6 rows of the data frame in Console
head(personaldata)
```

```
## # A tibble: 6 x 5
##      id lastname firstname startdate gender
##   <dbl> <chr>    <chr>     <chr>     <chr>
## 1   154 McDonald Ronald    1/9/2016  male
## 2   155 Smith    John      1/9/2016  male
## 3   111 Newton   Isaac     1/9/2016  male
## 4   282 Legend   John      1/9/2016  male
## 5   125 Franklin Benjamin  1/5/2016  male
## 6   153 Sanchez  Alejandro 1/1/2016  male
```

Joining (Merging) Data: Chapter Supplement

In addition to the `join` functions from the `dplyr` package covered in Joining (Merging) Data, we can use the `merge` function from base R to perform a *horizontal join*. Because this function comes from base R, we do not need to install and access an additional package like we do with the `join` functions, which some may find advantageous.

Functions & Packages Introduced

Function	Package
`names`	base
`merge`	base

Initial Steps

If required, please refer to the Initial Steps section from the chapter for more information on these initial steps.

```
# Set your working directory
setwd("H:/RWorkshop")
```

```
# Install readr package if you haven't already
# [Note: You don't need to install a package every
# time you wish to access it]
install.packages("readr")
```

```
# Access readr package
library(readr)
```

```
# Read data and name data frame (tibble) objects
personaldata <- read_csv("PersData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   lastname = col_character(),
##   firstname = col_character(),
##   startdate = col_character(),
##   gender = col_character()
## )
```

```
performancedata <- read_csv("PerfData.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   perf_q1 = col_double(),
##   perf_q2 = col_double(),
##   perf_q3 = col_double(),
##   perf_q4 = col_double()
## )
```

```
# View the names of the variables in the data frame (tibble) objects
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
names(performancedata)
```

```
## [1] "id"      "perf_q1" "perf_q2" "perf_q3" "perf_q4"
```

```
# View data frame (tibble) objects
personaldata
```

```
## # A tibble: 9 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   153 Sanchez    Alejandro 1/1/2016  male
## 2   154 McDonald   Ronald    1/9/2016  male
## 3   155 Smith      John      1/9/2016  male
```

```
## 4   165 Doe        Jane      1/4/2016  female
## 5   125 Franklin   Benjamin  1/5/2016  male
## 6   111 Newton     Isaac     1/9/2016  male
## 7   198 Morales    Linda     1/7/2016  female
## 8   201 Providence Cindy     1/9/2016  female
## 9   282 Legend     John      1/9/2016  male
```

```
performancedata
```

```
## # A tibble: 6 x 5
##      id perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   153     3.9     4.8     4.9     5
## 2   125     2.1     1.9     2.1     2.3
## 3   111     3.3     3.3     3.4     3.3
## 4   198     4.9     4.5     4.4     4.8
## 5   201     1.2     1.1     1       1
## 6   282     2.2     2.3     2.4     2.5
```

```
# Remove case with id variable equal to 153
personaldata <- personaldata[personaldata$id != 153,]
```

merge Function from Base R

We will use the `merge` function to horizontally match cases from the `personaldata` and `performancedata` data frames using `id` as a key variable. To identify what the key variable is, let's use the `names` function from base R to retrieve the list of variable names from the two data frames, which we already did above. Nevertheless, let's call up those variable names once more. Simply enter the name of the data frame as a parenthetical argument in the `names` function.

```
# Retrieve variable names from first data frame
names(personaldata)
```

```
## [1] "id"        "lastname"  "firstname" "startdate" "gender"
```

```
# Retrieve variable names from second data frame
names(performancedata)
```

```
## [1] "id"      "perf_q1" "perf_q2" "perf_q3" "perf_q4"
```

As you can see in the variable names listed above, the `id` variable is common to both data frames, and thus it will serve as our key variable.

Let's begin with what is referred to as an **inner join**:

1. Use the `<-` operator to name the joined data frame that we create using the `merge` function. For this example, I name the new joined data frame `mergeddf`, which is completely arbitrary; you could name it whatever you would like. Type the name of the new joined data frame to the *left* of the `<-` operator.
2. To the *right* of the `<-` operator, type the name of the `merge` function. Within the `merge` function parentheses, we will provide the arguments needed to make this join a reality. First, enter the name of one of the data frames (e.g., `personaldata`), followed by a comma. Second, enter the name of of the other data frame (e.g., `performancedata`), followed by a comma. Third, use the `by=` argument to indicate the name of the key variable (e.g., `id`); make sure the key variable is in quotation marks (`" "`), and remember, object and variable names in R are case and space sensitive.

```
# Inner join
mergeddf <- merge(personaldata, performancedata, by="id")

# View the joined data frame
mergeddf
```

```
##    id   lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
## 1 111     Newton     Isaac  1/9/2016   male     3.3     3.3     3.4     3.3
## 2 125   Franklin  Benjamin  1/5/2016   male     2.1     1.9     2.1     2.3
## 3 198    Morales     Linda  1/7/2016 female     4.9     4.5     4.4     4.8
## 4 201 Providence     Cindy  1/9/2016 female     1.2     1.1     1.0     1.0
## 5 282     Legend      John  1/9/2016   male     2.2     2.3     2.4     2.5
```

Now, let's revisit the original data frame objects that we read in initially.

```
# View the first original data frame
personaldata
```

```
## # A tibble: 8 x 5
##      id lastname   firstname startdate gender
##   <dbl> <chr>      <chr>     <chr>     <chr>
## 1   154 McDonald   Ronald    1/9/2016  male
## 2   155 Smith      John      1/9/2016  male
## 3   165 Doe        Jane      1/4/2016  female
```

```
## 4   125 Franklin   Benjamin  1/5/2016  male
## 5   111 Newton     Isaac     1/9/2016  male
## 6   198 Morales    Linda     1/7/2016  female
## 7   201 Providence Cindy     1/9/2016  female
## 8   282 Legend     John      1/9/2016  male
```

```
# View the second original data frame
performancedata
```

```
## # A tibble: 6 x 5
##      id perf_q1 perf_q2 perf_q3 perf_q4
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   153     3.9     4.8     4.9     5
## 2   125     2.1     1.9     2.1     2.3
## 3   111     3.3     3.3     3.4     3.3
## 4   198     4.9     4.5     4.4     4.8
## 5   201     1.2     1.1     1       1
## 6   282     2.2     2.3     2.4     2.5
```

In the output, first, note how all of the variables from the original data frames (i.e., `personaldata`, `performancedata`) are represented in the merged data frame (i.e., `mergeddf`). Second, note how the cases are matched by the `id` key variable. Third, note that the `personaldata` data frame has 8 cases, the `performancedata` data frame has 6 cases, and the `mergeddf` data frame has 6 cases. By default, the `merge` function performs an *inner join* and retains only those matched cases that have data in *both* data frames. Because cases whose `id` values were `154`, `155`, and `165` had data in `personaldata` but not `performancedata` and because the case with an `id` value equal to 153 was in `performancedata` but not `personaldata`, only the 5 cases that had available data in both data frames were retained.

To perform what is referred to as a **full join** in which we retain all cases and available data, we can add the `all=` argument to our previous code and specify the logical value `TRUE`.

```
# Full join
mergeddf <- merge(personaldata, performancedata, by="id", all=TRUE)

# View the joined data frame
mergeddf
```

```
##    id   lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
## 1 111     Newton     Isaac  1/9/2016   male     3.3     3.3     3.4     3.3
## 2 125   Franklin  Benjamin  1/5/2016   male     2.1     1.9     2.1     2.3
## 3 153       <NA>      <NA>      <NA>   <NA>     3.9     4.8     4.9     5.0
```

```
## 4 154   McDonald    Ronald  1/9/2016   male      NA      NA      NA      NA
## 5 155      Smith      John  1/9/2016   male      NA      NA      NA      NA
## 6 165        Doe      Jane  1/4/2016 female      NA      NA      NA      NA
## 7 198    Morales     Linda  1/7/2016 female     4.9     4.5     4.4     4.8
## 8 201 Providence     Cindy  1/9/2016 female     1.2     1.1     1.0     1.0
## 9 282     Legend      John  1/9/2016   male     2.2     2.3     2.4     2.5
```

Note how the `full_join` function retains all available cases that had available data in at least one of the data frames, which in this example is 9 cases. *When in doubt, I recommend using the `full_join` function to retain all available data.*

To perform what is referred to as a **left join** in which we retain only those cases with data available in the first (left, x) data frame (`personaldata`), we use the `all.x=TRUE` argument instead.

```
# Left join
mergeddf <- merge(personaldata, performancedata, by="id", all.x=TRUE)

# View the joined data frame
mergeddf
```

```
##    id   lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
## 1 111     Newton     Isaac  1/9/2016   male     3.3     3.3     3.4     3.3
## 2 125   Franklin  Benjamin  1/5/2016   male     2.1     1.9     2.1     2.3
## 3 154   McDonald    Ronald  1/9/2016   male      NA      NA      NA      NA
## 4 155      Smith      John  1/9/2016   male      NA      NA      NA      NA
## 5 165        Doe      Jane  1/4/2016 female      NA      NA      NA      NA
## 6 198    Morales     Linda  1/7/2016 female     4.9     4.5     4.4     4.8
## 7 201 Providence     Cindy  1/9/2016 female     1.2     1.1     1.0     1.0
## 8 282     Legend      John  1/9/2016   male     2.2     2.3     2.4     2.5
```

Note how the *left join* retains only those cases for which the first (left, x) data frame (i.e., `personaldata`) has complete data, which in this case happens to be 8 cases. Notably absent is the case associated with `id` equal to 153 because the first (left, x) data frame (i.e., `personaldata`) lacked that case. An `NA` appears for each case from the second (right, y) data frame that contained missing values on variables from that data frame.

To perform what is referred to as a **right join** in which we retain only those cases with data available in the second (right, y) data frame (`performancedata`), we use the `all.y=TRUE` argument instead.

```
# Right join
mergeddf <- merge(personaldata, performancedata, by="id", all.y=TRUE)

# View the joined data frame
mergeddf
```

```
##    id   lastname firstname startdate gender perf_q1 perf_q2 perf_q3 perf_q4
## 1 111     Newton     Isaac  1/9/2016   male     3.3     3.3     3.4     3.3
## 2 125   Franklin  Benjamin  1/5/2016   male     2.1     1.9     2.1     2.3
## 3 153       <NA>      <NA>      <NA>   <NA>     3.9     4.8     4.9     5.0
## 4 198    Morales     Linda  1/7/2016 female     4.9     4.5     4.4     4.8
## 5 201 Providence     Cindy  1/9/2016 female     1.2     1.1     1.0     1.0
## 6 282     Legend      John  1/9/2016   male     2.2     2.3     2.4     2.5
```

Note how the *right join* retains only those cases for which the joined (second, right, y) data frame (i.e., `performancedata`) has complete data. Because the first (left, x) data frame lacks data for the case in which `id` is equal to 153, an `NA` appears for each case from the first data frame that contained missing values on variables from that data frame.

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2020). *rmarkdown: Dynamic Documents for R*. R package version 2.3.

Bache, S. M. and Wickham, H. (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5.

Bauer, T. N., Erdogan, B., Caughlin, D. E., and Truxillo, D. M. (2020). *Fundamentals of human resource management: People, data, and analytics*. Sage, Thousand Oaks, California.

Deloitte (2018). Global human capital trends report 2018.

Gerbing, D. (2020). *lessR: Less Code, More Results*. R package version 3.9.6.

Muthén, B. O. and Muthén, L. K. (1998-2018). *Mplus version 8.3*. Los Angeles, California.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

RStudio Team (2020). *RStudio: Integrated Development Environment for R*. RStudio, PBC., Boston, MA.

Wickham, H. (2019). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.3.0.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wickham, H. and Bryan, J. (2019). *readxl: Read Excel Files*. R package version 1.3.1.

Wickham, H., François, R., Henry, L., and Müller, K. (2020). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.0.

Wickham, H. and Grolemund, G. (2017). *R for data science: Visualize, model, transform, tidy, and import data*. O'Reilly Media, Inc., Sebastopol, California.

Wickham, H., Hester, J., and Francois, R. (2018). *readr: Read Rectangular Text Data*. R package version 1.3.1.

Xie, Y. (2014). knitr: A comprehensive tool for reproducible research in R. In Stodden, V., Leisch, F., and Peng, R. D., editors, *Implementing Reproducible Computational Research*. Chapman and Hall/CRC. ISBN 978-1466561595.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

Xie, Y. (2020a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.20.

Xie, Y. (2020b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.29.

Xie, Y., Allaire, J., and Grolemund, G. (2018). *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 9781138359338.