

Tabular data analysis with R and Tidyverse: Environmental Health



Jean-Yves Sgro, PhD¹ Kristen Malecki, PhD, MPH²

Last updated: 30 June, 2023

¹Biotech Center, Biochemistry Dept., jsgro@wisc.edu

²Environmental and Occupational Health Sciences, kmalecki@uic.edu

Updated 2023 from original work for:
2020 Summer Research Opportunities Program (SROP) for sophomores
University of Wisconsin-Madison
Madison WI 53706 USA

©2020-2023 Jean-Yves Sgro

CC BY-NC 4.0

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/)
“Attribution-NonCommercial-ShareAlike 3.0 Un-
ported” license.



Cover: Pixabay.com image by Bessi

List of Figures

1.1	Adding packages is like adding gears for a more powerful engine.	3
1.2	NHANES logo.	4
2.1	R objects are containers holding data, variables, tables, etc. . . .	9
2.2	R holds workspace, objects in RAM.	10
3.1	The 4 quadrants of RStudio interface.	12
3.2	Adding comments to scripts makes them easier to share with others, or your future self.	15
3.3	Files Tab shows files and path.	16
3.4	More menu provides easy navigation to working directory. . . .	17
4.1	The assignment operator <- is used to create R objects.	20
4.2	A jar as a metaphor for an R objects.	21
4.3	R objects are conveniently listed within the *Environment* Tab in RStudio.	23
4.4	A dozen often referred to eggs.	24
4.5	A function typically takes input and provides output.	26
4.6	A function is <i>always</i> written with parenthesis even if they remain empty.	26

4.7	A function has default arguments. Options and additional arguments may modify its behavior.	27
4.8	Function plot() automatically generated scatter plot.	45
4.9	Split screen plots.	46
4.10	Function plot() automatically generated boxplot.	46
5.1	Datase 'airquality' is a daily record of daily air quality measurements in New York, May to September 1973.	50
5.2	Default boxplot of airquality dataset.	57
5.3	Comparing the plot of 2 subset formats.	58
5.4	Boxplot of temperature of airquality dataset.	59
5.5	Boxplot of temperature as a function of the month of airquality dataset.	60
5.6	Boxplot of temperature as a function of the month of airquality dataset with simple colors.	61
5.7	Using levels to automatically color boxplot.	62
5.8	A scatter plot can show trend.	63
5.9	Adding month levels both as color and number plotted.	64
5.10	Adding month levels both as color and number plotted.	65
5.11	26 pch geometric symbols for plots are numbered 0 to 25. Default is number 1: open circle.	66
5.12	Adding the simple regression line on the scatter plot.	67
5.13	qplot version of Temperature vs Month.	70
5.14	Better qplot version of Temperature vs Month.	72
5.15	Scatter plot for Ozone vs Temperature.	75
5.16	Scatter plot for Ozone vs Temperature, linear regression for each month.	76

5.17	Scatter plot for Ozone vs Temperature. Linear regression for all months.	77
5.18	Scatter plot for Ozone vs Temperature. Linear regression for all months.	78
6.1	Perfluorooctanoic acid (PFOA) is used worldwide as an industrial surfactant in chemical processes and as a material feedstock, and is a health concern and subject to regulatory action and voluntary industrial phase-outs.	83
6.2	Finding NHANES 2015-2016 data.	86
6.3	Summary of 4 attempts	91
6.4	PFAS_I boxplot with log values for odd columns and rotated labels.	93
6.5	PFAS_I histogram for summed values in column 21 labeled LBXMFOS.	94
6.6	Creating multiple histograms with one command	96
6.7	PFAS_I boxplot with log values for 4 columns.	99
6.8	PFAS_I boxplot with log values for 10 data columns.	100
6.9	Combining NHANES data into a single file is necessary for detailed analysis.	101
6.10	We have combined NHANES data for each individual from 2 separate files.	104
7.1	Boxplot and histogram of log ₁₀ transformation of PFAS sum data after creatinine adjustment.	114
7.2	Histogram of BMI values and log ₁₀ values.	116
7.3	Histogram of BMI values and log ₁₀ values.	117
7.4	Histogram of BMI values and log ₁₀ values with added linear regression.	118

7.5	Stretching horizontally makes the linear regression appear more horizontal	119
7.6	Histogram of BMI log ₁₀ values, linear regression (blue) and standard error (gray.)	120
8.1	Imagining the data stream as a flow of water in pipes.	126
8.2	The pipe operator is the conduit for the data stream.	126
8.3	Is the Tibble logo a hint on Star Trek?	128
8.4	A pipeline to recreate scatter plot of BMI values as a function of log ₁₀ RATIO creatinine adjustment for the sum of PFAS data column LBXMFO5.	130
8.5	Data is first injected in the pipeline (Hydroelectric power station, Huanza, Peru.)	131
11.1	ggplot2 constructs graphs in layers using a grammar of graphics.	158
11.2	Bar plot showing total count by age group without gender distinction.	162
11.3	With facet_grid() the age distribution by gender is on two separate graphs.	163
11.4	Bar plot showing age group distribution by gender. Stack bars is the default.	164
11.5	Side by side bar of gender count by age group is possible with the dodge or dodge2 options.	166
11.6	Side by side in each facet.	167
13.1	Reproducible research is more about computer analysis, replicable research is about reproducing research results.	182
13.2	HTML output of Tiny Rmd as knitr output.	191
13.3	HTML output for Tinyest R markdown conversion with Knitr button.	192

F.1	proposed nomenclature for perfluoroalkyl and polyfluoroalkyl substances	226
-----	--	-----

List of Tables

5.1	Airquality dataset variables	50
5.2	Details of the airquality dataset readings	51
6.1	Base R read functions	80
6.2	PFAS_I codes for sum data	87
6.3	Cholesterol (Total, HDL, LDL & triglycerides) in 2015-2016 NHANES	101
7.1	NHANES 2015-2016 albumin/creatinine data	109
7.2	Codes for albumin and creatinine ALB_CR_I file	110
9.1	Lectures on data wrangling: Tidyverse tidyr and dplyr packages.	134
10.1	The DDMARTL codes from NHANES DEMO_I	147
12.1	Chosen columns and their description	172
13.1	A course on reproducible research using R	182
13.2	Basic Markdown Syntax	185
15.1	Free online images and illustrations	208
15.2	Table format conversions including Excel and markdown.	209

B.1	Arithmetic operators and their symbols in R	214
B.2	Boolean values	215
B.3	Rational operators	215
E.1	PFAS_I analysis code	223
G.1	ggplot2 tutorials online	227
G.2	Tutorials on bar graph	228
H.1	Rmarkdown tutorials online	229
H.2	Rmarkdown templates	230

Contents

Preamble	19
Learning goals	20
Software used during this tutorial	20
1 Introduction	1
1.1 Software installation	2
1.2 Installing R packages	2
1.3 Datasets: NHANES	3
1.3.1 NHANES 2015-2016	4
1.4 Datasets: included in R	5
2 How R works	7
2.1 R is a software	7
2.2 R is a language	8
2.2.1 Classic R vs Tidyverse	8
2.3 Working with R: objects and workspace	9
3 Getting started	11
3.1 Launch RStudio	11

3.2	Organize with an RStudio project	13
3.3	Creating an R script	14
3.3.1	Script Editor	14
3.3.2	Comments	14
3.3.3	Executing commands	15
3.4	Working directory	16
4	Working with R	19
4.1	Creating R objects	20
4.2	Functions and their arguments	25
4.3	Built-in functions	27
4.3.1	list: ls()	27
4.3.2	class()	27
4.3.3	combine: c()	28
4.3.4	length()	29
4.3.5	Working directory: getwd() and setwd()	29
4.4	Getting help	29
4.5	Vectorisation	30
4.6	More complex data	30
4.6.1	Vectors	31
4.6.2	Matrix	31
4.6.3	Combining vectors to create a matrix	32
4.7	Dataframes	34
4.7.1	Dataframe manipulation	35
4.8	Generating data	36

CONTENTS	13
4.8.1 Regular sequences	36
4.8.2 Repeat and sequence functions:	37
4.8.3 Levels: <code>gl()</code> and <code>expand.grid()</code>	38
4.8.4 Random numbers	40
4.9 Conditional statements	43
4.9.1 Function <code>ifelse()</code>	43
4.10 Simple graphics with <code>plot()</code>	44
5 Working with tabular data in R	49
5.1 Airquality dataset	50
5.2 Exploring airquality	51
5.3 Subsetting	54
5.4 Base R Graphics exploration	57
5.5 Boxplots	59
5.6 Scatter plots	63
5.7 Simple linear regression	66
5.8 Fancier Graphics exploration	68
5.8.1 Boxplots	69
5.8.2 Scatter plots	74
6 Importing data	79
6.1 Importing from local files	80
6.2 Downloading Nhanes data	81
6.2.1 PFAS_I	83
6.3 Exploring PFAS_I data	87

6.3.1	PFAS_I boxplot	88
6.3.2	PFAS_I histogram	92
6.3.3	Fancier boxplot with qplot	97
6.4	Merging data files	100
6.4.1	Merge() function	103
6.4.2	Merging demographics data	104
7	Creatinine adjustment	107
7.1	Creatinine data	109
7.1.1	Downloading, merging PFAS and creatinine	110
7.2	Analyte measurement units	111
7.3	Reduced set	112
7.4	Computing Analyte / Creatinine ratio	112
7.5	Exposure - Outcome	114
7.5.1	Illusions	119
7.5.2	qplot version	120
7.6	Creating a master data file	120
8	Tidyverse: another R Universe	123
8.1	Magrittr - pipe and pipelines	125
8.2	Tibble	127
8.3	dplyr - overview	128
8.3.1	Demo 1: all together pipeline	129
9	Intermission: data wrangling	133
9.1	Part 3 here	134

CONTENTS	15
10 dplyr - data manipulation	137
10.1 selecting columns	138
10.2 Filtering rows	139
10.3 Arrange data	141
10.4 mutating data	142
10.4.1 mutate with conditional statement	143
10.5 Summarising and grouping data	145
10.6 Recoding: string replacement	146
10.7 Getting it all together	148
10.7.1 Example 1: by gender	149
10.7.2 Example 2: by gender and age	150
10.7.3 Base R Bar plot	153
10.7.4 ggplot2 versions	155
11 ggplot2	157
11.1 Tutorials	159
11.2 ggplot2 using dplyr chapter results	160
11.2.1 Barplot with qplot / ggplot	160
11.2.2 Error bars and meanTChol	165
12 Using NHANES weights	169
12.1 Header comments and packages	170
12.2 Acquiring NHANES data	171
12.3 Data wrangling: renaming and selecting data	171
12.3.1 Renaming columns	171

12.3.2	Selecting columns	173
12.3.3	Changing variable status to a factor	173
12.3.4	Adding the weight information	177
12.3.5	Statistics	178
13	Markdown and Reproducible research	181
13.1	Markdown	183
13.1.1	Markdown syntax	184
13.2	R markdown magic	186
13.2.1	Before your start	187
13.2.2	How to create an R markdown file	187
13.2.3	Adding R code	189
13.2.4	Very tiny Rmd file: Inline code	191
13.3	Other formats	193
13.4	A word on YAML	194
13.4.1	Limits	194
13.4.2	Indentation and White space	194
13.4.3	Automatic modifications	195
13.4.4	Quotes	196
13.4.5	Date	196
13.4.6	YAML resources	197
14	Report-template	199
14.1	Overall template format	200
14.2	YAML example	201

CONTENTS	17
14.3 General chunk options	202
14.4 Preamble, Preface and Introduction	203
14.5 Activating packages	203
14.6 Live web links	203
14.7 Embedding graphs	203
14.8 Inline code	204
14.9 Math formula	204
14.10 Addendum	205
15 Report resources	207
15.1 Illustrations	207
15.1.1 Adding and sizing images	208
15.2 Markdown tables	209
Appendix	209
A The story of R	211
B Simple math	213
B.1 Arithmetic operators	213
B.2 Boolean values	215
B.3 Rational operators	215
B.4 Logical operators	215
C Import NHANES sample code	217

D Merge Downloads into a Master file	219
D.1 Download into R object with NHANES code	219
D.2 Combine files into Master	221
D.3 Save/Write Master file to disk	221
D.4 Alternate download to R object with haven	221
D.5 Download, save XPT files to hard drive	222
E PFAS_I codes	223
F Perfluoroalkyl and polyfluoroalkyl	225
G ggplot2 tutorials online	227
H Rmarkdown resources	229
I The Story of Vector V: an R markdown example	231
About the authors	235
Acknowledgments	241
I.1 R packages used for the book	241
I.2 Extra Icons used:	241
I.2.1 Exercise / Homework	241
I.2.2 Study at home:	242

Preamble

The course book is based on a tutorial course for the 2020 “Summer Research Opportunities Program” (SROP) for “Underrepresented Racial Minority” (URM) at the University of Wisconsin-Madison ([of the Vice Provost \(2013\)](#), and [Archived](#))

The main objective of this course is to learn how to analyze tabular datasets of environmental health data using the software **R** within the **RStudio** interface.

This course is also a preparation on *reproducible research* using *dynamic documents* for the analysis of environmental health data from the “Center for Disease Control and Prevention” (CDC) “National Center for Health Statistics” (NCHS) repository of “National Health and Nutrition Examination Survey” (NAHANES) datasets. This type of large tabular data is typical and will provide a number of useful examples.

A special distinction between “classic R” and “Tidyverse” nomenclature will be highlighted.

This course book is available online in 2 formats on link shown below as a shortened URL:



- HTML: <https://go.wisc.edu/9zu8ud>
- PDF: <https://go.wisc.edu/4zzw73>

HTML is the primary format for easier Copy/Paste interaction. PDF is easier to print or download and contains a useful **Index**.

“Environmental Health is the field of science that studies how the environment influences human health and disease.”

National Institute of Environmental Health Science [NIEHS](#)

Data and observations are usually collected in the form of numbers and gathered into tables representing the data in columns and rows.

Learning goals

During this course we'll acquire new skills:

- Install and run R and Rstudio software with additional packages
- Understand programming concepts such as *variables*, *conditional statements*, *data stream*, and *pipelines*
- Examine, compare and contrast data
- Illustrate analyzes with graphics and plots
- Compose reproducible reports that can be automated

At the end of the course you'll have acquired sufficient proficiency and independence to use the software R within the RStudio graphical interface to analyze complex environmental datasets in tabular form and create useful and reproducible reports with annotated graphics.

Software used during this tutorial

- R - from [The Comprehensive R Archive Network](#) at cran.r-project.org
- RStudio - from rstudio.com

We'll also install additional “modules” within R called “packages” to add functionality and make analysis easier.

Chapter 1

Introduction

In this class we'll use the software called R inside another software called RStudio that provides a great graphical and intuitive user interface.

R is the name of the software itself, but also the name of the *programming language* that is used within the software.

In this chapter:

- Software installation
- Installing R packages
- NHANES datasets

Just like a cooking recipe is a series of tasks to prepare a dish starting with specific ingredients, a *program* is simply a *list of instructions* to be performed and the ingredients are the data that are provided within a dataset. The instructions are written with a *programming language*, in our case R.

This 5 minutes video *R - Coding - 4.1*¹ explains how R is useful in data science.

¹<https://youtu.be/xp1l7utYFGs>

1.1 Software installation

Students should install the following two software on their computer. Both R and RStudio have versions for the three main types of computers. Once installed, working within the software is the same on all computer platforms.



TASK: *Install the software on your computer.*

- R - from [The Comprehensive R Archive Network](https://cran.r-project.org) at cran.r-project.org
- RStudio - from rstudio.com

Choose the version for your computer and follow installation instructions.

The installation process is rather intuitive. If you need more guidance the following step-by-step videos would be useful:

- [Installing R and RStudio on Windows 10](#) (March 20, 2020 - 3min 23sec)
- [Installing R and Rstudio on MacOS](#) (Mar 22, 2020 - 4min)

1.2 Installing R packages

Packages are modular additions to the R software that add functionality in the form of new functions, included datasets, documentation, etc. The standard repository of R packages *The “Comprehensive R Archive Network”* (CRAN) will likely be the most used for environmental health.

One of a “suite” of packages that we’ll use is called Tidyverse and it should preferably be installed before classes start. While Tidyverse is a suite of

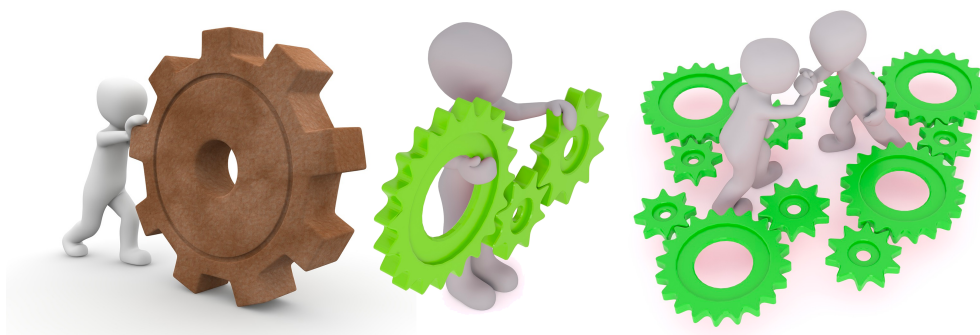


Figure 1.1: Adding packages is like adding gears for a more powerful engine.

multiple packages, this can be installed just like a single package with that single name.

The method to add a package is rather simple:

- Copy the following command in the R console: `install.packages("tidyverse")`
- Alternatively use the Packages pane in RStudio to do the installation with the graphical interface

See also section 3 to get oriented in RStudio.

To install in RStudio follow this video [Installing Packages in R Studio](#) (Nov 20, 2012 - 2.52 min) and use the package name Tidyverse instead.

To install in R follow the demonstration in the video [How to Install Packages in R](#) (Aug 9, 2013 - 6:24min)

1.3 Datasets: NHANES

Exercises in this book will be from the National Health and Nutrition Examination Survey (NHANES)² a survey research program conducted by the National

²<https://www.cdc.gov/nchs/nhanes/>

Center for Health Statistics (NCHS)³ to assess the health and nutritional status of adults and children in the United States, and to track changes over time.

An article in FAQs.org (Beals (2008)) details the history of NHANES and how the collected data is used.



Figure 1.2: NHANES logo.

1.3.1 NHANES 2015-2016

NHANES data is collected in datasets and we'll use datasets from the 2 year collection between 2015 and 2016.



IMPORTANT NOTE:

³https://en.wikipedia.org/wiki/National_Center_for_Health_Statistics

NHANES datasets are complex and in some cases the data may not be used *as is* and may require careful considerations before any conclusion is reached. Attention should be given to the existence of sub-groups. In other cases comparisons need to include sub-group weights that are included within the dataset.

See chapter 12.



NHANES file names:

The NHANES data files have succinct names, for example DEMO for demographics, with an appended *suffix* that is specific to the series. For example, 2015-2016 have the suffix `_I` and the actual file will have the root name `DEMO_I` while the demographics for other series would be different. For example the 2017-2018 series has suffix `_J` and the very first series in 1999-2000 had suffix `_A`. The pattern is therefore to go to the next letter each time a new series is published.

Video⁴: How are the data collected from the participant's point of view [NHANES Participants \(English\)](#) 2:22min

See also⁵: [The Latest Data Release and Reports from the National Health and Nutrition Examination Survey](#) May 21, 2020 - 57:29 min

1.4 Datasets: included in R

A number of small datasets are included with R during installation. We might make use of one or more.

⁴<https://youtu.be/xYBWlSGzVZM>

⁵<https://youtu.be/CXKFSdCXrFI>

There is no further installation required to access the included datasets.

Chapter 2

How R works

R is both a software and a language. There are just a few things that the user should be aware of to understand how R works that we can separate between these 2 aspects.

In this chapter:

- R is a software
- R is a language
- Working with R: objects and workspace

2.1 R is a software

R is a software that:

- should be installed on the computer prior to class (see section [1.1.](#))
- has a *basic* set of capabilities that can be enhanced by adding *packages*
- can be used as a computation engine by other software such as RStudio
- is mostly run by commands entered with the keyboard

2.2 R is a language

As a language R has capabilities to:

- use mathematical and logical operators
- create and use variables with the assignment operator `<-`
- recognizes and handles different data types including scalars, vectors (numerical, character, logical), matrices, data frames, and lists.
- can create plots and graphics
- import data from many file types, from text to databases or from the web
- use built-in functions to carry out a specified task on the data or variables.
- add new collections of R functions, data, and compiled code in a well-defined format in the form of additional packages.
- provide a comprehensive built-in help system.

2.2.1 Classic R vs Tidyverse

The way R works with commands can be confusing for multiple reasons. Statements written in the original R language now called “Classic R” can be sometimes cumbersome and lacking in clarity. The newer set of packages that make up a “Suite” called **Tidyverse** has created a new set of language format that is more modular and often easier to understand.

After reviewing basic “Classic R” we’ll also review Tidyverse methods when working on datasets later on.

2.3 Working with R: objects and workspace

The *workspace* is a *working environment* where R will store and remember *user-defined objects*: vectors, matrices, data frames, lists, functions, variables etc. At the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started.

Except for functions most of the *user-defined objects* are usually referred to as **R objects** as a way to designate them. An R object makes it easy for humans to designate its *content* which could be a single numerical value or a large table of data.

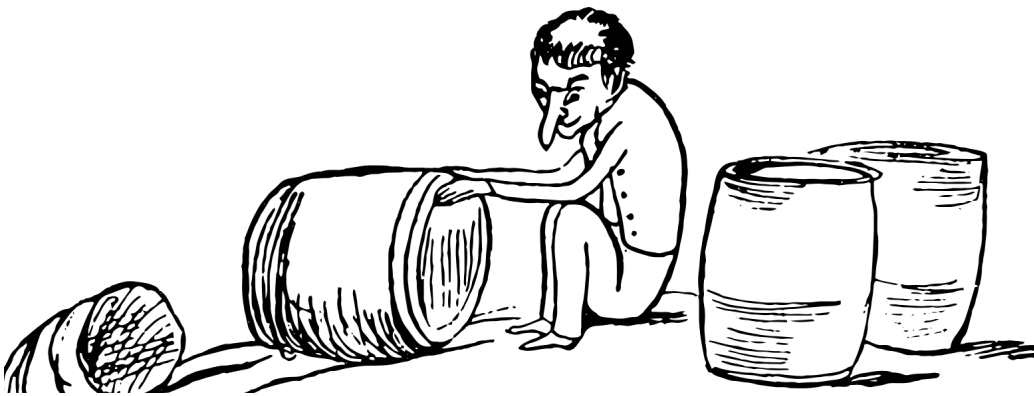


Figure 2.1: R objects are containers holding data, variables, tables, etc.

R tends to keep all data in the computer memory (RAM) which used to limit the ability to work on large datasets. However, there are now special packages and functions that can help with this aspect.

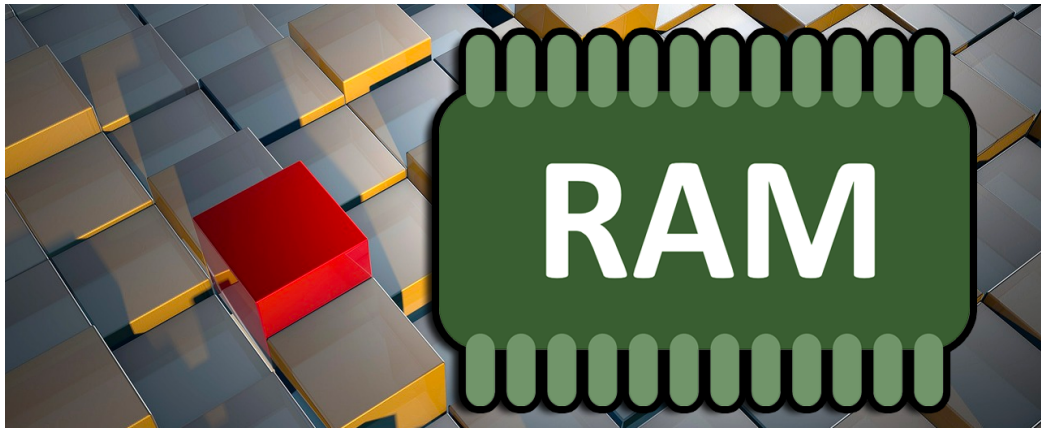


Figure 2.2: R holds workspace, objects in RAM.

Chapter 3

Getting started

We will use RStudio as a global interface to R. We'll write code, open and save files, create and visualize plots, keep track of variables and R objects all within the same RStudio window.

In this chapter:

- Launching Rstudio
- Organize with an Rstudio Project
- Create an R script
- Working directory

3.1 Launch RStudio



TASK: *Launch RStudio*

To get started launch RStudio on your computer and it will access R automatically.

Both R and RStudio should be installed prior to class (see section 1.1.)

RStudio will open with 3 sections (called *panes*) if it is the first time you are using it:

- the R Console (left),
- Workspace area with Environment/History (top-right), and
- Files/Plots/Packages/Help/Viewer (bottom-right).

When we invoke the **Source** text editor it will automatically show at the top of the R Console on the left (see figure 3.1.)

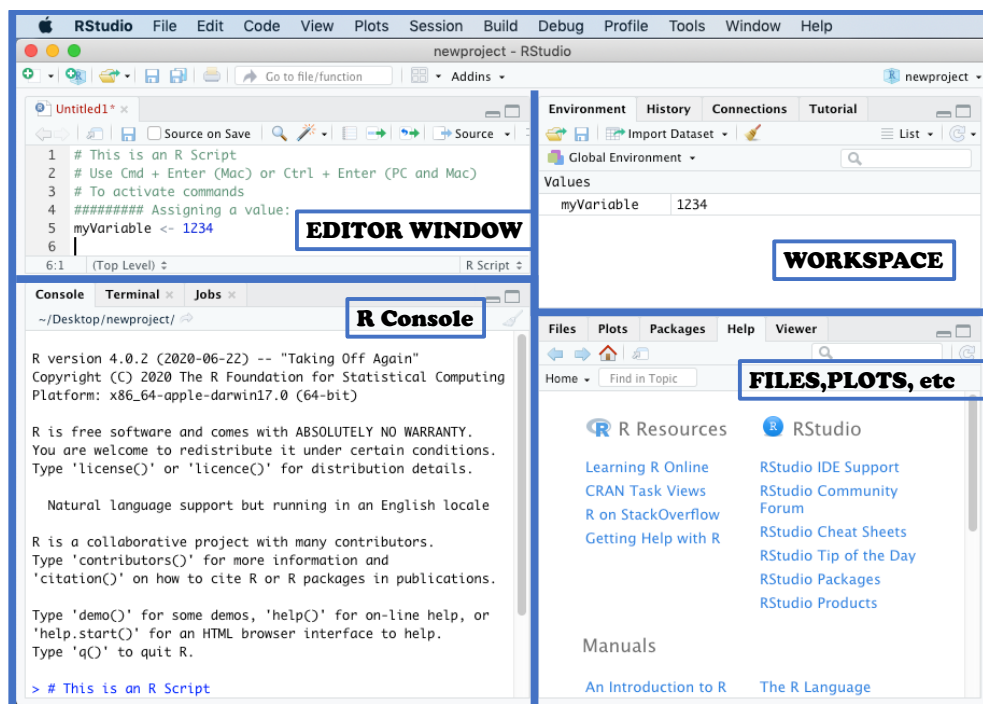


Figure 3.1: The 4 quadrants of RStudio interface.



Note: the relative position and content of each pane can be customized with the menu cascade:

Tools -> Global Options -> Pane Layout

3.2 Organize with an RStudio project

It is a good habit to immediately create a project for handling the analysis of new data and keep everything together. RStudio can create the project directory if it does not exist, or it can use a directory that has already been set-up.

Having a single top-level directory to move the project to a different drive or into another directory, or to share it with collaborators. Subdirectories can also be used to better separate various files and data.



TASK: *Create an R project*

Start with menu cascade: **File > New Project** and then choose:

- **New Directory**
- **Empty Project**
- Chose a name for the directory, for example **learn-R**
- Keep the suggestion to create the project in **~/Desktop**
- Click on **Create Project**

From there it would be possible to create sub-directories from within the RStudio interface under the **Files** tab in the bottom-right pane. Alternatively the sub-directories could also be created from the familiar computer operating system interface.

For more complex project it may be useful to create sub-directories to contain data, scripts and other documents separately. For very complex data more sub-directories could be added such as an output directory for example.

For now we'll just keep things under the same project folder.

3.3 Creating an R script

To more easily keep a record we'll create a new text file with the built-in Editor.



TASK: *Create script file*

Use the following menu cascade to create a new script:

File > New File > R Script

You should now have a blank space on the left hand side as in figure 3.1.

This is where we'll write R code and the file can then be saved to a plain text file that can be used again later. When saved the file will have a .R filename extension.

3.3.1 Script Editor

The editor is a plain text editor (no bolds or italics) but it offers color-coding of the text depending on what is written (syntax highlighting.)

3.3.2 Comments

While the code we write always makes sense *right now* it may not be obvious to others, including ourselves in the future. It is therefore extremely useful to comment

the script with information such as giving a title to sections or plain comments on the goal we are trying to accomplish.

Commenting is accomplished easily: each line with a a hashtag (#) is a comment and is ignored by R.

For example:

```
# This is a comment line  
# I can write many comments to make sure I remember what I am doing  
# The dir() function lists the content of the current directory  
dir() # comments can even be added here
```



Figure 3.2: Adding comments to scripts makes them easier to share with others, or your future self.

3.3.3 Executing commands

Commands that are written within the script can be executed by using the **Ctrl** + **Enter** shortcut (on Macs, **Cmd** + **Enter** will also work.) This will execute the

command on the current line (indicated by the cursor) or all of the commands in the currently selected text.

This action will send the selected text to the R console that will run the commands. It is therefore the same as a Copy/Paste action from the script to the console, but easier.

3.4 Working directory

We created a new RStudio project earlier and the Files Tab on the bottom right pane shows the location of files and the “path” to this directory (red circle in figure 3.3.) It is possible to navigate the whole hard drive by using the 2 dots `..` next to the “up” green arrow.

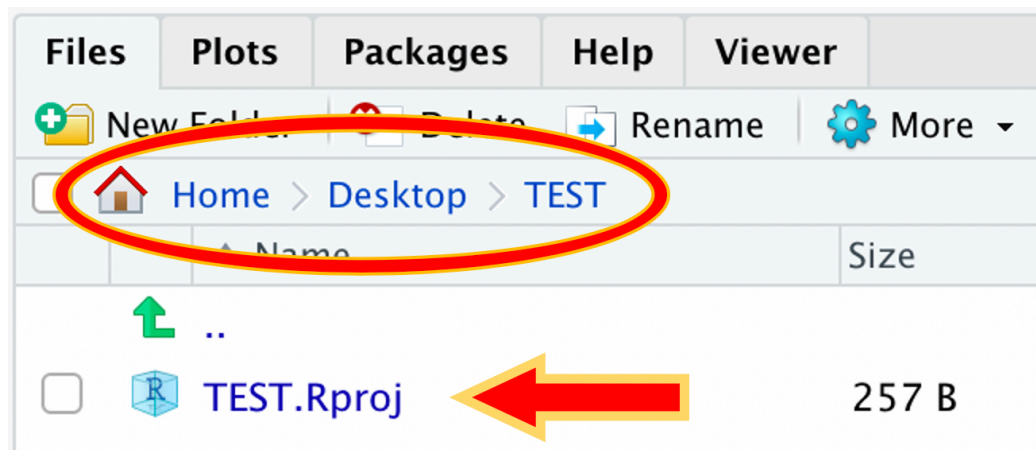


Figure 3.3: Files Tab shows files and path.

The **More** pull-down (3.4) menu makes it easy to return to the working directory if we got lost navigating the hard drive directory, to choose a new directory, or even to open the working directory on the computer graphical interface

In a section 4.3 we'll learn command functions that can let us find or change the working directory from the R console.

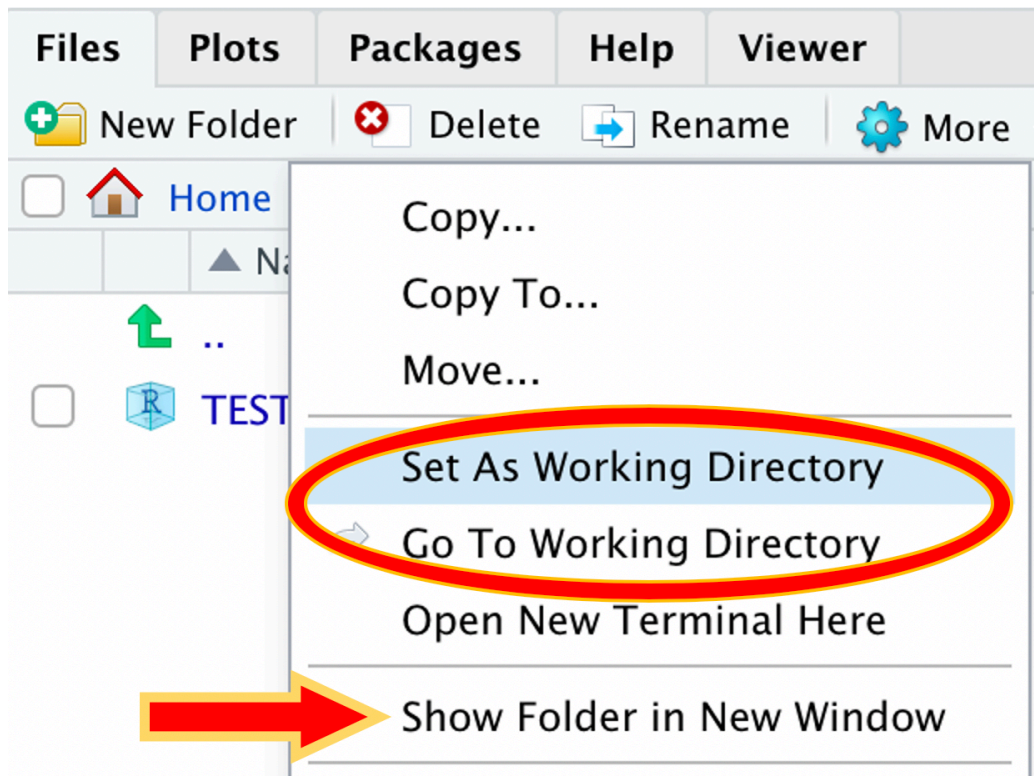


Figure 3.4: More menu provides easy navigation to working directory.

Chapter 4

Working with R

This section will be kept brief as there is a large set of introduction material online. For example this online book: “[Introduction to R](https://cengel.github.io/R-intro/)”¹. There are indeed a few principles in “Classic R” that should be understood such as creating **R objects** (section 4) and using basic R functions.

In this chapter:

- Creating user-defined R objects
- Functions and their arguments
- Vectorization
- Data frames tabular format
- Generating data
- Simple graphics with `plot()`

¹<https://cengel.github.io/R-intro/>

4.1 Creating R objects

User-created R objects are a method to handle data. It can be thought of as two actions:

- Read the data into a container, or jar
- label the jar with the content

Regardless of the size of the data (and perhaps with a little magic?) the container will adopt the required size to contain all of the data.

The user will then define a name for the container to easily call it back later.

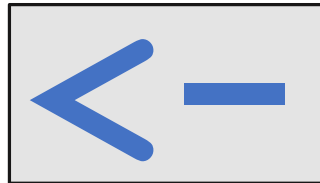


Figure 4.1: The assignment operator `<-` is used to create R objects.



NOTE

The assignment operator can be replaced with the equal sign `=` in most cases but “R purists” prefer the standard `<-` assignment code.

For a more complex discussion see [What are the differences between “=” and “<-” assignment operators in R?](#)²

Here is a simple illustration: we'll place the word `strawberry` into an jar called `jam`. In order to do the job we need to use the “assignment” symbol `<-` that could be read as “assign..” or “place into” or “read in” etc. Since `strawberry` is a word and not a number it has to be placed between quotes.

```
jam <- "strawberry"
```

we now have an R object called `jam` that contains the character string `strawberry`. In the top right panel in RStudio the new object is now listed as shown in figure 4.3.



Figure 4.2: A jar as a metaphor for an R objects.

As we just saw, characters have to be placed within quotes. The following data types occur often with routine R calculations:

- Numeric
- Integer
- Complex
- Logical

- Character

An R object can contain many types of data. It is easier to understand this with numbers. Let's make another object: we'll assign the number 12 to an object labeled dozen. Since 12 is a number we do not use quotes.

```
dozen <- 12
```

Since dozen contains and represents the number 12 we can also use mathematical operators on it. for example we can calculate how much are 2 dozens: the result is calculated by R using dozen as a *variable*.

```
# Two dozens are:  
dozen * 2
```

```
[1] 24
```

The result will be printed on the screen. Since there is only one value, the first line on the result is [1].

The choice of the label (or name) of the R object should be helpful. Here dozen is very specific and one would not want to use that label for containing any other number than 12.

A more useful name might be multiplier? perhaps not, maybe we would want to use that object to *divide*!

Let's choose a more generic label. Some people like to add **my** as part of the chosen name to make sure that they are not inadvertently using the same name as another program. for example let's use myNum to represent *my number*:

```
myNum <- 12
```

We can again make use of this object that will replace the value it contains. Here are some examples with arithmetic operators: add, subtract, multiply, divide. (See Appendix @ref=(arithmeticoperators).)

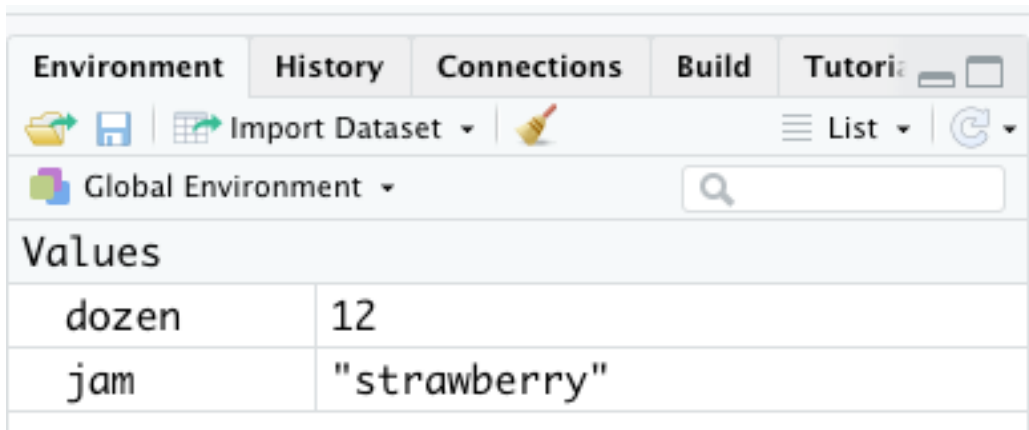


Figure 4.3: R objects are conveniently listed within the *Environment* Tab in RStudio.

```
# add:  
myNum + dozen
```

```
[1] 24
```

```
# subtract:  
myNum - dozen
```

```
[1] 0
```

```
# multiply:  
myNum * dozen
```

```
[1] 144
```

```
# divide:  
myNum / dozen
```

```
[1] 1
```

We can also ask if the two objects are “equal”, a question that can only result as TRUE or FALSE. This comparison requires using *relational* operators (see Appendix B.3.) It is noteworthy that such comparison is not limited to objects containing numbers.

```
# compare:  
myNum == dozen
```

```
[1] TRUE
```



Figure 4.4: A dozen often referred to eggs.

Exercise 4.1. Exercise: *calculate a price*

The price of one egg is 20 cents.

The price of a dozen is discounted 10%.

We want to buy 3 dozen.

How much will this cost?

Can you write the code to easily change the number of dozen purchased? or if the discount is changed later?

```
# here are some hints  
  
egg <- 0.2 # 20 cents in $
```



```
dozen <- 12
discount <- 0.10 # 10% in decimal
myNum <- 3 # how many I want now
```

Of course this could be calculated with just the numbers. But it makes computing changes easier if we use variables. Later we can change the variable assignment.

Price without discount: \$ 7.2

Discount: \$ 0.72

Discounted price = \$ 6.48



CAUTION

R objects cannot have a name that start with a number and cannot contain a dash as it is interpreted as a minus sign.

The name of an object must start with a letter (A–Z or a–z) but can include letters, digits (0–9), dots (.), and underscores (_). R is **case sensitive** and discriminates between uppercase and lowercase letters in the names of the objects, so that **a** and **A** can name two distinct objects (even under Windows).

4.2 Functions and their arguments

We just saw examples on how to use R with numbers to do some calculations. More complicated calculations, and computations, are handled with **functions** many of which are installed as part of base R installation. More functions can be added as we'll see later.

Functions perform a **task** to “accomplish something.” The “something” could be

the transformation of data, for example calculating the logarithmic value of a provided number. Most of the time the function **returns** and **output**.

Therefore one can think of a function taking an **input** and usually providing an **output**.

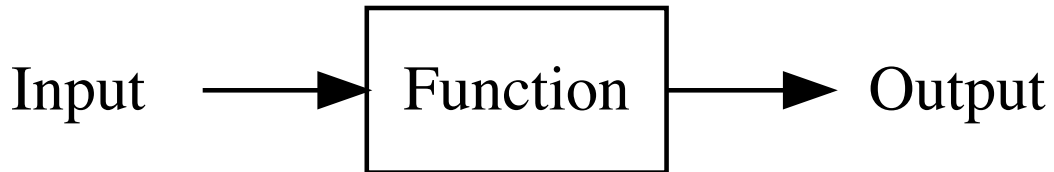


Figure 4.5: A function typically takes input and provides output.

The input is provided in the form of **argument** which can be R objects, variables, numbers, etc.

A function will typically have a default behavior that can be modified with optional arguments.

A function is always written as its name followed by parenthesis, even if these remain empty. For example the function to **list** all the R object currently within the workspace is the **list function** and it written as `ls()`.

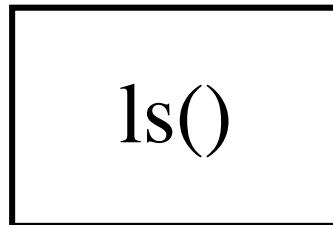


Figure 4.6: A function is *always* written with parenthesis even if they remain empty.

Most function will have a *default* behavior as determined by *default arguments*. Additional arguments and options may be added to a function to modify its behavior. The *input* is typically one of the arguments provided. Arguments can be anything expected by the function and can be numbers, filenames, but also other ob-

jects. The meaning of each required or optional *argument* may differ depending on the function and can be looked up in the documentation.

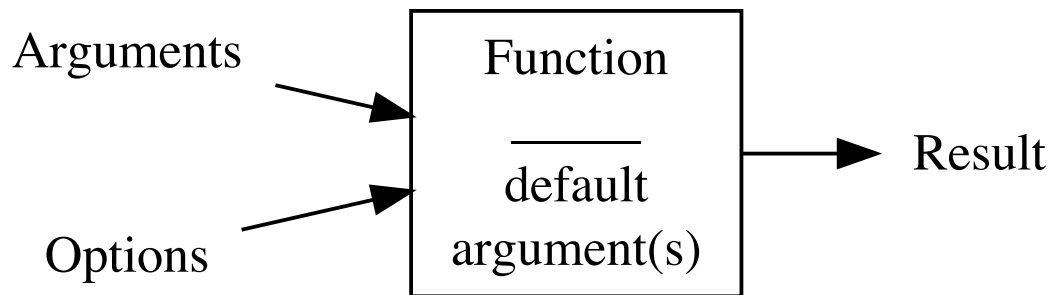


Figure 4.7: A function has default arguments. Options and additional arguments may modify its behavior.

4.3 Built-in functions

An R function is invoked by its name, then followed by parenthesis. Parenthesis contain mandatory or optional arguments to pass to the function. Parenthesis are always written even if they remain empty.

4.3.1 list: `ls()`

For example we can now *list* the R objects that we created above with the function `ls()`:

```
ls()
```

```
[1] "colorize" "discount" "dozen"    "egg"      "jam"      "myNum"
```

4.3.2 class()

We can verify the type, or class of these variables with the function `class()`

```
class(jam)
```

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

```
class(myNum)
```

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

4.3.3 combine: c()

The combine function is essential in R.

For example the following three numeric values are combined into a *vector*. (More on vectors below, section [4.6.1.](#))

```
c(1, 2, 3)
```

```
[1] 1 2 3
```

Since we did not assign to a user-defined object or a variable name the output is immediately printed out.

Here is the same vector assigned to variable `v`

```
v <- c(1, 2, 3)
```

This time no out put is produced but the data is stored in memory and can be called again.

However it is possible to obtain **both** actions at the same time: placing the assignment code within parenthesis:

```
(v <- c(1, 2, 3))
```

```
[1] 1 2 3
```

4.3.4 length()

It may be useful to know the length of an object:

```
length(v)
```

```
[1] 3
```

4.3.5 Working directory: getwd() and setwd()

In section 3.4 we saw how choose a new directory or return to it.

Function `getwd()` will *get* the *working directory* and print it on the console.

```
getwd()
```

Function `setwd()` will take as argument the *absolute* or *relative path* to the new chosen directory as defined by your operating system. Mac, Unix and Linux users use the forward slash (/) as a separator. This also works in Windows. However Windows users need to double back slashes (\\) if they use the backslash (/) as a separator. See Appendix C for sample code example that is also suited for Windows users.

4.4 Getting help

R provides extensive documentation. Depending on the installation method or how you access R the results may appear either in plain text within the R console, an HTML page, or within the Help tab on RStudio etc.

For example, entering `?c` or `help(c)` at the prompt provides documentation of the combine function `c()`.



NOTE Within `help, . . .` often means that arguments can be passed along by other functions. `index{Symbols!...}`

4.5 Vectorisation

R calculations are “vectorized” in the sense that any calculation can be applied to **all elements** of *e.g.* a vector. For example:

```
# multiply elements of vector v by 10:  
v * 10
```

```
[1] 10 20 30
```

```
# divide elements of vector v by 2:  
v / 2
```

```
[1] 0.5 1.0 1.5
```

This is a very important aspect of R.

4.6 More complex data

There exist other types of more complex data that R can handle, most of them can be tabular or multidimensional:

- Vector
- Matrix
- List
- Data Frame

Tabular data is a very common form to collect information and most useful in data analysis.

4.6.1 Vectors

We already created a one-dimensional vector `v` above containing *numeric* values. But vectors can also contain characters or logical data. However, all data in one vector have to be of the same nature.

For example here is a vector made of characters:

```
# create a vector of character  
vc <- c("a", "b", "c")
```

4.6.2 Matrix

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. All elements have to be of the same nature, *e.g.* numeric or character.

The function `matrix()` can be used to create a new matrix object.

```
matrix(c(1,2,3,4,5,6), nrow=2)
```

	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	6

However, some more information needs to be given, for example how many rows should the matrix have, this is done by the `nrow=` option. Obviously the number of elements given should be in the number of expected row by columns. The default values are `nrow = 1`, `ncol = 1` and the default filling method is by column since the default is `byrow = FALSE`.

**EXERCISE**

Try to change some of the defaults. For example change `byrow = FALSE` to `byrow = TRUE`.

Your results:

4.6.3 Combining vectors to create a matrix

Another way to create a matrix is by combining vectors of the *same length* with the functions `cbind()` or `rbind()` to combine by column or row.

**EXERCISE**

Try these commands on the vectors `v` and `vc` - for example:


```
# with v
cvv <- cbind(v,v)

rvv <- rbind(v,v)

cvvvc <- cbind(v,v,v)

# with character vector vc
vc2 <- cbind(vc,vc)

# with both v and vc
vc3 <- cbind(v,vc)
```

Your results:

What happened when using both v and vc (hint: class().)

4.7 Dataframes

Dataframes are a type of table that allows each column to contain a different variable type. For example one column can contain characters and another column can contain numbers.

This type of tabular data is extremely useful in data analysis.

We can use the function `data.frame()` to construct a dataframe starting with and combining vectors.

```
# num: a vector of numbers
num <- c(2, 3, 5)

# let: a vector of letters
let <- c("aa", "bb", "cc")

# tf: a vector of logicals true or false
tf <- c(TRUE, FALSE, TRUE)

# df is a data frame
df = data.frame(num, let, tf)
```

We can inquire about `df`: the class of the object, its dimensions, the name of the headers for the columns.

```
class(df)
```

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

```
dim(df)
```

```
[1] 3 3
```

```
names(df)
```

```
[1] "num" "let" "tf"
```

4.7.1 Dataframe manipulation

As just as simple demonstration we'll change the name of the rows.

For now the dataframe looks like this:

```
df
```

```
  num let  tf
1   2  aa TRUE
2   3  bb FALSE
3   5  cc TRUE
```

and if we ask the name of each row we get the current list:

```
rownames(df)
```

```
[1] "1" "2" "3"
```

In R things can change by reassigning new values, so we can indeed change the row names with the function `rownames()` and giving new values. For example:

```
row.names(df) <- c("row1", "row2", "row3")
```

```
# print df
```

```
df
```

	num	let	tf
row1	2	aa	TRUE
row2	3	bb	FALSE
row3	5	cc	TRUE

In the same way we could change the column names:

```
colnames(df) <- c("numbers", "letters", "logical")
```

Note: functions `row.names` and `rownames` exist for rows, but only `colnames` exist for columns.

In this final version the data itself is not altered but we changed both the column and row names:

```
df
```

	numbers	letters	logical
row1	2	aa	TRUE
row2	3	bb	FALSE
row3	5	cc	TRUE

4.8 Generating data

There are many ways to generate data from within R as series of numbers, in sequence or as random numbers. This section is purposefully kept simple.

4.8.1 Regular sequences

The generation of numbers in sequence can be useful to create lists.

The following command will generate an object with 10 elements; a regular sequence of integers ranging from 1 to 10, saved within variable `x` thanks to the operator `:` :

```
x <- 1:10  
x
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

Various options can be used to alter the results, for example requesting 11 values, starting with 3 and ending at 5.

```
seq(length=11, from=3, to=5)
```

```
[1] 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0
```

4.8.2 Repeat and sequence functions:

It may be useful to print a number multiple times. This can be done with the `rep()` function. For example:

```
rep(1,15)
```

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

The function `sequence()` creates a series of sequences of integers each ending by the numbers given as arguments.

```
sequence(2:5)
```

```
[1] 1 2 1 2 3 1 2 3 4 1 2 3 4 5
```

For clarity here is the result with * separators added:

```
[1] 1 2 *1 2 3* 1 2 3 4 *1 2 3 4 5*
```

To understand this output it is useful to also remember that `2:5` means **2, 3, 4, 5** and that the function will apply to each of these digits in turn.

4.8.3 Levels: `gl()` and `expand.grid()`

These two functions are very useful for creating tables containing experimental data.

The function `gl()` generates “levels” series of “factors” or “categories” as values or labels. The following example will generate 4 each of 2 levels:

```
gl(2, 4, labels = c("Control", "Treat"))
```

```
[1] Control Control Control Control Treat Treat Treat Treat
Levels: Control Treat
```

The function `expand.grid()` creates a data frame with *all possible combinations* of vectors or factors given as arguments.

This example

```
expand.grid(h=c(60,80), w=c(100, 300), sex=c("Male", "Female"))
```

	h	w	sex
1	60	100	Male
2	80	100	Male
3	60	300	Male

```
4 80 300 Male
5 60 100 Female
6 80 100 Female
7 60 300 Female
8 80 300 Female
```

Note: The arguments are rotated as a function of their position in the command.

**EXERCISE**

Try the following:

```
expand.grid(sex=c("Male", "Female"), h=c(60,80), w=c(100, 300))
```

How many lines is the table (not counting the header? (hint: row numbers))

The use of `seq()` can also be useful in this context.

**EXERCISE**

Try the following examples.

```
expand.grid(sex=c("Male", "Female"), h=c(60,80), w=c(100, 300))
```

```
expand.grid(height = seq(3, 3, 5),
             weight = seq(100, 250, 50),
             sex = c("Male", "Female"))
```

How many lines is the table (not counting the header? (hint: row numbers)

Add one more variable `treatment = c("control", "drug")` and see how much the table expands:

```
expand.grid(height = seq(3, 3, 5),  
            weight = seq(100, 250, 50),  
            sex = c("Male", "Female"))
```

How many lines is the table (not counting the header? (hint: row numbers)

Note: the function `dim()` can be applied directly as well, for example:

```
dim(expand.grid(sex=c("Male", "Female"),  
               h=c(60,80),  
               w=c(100, 300)))
```

4.8.4 Random numbers

Most of the statistical functions are available within R such as Gaussian (Normal), Poisson, Student *t*-test etc.

To generate random numbers, the function based on the Normal distribution we use the function `rnorm()` (r for random and norm for Normal.) The number of desired random numbers is given as argument.

Since these are random, *the answers are never the same!*

**EXERCISE**

Perform the following command requesting a single random number a few times (*e.g.* 5 times) in a row:

```
rmnorm(1)
```

Do you get the same result every time?

☐ Yes

☐ No

To provide means of reproducible the function `set.seed()` can be used to obtain the same result every time. The seed is a number chosen by the author. Here is an example selecting three numbers.

```
set.seed(33); rmnorm(3)
```

```
[1] -0.13592452 -0.04079697 1.01053901
```

```
set.seed(33); rmnorm(3)
```

```
[1] -0.13592452 -0.04079697 1.01053901
```

```
set.seed(33); rmnorm(3)
```

```
[1] -0.13592452 -0.04079697 1.01053901
```

However, changing the seed value will change the results:

```
set.seed(22); rnorm(3)
```

```
[1] -0.5121391  2.4851837  1.0078262
```



*Important note*³ “[these] Pseudo Random Number Generators because they are in fact *fully algorithmic*: given the same seed, you get the same sequence. And that is a *feature* and not a bug.”

One R method for choosing letters at random is with the function `sample()`. The term `LETTERS` represents the alphabet and is built-in R.

```
sample(LETTERS, 5)
```

```
[1] "Q" "E" "K" "C" "P"
```

```
sample(LETTERS, 5)
```

```
[1] "T" "P" "H" "Z" "A"
```

In the same way as before setting a seed will reproduce the same result every time.

```
set.seed(42); sample(LETTERS, 5)
```

```
[1] "Q" "E" "A" "J" "D"
```

```
set.seed(42); sample(LETTERS, 5)
```

```
[1] "Q" "E" "A" "J" "D"
```

4.9 Conditional statements

Making choices or decisions are what conditional statements are all about in programming.

There are multiple ways of writing a conditional statement in R using different functions

4.9.1 Function `ifelse()`

Function `ifelse()` has the same functionality as the IF statement in Excel and required 3 arguments:

1. a logical test that is either TRUE or FALSE
2. an answer if the logical test is TRUE
3. and alternate answer if the logical test is FALSE

This is best understood by an example:

```
# Logical test is TRUE: print first option  
ifelse(5 > 4, "YES! 5 is greater than 4", "NO! 5 is not smaller than 4")
```

```
[1] "YES! 5 is greater than 4"
```

```
# Logical test is FALSE: print second option  
ifelse(5 <= 4, "YES! 5 is greater than 4", "NO! 5 is not smaller than 4")
```

```
[1] "NO! 5 is not smaller than 4"
```

This will be revisited later in the Tidyverse section (10.4.1.)

Other conditional statements can be learned elsewhere. For example:

- [MODULE 4.5 Conditional Statements in R](#) (Utah Sate Univ.)⁴
- [Conditional statements and loops in R](#)⁵.

4.10 Simple graphics with plot()

We will create a very simple graphic output from generated random numbers:

Create a data vector of 100 random numbers (note: if you choose the same seed number your final plot will be identical.)

```
set.seed(9)
data <- rnorm(100)
```

The `plot()` function will create a simple scatter plot with circles as the default symbol.

```
plot(data)
```

It is possible to include more than one plot on the same figure/page with the parameter function modifying the number of rows and columns planned for plotting: `par(mfrow=c(1,1))` by default.

As a brief example we'll replot these data points as points, lines, both, and overlay. The labels for the axes are rendered blank to make the final layout less cluttered.

⁴http://learnr.usu.edu/base_r/data_manipulation/4_5_conditionals.php

⁵<https://youtu.be/2evtsnPaoDg>

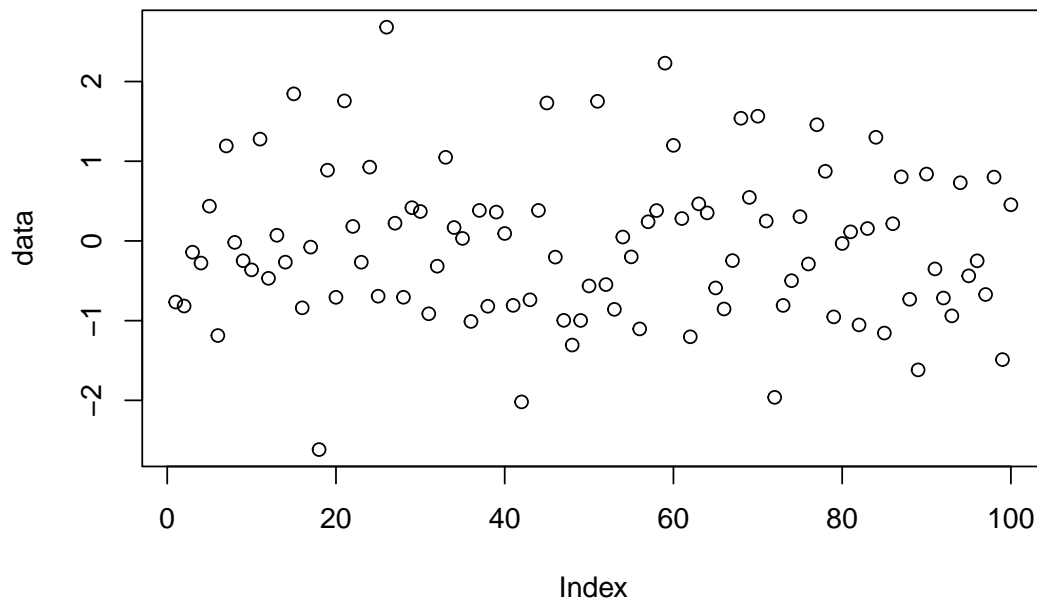


Figure 4.8: Function `plot()` automatically generated scatter plot.

```
par(mfrow = c(2,2))

plot(data, type = "p", main = "points", ylab = "", xlab = "")
plot(data, type = "l", main = "lines", ylab = "", xlab = "")
plot(data, type = "b", main = "both", ylab = "", xlab = "")
plot(data, type = "o", main = "both overplot", ylab = "", xlab = "")
```

Afterwards it is useful to reset the number of plots per page to 1:

```
par(mfrow = c(1,1))
```

Other types of default plots are available. For example a box plot.

```
boxplot(data)
```

R default graphics are useful for exploring the data. However, more modern additional packages can be added to make plots more appealing while at the same

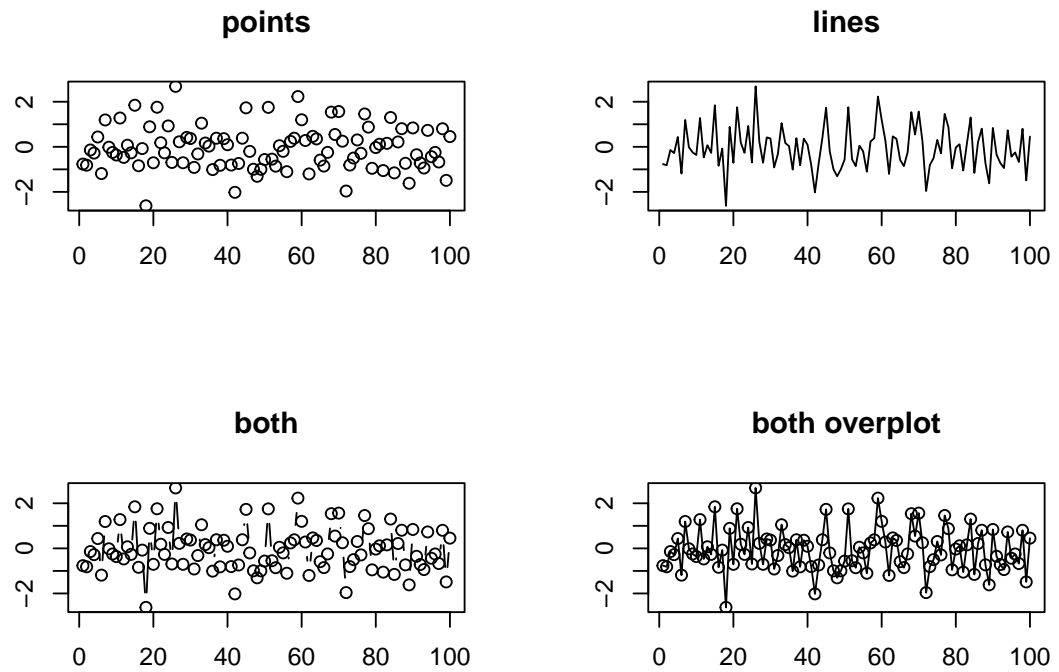
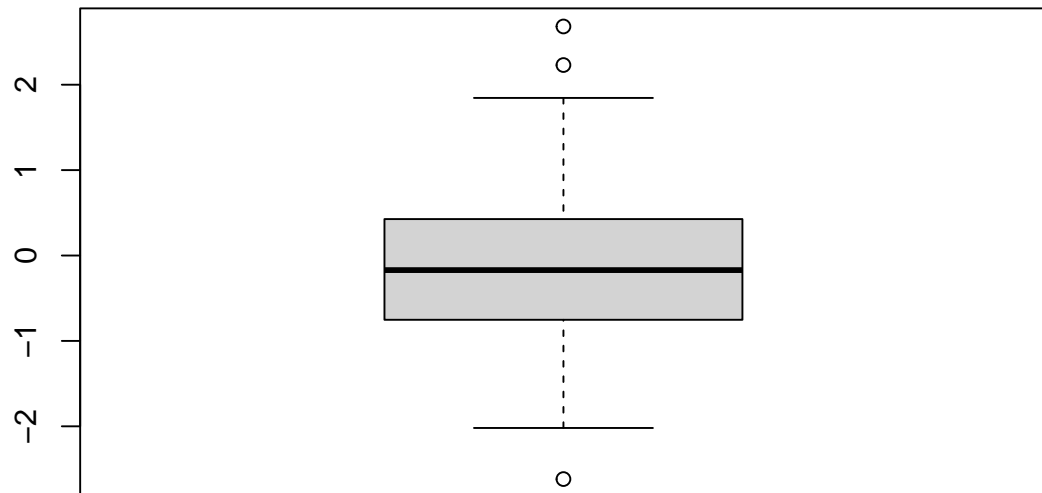


Figure 4.9: Split screen plots.

Figure 4.10: Function `plot()` automatically generated boxplot.

time trying to make it easier to create them.

Chapter 5

Working with tabular data in R

Before working with your own data, it helps to get a sense of how R works with tabular data from a built-in R data set. We'll use the data set `airquality` to do this exploration. Along the way we'll learn simple functions or methods that help explore the data or extract subsets of data.

In this chapter:

- `airquality` dataset
- Learning base R commands while exploring `airquality`
- Graphical exploration: boxplot and histogram



NOTE

You can search through the pre-installed data sets with the function `data()`.

5.1 Airquality dataset

The `airquality` dataset is built-in R so there is nothing to install or prepare, it is already there as an R object. This data is small compared to environmental data sets.

We can learn more about the dataset with the `help(airquality)` command and we'll learn that it is *Daily air quality measurements in New York, May to September 1973* stored within a *A data frame with 153 observations on 6 variables*. The source of the data: *obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data)* and cited by [Chambers et al. \(1985\)](#).



Figure 5.1: Dataset ‘airquality’ is a daily record of daily air quality measurements in New York, May to September 1973.

Table 5.1: Airquality dataset variables

Column	Name	Type	Details
[,1]	Ozone	numeric	Ozone (ppb)
[,2]	Solar.R	numeric	Solar R (lang)
[,3]	Wind	numeric	Wind (mph)
[,4]	Temp	numeric	Temperature (degrees F)
[,5]	Month	numeric	Month (1–12)
[,6]	Day	numeric	Day of month (1–31)

The values are *daily readings of the air quality values for May 1, 1973 (a Tuesday) to September 30, 1973*.

Table 5.2: Details of the `airquality` dataset readings

Details: Daily readings

Ozone: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island

Solar.R: Solar radiation in Langley's in the frequency band 4000–7700

Angstroms from 0800 to 1200 hours at Central Park

Wind: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport

Temp: Maximum daily temperature in degrees Fahrenheit at La Guardia Airport.

5.2 Exploring airquality



Base R

This section uses the default R installation. This is sometimes called “base R” and the code may be referred to as “Classic R” as compared to more modern methods that we’ll explore later.

We can look at the first and last few lines of that `airquality` tabular data. We already know that column names but we can list them with:

```
colnames(airquality)
```

```
[1] "Ozone" "Solar.R" "Wind" "Temp" "Month" "Day"
```

Using functions `head()` and `tail()` we can show the default of 6 lines of data presented with the column headers:

```
head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

Both commands can be easily modified to select the desired number of lines:

```
tail(airquality, 4)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
150	NA	145	13.2	77	9	27
151	14	191	14.3	75	9	28
152	18	131	8.0	76	9	29
153	20	223	11.5	68	9	30

In both cases we see that some data is missing, as represented by NA. It is often important to know about missing data and many functions provide default and optional arguments to deal with that.

We can use the function `colSums()` to easily report the existence and number of NA for each column:

```
colSums(is.na(airquality))
```

Ozone	Solar.R	Wind	Temp	Month	Day
37	7	0	0	0	0

We can get an idea of the size of the table with the function that prints its dimensions:

```
dim(airquality)
```

```
[1] 153  6
```

Interestingly the length is the number of columns:

```
length(airquality)
```

```
[1] 6
```

We can also check the *structure* of the dataset with:

```
str(airquality)
```

```
'data.frame':  153 obs. of  6 variables:
 $ Ozone   : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind    : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp    : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month   : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day     : int  1 2 3 4 5 6 7 8 9 10 ...
```

This provides insight telling us that `airquality` is a of class `data.frame`, the number of observation, the number of variables, and further details about each variable and the first 10 values in each column.

The `summary()` function provides a standard statistical output for each column:

```
summary(airquality)
```

Ozone	Solar.R	Wind	Temp
Min. : 1.00	Min. : 7.0	Min. : 1.700	Min. : 56.00
1st Qu.: 18.00	1st Qu.: 115.8	1st Qu.: 7.400	1st Qu.: 72.00
Median : 31.50	Median : 205.0	Median : 9.700	Median : 79.00
Mean : 42.13	Mean : 185.9	Mean : 9.958	Mean : 77.88
3rd Qu.: 63.25	3rd Qu.: 258.8	3rd Qu.: 11.500	3rd Qu.: 85.00
Max. : 168.00	Max. : 334.0	Max. : 20.700	Max. : 97.00
NA's : 37	NA's : 7		
Month	Day		
Min. : 5.000	Min. : 1.0		
1st Qu.: 6.000	1st Qu.: 8.0		
Median : 7.000	Median : 16.0		
Mean : 6.993	Mean : 15.8		
3rd Qu.: 8.000	3rd Qu.: 23.0		
Max. : 9.000	Max. : 31.0		

For each variable (*i.e.* each column) this provides the *minimum* and *maximum* value, the *mean*, the *median*. The quartile values divide the number of data points into four more or less equal parts, or quarters.

5.3 Subsetting

It is often desirable to access only some portion of the data. Hence there are ways to select just some columns or rows with the *square bracket* `[]` subsetting method.

The first number in the brackets represents the choice of column(s). If there is a second number after a comma, that number represents the choice for row(s). Omitting a number means that we want the whole. Here are useful examples adapted from “[Introduction to R](https://cengel.github.io/R-intro/)”¹.

¹<https://cengel.github.io/R-intro/>

**SUBSETTING**

Take the time to explore the following commands:

```
airquality[]           # the whole data frame (as a data.frame)
airquality[1, 1]       # first element in the first column (as a vector)
airquality[1, 6]       # first element in the 6th column (as a vector)
airquality[, 1]        # first column in the data frame (as a vector)
airquality[1]          # first column in the data frame (as a data.frame)
airquality[1:3, 3]     # first three elements in the 3rd column (as a vector)
airquality[3, ]        # the 3rd row (as a data.frame)
airquality[1:6, ]      # the 1st to 6th rows, equivalent to head(airquality)
airquality[c(1,4), ]   # rows 1 and 4 only (as a data.frame)
airquality[c(1,4), c(1,3)] # rows 1 and 4 and columns 1 and 3 (as a data.frame)
airquality[, -1]       # the whole data frame, excluding the first column
airquality[-c(3:153),] # equivalent to head(airquality, 2)
```

Here is an example using this method to compute the average temperature (variable Temp) in the 4th column by giving the subset as an argument to the `mean()` function:

```
mean(airquality[, 4])
```

```
[1] 77.88235
```

This notation is useful and does the job. The command could be understood as the English phrase: “take the mean of all the values located in the 4th column of the *airquality* dataset.”

Another subsetting method typical in R is to use the name of the object and the name of the column separated by a \$ sign. For example the column for tempera-

ture would be designated as `airquality$Temp`. So we could also use that notation to compute. This time let's calculate the median:

```
median(airquality$Temp)
```

```
[1] 79
```

Here is another example calling for the summary of just one column, here the Ozone column.

```
summary(airquality$Ozone)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
1.00	18.00	31.50	42.13	63.25	168.00	37

However, it may easier to work with the `with()` function that allows to simply use the column name:

```
with(airquality, mean(Temp))
```

```
[1] 77.88235
```

This command could be spoken in English as “*working with the dataset `airquality` calculate the average of the values in the column labeled `Temp`.*”

**NOTE:**

The more modern methods for working with tabular data is to use the Tidyverse package `dplyr` as will be explored later.(Section 8.)

5.4 Base R Graphics exploration

R provides useful default plotting mechanisms that are useful to explore the data the most rapidly. Other packages can later be used to make the plots prettier.

Most R graphics functions will have defaults that help provide the most meaningful plot. For example we can ask for a boxplot:

```
boxplot(airquality)
```

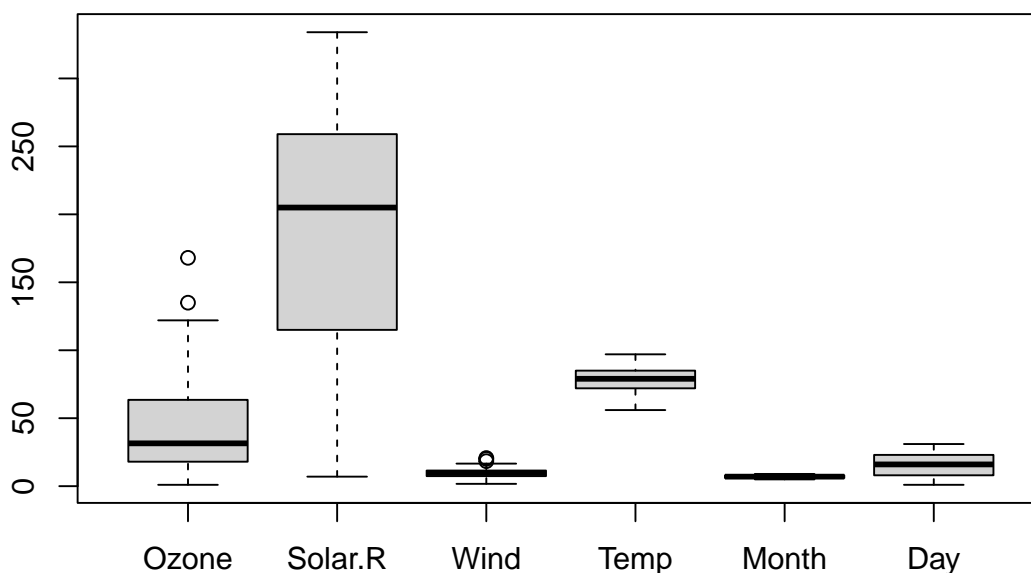


Figure 5.2: Default boxplot of airquality dataset.

The result is alright but it is clear to see that the scale has been chosen to plot the largest values which are from the `Solar.R` column, therefore “crushing” the other, smaller values.

Let’s compare the results of plotting the temperature from column 4 with the two subsetting methods we just learned. For this we’ll split the graphical page to 1 row and 2 columns first, and then issue the plotting commands:

```
par(mfrow = c(1,2))  
hist(airquality[,4])  
with(airquality, hist(Temp))
```

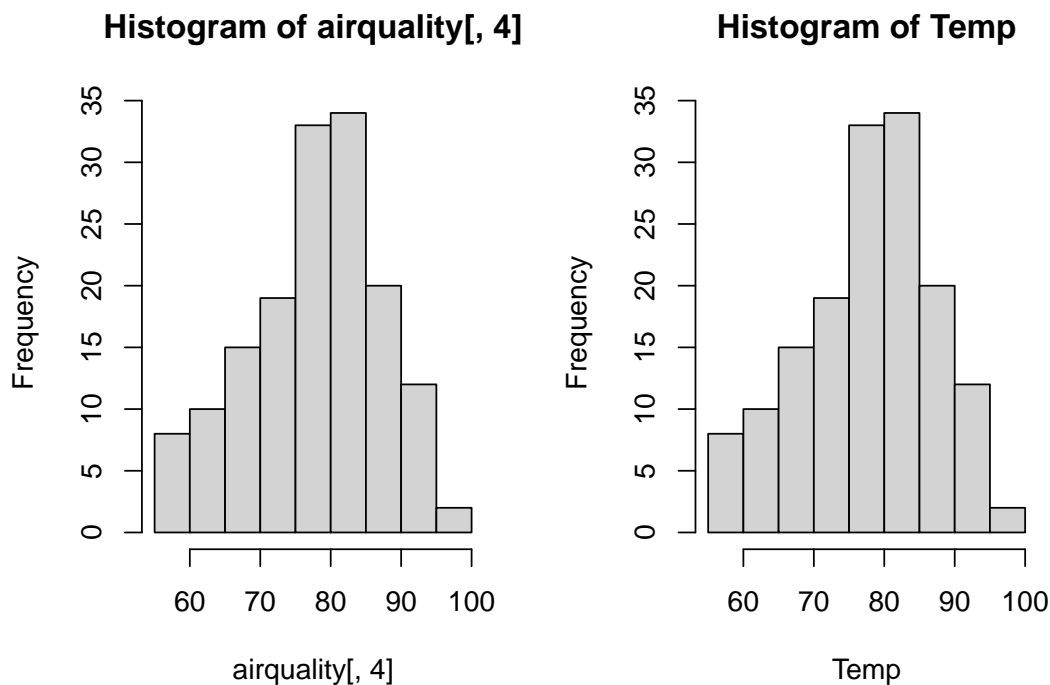


Figure 5.3: Comparing the plot of 2 subset formats.

```
par(mfrow = c(1,1))
```

We can note that the title of the plot and the name for the horizontal axis reflect what is written within the `hist()` function. This is just a default. There are ways to change what is written there as detailed in the help.

Now we may rather want to see a boxplot for the temperature.

```
with(airquality, boxplot(Temp))
```

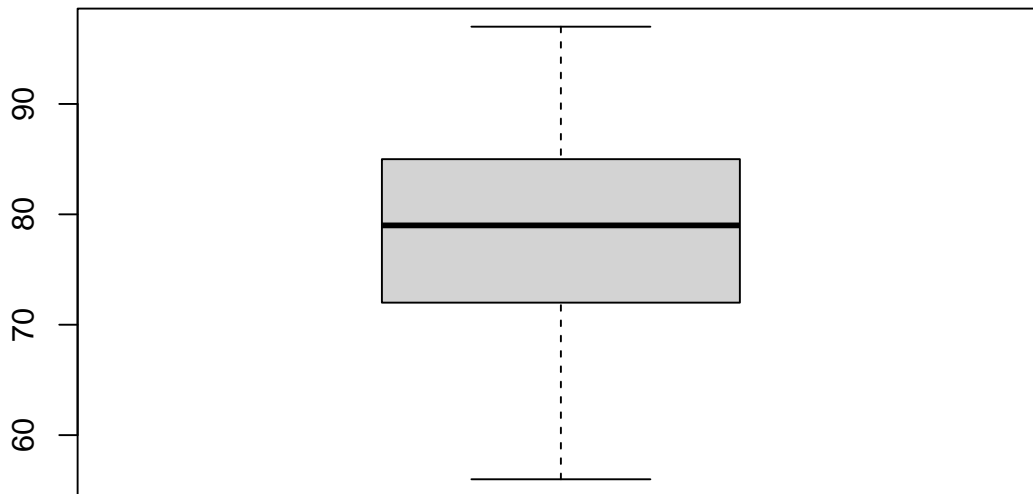


Figure 5.4: Boxplot of temperature of airquality dataset.

This is not super informative, and simply is a larger version of just the Temp values seen in figure 5.2.

5.5 Boxplots

It would be more interesting to plot the temperature separately for each month. This is possible by adding one more term that specifies that we want to “plot temperature as a function of the month.” This is accomplished with the *tilde* symbol ~ between the two variables that can be read in English with the phrase “as a function of.”

```
with(airquality, boxplot(Temp ~ Month))
```

It would be possible to add a color, choosing from the default 9 colors in R that are numbered 0-8. 0 is the default white. The next colors have also a name that can be printed by the `palette()` function:

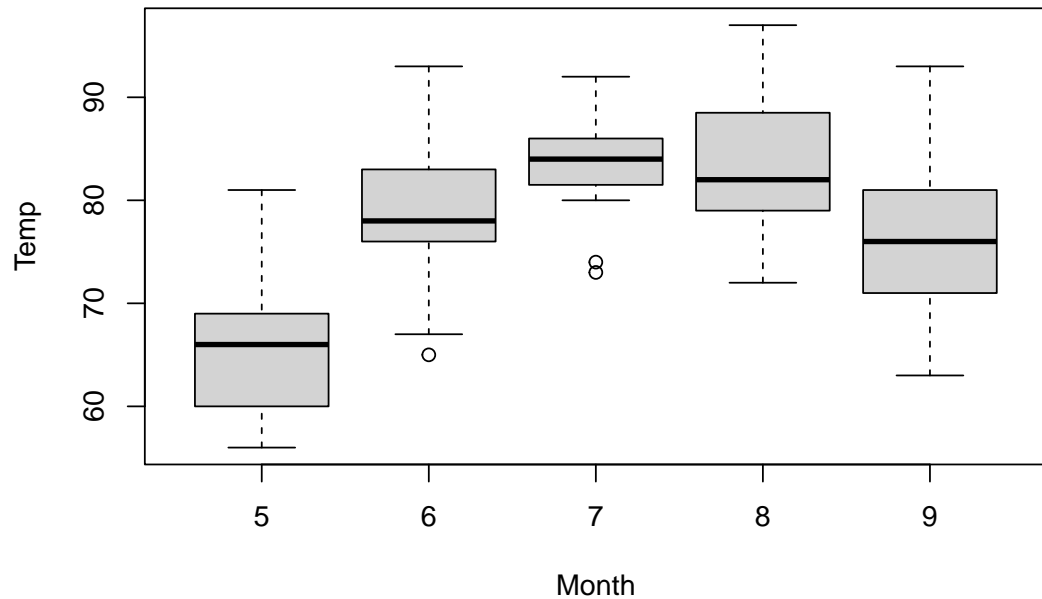


Figure 5.5: Boxplot of temperature as a function of the month of airquality dataset.

```
palette()
```

```
[1] "black"    "#DF536B" "#61D04F" "#2297E6" "#28E2E5" "#CD0BBC" "#F5C710"
[8] "gray62"
```

Therefore we could color the boxes individually by simply specifying a vector of number as we learned to do with the combine `c()` function:

```
with(airquality, boxplot(Temp ~ Month, col = c(1,2,3,4,5)))
```

This can help to understand the notion of **factor**, used for *categorical* variable stored it as *levels*. We can force R to consider a variable as `.factor` and that will also list the different *levels* of that factor.

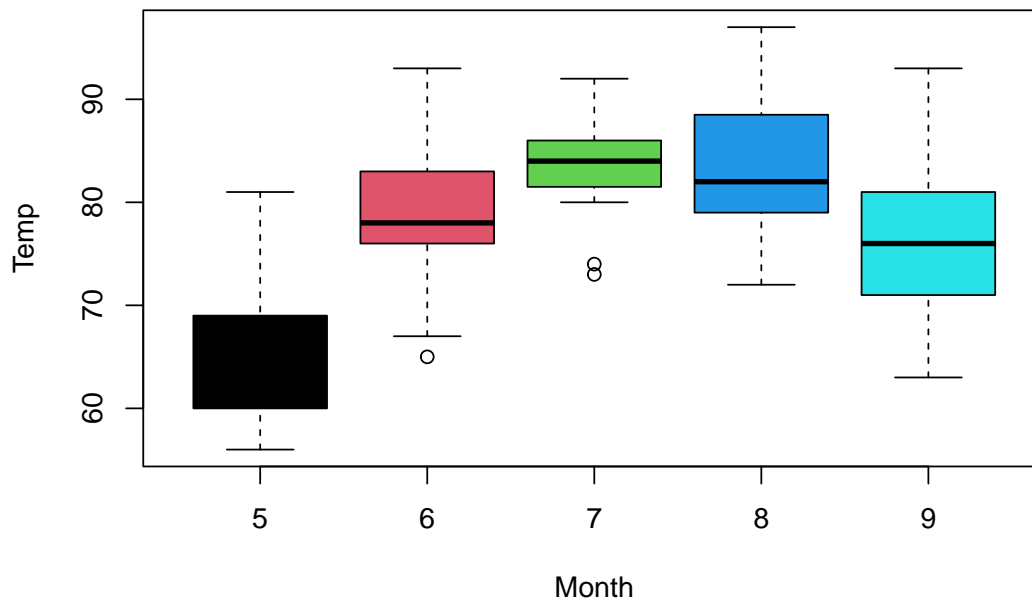


Figure 5.6: Boxplot of temperature as a function of the month of airquality dataset with simple colors.

```
with(airquality, as.factor(Month))
```

[illegible]

We can the refine the command to ask for just the levels:

```
levels(with(airquality, as.factor(Month)))
```

```
[1] "5" "6" "7" "8" "9"
```

With this knowledge we could now color the boxplot without having to type specific colors, or know how many to use by specifying that we want to color by level:

```
with(airquality,
      boxplot(Temp ~ Month,
              col = levels(with(airquality,
                                as.factor(Month))))))
```

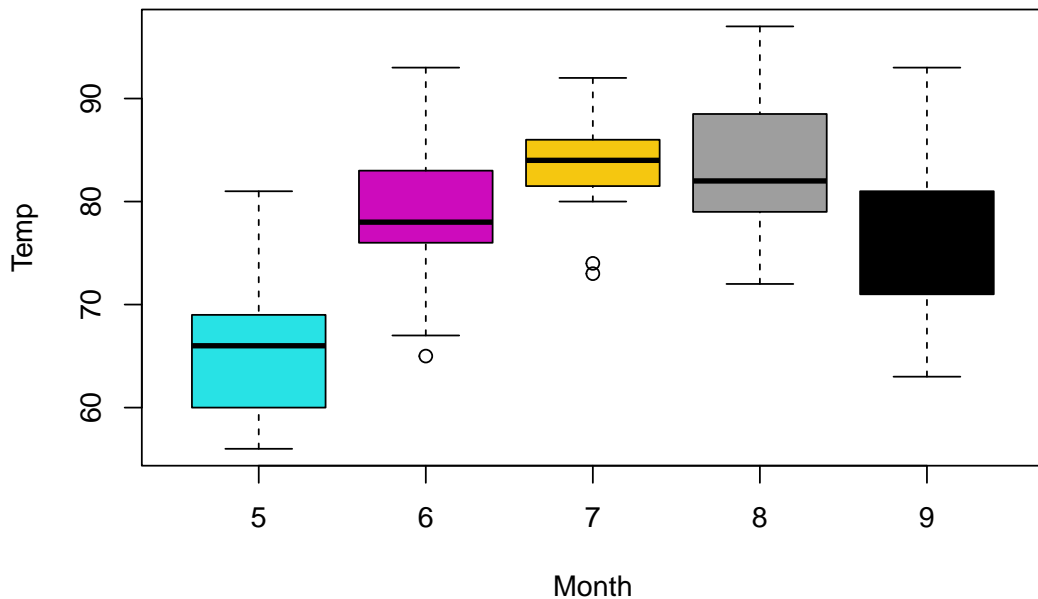


Figure 5.7: Using levels to automatically color boxplot.

Since the levels are 5, 6, 7, 8, 9 the colors of this plot are different than the plot in figure 5.6. Of course this command is not easy to understand as it is. It is usual to create intermediate variables to make the code easier to read. For example we could create a variable called `MyCol` to contain the levels.

5.6 Scatter plots

Another type of useful plot is a *scatter plot* where “points” with an “*x*” and a “*y*” coordinates are plotted. For example we could plot the Ozone levels *as a function* of the temperature Temp. This can be written using the `with()` function:

```
with(airquality, plot(Ozone ~ Temp))
```

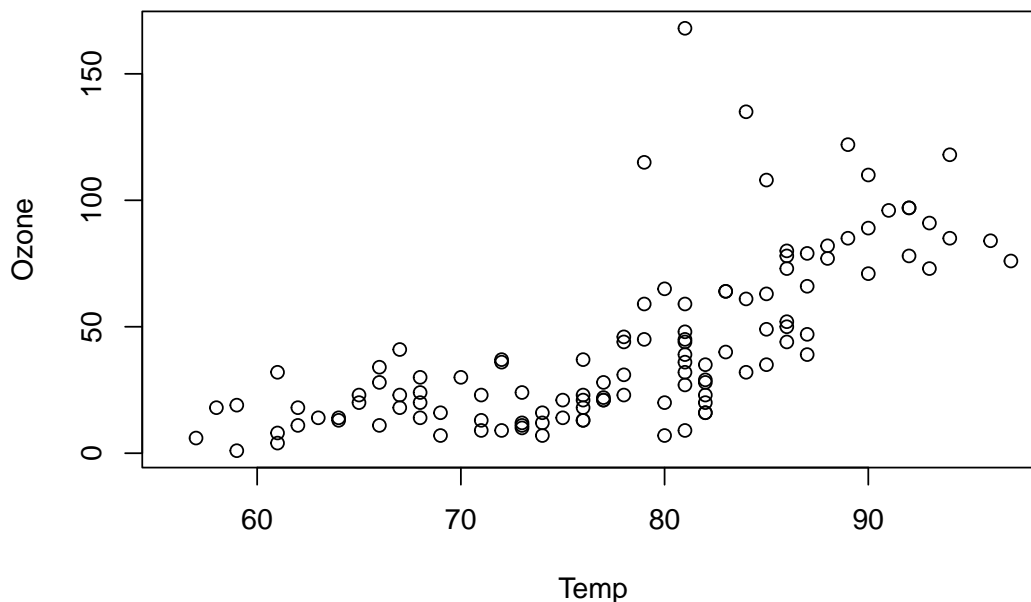


Figure 5.8: A scatter plot can show trend.

As we did with the boxplot we could also color each circle as a function of the month. We could also change the circle with another geometrical form also based on the month. Typically, to avoid “crowded” commands with too many things going on, it is best to decompose the options on separate commands.

All we need to do is assign the levels of the months into a separate variable or a user-defined R object we can call `mlev` for “month levels” for example:

```
mlev <- levels(with(airquality, as.factor(Month)))
```

This command *extracts* the level values but the `mlev` is of class character and contains 5, 6, 7, 8, 9 which are just the numbers shown as characters.

We have seen that for the `plot()` function the color option is called `col`. For the shape option it is called `pch` which stands for **p**rint **c**haracter. We can use those values to change both the color and the character to be displayed:

```
with(airquality, plot(Ozone ~ Temp,
  pch = mlev,
  col = mlev))
```

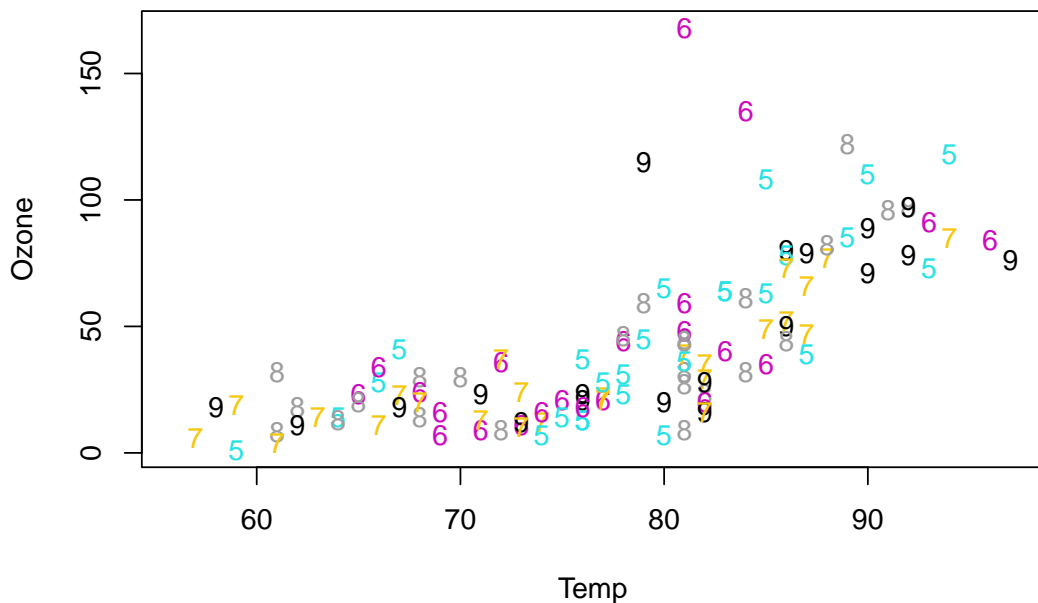


Figure 5.9: Adding month levels both as color and number plotted.

To make use the values within `mlev` to change the geometric shape we can also force them as a numeric value:


```
with(airquality, plot(Ozone ~ Temp,
                     pch = as.numeric(mlev),
                     col = mlev))
```

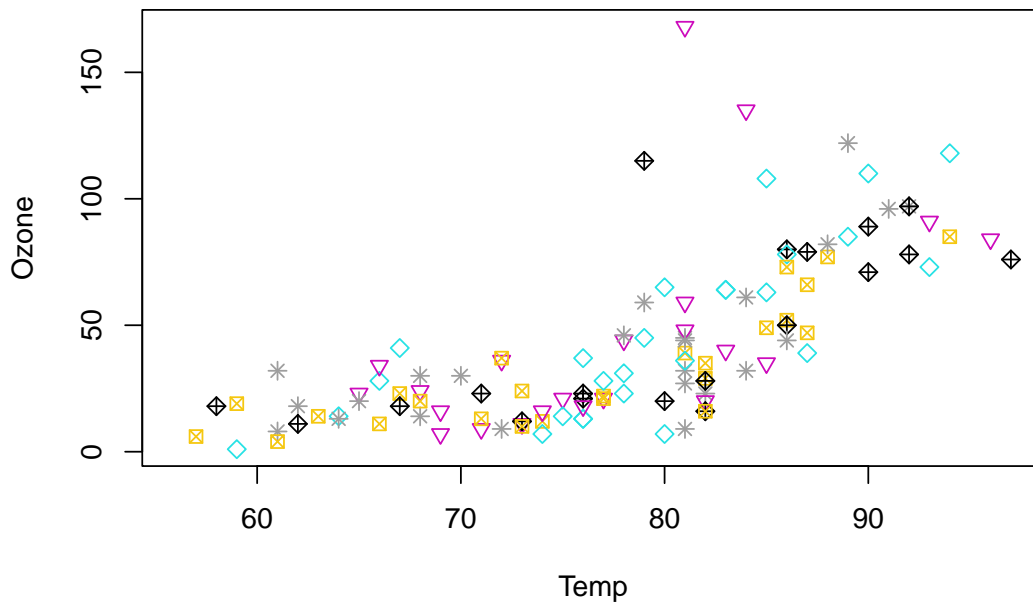


Figure 5.10: Adding month levels both as color and number plotted.

This will call one of the predefined geometric plot characters built in R.



Plot symbols

There are 26 default geometric symbols in R called with `pch=` option. Points can be omitted from the plot using `pch = NA`. `pch 21 to 25` are open symbols that can be filled by a color.

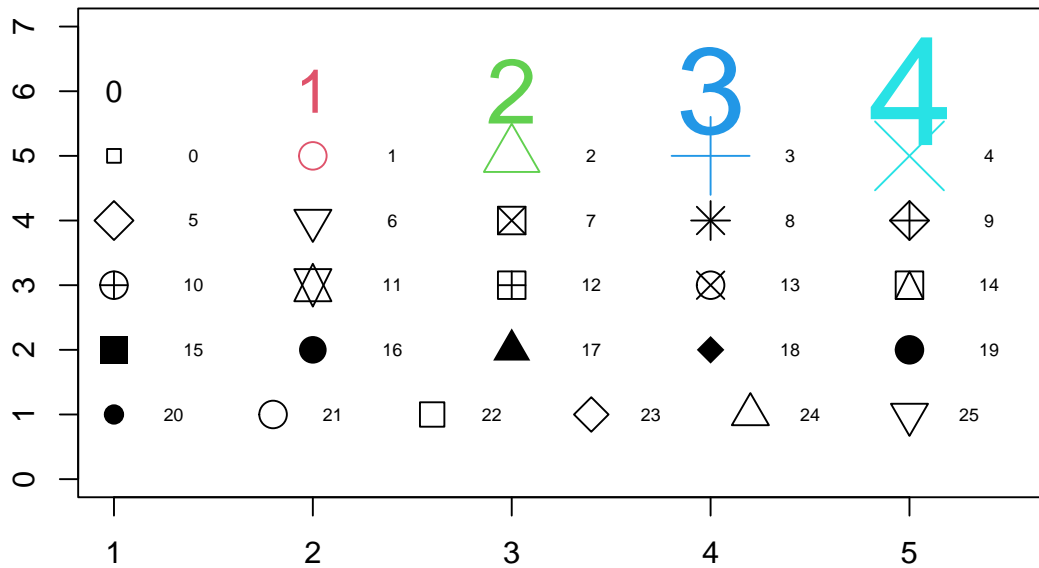


Figure 5.11: 26 pch geometric symbols for plots are numbered 0 to 25. Default is number 1: open circle.

5.7 Simple linear regression

The simple function `lm()` creates a *linear model* of the data and will omit NAs if any automatically. For this example it suffices. Other options exists, or computations can ne one to *impute* the missing data, for example replacing each NA with the average (mean) of all values. The result of `lm()` is a slope and an intercept which describes a regression line. This can help show a trend, but it is also important to keep in mind that `lm()` is a simple model and that other regression methods exist.

We can compute a simple regression line for the Ozone vs Temp by providing the values, as in a subset. The most elegant writing is by using the `with()` function:

```
model1 <- with(airquality, lm(Ozone ~ Temp))
model1
```

```
Call:
lm(formula = Ozone ~ Temp)

Coefficients:
(Intercept)      Temp
   -146.995      2.429
```

We could use `str()` on the new `model1` object to note that it has a complex structure. Suffice to mention for now that the 2 most important values can also be called with `model1$coefficients`

We can now add the regression line to the existing scatter plot with the `abline()` function used to add one or more straight lines through the current plot.

```
with(airquality, plot(Ozone ~ Temp, pch = mlev, col = mlev))
abline(model1, col = "blue", lwd = 3)
```

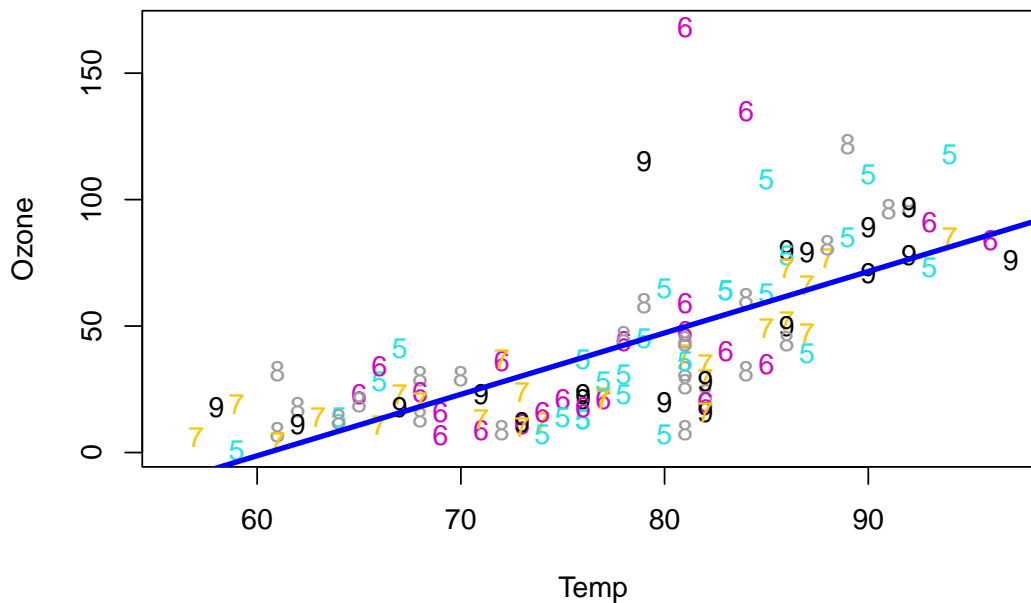


Figure 5.12: Adding the simple regression line on the scatter plot.

**Exercise:**

We saw that Ozone increases with Temp.

Using a scatter plot and an optional linear model regression can you tell what is the effect of Wind?

5.8 Fancier Graphics exploration



For this section we'll call on ggplot2 which is a package included in the Tidyverse suite. If you need to install this go to section 1.2 and proceed with the installation.

It may be useful to skip this section and review chapter 11 before spending too much time if this section proves difficult.

The ggplot2 package is now the “new standard” and while it is useful to know the graphics commands from R base, it is becoming more and more important to learn how to use this package.

There is a main command called `ggplot()` and a somewhat simpler command called `qplot()` (with a single q) that resembles a little more the graphics commands we saw earlier. (`qplot()` is short for “quick plot”.)

First we need to *activate* or *load* into memory the ggplot2 package. This is accomplished with the `library()` function.

```
library(ggplot2)
```

If you have an error make sure that you have previously installed Tidyverse or the single ggplot2 package (see section 1.2.)

5.8.1 Boxplots

Let's start by trying to reproduce some of the plots with these new commands. Here is how to create a box plot of the temperature (columnTemp) as a function of the month (column Month.) We also need to specify that we want to use Month as the coloring factor. We don't need to specify that we want the *levels* as qplot is smart enough to understand that. To obtain a boxplot we ask for a type of plot "geometry".

```
qplot(Month, Temp, data = airquality,  
      geom = "boxplot", color = as.factor(Month))
```

Warning: `qplot()` was deprecated in ggplot2 3.4.0.

This warning is displayed once every 8 hours.

Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.

Note the order of the variables that are written here in reverse order as compared to the Base R commands of figure 5.7.

Exercise 5.1. What would happen if the order of the variables Month and Temp were inverted here?

What about the base R version that created figure 5.7?

Some improvement and tweaking are always possible, but for a first plot it is not bad.

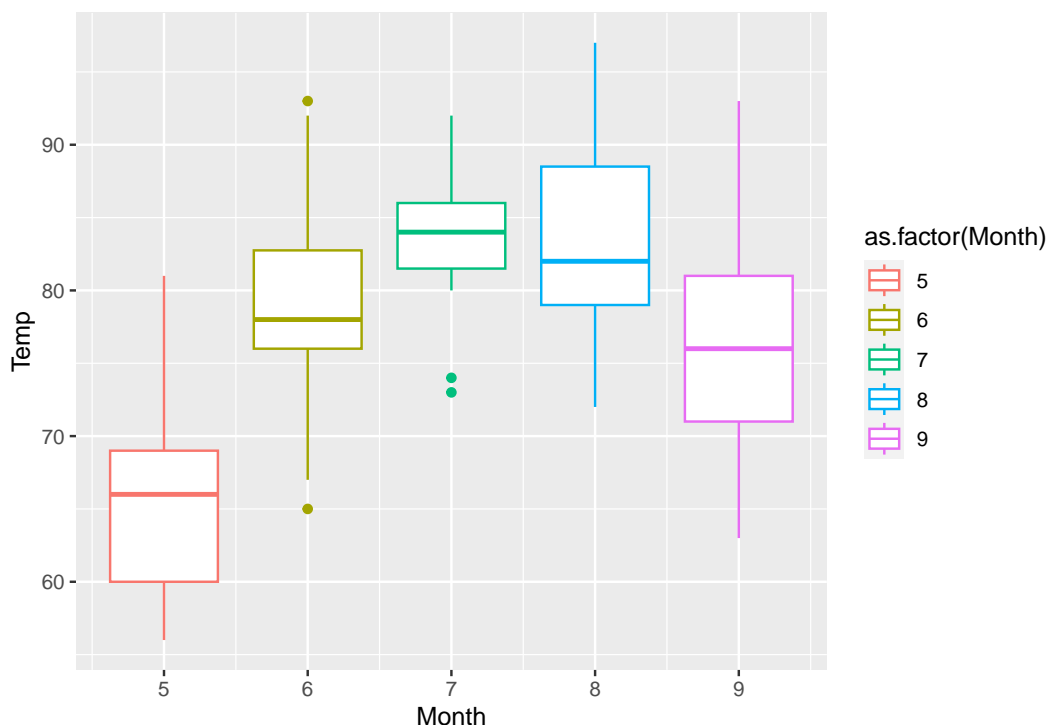


Figure 5.13: qplot version of Temperature vs Month.

We can improve the plot by transforming some of the data, namely by making the Month variable a factor rather than just a numeric entry as was shown by the `str()` function in section 5.2.

To avoid making changes to the original data, we'll copy the `airquality` data into a new object that we can call `aq` for simplicity. From that point the original dataset will not be changed and we'll only affect the `aq` object.

```
aq <- airquality
```

We can now transform the Month column using one of the subsetting methods we saw previously (section 5.3.) Both `aq$Month` and `aq[, 5]` would work. The following command will overwrite the Month column with its modified status as a factor. The command `aq$Month <- factor(aq$Month)` would provide the factor definition. But we can add a modification that will change the “label” of the

factors from numbers to the name of the month in the calendar thanks to the `month.abb` parameter that can convert the month number into an abbreviated English name.

```
aq$Month <- factor(aq$Month,  
                  levels = 5:9,  
                  labels = month.abb[5:9],  
                  ordered = TRUE)
```

Let's see if that worked with some test commands for both `aq` and `airquality`:

```
# class  
class(airquality$Month)
```

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

```
class(aq$Month)
```

```
[1] "ordered" "factor"
```

```
# levels  
levels(airquality$Month)
```

```
NULL
```

```
levels(as.factor(airquality$Month))
```

```
[1] "5" "6" "7" "8" "9"
```

```
levels(aq$Month)
```

```
[1] "May" "Jun" "Jul" "Aug" "Sep"
```

We can now redo the plot:

```
qplot(Month, Temp, data = aq, geom = "boxplot", color = Month) +  
  theme(legend.position = "none")
```

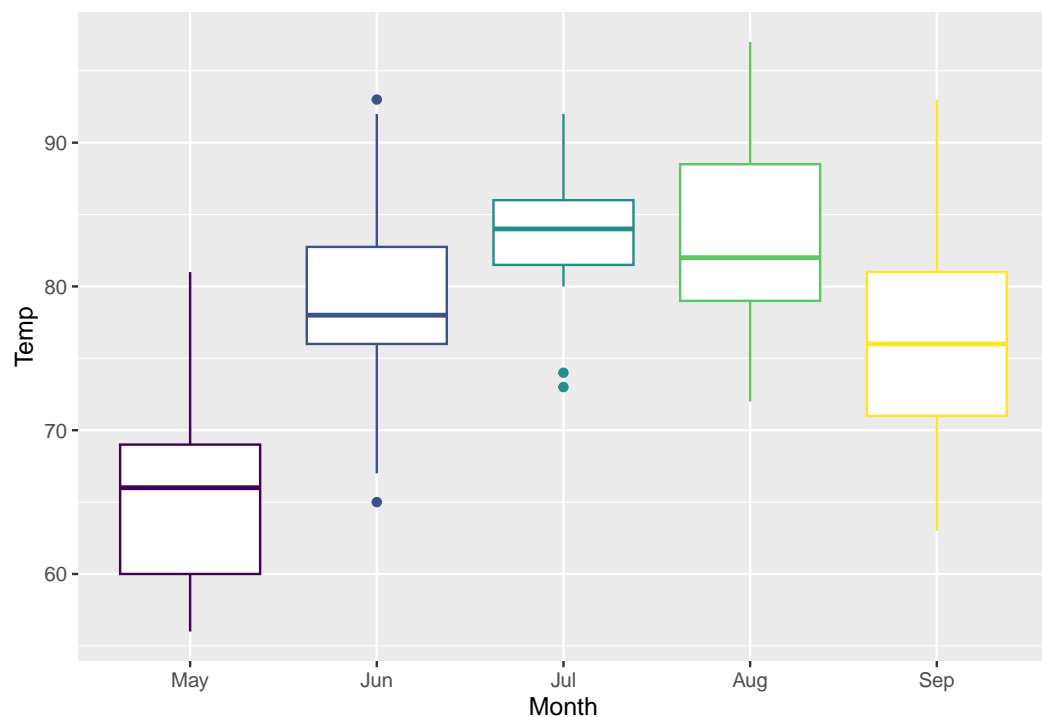


Figure 5.14: Better qplot version of Temperature vs Month.

Exercise 5.2. Exercise

Create 4 plots of boxplot on a single page colored by month for the following:

- Ozone vs Month
- Solar.R vs Month
- Temp vs Month

- Wind vs Month

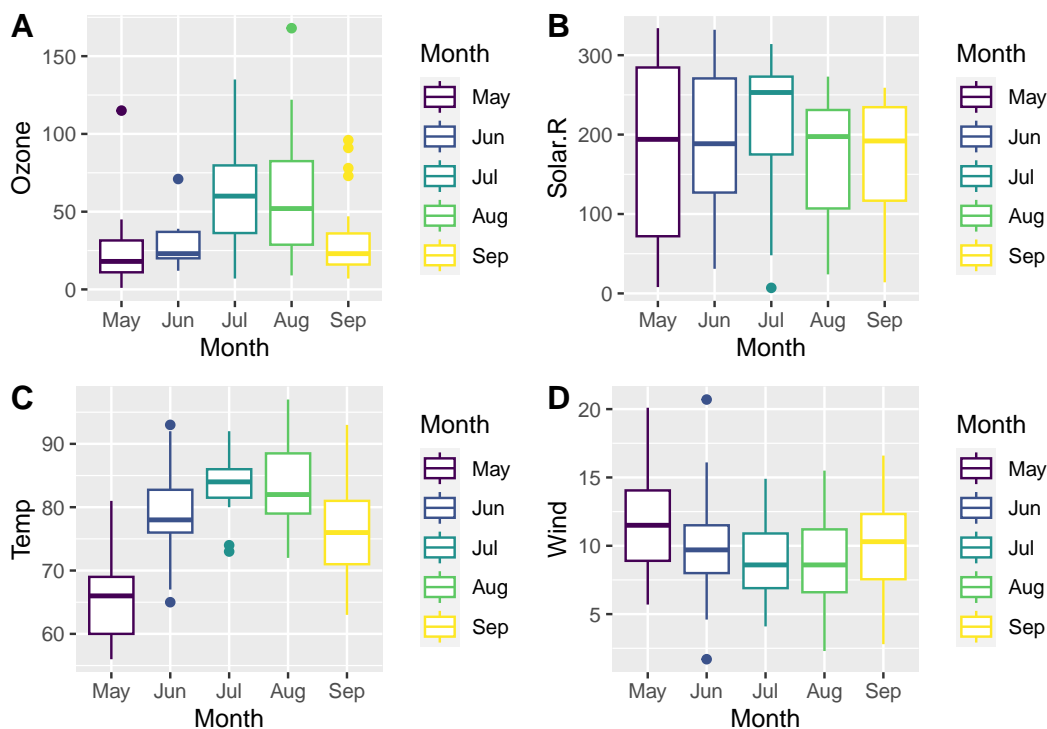
Unlike Base R graphics the `par(mfrow = c(2,2))` command would not do the job.

For this we need to rely on a newer package that helps publish ggplot style graphics.

Therefore we need to install the package `ggpubr` for example with command `install.packages("ggpubr")` (dependent packages will also be updated.) From this package, the function `ggarrange()` can be used to list the plots sequentially, specifying the number of rows and columns on the final page at the end. We can also optionally add large labels.

```
library(ggpubr)

ggarrange(
  qplot(Month, Ozone, data = aq, geom = "boxplot", color = Month),
  qplot(Month, Solar.R, data = aq, geom = "boxplot", color = Month),
  qplot(Month, Temp, data = aq, geom = "boxplot", color = Month),
  qplot(Month, Wind, data = aq, geom = "boxplot", color = Month),
  labels = c("A", "B", "C", "D"),
  ncol = 2, nrow = 2)
```



In this example the legend is repetitive and could be omitted from at least 3 of the plots. This can be accomplished by adding `theme(legend.position="none")` for each of the plots for which we want to remove the legend. For example:

```
qplot(Month, Temp, data=aq, geom="boxplot", color=Month) +
  theme(legend.position="none")
```

5.8.2 Scatter plots

We can also create a scatter plot easily. Remember that we made `Month` a factor above (5.8.1.)

```
qplot(Temp, Ozone, data = aq, col = Month)
```

Warning: Removed 37 rows containing missing values (``geom_point()``).

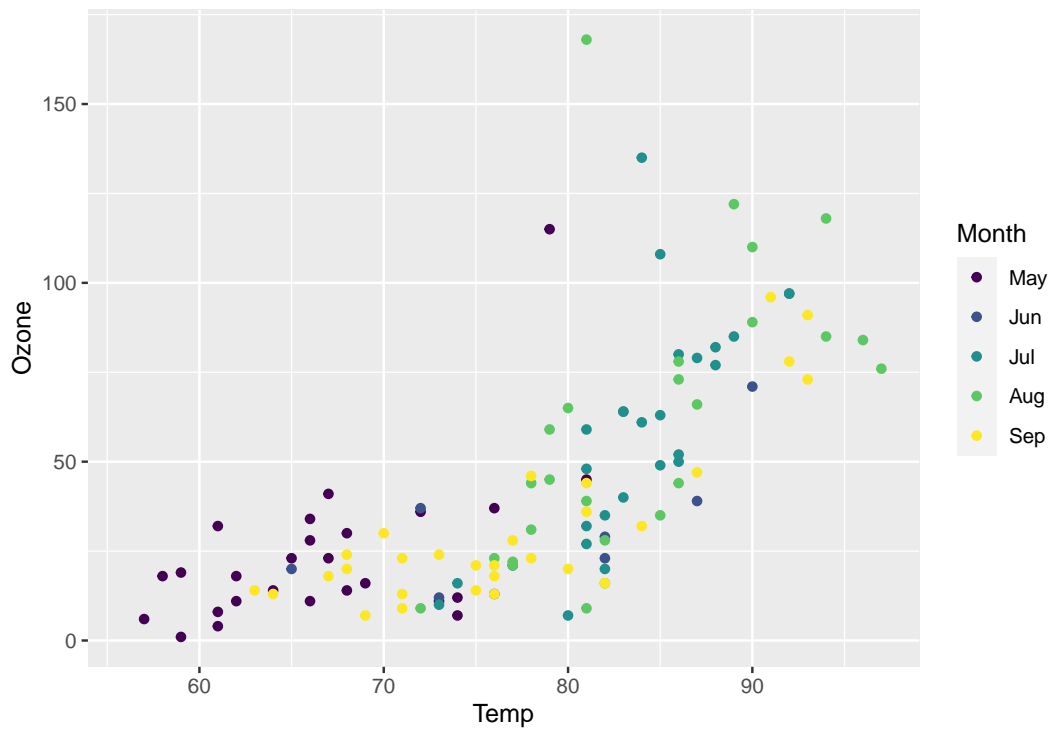


Figure 5.15: Scatter plot for Ozone vs Temperature.

We can also add a linear regression which will be calculated directly by specifying the method as "lm". Since Month is a factor the linear regression will be calculated separately for each month automatically. The SE option is a request to not print the standard error that would make the plot cluttered.

```
qplot(x = Temp, y = Ozone, data = aq,
      col= Month,
      geom = c("point", "smooth"),
      method = "lm",
      se = FALSE)
```

```
`geom_smooth()` using formula = 'y ~ x'
```

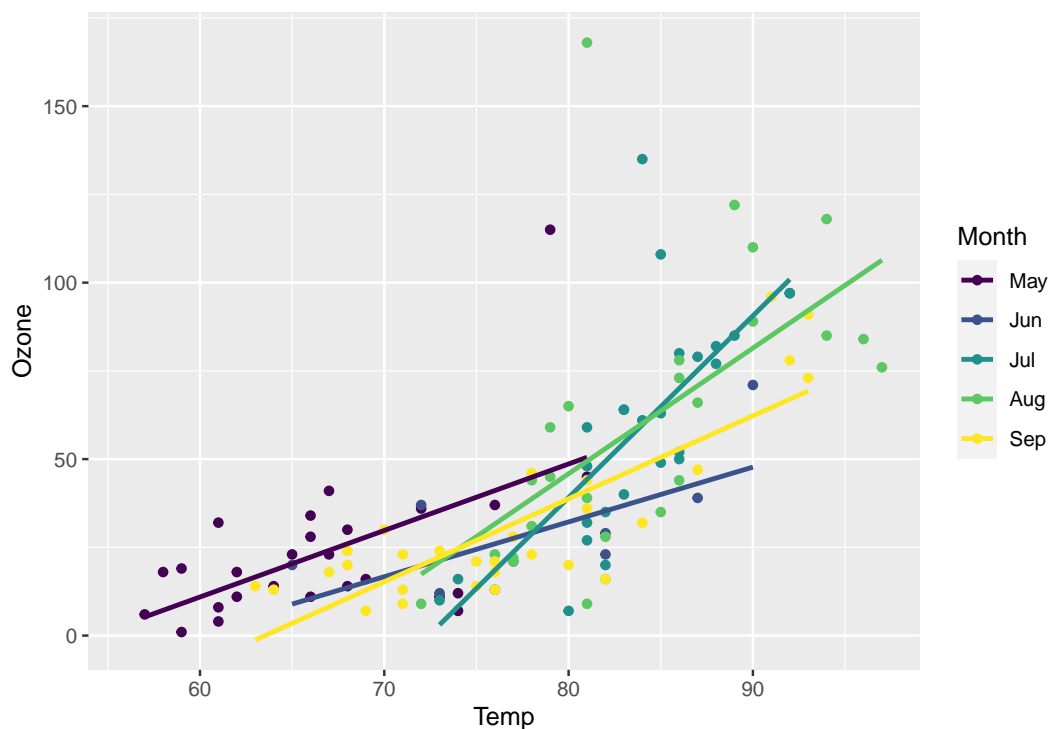


Figure 5.16: Scatter plot for Ozone vs Temperature, linear regression for each month.

To compute the linear regression as we did with the classic R plot all we need to do is to specify that we want to use month as a numeric value. We can also now turn SE to TRUE if we wish:

```
qplot(x=Temp, y=Ozone, data=aq,
      col=as.numeric(Month),
      geom=c("point", "smooth"),
      method="lm",
      se = T)
```

```
`geom_smooth()` using formula = 'y ~ x'
```

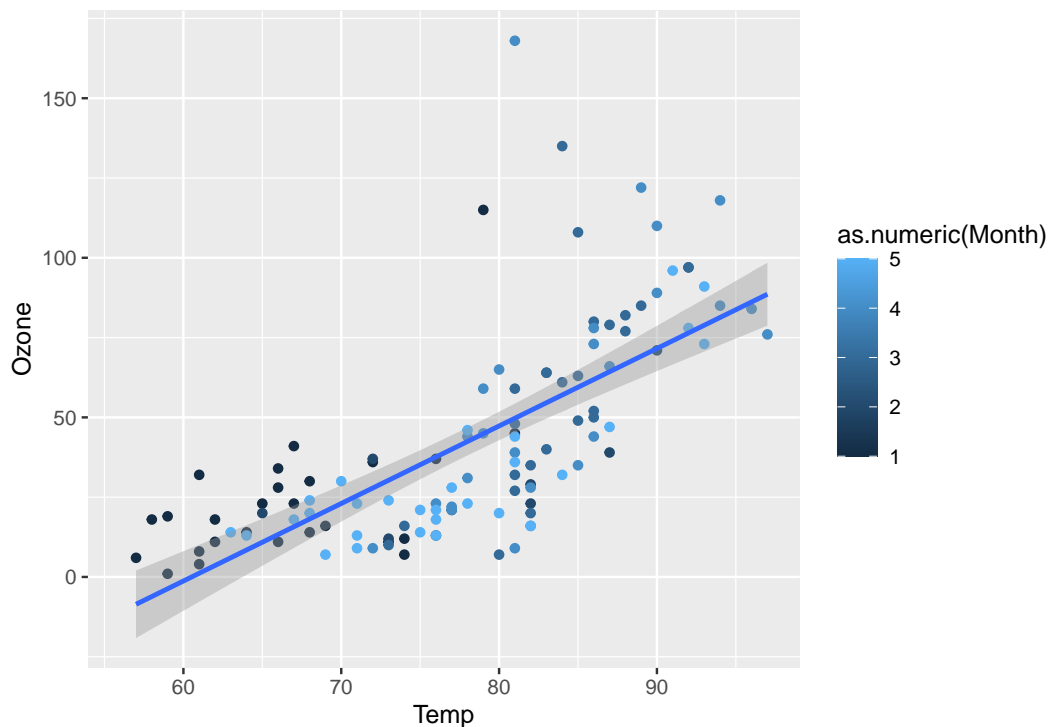


Figure 5.17: Scatter plot for Ozone vs Temperature. Linear regression for all months.

The result is that the legend now reports Month as a continuous data, which is not correct. The legend could be removed by adding `theme(legend.position="none")` as we saw above.

If we do not specify the *method* by removing `method="lm"` we obtain the default, more complex, non linear regression line. In that case the “loess” regression is used.

```
qplot(x=Temp, y=Ozone, data=aq,
      col=as.numeric(Month),
      geom=c("point", "smooth"),
      se = T) +
  theme(legend.position="none")
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

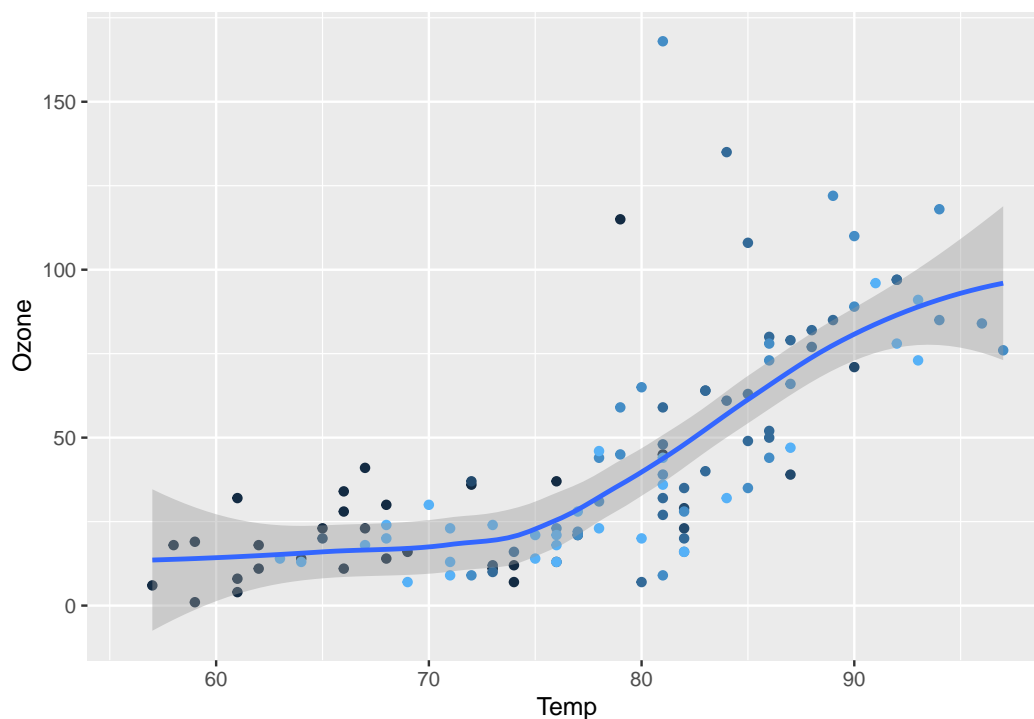
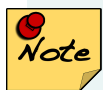


Figure 5.18: Scatter plot for Ozone vs Temperature. Linear regression for all months.



These examples above are to show what is possible with `qplot` which is the *quick plot* version of the more fancy `ggplot`.

Using Internet search is useful to find examples of code that help. For example the linear regression addition was found on this stack overflow page:

[I need to add linear regression trend lines to qplot²](#).

Chapter 6

Importing data

Importing data is rather easy in R but that may also depend on the nature of the data to be imported and from what format. For environmental studies data are usually in tabular form such as a spreadsheet or a comma-separated file (.csv.)

In this chapter:

- Importing from local files
- Downloading Nhanes data
- Exploring PFAS_I data
- Merging data files

The **way** or method used to import the data in R will have fundamental implications on the `class` of the object containing the data just read and therefore what methods can later be used to analyze the data. In Classic R we'll most likely want to have the data in the `data.frame` class as it is the most versatile and useful.

Base R has a series of *read* functions to import tabular data from plain text files with columns delimited by: *space*, *tab*, *comma*, with or without a *header* containing the column names. With an added package it is also possible to import directly from a Microsoft Excel spreadsheet format or other *foreign* formats from various sources.

6.1 Importing from local files

In base R the standard commands to read text files are based on the `read.table()` function. The derived functions exist in 2 flavor to accommodate USA and European conventions for decimal point (a comma in Europe) and comma separator (a semicolon in Europe.) The following table lists the collection of the base R “read” functions. For more details use the help command `help(read.table)` that will display help for all.

Table 6.1: Base R read functions

Function name	Assumes header	Separator	Decimal	File type	Comment
<code>read.table()</code>	No	none	.	.txt	USA
<code>read.csv()</code>	Yes	,	.	.csv	USA
<code>read.csv2()</code>	Yes	;	,	.csv	Europe
<code>read.delim()</code>	Yes	Tab	.	.txt	USA
<code>read.delim2()</code>	Yes	Tab	,	.txt	Europe

A similar approach is used to **write** the data out but the `*delim()` version do not exist, but can be managed with specifying the tab delimiter within the `write.table()` function.

Assuming that you have a file name `test.csv` containing these 5 columns of data

```
c1,c2,c3,c4,c5
1.481,3.478,4.246,3.687,6.051
1.73,5.825,4.526,6.754,0.15
2.556,6.275,2.525,6.368,5.479
2.828,4.77,5.12,3.744,4.01
2.989,4.396,2.078,4.237,4.618
3.122,6.317,5.414,3.551,5.607
```


The command to read such a file into a user defined object named `test` would be:

```
# Do not run  
test <- read.csv("test.csv")
```

6.2 Downloading Nhanes data

R is a great “statistical software for data analysis” but there are other competing software in Industry that can even be expensive such as SPSS, STATA, JMP, Matlab, and SAS.

NHANES data is saved in a SAS transport file (.xpt) created by the SAS XPORT engine. This is what is available on the NHANES web site. Fortunately they also provide methods to import this data in R by using the [foreign package](#) (see Appendix C.)



TASK: *Install package. foreign.*

It is necessary to install this package to be able to follow the code below and import the NHANES data. You can use `install.packages("foreign")` or follow alternate direction in section 1.2.

See also Appendix D.4 for an alternate method using the haven package instead. See Appendix D.5 for code to download and save .XPT files onto your computer.



Relax

You may be given a pre-downloaded data set for your homework exercise(s).

However, it is always useful to know where and how to get your own data. This is the purpose of this section.



Other options See Appendix D and section 6.4 for more details.

The relevant .XPT files used in this book were combined into a .zip archive that can be downloaded: [XPTs.zip](#)

The combined “Master4” file can also be downloaded as well as a .csv file with either:

- [Master4.csv](#)
- [Master4.csv.zip](#)

6.2.1 PFAS_I

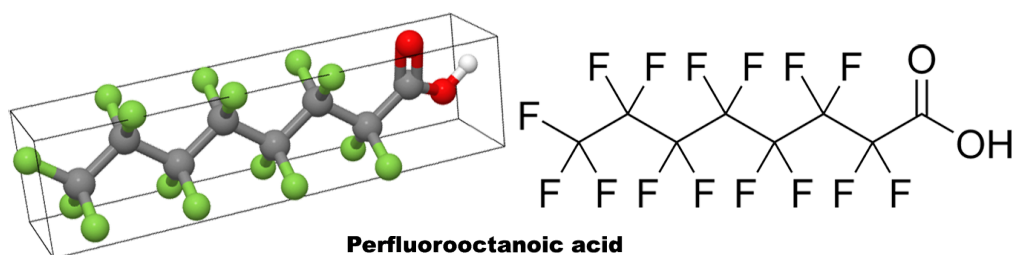


Figure 6.1: Perfluorooctanoic acid (PFOA) is used worldwide as an industrial surfactant in chemical processes and as a material feedstock, and is a health concern and subject to regulatory action and voluntary industrial phase-outs.

As an example we'll download the file resulting of the blood serum analysis of *Perfluoroalkyl and Polyfluoroalkyl substances (PFAS_I)* used in multiple commercial applications including surfactants, lubricants, paints, polishes, food packaging and fire-retarding foams. More information can be read from the [documentation page](#)¹ (that also contains a link describing all the details for the laboratory methods used.)

¹https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/PFAS_I.htm



For more information:

- [Perfluoroalkyl and Polyfluoroalkyl Substances²](#) on NIEHS / NIH web site.
- [Perfluoroalkyl and Polyfluoroalkyl Substances in the Environment: Terminology, Classification, and Origins³](#) by Buck et al. (2011). (See their Table 1 in Appendix F.)

The NHANES tutorial R code (Appendix C) is for the **demographic** data in file DEMO_I.XPT:

```
# DO NOT RUN
# Download NHANES 2015-2016 to temporary file
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/DEMO_I.XPT",
              tf <- tempfile(),
              mode="wb")

# Create Data Frame From Temporary File
DEMO_I <- foreign::read.xport(tf)
```

The first command using the `download.file()` function accomplishes 2 tasks, it will:

1. **download** the file from the web link
2. **save** the file to a temporary file named `tf`, using the transfer mode coded `w` for *write* and `b` for *binary*. (For more info see detail on *file open* `fopen` options⁴.)

⁴<http://www.cplusplus.com/reference/cstdio/fopen/>

This “trick” avoids saving the file locally to the hard drive. Should one want to do that the command could be simplified by replacing `tf <- tempfile()` with the name of a file within quotes such as `DEMO_I.XPT`.

The second command uses the `foreign` package function `read.xport()` to read the data into a data frame named `DEMO_I`. If we had saved the file to the local drive we would replace `tf` with `DEMO_I.XPT`.



NOTATION: The use of `::` notation in `foreign::read.xport(tf)` tells R to use the function `read.xport()` from the `foreign` package **without** the need to use the `library()` function first. This is common for cases where we only want to use a function once.

Alternate package: An alternate option to the code provided is using the package `haven` and download the file instead with:

```
DEMO_I <- read_xpt(url(
  "https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/DEMO_I.XPT"))
```

The data can be found on the web site starting at: <https://wwwn.cdc.gov/nchs/nhanes/ContinuousNhanes/> and then:

- Click on “Laboratory data” (figure 6.2)
- Scroll and find the 2015-2016 entry

The entry is specified to be only 376.7 Kb in size.



TASK: Download file `PFAS_I.XPT`.

The data file direct link is:

https://wwwn.cdc.gov/nchs/nhanes/2015-2016/PFAS_I.XPT

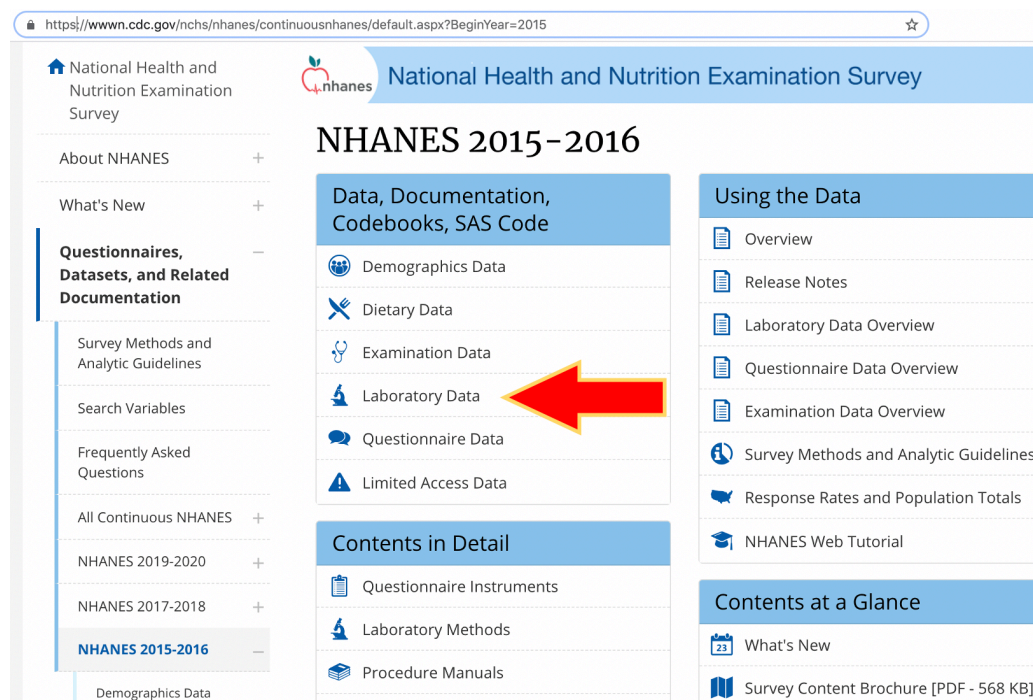


Figure 6.2: Finding NHANES 2015-2016 data.

We can download the file as in the above example without the need to save the .XPT file on the local drive.

```
# Download NHANES PFAS_I 2015-2016 data to temporary file
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/PFAS_I.XPT",
              tf <- tempfile(),
              mode="wb")

# Create Data Frame PFAS_I From Temporary File
PFAS_I <- foreign::read.xport(tf)

class(PFAS_I)
```

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

We now have a data frame named `PFAS_I`.

6.3 Exploring PFAS_I data

`PFAS_I` is of class `data.frame`. The meaning of the column headers can be found in the NHANES documentation page https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/PFAS_I.htm (also found in Appendix E.)

The first and last three codes are in the table shown here.

Table 6.2: PFAS_I codes for sum data

Code	Description
SEQN	Respondent sequence number
LBXMFOS	Sm-PFOS (ng/mL)
LBDMFOSL	Sm-PFOS Comment Code

If we read the code information on the web page or on Appendix E we can see that some columns are “*comment*” columns. These report the “*success*” of the analysis with a value of 0 *at or above the detection limit*, a value of 1 *below lower detection limit* and a dot . for missing values. One more thing to notice is that the text entry describing these columns is written in multiple ways:

- Comment Code: 5 times
- Comt Code: 4 times
- comment : 1 time

Therefore there are 10 “*comment*” columns that alternate with data columns.

We can also note that the data columns all contain an X in their name while the comment columns contain the letter D. The first 2 columns of SEQN and WTSB2YR are not part of that naming pattern.



REVIEW classic R methods:

This is the perfect time to review the classic methods that are built-in R that we explored with the `airquality` dataset with functions:

`dim()`, `length()`, `str()`, `summary()`, `colnames()`, `head()` and `tail()`, `colSums(is.na())`. etc.

Since they were already visited those exact commands will not be developed again but we'll see how to do some specific manipulations by adding optional arguments in some of the commands.

We'll start by exploring the data graphically.

6.3.1 PFAS_I boxplot

The base R graphics functions are rather smart to make a graph quickly with little information. We could for example use `boxplot(PFAS_I)` but we would quickly note that the first column SEQN would “crush all other columns simply because the values for this column that represents the “Respondent sequence number” (the code for each individual) has range 83736 to 93700 as reported by command `summary(PFAS_I[,1])`. Therefore as a first approach it would be useful to be able to plot everything **without** the first column. This is accomplished with subsetting (section 5.3) but using a **minus sign** to indicate that we want to remove the designated column. The command to remove the first command would be:


```
# Remove the first columns  
boxplot(PFAS_I[, -1])
```

We just need to remember that within the square brackets the first item represents rows and the second represents columns. Nothing written means take everything. In that sense `PFAS_I[,]` is exactly the same as simply `PFAS_I`. We just specify `-1` for columns to remove the first one.

However, the next “annoying” thing will be that data from column 2 is now “crushing” the other boxes. So we now want to remove the first 2 columns: `SEQN` and `WTSB2YR`. How do we do that? there are only 2 spots within the square bracket.

To remove the 2 columns we can take advantage of the combine function `c()` to gather the numbers of the columns and add a minus sign before it to specify their removal. We can thus write:

```
# Remove the first and second columns  
boxplot(PFAS_I[, -c(1:2)])
```

This is still not satisfying as there are a lot of “outliers” *i.e.* values that extend beyond the box. And once again the boxes are all “crushed”.

One way to have a better image is to limit the values that are plotted in the vertical (y) axis by using the optional parameter `ylim` = which requires 2 numbers specifying a lower and an upper limit, for example from 0 to 10. Note that once again we need to use the combine `c()` function, something that is ubiquitous in R code:

```
# Remove the first and second columns  
# limit vertical axis with ylim  
boxplot(PFAS_I[, -c(1:2)], ylim = c(0,10))
```

All four attempts can be summarized as:

```

par(mfrow = c(2,2))
# All data
boxplot(PFAS_I)
# Remove the first columns
boxplot(PFAS_I[, -1])
# Remove the first and second columns
boxplot(PFAS_I[, -c(1:2)])
# limit vertical axis with ylim
boxplot(PFAS_I[, -c(1:2)], ylim = c(0,10))

```

```

par(mfrow = c(1,1))

```

This has not been so useful yet, but we are getting closer perhaps!

We noted earlier (section 6.3) that besides the first 2 columns, every other column was a “comment” column containing only 0, 1, and a few .. Therefore it is not very useful to include them in the plot. We can further realize that they are all columns with an *odd* number, and we can create a list of these numbers from the `seq()` function (section 4.8.1).

We want to omit columns 1 and 2, and finish with column 21 since the last column 22 is a “comment” column. So starting with 3 and stepping by 2 will provide all *odd* numbers between 3 and 21.

```

seq(3,21, by = 2)

```

```

[1] 3 5 7 9 11 13 15 17 19 21

```

We can also remember that using a `log()` function may give the data a better spread. If this does not work for the boxplot perhaps it will work for a histogram. We could also noted in the previous attempts that the labels for the columns are printed on the horizontal axis, but not all of them due to spacing. We can therefore add an additional option which will rotated the bottom labels by 90 degrees

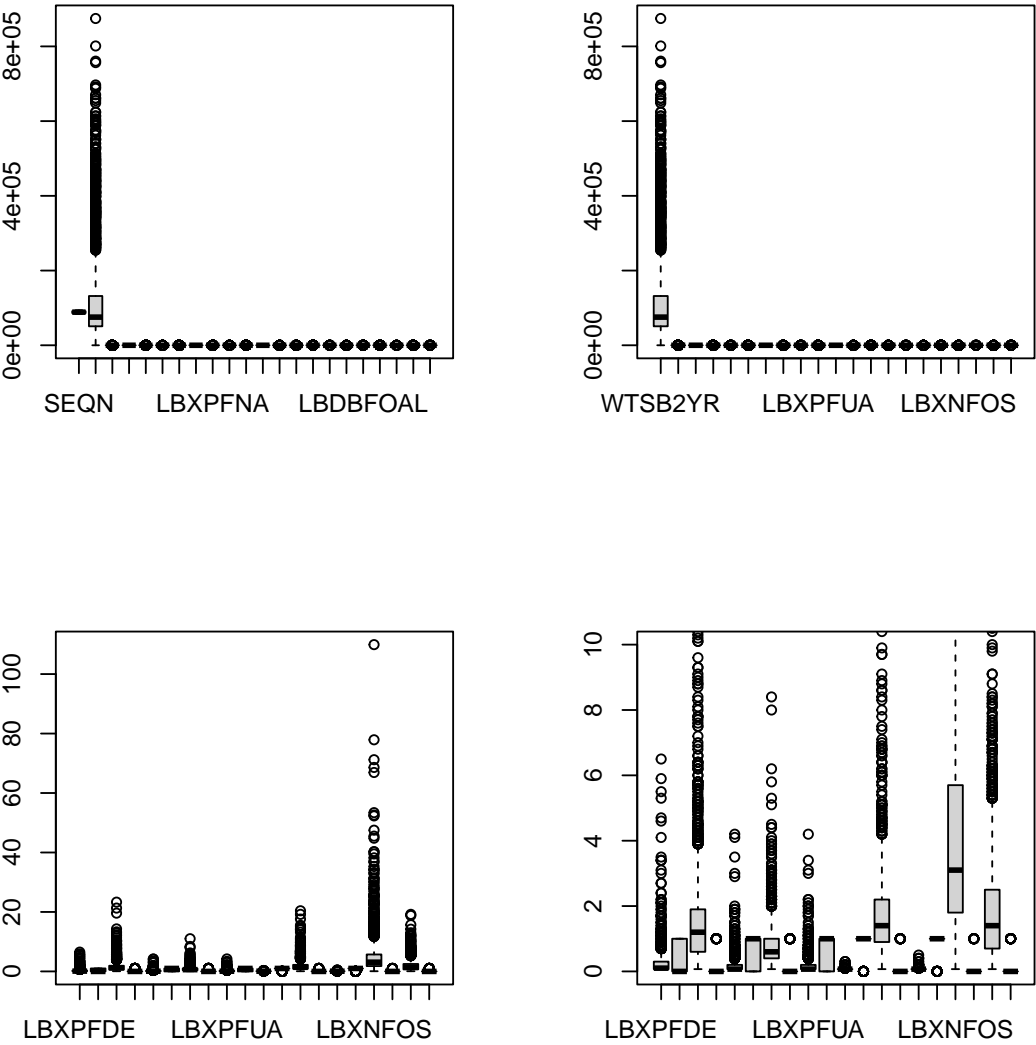


Figure 6.3: Summary of 4 attempts

so that all of them can be printed. To find this option one would have to learn about it in an example, as finding it by help is tricky unless we know where to look, which would be the list of parameters for graphics found with `help(par)`. The command is `las=2` and that is most likely an abbreviation for **l**abel **a**xis **s**tyl**e**.

The following command combines all of that. We are asking for a boxplot from `PFAS_I`, the values will be changed to the natural log, only for columns that are *odd* and the label will be rotated on the horizontal axis:

```
boxplot(log(PFAS_I[, seq(3,21, by = 2)]), las=2)
```

There are many fancy ways to alter base R graphics, a very detailed example can be found in [this blog](#)⁵ or this published “exercise” [Fixing Axes and Labels in R Plot Using Basic Options](#)⁶.

However, most people now are switching to more modern plotting methods for making the final, fancy or published version. However, it is still very useful to know how to use R base graphics to explore data as they are usually simpler to apply at first.

6.3.2 PFAS_I histogram

We can perhaps quickly apply what we just learned to making histograms of the data.

The difference here is that each histogram would need to be a separate graph. So just replacing `boxplot` by `hist` will not work.

Let’s start by looking at just one of them. Column 21 is the sum of all others. We can test also if it is necessary to use the `log()` function. We can plot both original and logged values in one image. We can also use `remember` and use the other

⁵<https://www.tenderisthebyte.com/blog/2019/04/25/rotating-axis-labels-in-r/>

⁶<https://rpubs.com/riazakhan94/297778>

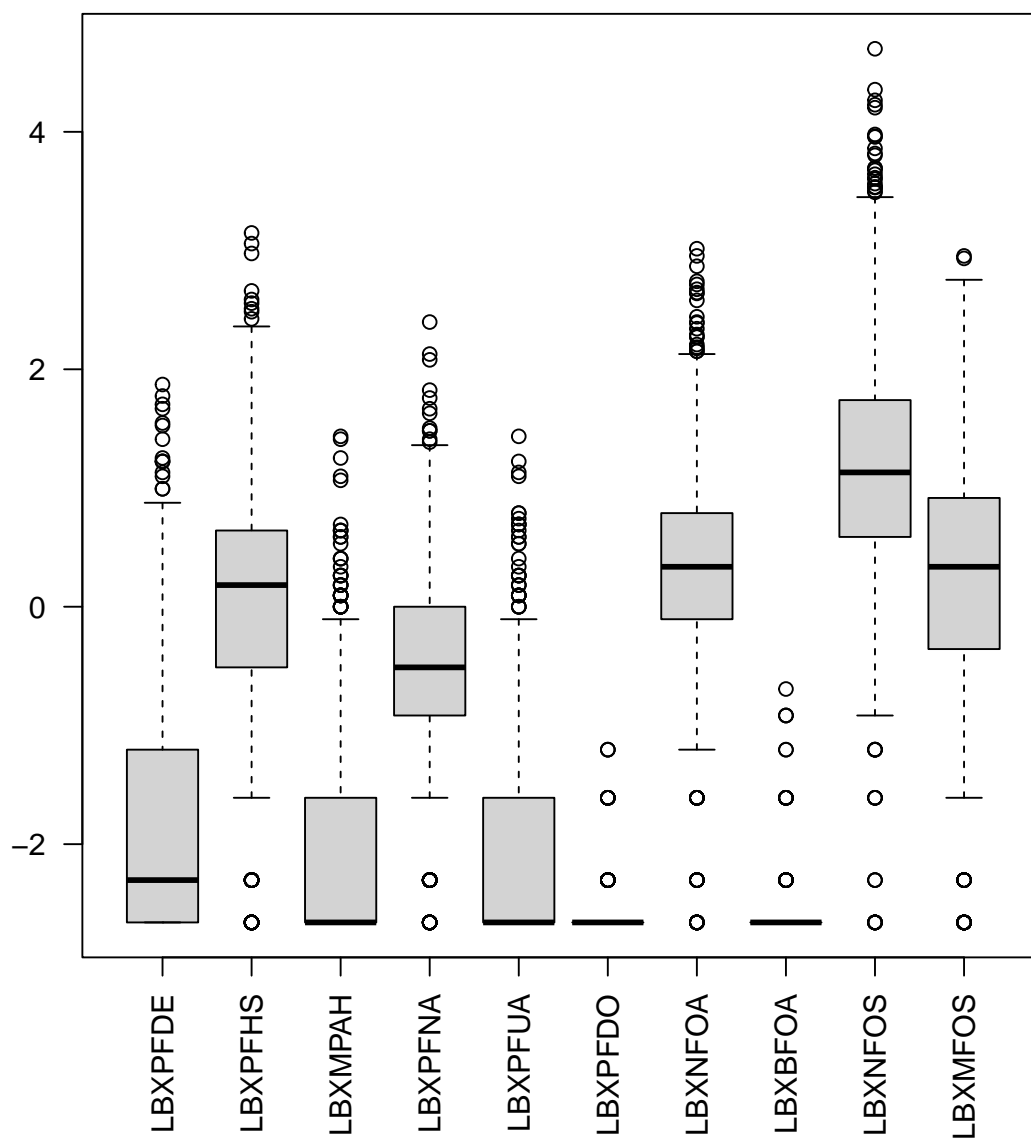


Figure 6.4: PFAS_I boxplot with log values for odd columns and rotated labels.

subset method using the `with()` function which will make the title of the plot nicer (see (section 5.4.))

```
par(mfrow = c(1,2))  
# original data  
# hist(PFAS_I[,21])  
with(PFAS_I, hist(LBXMFOFOS))  
# natural log applied  
# hist(log(PFAS_I[,21]))  
with(PFAS_I, hist(log(LBXMFOFOS)))
```

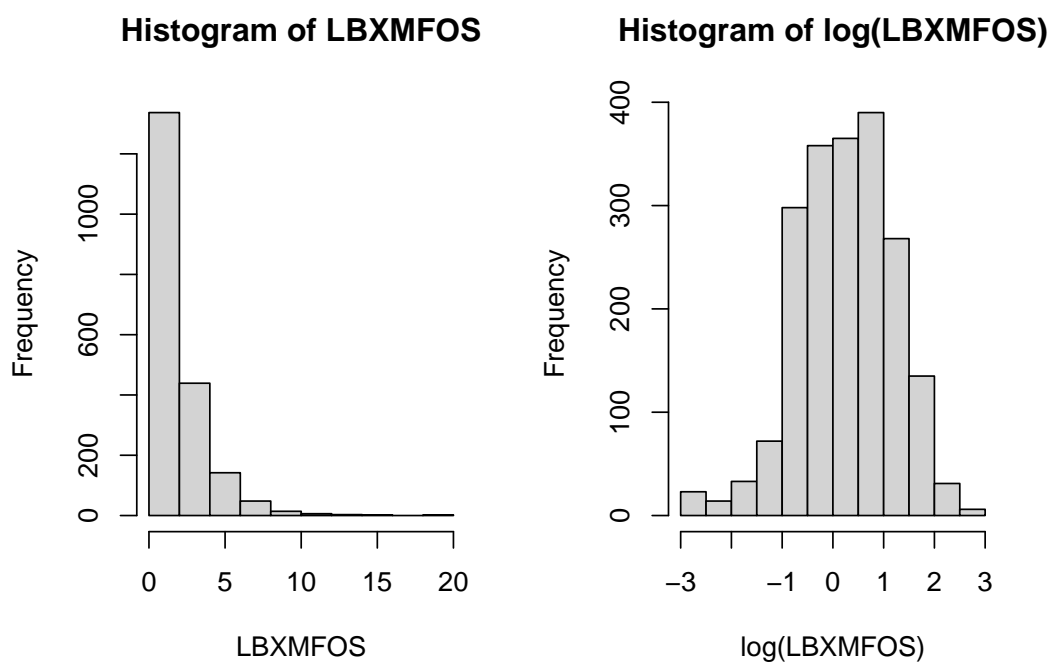


Figure 6.5: PFAS_I histogram for summed values in column 21 labeled LBXMFOFOS.

```
par(mfrow = c(1,1))
```



NOTE For histograms the options `breaks = 25` could be added (possibly with a different number) to bin into smaller portions and make a finer plot. By default the histogram will be a “frequencies” version that can be changed to a “densities” version so that the histogram has a total area of one. This is done by adding `freq = FALSE`. See `help(hist)`.

So how do we plot all of the histograms for all columns? This is more complicated than it appears at first thought. On discussion forums it would be possible to find answers that have R code on many lines looking like a full program. However, there is a simple solution but it uses a rather challenging base R function that is difficult to understand for most people.

It would be very difficult to also apply the `log()` function to the data. Let’s create a small subset of 4 of the 10 data columns. We’ll store that data in a simple named object `L` after which we can verify some properties:

```
L <- log(PFAS_I[, c(5,9,15,21)])  
class(L)
```

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

```
colnames(PFAS_I[, c(5,9,15,21)])
```

```
[1] "LBXPFHS" "LBXPFNA" "LBXNFOA" "LBXMFOS"
```

Taking into account that we’ll have 4 plots the “magical” command is now simply: `lapply(L, hist)`.

```
par(mfrow = c(2,2))
lapply(L, hist)
```

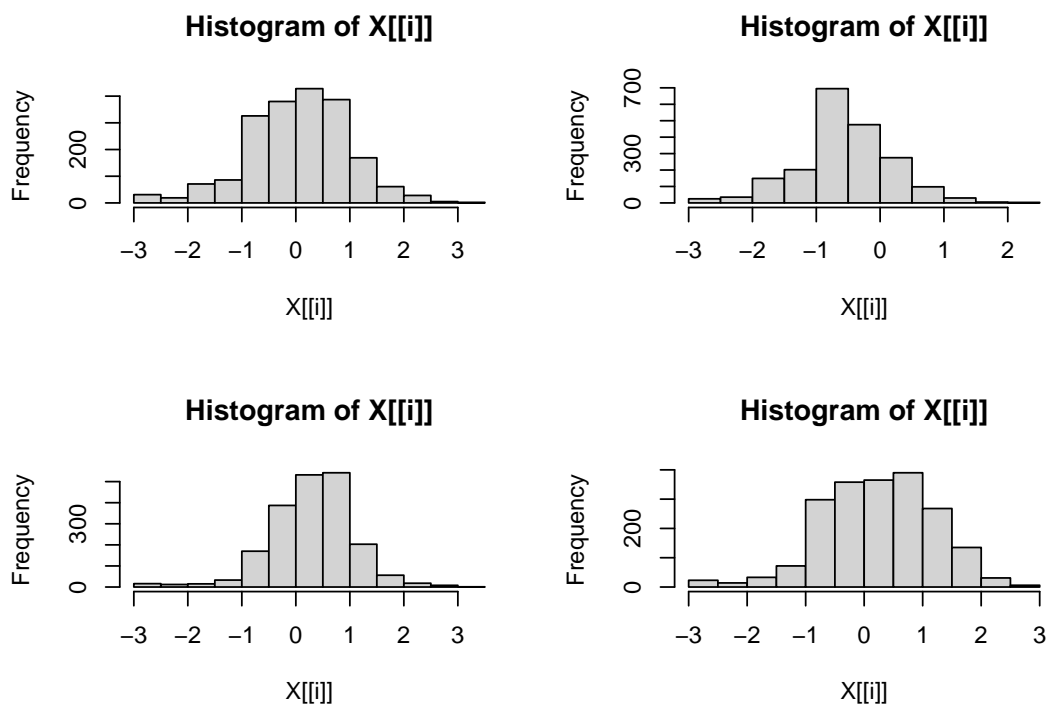


Figure 6.6: Creating multiple histograms with one command

```
par(mfrow = c(1,1))
```

Some additional options can be added to change the histogram but the format is different that when the `hist()` function is used. In this case the additional parameters would need to follow `hist` separated by a comma. For example to add 2 parameters:

```
lapply(L, hist, breaks=25, freq = FALSE).
```

Each title “Histogram of X[[i]]” could be changed to the same title for all with *e.g.* `main = "Histogram"` or completely suppressed with `main = ""`. But to provide the name of the column either in the title or on the axis would require even more sophisticated commands.



ADVANCED The solution for this calls on the `lapply()` function that can be difficult to understand. This whole *family* of functions is described in details on the [guru99](https://www.guru99.com/r-apply-sapply-tapply.html) web site:

<https://www.guru99.com/r-apply-sapply-tapply.html>

These functions can be very useful and are found as suggestions on forums.

6.3.3 Fancier boxplot with qplot



This section uses `ggplot2` which is a package included in the Tidyverse suite. If you need to install this go to section 1.2 and proceed with the installation.



This section is here to illustrate another way to create the same or similar plots as we did with base R. It may be confusing at first, in which case this section can be skipped to better come back later. Perhaps after learning more from links in chapter 11.

Most `ggplot` examples online assume that some of the columns of the data can be used to plot *against* other columns as we did for the `airquality` plots using the easier `qplot()` (section 5.8.)

The data we have in `PFAS_I` are all number data and we'd like to plot them all as we did with the base R graphic function `boxplot()`.

This section with a **quick solution is here** as it is difficult to find examples that match the data style we have here *i.e.* many columns that need to be plotted. The solution involves the `stack()` function that is part of the `utils` “utilities library” for data frames.

This solution was found on this forum page: [Building a box plot from all columns of data frame with column names on x in ggplot2](https://stackoverflow.com/questions/27109347/building-a-box-plot-from-all-columns-of-data-frame-with-column-names-on-x-in-ggplot2)⁷.

To understand what it does we can simply look at its effect on the `L` object:

```
head(stack(L))
```

	values	ind
1	-0.5108256	LBXPFHS
2	0.3364722	LBXPFHS
3	1.5892352	LBXPFHS
4	0.1823216	LBXPFHS
5	-1.6094379	LBXPFHS
6	-0.6931472	LBXPFHS

Note: this could be written `head(utils::stack(L))` in case of ambiguity with other commands.

The effect is to create a simpler but longer format as long as the number of columns multiplied by the number of rows. For `L` this would be $2170 * 4 = 8680$. The column names for this format is always `values` and `ind` (independent variable) which are names to be reported in `qplot()`:

```
# library(ggplot2)
qplot(ind, values, data = stack(L), geom = "boxplot")
```

Warning: Removed 708 rows containing non-finite values (``stat_boxplot()``).

⁷<https://stackoverflow.com/questions/27109347/building-a-box-plot-from-all-columns-of-data-frame-with-column-names-on-x-in-ggplot2>

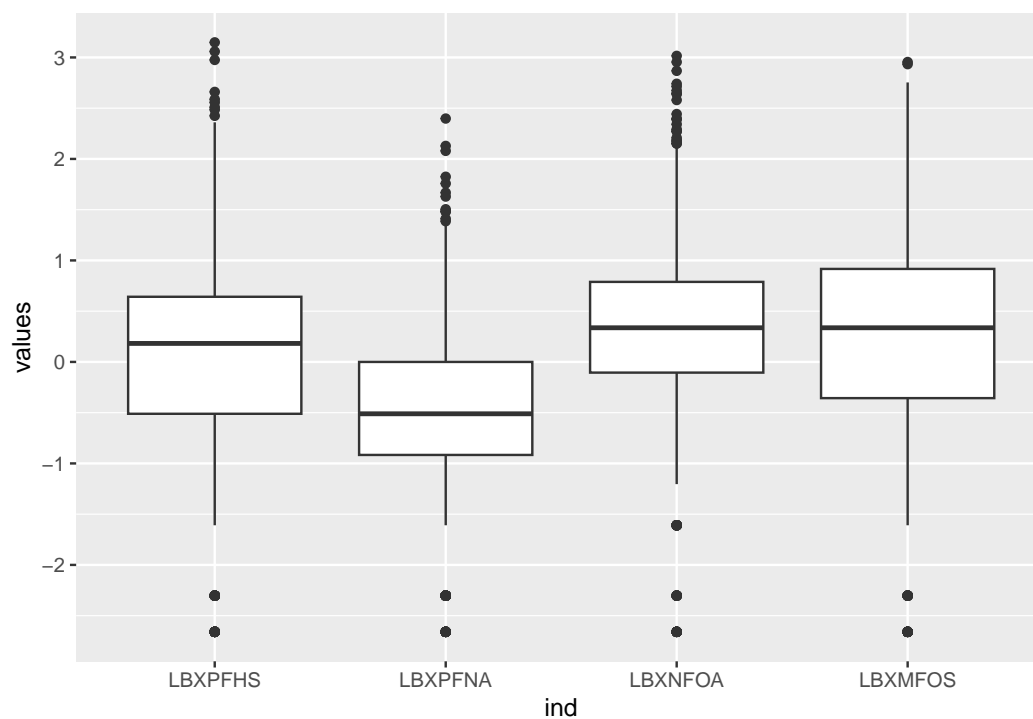


Figure 6.7: PFAS_I boxplot with log values for 4 columns.

We'll see later how this work, but the ggplot version for all 10 odd-numbered columns could be written as:

```
# library(ggplot2)
ggplot(stack(log(PFAS_I[, seq(3,21, by = 2)])),
  aes(x = ind, y = values)) +
  geom_boxplot()
```

Warning: Removed 1770 rows containing non-finite values (``stat_boxplot()``).

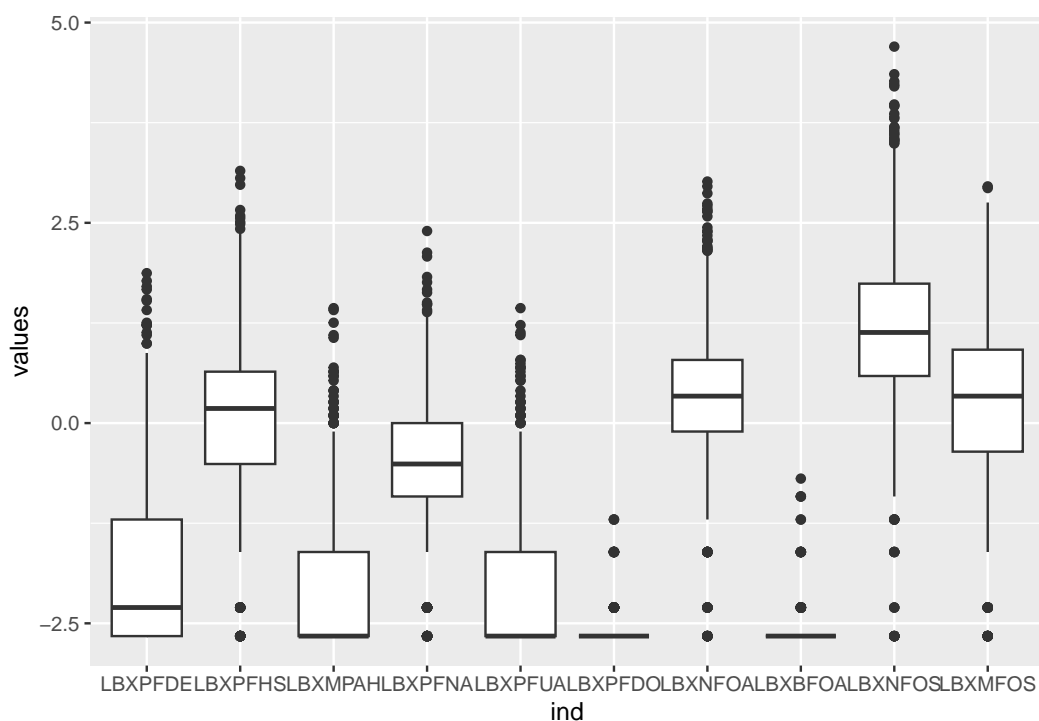


Figure 6.8: PFAS_I boxplot with log values for 10 data columns.

6.4 Merging data files

NHANES data are split into multiple files to provide flexibility and modularity in the choice of data. This makes the data easier to handle in small portions rather than a huge, single data file. On the other hand it is often necessary to merge one or more data file in order to obtain all of the data required for an analysis so that all of the data for an individual scattered among multiple files be found on a **single row** in the new, combined data file.

All that is required and necessary to merge data is at least a single column with the same name. All NHANES data pertinent to individuals (excluding special files with pooled data) start with the SEQN column that identify individuals with a unique number.

PFAS may disrupt lipid regulation and gathering data that have both PFAS and

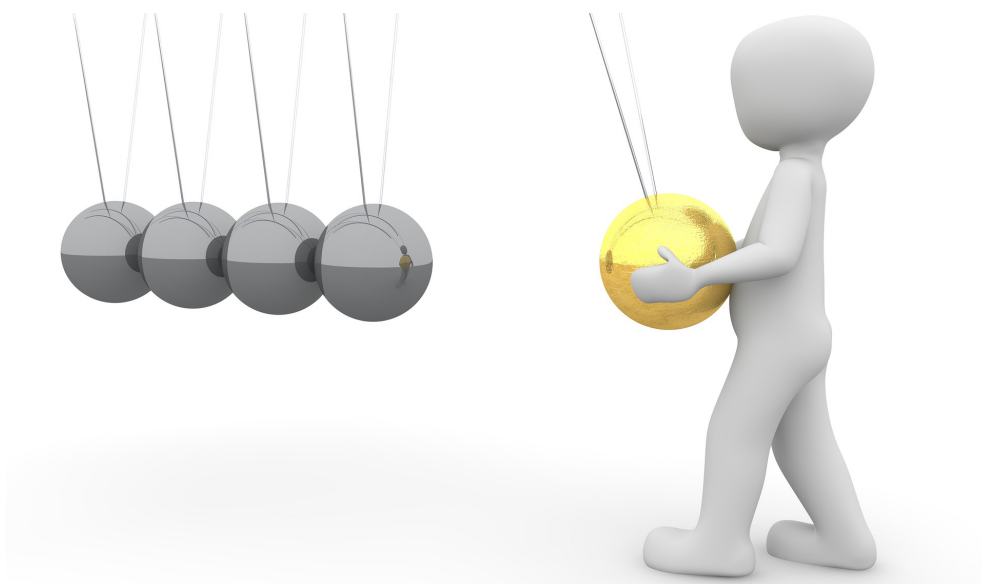


Figure 6.9: Combining NHANES data into a single file is necessary for detailed analysis.

cholesterol or triglyceride data would help in a study. As an example we'll merge the PFAS_I file with another file containing cholesterol data. There are 3 files containing cholesterol data for 2015-2016, separated by type.

Table 6.3: Cholesterol (Total, HDL, LDL & triglycerides) in 2015-2016 NHANES

Data File Name	Doc File	Data File	Date Published
Cholesterol - High-Density Lipoprotein (HDL)	HDL_I Doc	HDL_I Data [XPT - 189.2 KB]	September 2017
Cholesterol - Low - Density Lipoprotein (LDL) & Triglycerides	TRIGLY_I Doc	TRIGLY_I Data [XPT - 151.2 KB]	January 2019

Data File Name	Doc File	Data File	Date Published
Cholesterol - Total	TCHOL_I Doc	TCHOL_I Data [XPT - 189.2 KB]	September 2017

The number of observation within each file is different with 8021 (765 missing) for both *total cholesterol* and *HDL* files, and only 3191 (468 missing) for *LDL + Triglyceride*. The latter is probably due to NHANES method to use subsets of a population as a cost saving method.

The PFAS_I data contains 2170 entries with 177 missing. During the merging of the data, only rows that have a corresponding SEQN entry will be saved.

For simplicity we'll use the total cholesterol data. It contains only three columns but none of them are “*comment*” columns:

- SEQN - Respondent sequence number
- LBXTC - Total Cholesterol (mg/dL)
- LBDTC SI - Total Cholesterol (mmol/L)

The LBDTC SI variable was derived from LBXTC: *The total cholesterol in mg/dL (LBXTC) was converted to mmol/L (LBDTC SI) by multiplying by 0.02586.*

The file has to be downloaded into a new user-defined R object as we did with the PFAS data:

```
# Download NHANES TCHOL_I 2015-2016 data to temporary file
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/TCHOL_I.XPT",
              tf <- tempfile(),
              mode="wb")

# Create Data Frame PFAS_I From Temporary File
TCHOL_I <- foreign::read.xport(tf)
```

```
class(TCHOL_I)
```

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

6.4.1 Merge() function

The `merge()` function has many optional parameters that permit variations of the merging. Details on all options for this function are available as `help(merge)` or simply `?merge`.

All we want for now is specify a common column, in our case `SEQN` and just keep the entries that exist for both data files. Therefore the total number of entries cannot be larger than that of `PFAS_I`, that is 2170.

The specification of the chosen columns is done with the `by.x` and `by.y` options. In our case the chosen name will be the same. But if by chance the name was different in the 2 files, specifying the name of the column in this way would still work.

So, let's combine the 2 files:

```
# Merging PFAS and total cholesterol TCHOL data frames  
M1 <- merge(PFAS_I, TCHOL_I, by.x = "SEQN", by.y = "SEQN")
```

```
class(M1)
```

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

```
dim(M1)
```

```
[1] 2170  24
```

Since TCHOL_I is the second argument (y in documentation) its columns are added at the **end** of PFAS_I. Inverting the arguments would place the TCHOL_I columns at the beginning. However, in all cases the SEQN column remains the first column.

We can check the name of just a few columns, for example with:

```
colnames(M1)[c(1,20:24)]
```

```
[1] "SEQN"      "LBDNFOSL" "LBXMFOS"  "LBDMFOSL" "LBXTC"    "LBDTCISI"
```

The last 2 names are indeed the code names found in the documentation for TCHOL_I.

We now have merged 2 data files using the unique number for each individual. The process would be the same to add more data, for example adding the other cholesterol data files for HDL and LDL+Triglycerides. But this can be done as an exercise if it proves useful.

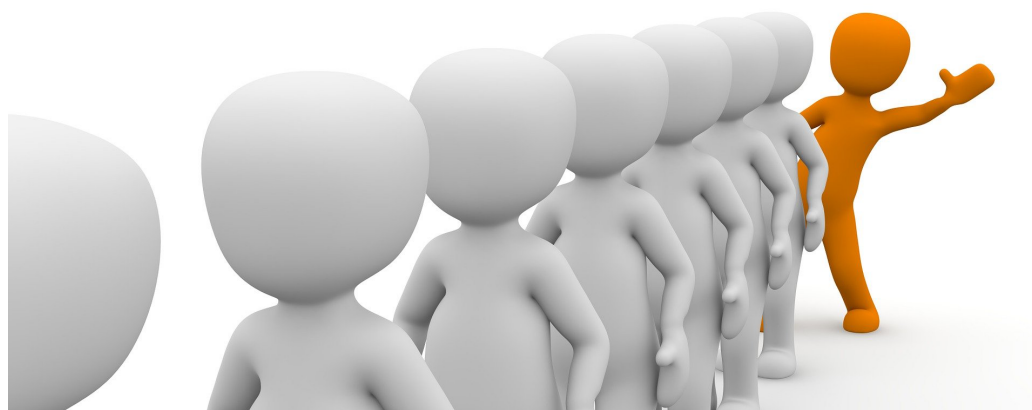


Figure 6.10: We have combined NHANES data for each individual from 2 separate files.

6.4.2 Merging demographics data

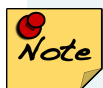
Most studies are set to compare and analyze data for different population. Therefore adding the “demographics” data file would be useful.



Study: Merge your data with the demographics file if your study requires it.

The procedure would be exactly the same using the `merge()` function through the common SEQN column for individuals.

The code to download the demographics data was the example we saw earlier in section [6.2.1](#).



If you are struggling with downloading data file or creating a master file with multiple merges check code in Appendix [D](#) (using NHANES code with foreign package) and [D.4](#) (using haven package.)

Chapter 7

Creatinine adjustment

Many of the NHANES samples are derived from urine sample analysis. In order to compensate for most variations between individuals it is often necessary to proceed to an adjustment with the level of *creatinine* a metabolite that has a rate of excretion rather constant and can serve as an indicator of urine dilution.

In this chapter:

- Creatinine adjustment rationale
- Download and explore creatinine data
- Converting weight/volume units
- Merging and reducing data
- Computing and saving creatinine adjustment

An older NHANES document had information about this process:

The concentrations of environmental chemicals per whole weight of serum are also on the laboratory file and can be used for comparison with other published studies that have investigated these chemicals.

The current NHANES urine collection protocol provides ‘spot’ urine samples because these are collected at different times of the day (depending on the examination session) and only one specimen is collected from each survey participant. The laboratory measures of environmental chemicals in urine are provided on the data files as concentrations per volume of urine. Each data set for environmental chemicals measured in urine, also includes a variable for urinary creatinine concentration.

Urine dilution may vary markedly from person to person, time to time, and because of other conditions, including fluid consumption, physical workload, and health. Creatinine is produced as a result of muscle metabolic processes, and excreted from the body at a fairly constant rate (though extreme diets may affect urine creatinine levels). The effect of urinary dilution can be accounted for by determining the amount of the environmental chemical per amount of urinary creatinine in a given volume of urine.

The equation for creatinine adjustment is:

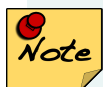
Analyte concentration per gram of creatinine =

$$\frac{\text{Concentration of environmental chemical in urine (wt/vol)}}{\text{Concentration of creatinine in urine (wt/vol)}}$$



WARNING Creatinine is related to lean body mass and renal function of individuals, and varies by age, gender, and race/ethnicity group.

It is recommended that one compare the creatinine-corrected environmental chemical concentrations among individuals of similar demographic groups rather than the whole population because urinary creatinine levels differ according to age, gender, and race/ethnicity. Alternatively, multiple regression analyses can be conducted using urinary creatinine as an independent variable (in addition to variables for age, gender, and race/ethnicity), so that the environmental chemical concentrations comparisons can be based on adjustment for urinary dilution and demographic differences.



The current NHANES web site no longer has this information in this format. It is available as a copy in a page titled “[Using Blood Lipid or Urine Creatinine Adjustments in the Analysis of Environmental Chemical Data](#)”¹ as the link within that page titled “*Key Concepts about Blood Lipid or Using Urine Creatinine Adjustments of Environmental Chemical Data*”²

I have archived both pages at [archive.org](#) to preserve the availability of these pages. Searching with the original links within the archival site will retrieve these original files.

7.1 Creatinine data

The 2015-2016 document file is listed as ALB_CR_I.doc as it also contains information for albumin.

Table 7.1: NHANES 2015-2016 albumin/creatinine data

Data File			
Name	Doc File	Data File	Date Published
Albumin & Creati- nine - Urine	ALB_CR_I Doc	ALB_CR_I Data [XPT - 539.8 KB]	Updated June 2019

The data file contains 8608 data points (328 missing.)

The listed codes within ALB_CR_I.doc are:

Table 7.2: Codes for albumin and creatinine ALB_CR_I file

Code	Description
SEQN	Respondent sequence number
URXUMA	Albumin, urine (ug/mL)
URDUMALC	Albumin, urine comment code
URXUMS	Albumin, urine (mg/L)
URXUCR	Creatinine, urine (mg/dL)
URDUCRLC	Creatinine, urine comment code
URXCRS	Creatinine, urine (umol/L)
URDACT	Albumin creatinine ratio (mg/g)

7.1.1 Downloading, merging PFAS and creatinine

As an example we'll continue working on the PFAS_I urine metabolite and therefore we need to combine it with the creatinine data, again with the SEQN individual column. `index{urine metabolite}`

```
# Download NHANES ALB_CR_I 2015-2016 data to temporary file
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/ALB_CR_I.XPT",
              tf <- tempfile(),
              mode="wb")
# Create Data Frame ALB_CR_I From Temporary File
ALB_CR_I <- foreign::read.xport(tf)
```

Once the ALB_CR_I data frame is created we can merge it with the PFAS_I data frame.

```
# Merging PFAS and total cholesterol TCHOL data frames
M2 <- merge(PFAS_I, ALB_CR_I, by.x = "SEQN", by.y = "SEQN")

dim(M2)
```

[1] 2170 29

7.2 Analyte measurement units

The ratio of analyte to creatinine has to be performed using the same unit of weight by volume as detailed in the formula we have seen. Therefore we must look at the unit values provided in the HTML .DOC pages and find:

- LBXMFOS - Sm-PFOS (ng/mL)
- URXUCR - Creatinine, urine (mg/dL)

The two weight/volume are not one the same scale so we need to convert from one to the other or to a common version. Creatinine is abundant and expressed in milligrams per deciliter (mg/dL). PFAS is on a smaller scale in nanogram per milliliter.

- $1ng = 10^{-9}gram$
- $1mg = 10^{-3}gram$
- $1ml = 10^{-3}liter$
- $1dL = 10^{-1}liter$

Hence $1ng/ml = 10^{-9}g/10^{-3}l$
and $1mg/dL = 10^{-3}g/10^{-1}l$

To avoid having too small values in decimals, the best option may be to convert the creatinine values to the same unit as the PFAS knowing that $1mg/dL = 10000ng/ml$ as can be deducted from the ratio of the units. Therefore in a conversion in this context would need to add a factor of 10^4 for the creatinine current values:

$$\frac{Analyte}{Creatinine * 10^4} = \frac{Analyte}{Creatinine} * 10^{-4}$$

7.3 Reduced set

The new M2 merged data contains 29 columns, but we only need to keep a smaller number of them for this demonstration: SEQN as well as the column for the sum of PFAS (LBXMFOS) and those of creatinine. But now we know that we need to use the creatinine column data URXUCR with *mg/dL*.

We can create a subset data frame by choosing these columns as we have seen before (section 5.3.)

```
# new subset containing SEQN, PFAS sum and creatinine columns
PFAS_CRE <- M2[,c(1,21,26)]
class(PFAS_CRE)
```

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

```
dim(PFAS_CRE)
```

```
[1] 2170    3
```

7.4 Computing Analyte / Creatinine ratio

We now need to do 2 things:

1. compute the ratio value from the established formula
2. add this as a new column to PFAS_CRE

The computation will take values from column 2 (LBXMFOS) and divide it by the values of column 3 (URXUCR) and multiply that with 10^{-4} . This could be written as:

$$ratio = \frac{LBXMFOS}{URXUCR} * 10^{-4}$$

We'll write the resulting number, for each row, in a column called `RATIO` (as an example) adding a column to the existing user-defined R object that we could call `PFAS_CRE`. All we need to do is assign the name of the new column using the sub-setting method using a `$` sign: `PFAS_CRE_R$RATIO`. The column will be created on that demand and populated with the values that are computed.

```
# add new column RATIO with computer values
PFAS_CRE$RATIO <- with(PFAS_CRE, (LBXMFOS / URXUCR) * 10^-4)
# check results
head(PFAS_CRE)
```

	SEQN	LBXMFOS	URXUCR	RATIO
1	83736	0.6	315	1.904762e-07
2	83745	0.8	178	4.494382e-07
3	83750	1.9	81	2.345679e-06
4	83754	5.4	148	3.648649e-06
5	83762	0.4	317	1.261830e-07
6	83767	1.0	65	1.538462e-06

We can check the distribution of values with boxplot and histogram. This time we can use log base 10 with function `log10()` as it may better reflect the negative powers of 10 in the data. (Note: the code below is indented for easier reading.)

```
par(mfrow=c(1,2))
boxplot(log10(PFAS_CRE$RATIO),
        main = "PFAS/Creat. log10 ratio")

hist(log10(PFAS_CRE$RATIO),
     freq=FALSE,
     breaks = 50,
     main = "Density Histogram ",
     xlab = "log10 of PFAS/creatinine ratio")
```

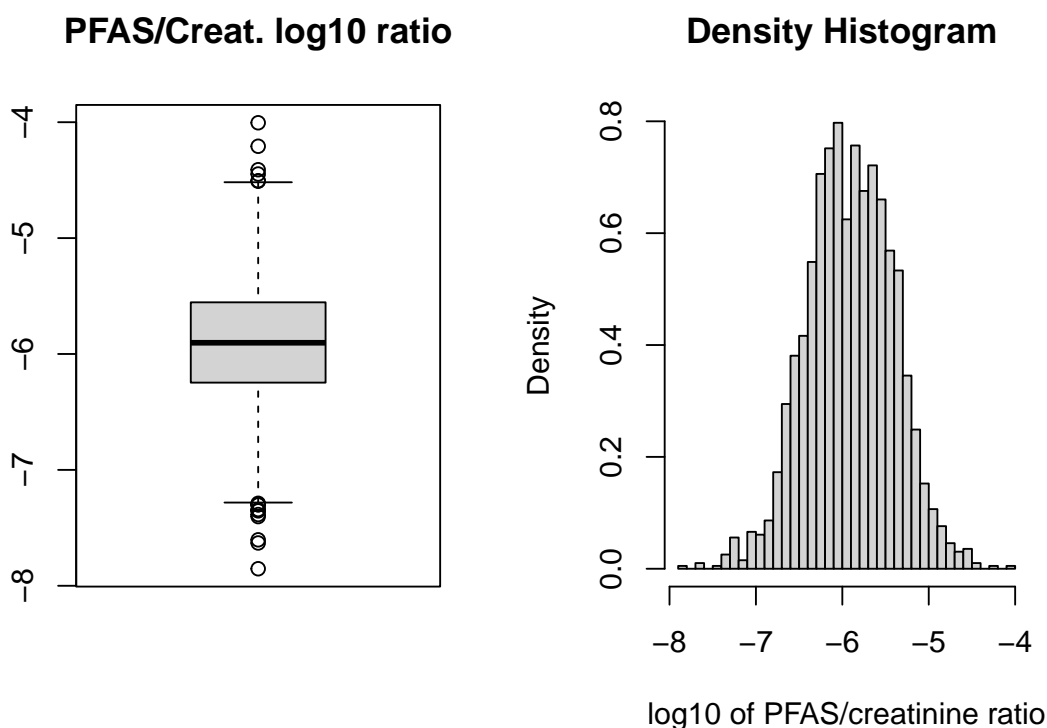


Figure 7.1: Boxplot and histogram of log10 transformation of PFAS sum data after creatinine adjustment.

```
par(mfrow=c(1,1))
```

7.5 Exposure - Outcome

Background³: “The key to understanding the environmental fate and transport of PFAS compounds is their surface-active behavior. The fluorinated backbone is both hydrophobic (water repelling) and oleophobic/lipophobic (oil/fat repelling) while the terminal functional group is hydrophilic (water loving). This means that PFAS compounds tend to partition to interfaces, such as between air and water with the fluorinated backbone residing in air and

³[https://clu-in.org/contaminantfocus/default.focus/sec/Per-_and_Polyfluoroalkyl_Substances_\(PFASs\)/cat/Chemistry_and_Behavior/](https://clu-in.org/contaminantfocus/default.focus/sec/Per-_and_Polyfluoroalkyl_Substances_(PFASs)/cat/Chemistry_and_Behavior/)

the terminal functional group residing in water. The PFAS partitioning behavior also is affected by the alkyl chain length and the charge on the terminal functional group. In general, PFASs with shorter alkyl chain length are more water soluble than those with longer lengths."

One question that may arise is whether PFAS compounds could accumulate in the fat tissue in the body. We can explore this option thanks to the "Body Measures (BMX_I)" NHANES data, at least on a broad sense. But we first need to download the file:

```
#BMX_I - 1.9 MB
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/BMX_I.XPT",
             tf <- tempfile(),
             mode="wb")
BMX_I <- foreign::read.xport(tf)
# Dimensions
dim(BMX_I)
```

```
[1] 9544  26
```

For this test we'll only keep the BMXBMI column "**Body Mass Index (kg/m**2)**" which is the 11th column. We also need to keep the common SEQN column. We can merge these 2 columns with the dataset containing the creatinine adjustment of the PFAS sum data we just made earlier:

```
PFAS_CRE_BMI <- merge(PFAS_CRE, BMX_I[, c(1,11)], by.x = "SEQN", by.y = "SEQN")
```

We can take a quick look at the BMI *distribution* with original and log10 values in a histogram form:

```
par(mfrow=c(1,2))
with(PFAS_CRE_BMI,hist(BMXBMI, breaks = 30))
with(PFAS_CRE_BMI,hist(log10(BMXBMI), breaks = 30)) ; par(mfrow=c(1,1))
```

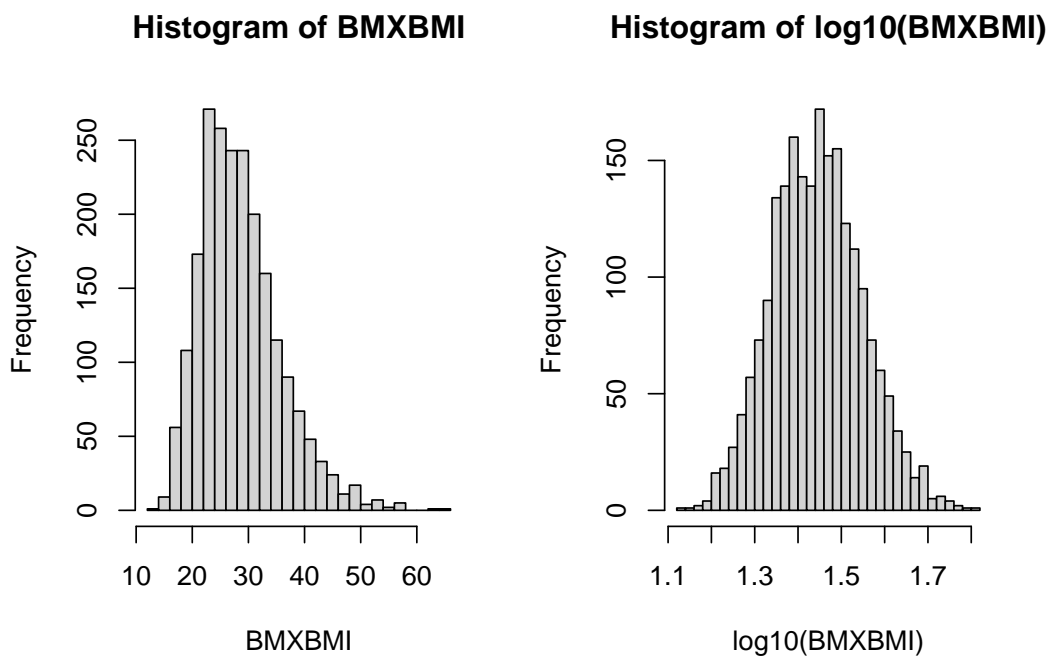


Figure 7.2: Histogram of BMI values and log10 values.

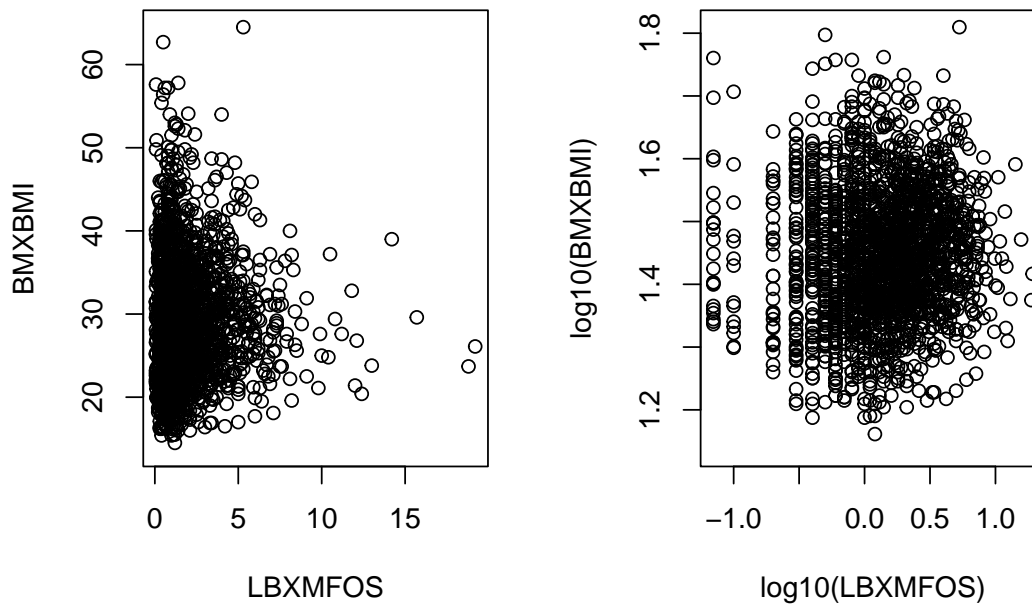
We can note that, as would be expected, the logged values have a more “bell-shaped” distribution.

We can now create a simple plot showing PFAS values as a function of BMI. We can plot both the original values as well as the log values.

```
par(mfrow=c(1,2))
with(PFAS_CRE_BMI, plot(BMXBMI ~ LBXMFOS))
with(PFAS_CRE_BMI, plot(log10(BMXBMI) ~ log10(LBXMFOS))); par(mfrow=c(1,1))
```

We can add a simple linear regression line for both plot as we have seen previously (section 5.7.)

```
lm1 <- with(PFAS_CRE_BMI, lm(BMXBMI ~ LBXMFOS))
lm2 <- with(PFAS_CRE_BMI, lm(log10(BMXBMI) ~ log10(LBXMFOS)))
# print out values
lm1; lm2
```

Figure 7.3: Histogram of BMI values and log₁₀ values.

```
Call:
lm(formula = BMXBMI ~ LBXMFOS)
```

```
Coefficients:
(Intercept)    LBXMFOS
  28.5857      0.1376
```

```
Call:
lm(formula = log10(BMXBMI) ~ log10(LBXMFOS))
```

```
Coefficients:
(Intercept) log10(LBXMFOS)
  1.4458      0.0154
```

```
abline(lm1, col="blue", lwd=3)
abline(lm2, col="blue", lwd=3)
```

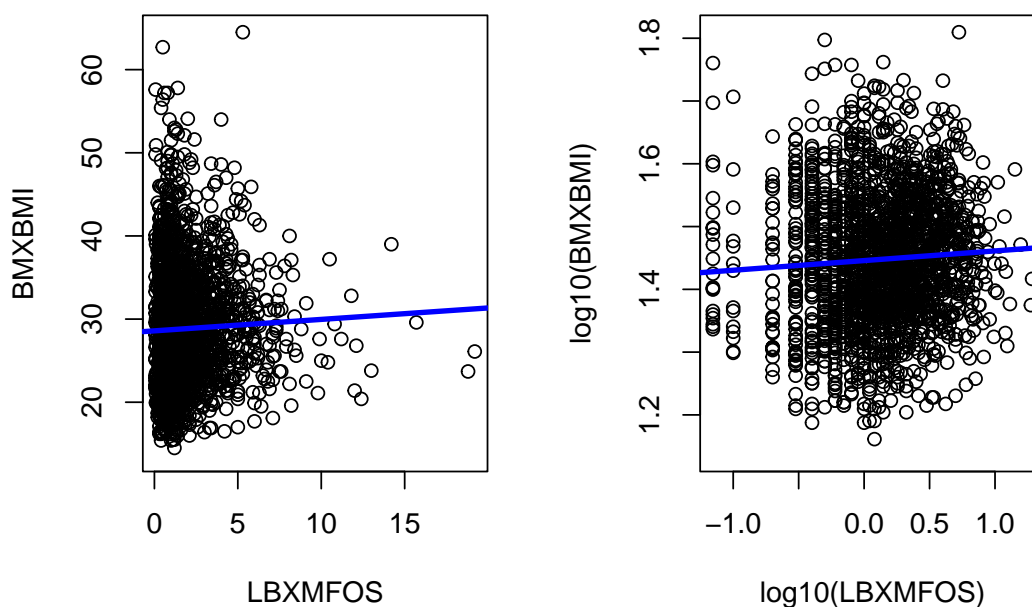


Figure 7.4: Histogram of BMI values and log₁₀ values with added linear regression.

The slope is faint in both cases and indeed the slope values found in the coefficients records (section 5.7) are low at about 0.138 for the normal values and 0.015 for the log values.

We can confirm that the correlation is faint by calculating the *Pearson correlation factor* which is the default method for the `cor()` function. The help tells us that we need to add `use="complete.obs"` to the command to avoid an NA result or errors due to missing values.

```
with(PFAS_CRE_BMI, cor(BMXBMI, LBXMFOS, use="complete.obs"))
```

```
[1] 0.03611288
```

```
with(PFAS_CRE_BMI, cor(log10(BMXBMI) ,log10(LBXMFO5), use="complete.obs"))
```

```
[1] 0.05981631
```

The values are indeed very small in accord with the very flat linear regression.

7.5.1 Illusions

It can be noted that if the plot is stretched horizontally, the line will look even flatter. This can be seen if we change the plotting position with `par(mfrow=c(2,1))` instead of `par(mfrow=c(1,2))`.

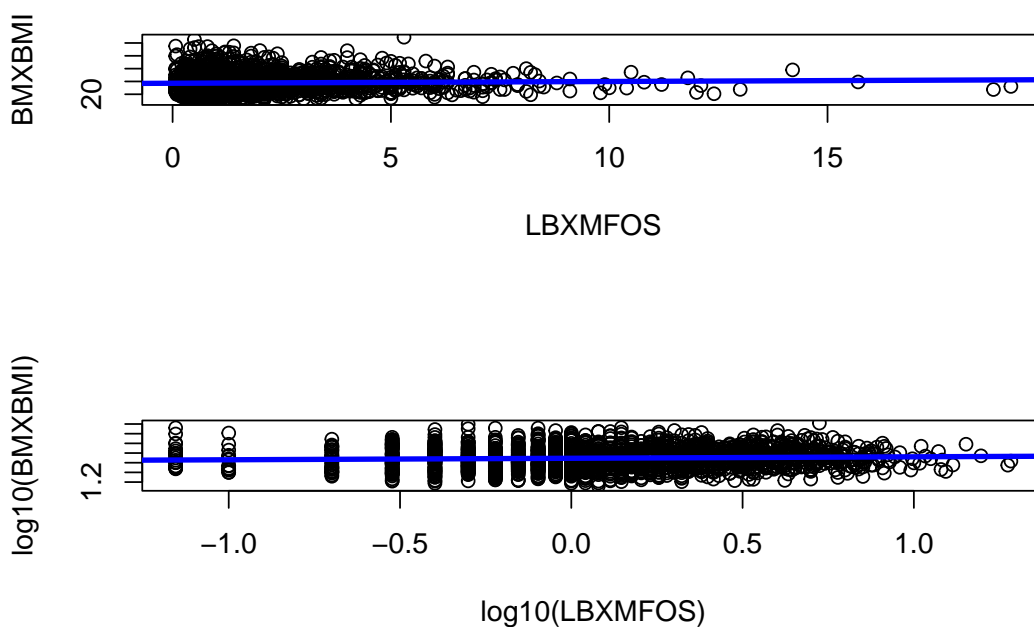


Figure 7.5: Stretching horizontally makes the linear regression appear more horizontal

7.5.2 qplot version

The `qplot()` version can be created by getting inspired to what was done in section 5.8.2.

```
`geom_smooth()` using formula = 'y ~ x'
```

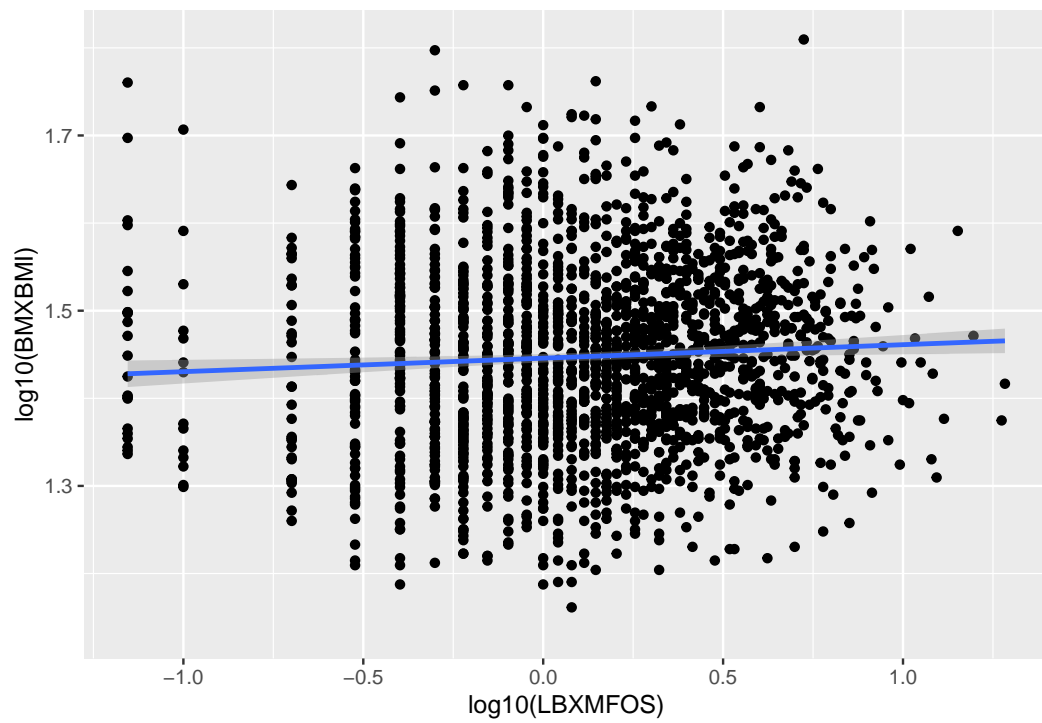


Figure 7.6: Histogram of BMI \log_{10} values, linear regression (blue) and standard error (gray.)

7.6 Creating a master data file



See Appendix D for a complete set of code to *download*, *merge* and *save* the master file.

We have learned how to combine two NHANES data files, but it is could useful to be able to merge more data into a large repository or “master” file from which smaller datasets can be created.

The `merge()` function can automatically recognize identical columns which will make the merging easier as it will not be necessary to use the `by.x` commands. In fact, doing so will *prevent* the `merge()` function to recognize columns that are identical and this will result in duplicate columns. However, to make the file a “master” we would need to keep all rows, and this is accomplished by adding the `all.x` option (see `?merge`.) Below we’ll download and merge more datasets, including some that we already have downloaded. This should just be a review.

We’ll add the file `BMX_I` for body/mass index and other body measurements as well as the demographic file `DEMO_I` and the `PFAS_I`,

```
# DEMO_I - 3.6 MB
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/DEMO_I.XPT",
             tf <- tempfile(),
             mode="wb")
DEMO_I <- foreign::read.xport(tf)
# Dimensions
dim(DEMO_I)
```

```
[1] 9971  47
```

We previously downloaded (or learned how to download) the albumin/creatinine file (`ALB_CR_I`) and that of total cholesterol `TCHOL_I`.

Let’s merge all 4 keeping all rows starting with `DEMO_I` so that it is on the left hand side.

```
Master1 <- merge(DEMO_I, BMX_I, all.x=TRUE)
Master2 <- merge(Master1, PFAS_I, all.x=TRUE)
Master3 <- merge(Master2, TCHOL_I, all.x=TRUE)
Master4 <- merge(Master3, ALB_CR_I, all.x=TRUE)
# dimensions
dim(Master1) ; dim(Master2); dim(Master3); dim(Master4)
```

```
[1] 9971 72
```

```
[1] 9971 93
```

```
[1] 9971 95
```

```
[1] 9971 102
```

The process would be the same to add more data file.

To save the master file into a comma separated file (.csv) use the `write_csv()` function of `dyplr` which is “about twice as fast as `write.csv()`, and never writes row names.” (See chapter 8 section 8.3 and chapter ??10 as we have not yet studied that package at this point.)

For example to save the data frame `Master4` in the current directory:

```
write_csv(Master4, "Master4.csv")
```

Using base R we would write similarly:

```
write.csv(Master4, "Master4.csv")
```

This would allow to read the data again without having to go through the process of creating the combined dataset.

Chapter 8

Tidyverse: another R Universe

Tidyverse *exists* and it is a *dialect* of R said **Hadley Wickham**¹ at the [RStudio::Conf 2017](#)² about this single package that is an umbrella name for a *coherent system of [multiple] packages for data manipulation, exploration and visualization that share a common design philosophy*.³

In this chapter:

- Tidyverse goal
- Tidyverse packages
- Magrittr: pipes
- dplyr: pipeline demonstration



Hadley Wickham's notes from the 2017 conference⁴ about **Tidyverse**:

¹Hadley Wickham is the Chief Scientist at RStudio, a member of the R Foundation, and Adjunct Professor at Stanford University and the University of Auckland. He builds tools (both computational and cognitive) to make data science easier, faster, and more fun. He develops packages for data science.

²<https://rstudio.com/resources/rstudioconf-2017/>

³<https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/>

1. It exists
2. It has a web site
3. It has a package
4. It has a book

Perhaps more importantly:

Goal: Solve complex problems by combining simple, uniform pieces.

The fundamental philosophy in Tidyverse is to separate **commands** and **queries**

A **commands** function **performs** an action

A **query** function **computes** a value

Examples:

Command: `print()`, `plot()`, `write.csv()`, `<-`

Query: `summary()`, `sqrt()`

Tidyverse is a package that installs a series of other packages. The fact that “*it has a package*” means that all packages composing Tidyverse can be installed with the single command:

```
install.packages("tidyverse")
```

instead of:

```
install.packages(c(
  "broom", "dplyr", "feather", "forcats", "ggplot2", "haven",
  "httr", "hms", "jsonlite", "lubridate", "magrittr",
  "modelr", "purrr", "readr", "readxl", "stringr", "tibble",
  "rvest", "tidyr", "xml2"
))
```



Study: Watch the first 30 minutes of Hadley Wickham’s keynote presentation at RStudio::Conf 2017 - February 10, 2017

[Data Science in the Tidyverse⁵](#)

In the next sections we’ll explore the packages that may be useful for analysis of tabular data such as NHANES data.

8.1 Magrittr - pipe and pipelines

In English a “*pipe*” can designate an object to smoke tobacco or house plumbing. In both cases it can be viewed as a hollow cylinder.

In computing a “*pipe*” is a method to create a *data stream* in the memory of the computer without the need to create intermediary files or R objects. In Unix the pipe is represented by a vertical bar: | but in R the pipe is represented by:

$$\%>\%$$

In English, when reading code, it is useful to replace the pipe with ***and then*** to better understand the successive passage of each step or function.

Once started with data from an *object* the resulting stream of data can be modified by a function and then passed on to the next function, and then the next etc. The flow of data can be conceptualized as a flow of water going through pipes until it exits (figure 8.1.)

The stream of data can be modified by successive function, each passing the data stream along the “pipe” to the next function until the final result (8.2.)

There can be more than one operation until the final result.

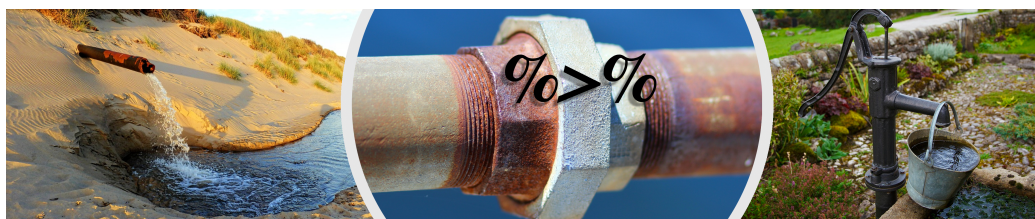


Figure 8.1: Imagining the data stream as a flow of water in pipes.

object `%>%` operation() \longrightarrow result

Figure 8.2: The pipe operator is the conduit for the data stream.



The command-query distinction is useful for pipes

The body is made up of **queries**

Every pipe is ended by a **command**

The use of *pipe* can help create *pipelines* to manipulate, convert, gather, select data in a way that ends in a final result without the need of intermediate items, as all happens while “in transit” within the conduits.

The pipe is widely used in the context of Tidyverse but it is not restricted to that Universe and can find its uses in writing R commands.



Study: Watch the 25 min RStudio::Conf 2017 by Bob Rudis:

[Writing Readable Code with Pipes⁶](#)



The name of the package is derived as a reference to the famous surrealist painter [René Magritte](#) 1929 image “[this is not a pipe](#)” as an image is not the object itself. This image is now at the Los Angeles County Museum of Art.

8.2 Tibble

A “tibble” is a data frame, but a modern reimagining of the `data.frame` class.
{tibble!data frame}



From the Tidyverse Tibble web page^a: *Tibbles are data.frames that are lazy and surly: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). This forces you to confront problems earlier, typically leading to cleaner, more expressive code. Tibbles also have an enhanced `print()` method which makes them easier to use with large datasets containing complex objects.*

^a<https://tibble.tidyverse.org/>

As far as we are concerned we do not have to worry about that as Tidyverse packages work fine with data frames. We'll just see the word “tibble” appear when working with the Tidyverse functions and that's simply what it is.

One difference in the print out of a table of data from a data frame in a *tibble* form is that we'll see the data type printed under the column name such as `<chr>` for character column, `<int>` for integers and `<dbl>` for “double-precision decimal number.”

Trivia

The Tibble logo font character for letters T and E are very close in shape (but not the B) but could the tibble name also be related to the famous sweet “tribble” creature on the original Star Trek. Or is it a New Zealander way of pronouncing “table”? Who know? (perhaps H W does?)



Figure 8.3: Is the Tibble logo a hint on Star Trek?

8.3 dplyr - overview



From dplyr.tidyverse.org/:

*dplyr is a grammar of data manipulation, providing a consistent set of **verbs** that help you solve the most common data manipulation challenges.*

`select()` picks variables based on their names. (columns)

`filter()` picks cases based on their values. (rows)

`arrange()` changes the ordering of the rows.

`mutate()` adds new variables that are functions of existing variables.

`summarise()` reduces multiple values down to a single summary.

These all combine naturally with `group_by()` which allows you to perform any operation “*by group*”.

8.3.1 Demo 1: all together pipeline

Before we go into details of the various **verbs** that make dplyr powerful, let's first create a *pipeline* as an example of the **power** of the Tidyverse methods: with **one series of commands and queries** we'll recreate one of the plots of figure 7.2 “*just like that!*” with no need of any intermediate steps or temporary objects.

We'll start with our master file Master4 “injected” into the pipeline and then we'll:

- *select* specific columns (automatic subset)
- *filter out* rows that have NA
- *compute* the RATIO for creatinine ajustment (*mutate*)
- *plot* the data with `qplot()` and include automatic linear regression.

Now here's the code - discussed further below:

First, we need to make sure that tidyverse is loaded:

```
library(tidyverse)
```

Then we run the pipeline:

```
# pipeline demo 1
Master4 %>%
  select(SEQN, LBXMFOS, URXUCR, BMXBMI) %>%
  filter(!is.na(LBXMFOS)) %>%
  # head() %>%
  mutate (RATIO = (LBXMFOS/URXUCR)*10^-4) %>%
  qplot(log10(RATIO), BMXBMI, data = ., geom = c("point", "smooth"))
```

```
`geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```

Here are a few more details about the code, and let's see if we follow the *The command-query distinction useful for pipes*

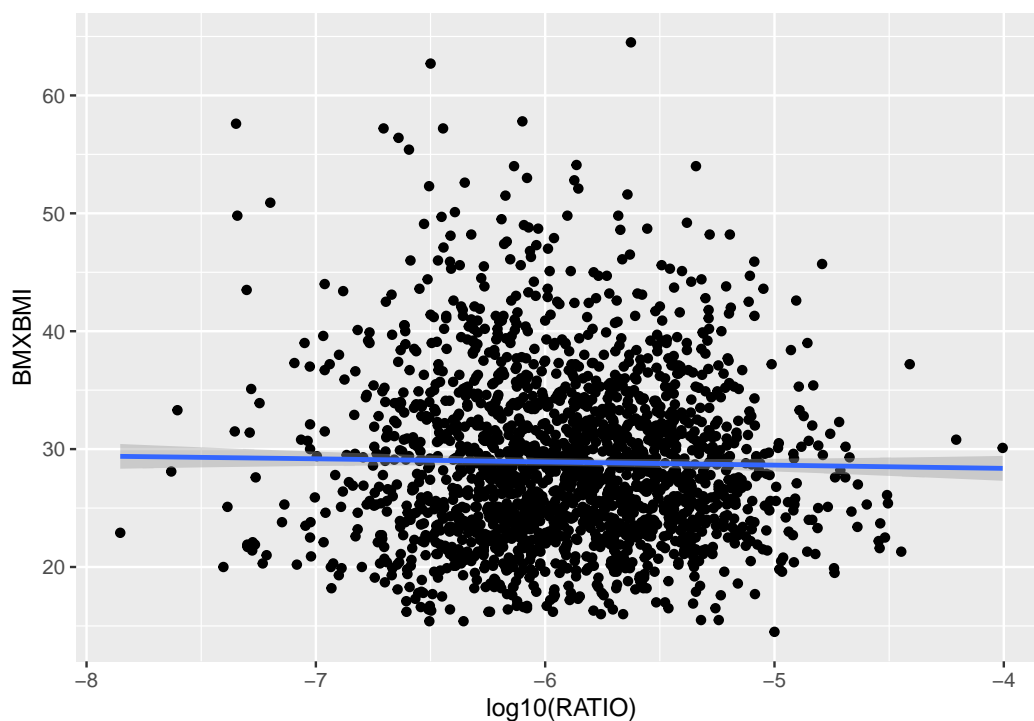


Figure 8.4: A pipeline to recreate scatter plot of BMI values as a function of \log_{10} RATIO creatinine adjustment for the sum of PFAS data column LBXMFO5.

- The body is made up of **queries**
- Every pipe is ended by a **command**

But what about the beginning?

The beginning of the pipe needs to start the “injection” of data. In the example we started with Master4 which is a very large dataset:

Master4 %>%: the *implied* function here is `print()` which is a **command** performing an action.

However, we could also have started with:

`select(Master4, SEQN, LBXMFO5, URXUCR, BMXBMI) %>%:` in this case the Master4 data is within the **query** function `select()`.



Figure 8.5: Data is first injected in the pipeline (Hydroelectric power station, Huanza, Peru.)

But in both cases we have *data starting to stream* down the pipeline.

`filter(!is.na(LBXMFO5)) %>%` uses a logical operator (Appendix B.4) to remove the rows that have NA within the LBXMFO5 column. `!` is *negating* the next statement is `.na` that checks if there is an NA value. This can be read in English as “*is not NA*”. This is a **query**.

`# head()` `%>%` is commented out and can be used for testing and just show the first 6 lines of data passing through. It does not hamper the pipeline to have a line commented out. Actual comments of explanation could therefore be included along the pipeline. `head()` is a **command** that would end the pipeline for testing.

`mutate (RATIO = (LBXMFO5/URXUCR)*10-4) %>%` computes the creatinine adjustment as was detailed in section 7.4 using the same formula. A new column named RATIO will be created to store the computation, just as it was done in base R. This is a **query**.

`qplot(log10(RATIO), BMXBMI, data = ., geom = c("point", "smooth"))` will make the plot, with default regression curve (to compute a linear model line see section 7.5.2. The `geom` portion could be removed to just get the points.)

The option `data = .` may appear “strange” and we have not seen this yet. Since we are in a pipeline, the data is symbolically represented by the dot `.` which is

useful, otherwise how would we specify where the data came from?

Did we follow the pipe rules? Overall **yes!**

`qqplot` is first a **command** that will perform the action of creating a plot. However, this function as well as its bigger version `qqplot2()` were created before the Tidyverse, and does not adhere completely to those rules as **internally** there will be some computation (hence **query**) to create the regression line or curve. However, one could argue that since the plotting of the line could be the final step, that would be the result of a **command**.

Chapter 9

Intermission: data wrangling

NHANES datasets are “curated” and are created following standard practice resulting in datasets listed in tabular data formatted in a way well suited for R.

This section is here as an “intermission” in the form of a lecture by Garrett Grolemund, Data Scientist and Master Instructor at RStudio, split into 4 YouTube videos. The **whole** four parts are listed here, but the most important for treating NHANES data would be **Part 3** about the `dplyr` Tidyverse package. Part 1 would review what was learned in the previous chapter (8) and Part 2 is about the `tidyr` package that helps reformat the data, a very useful tool but not really necessary for NHANES data.



Description of the RStudio videos:

Data wrangling is too often the most time-consuming part of data science and applied statistics. Two tidyverse packages, `tidyr` and `dplyr`, help make data manipulation tasks easier. These videos introduce you to these tools. Keep your R code clean and clear and reduce the cognitive load required for common but often complex data science tasks.

Table 9.1: Lectures on data wrangling: Tidyverse `tidyr` and `dplyr` packages.

Title	Link	Time
Part 1: What is data wrangling? Intro, Motivation, Outline, Setup	https://youtu.be/jOd65mR1zfw	8:26
Part 2: Tidy Data and <code>tidyr</code>	https://youtu.be/1ELALQlO-yM	17:36
Part 3: Data manipulation tools: <code>dplyr</code>	https://youtu.be/Zc_ufg4uW4U	19:34
Part 4: Working with Two Datasets: Binds, Set Operations, and Joins	https://youtu.be/AuBgYDCg1Cg	7:23

9.1 Part 3 here

HTML version has Part 3 embedded here:

Pt. 3: Data manipulation tools: `dplyr` https://youtu.be/Zc_ufg4uW4U

00:40 setup

02:00 - `dplyr::select`

03:40 - `dplyr::filter`

05:05 - `dplyr::mutate`

07:05 - `dplyr::summarise`

08:30 - `dplyr::arrange`

09:55 - Combining these tools with the pipe (Setup for the Grammar of Data Manipulation)

11:45 - dplyr::group_by

Chapter 10

dplyr - data manipulation

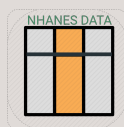
While base R has many tools that can do the job, dplyr and other Tidyverse packages can easily work together and allow the easy creation of pipelines to accomplish a task as was demonstration in the previous chapter [8.3.1](#).

In this chapter we'll explore a few dplyr *verbs*

- *select* to choose *columns*
- *filter* to check data on *rows*
- *arrange* to order data

- *mutate* to compute new values
- *summarise* to create condensed data
- *group_by* to select specific data

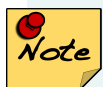
In addition we'll learn about some *conditional* selection within the *functions* described by these *verbs*.



We'll use the Master4 large data file that was created previously. (See all download and merge code in Appendix D if you need to recreated it.) Master4 is a merge, in order of the following datasets (links are to the documentation):

[DEMO_I](#), [BMI_I](#), [PFAS_I](#), [TCHOL_I](#), [ALB_CR_I](#)

10.1 selecting columns



To limit the output length most examples will be *piped in* the `head()` function to only print the column headers followed by 6 data lines. `%>% head()` may terminate the command but the pipeline can be extended further.

The *selection of columns* is easy and just requires the name of the data frame containing the data and specifying the column names that we want to keep for further use in an analysis. The total length (number of rows) would be printed, here limited by piping into `head()`.

```
select(Master4, SEQN, LBXMFOS, URXUCR, BMXBMI) %>% head()
```

	SEQN	LBXMFOS	URXUCR	BMXBMI
1	83732	NA	41	27.8
2	83733	NA	181	30.8
3	83734	NA	70	28.8
4	83735	NA	102	42.4
5	83736	0.6	315	20.3
6	83737	NA	64	28.6

We can note that in column LBXMFOS there are 5 NA, in the next section we'll see how to get rid of them.

Note that the square bracket subsetting also works, for example to select the first 7 columns we could write `select(Master4, 1:7)` which would be equivalent to `Master4[, 1:7]` in this case.

10.2 Filtering rows

The *filtering* of rows depends on the desired outcome. One step that is useful and often necessary is to remove the NA values. If we continue with the selected columns we can remove those with:

```
select(Master4, SEQN, LBXMFOS, URXUCR, BMXBMI) %>% head() %>%
  filter(!is.na(LBXMFOS))
```

	SEQN	LBXMFOS	URXUCR	BMXBMI
1	83736	0.6	315	20.3

Since the `head()` function was first, we are only filtering the first 6 rows, and 5 are therefore eliminated. If we move the `head()` function after, we'll select the first 6 rows that do not have any NA.

```
select(Master4, SEQN, LBXMFOS, URXUCR, BMXBMI) %>%
  filter(!is.na(LBXMFOS)) %>%
  head()
```

	SEQN	LBXMFOS	URXUCR	BMXBMI
1	83736	0.6	315	20.3
2	83745	0.8	178	25.0
3	83750	1.9	81	24.1
4	83754	5.4	148	43.7

5	83762	0.4	317	38.0
6	83767	1.0	65	26.3

We can use other *operator* (Appendix B) to filter the rows with *conditional statements*. For example we could ask to keep the BMI values below or equal to 25.0 and more, and at the same time removing NA values from chosen columns:

```
select(Master4, SEQN, LBXMFOS, URXUCR, BMXBMI) %>%
  filter(BMXBMI <= 25.0) %>%
  head()
```

	SEQN	LBXMFOS	URXUCR	BMXBMI
1	83736	0.6	315	20.3
2	83738	NA	100	18.1
3	83739	NA	25	15.7
4	83745	0.8	178	25.0
5	83746	NA	34	16.1
6	83748	NA	14	16.1

We could have the `filter()` function more than once within the pipeline, but we can also have multiple statements at the same time:

```
select(Master4, SEQN, LBXMFOS, URXUCR, BMXBMI) %>%
  filter(!is.na(BMXBMI),
         BMXBMI <= 25.0,
         !is.na(LBXMFOS),
         URXUCR == 25.0) %>%
  head()
```

	SEQN	LBXMFOS	URXUCR	BMXBMI
1	85253	2.8	25	24.2
2	86086	1.1	25	20.3

3	88829	0.5	25	20.1
4	89326	0.7	25	22.9
5	89399	1.1	25	24.0
6	90178	0.6	25	20.4

Note on style: writing each *conditional statement* on a separate line makes it easier to read and understand the code.



The dplyr function `drop_na()` would also remove NA from all rows, or specified rows.

10.3 Arrange data

For dplyr to *arrange* data is to sort data in order. The following example accomplishes many things at the same time and the “seed” makes the results always the same. To see just 6 lines of ordered data based on age, the data is first randomly sampled to keep only 100 rows with a dplyr function `sample_n()`. All rows in the age column `RIDAGEYR` without value are removed with `drop_na()`. Four columns are selected, and the age is filtered to keep only ages above 12. The data is then *arranged* (ordered) by the first (age) and then second column (creatinine) as designated. The data is first ordered by age and then by the second parameter.

```
set.seed(18) ;
Master4 %>%
sample_n(100) %>%
drop_na(RIDAGEYR) %>%
select(SEQN, URXUCR, BMXBMI, RIDAGEYR) %>%
filter(RIDAGEYR >= 12) %>%
```

```
arrange(RIDAGEYR, URXUCR) %>%
  head()
```

	SEQN	URXUCR	BMXBMI	RIDAGEYR
1	89664	114	16.1	12
2	91777	148	20.3	13
3	90183	330	26.2	13
4	92077	116	17.8	14
5	84253	144	20.6	14
6	92721	161	25.2	14

Note that the order in the pipe may be important, but the modularity of the code makes it easy to try if there is an effect. For example, since we are only looking at 6 values here, it makes no difference if the sampling of 100 rows occurs before or after the dropping of NA in the age column.

10.4 mutating data

As we explored in the previous demonstration (8.3.1) we can create a new column when computing data with the `mutate()` function. As an example we can compute a ratio of $\frac{height}{weight}$ as it is often a useful measure. In the `BMX_I` data we can find the relevant column names:

- `BMXWT` - Weight (kg)
- `BMXHT` - Standing Height (cm)
- `BMXBMI` - Body Mass Index (kg/m^{**2})

We can then make a selection of columns and compute the ratio which will be saved in a new column with the name that we define for example `HWRATIO`. As an example the data is then rounded to just 1 decimal point as most of the rest of the data:

```
Master4 %>%
select(SEQN, BMXWT, BMXHT, BMXBMI) %>%
mutate(HWRATIO = BMXHT / BMXWT,
       HWRATIO2 = round(HWRATIO, digit=1)) %>%
head()
```

	SEQN	BMXWT	BMXHT	BMXBMI	HWRATIO	HWRATIO2
1	83732	94.8	184.5	27.8	1.946203	1.9
2	83733	90.4	171.4	30.8	1.896018	1.9
3	83734	83.4	170.1	28.8	2.039568	2.0
4	83735	109.8	160.9	42.4	1.465392	1.5
5	83736	55.2	164.9	20.3	2.987319	3.0
6	83737	64.4	150.0	28.6	2.329193	2.3

10.4.1 mutate with conditional statement

When analyzing tabular data we may have the choice will be made for each row. Here is an example found online: [Creating New Variables in R with mutate\(\) and ifelse\(\)](https://rpubs.com/daranzolin/mutateifelse)¹.

```
# Create 3 vectors
section <- c("MATH111", "MATH111", "ENG111")
grade <- c(78, 93, 56)
student <- c("David", "Kristina", "Mycroft")
# Combine vectors into a dataframe
gradebook <- data.frame(section, grade, student)
# Test of grade level and assign an outcome
mutate(gradebook, Pass.Fail = ifelse(grade > 60, "Pass", "Fail"))
```

¹<https://rpubs.com/daranzolin/mutateifelse>

	section	grade	student	Pass.Fail
1	MATH111	78	David	Pass
2	MATH111	93	Kristina	Pass
3	ENG111	56	Mycroft	Fail

Results are printed out

By “nesting” multiple `ifelse()` statements we can manage to provide multiple choices: the second argument is replaced by another `ifelse()` statements in succession. The final iteration has a single option that is not an `ifelse()` statements. In the example below, from the same source, the gradebook is read one row at a time, and then we need to provide a grade based on the letter system: A through D or F.

```
mutate(gradebook,
  letter =
    ifelse(grade %in% 60:69, "D",
      ifelse(grade %in% 70:79, "C",
        ifelse(grade %in% 80:89, "B",
          ifelse(grade %in% 90:99, "A",
            "F")))))
```

	section	grade	student	letter
1	MATH111	78	David	C
2	MATH111	93	Kristina	A
3	ENG111	56	Mycroft	F

This makes use of the `%in%` operator that can easily be understood as being able to test if a value is *within* the proposed range. On the first line we ask if the test of column `grade` for each row is between 60 and 69. If the answer is “yes” which means the test is `TRUE`, then the value will be “D”. If the test is `FALSE` we’ll go to the next question and so on until the last line where the final choice is F. (Note that for

readability the code is indented, and “F” is the alternate option for the test grade %in% 90:99.)



We'll use this statement is a very useful way in a future section (10.7.)

10.5 Summarising and grouping data

The following data can be found in the DEMO_I demographic data file:

- RIAGENDR: 1-male, 2-female (none missing)
- DDMARTL: 1 Married, 2 Widowed, etc.
- DMDEDUC2 - Education level - Adults 20+
- INDHHIN2 - Annual household income

We want to get some information by group, in this case we'll group by marital status and then count the total number of observations for each case and we'll store this in column `Counts`. For each marital status code, we'll then count how many women and men are in each category that we'll report in columns `Mnum` and `Wnum` (the sum of these 2 on each line should be equal to the reported `Counts` column.)

For simplicity with *income* and *education* we'll simply compute the mean of the codes, which should still give us an indication of the level of income and education for each category of marital status.

We'll place the results in a user-defined R object so that we can reuse it in the next section without re-writing the complete pipeline. Let's call it `Xsum` for example

```
Xsum <- Master4 %>%  
select(SEQN, RIAGENDR, DDMARTL, DMDEDUC2, INDHHIN2) %>%  
drop_na() %>%
```

```
group_by(DMDMARTL) %>%
summarise(Counts = n() ,
          Mnum = sum(RIAGENDR == 1),
          Wnum = sum(RIAGENDR == 2),
          MeanIncCode = mean(INDHHIN2),
          MeanEducCode = mean(DMDEDUC2))

Xsum # print output
```

```
# A tibble: 8 x 6
  DMDMARTL Counts  Mnum  Wnum MeanIncCode MeanEducCode
    <dbl>   <int> <int> <int>      <dbl>         <dbl>
1         1   2792  1481  1311      12.5          3.52
2         2    401   106   295      11.1          3.00
3         3    597   239   358      10.1          3.49
4         4    184    66   118       9.48          2.80
5         5    999   475   524      11.6          3.60
6         6    533   270   263      10.7          3.25
7        77     2     1     1       42           4
8        99     1     0     1       99           9
```

10.6 Recoding: string replacement

The above results are OK but it would be nice to be able to change some of the code numbers to actual English word such as “married” or “single” in text. This can be accomplished in many ways in `dplyr` but one of the simplest is to use `recode()` as a special case within `mutate()` to *overwrite* a column.

Table 10.1: The DMDMARTL codes from NHANES DEMO_I

Code or Value	Value Description	Count	Cumulative
1	Married	2886	2886
2	Widowed	421	3307
3	Divorced	614	3921
4	Separated	192	4113
5	Never married	1048	5161
6	Living with partner	555	5716
77	Refused	2	5718
99	Don't Know	1	5719
.	Missing	4252	9971

We can then *recode* the DMDMARTL column of Xsum with the following pipeline:

```
Xsum2 <- Xsum %>% mutate(DMDMARTL = recode(DMDMARTL,
  `1` = "Married",
  `2` = "Widowed",
  `3` = "Divorced",
  `4` = "Separated",
  `5` = "Nevermarried",
  `6` = "Living with partner",
  `77` = "Refused",
  `99` = "Don' Know")
)
Xsum2 # print out
```

```
# A tibble: 8 x 6
  DMDMARTL      Counts  Mnum  Wnum MeanIncCode MeanEducCode
  <chr>      <int> <int> <int>      <dbl>      <dbl>
1 Married      2792  1481  1311      12.5        3.52
2 Widowed       401   106   295      11.1        3.00
```

3 Divorced	597	239	358	10.1	3.49
4 Separated	184	66	118	9.48	2.80
5 Nevermarried	999	475	524	11.6	3.60
6 Living with partner	533	270	263	10.7	3.25
7 Refused	2	1	1	42	4
8 Don' Know	1	0	1	99	9

(Credit: this example inspired by [How to Recode a Column with dplyr in R²](#))

In fact 2 of the columns are not well named and we should rename Mnum and Wnum to simply Men and Women in the table. This is done with the dplyr rename() function:

```
Xsum2 %>% rename(Men = Mnum, Women = Wnum)
```

```
# A tibble: 8 x 6
```

DMDMARTL	Counts	Men	Women	MeanIncCode	MeanEducCode
<chr>	<int>	<int>	<int>	<dbl>	<dbl>
1 Married	2792	1481	1311	12.5	3.52
2 Widowed	401	106	295	11.1	3.00
3 Divorced	597	239	358	10.1	3.49
4 Separated	184	66	118	9.48	2.80
5 Nevermarried	999	475	524	11.6	3.60
6 Living with partner	533	270	263	10.7	3.25
7 Refused	2	1	1	42	4
8 Don' Know	1	0	1	99	9

10.7 Getting it all together

We now know enough that we should be able to get it all together in an annotated pipeline starting with Master4.

²<https://cmdlinetips.com/2019/04/how-to-recode-a-column-with-dplyr-in-r/>

10.7.1 Example 1: by gender

Let's try to answer the question: **“What is the average level of total cholesterol in men and women from the Master4 dataset.?”**

To answer the question we'll need the following:

- o. create an object to contain the results and redirect to it
 1. Start with Master4 (inject data in the pipeline)
 2. Select relevant columns
 3. Remove NA rows
 4. Group by gender
 5. Summarize.

We now know how to do this in a data stream with a pipeline. Let's write it and add comment lines so we can remember later the purpose of the code:

```
TcholGender <- Master4 %>%  
# select columns  
select(SEQN, RIAGENDR, LBXTC ) %>%  
# filter all rows to remove NAs  
drop_na() %>%  
# Group by gender  
group_by(RIAGENDR) %>%  
summarise(  
  Men = sum(RIAGENDR == 1),  
  Women = sum(RIAGENDR == 2),  
  MeanTChol = mean(LBXTC))  
  
# Print results:  
TcholGender
```

```
# A tibble: 2 x 4
  RIAGENDR    Men Women MeanTChol
  <dbl> <int> <int>    <dbl>
1      1  3545     0    178.
2      2     0  3711    183.
```

The final table will be short but that is exactly what a summary should be.

The value for men is **177.75** based on a total of **3545** observations.

For women the we see is **182.65** based on a total of **3711** observations.

(See section 13.2.4 later to learn how these numbers were embedded within the text automatically without copy/paste!)

10.7.2 Example 2: by gender and age

Let's make things a bit more interesting with the question:

“Base on gender and age group, what is the mean and standard deviation of PFAS compounds, as well as the mean, standard deviation, minimum, and maximum values of total cholesterol in men and women?”

One of the key word is “age group” as in the NHANES data age is a single integer number with a range from 1 to 80 in RIDAGEYR column and therefore it is “up to us” to create the age groups! This can be done rather easily within a combination of `mutate()` and with nested `ifelse()` statements within. Nesting the `ifelse()` function within itself allows making multiple choices. (For a refresher on `ifelse()` see section 10.4.1.)

We'll have to use the appropriate columns of data, compute the creatinine adjustment and summarize these values *by age group* for men and women.

A new base R function is introduced here `formatC()` which will force the numbers to be printed in scientific exponent for better clarity. (Try without and see the difference! - this option was suggested on [Stack Overflow](#).)

The pipeline below is annotated to specify the function of each line and the results are saved within a user-defined R object `Example2`.

```
Example2 <- Master4 %>%
# select columns
select(SEQN, RIAGENDR, RIDAGEYR, LBXMFOS, URXUCR, LBXTC ) %>%
# filter all rows to remove NAs
drop_na() %>%
# Creatinine adjustment
mutate (RATIO = (LBXMFOS/URXUCR)*10^-4) %>%
# categorize ages in 5 groups:
# Children: G0T018, younger adults: G19T035,
# and older adults: G36T065, seniors: G66T079,
# and 80 and older: G80.
mutate(AGEGROUP = ifelse(RIDAGEYR %in% 0:18, "G0T018",
                        ifelse(RIDAGEYR %in% 19:35, "G19T035",
                              ifelse(RIDAGEYR %in% 36:65, "G36T065",
                                    ifelse(RIDAGEYR %in% 66:79,
                                            "G66T079", "G80"))))) %>%
#
# NOTE: this would be a place to split the pipeline in 2 sections
#
# Group by gender first, age second
group_by(RIAGENDR, AGEGROUP) %>%
# Summarize: count totals,
summarise(
  Men = sum(RIAGENDR == 1),
  Women = sum(RIAGENDR == 2),
  MeanPFASR = formatC(mean(RATIO), format = "e", digits = 2),
  sdPFASR = formatC(sd(RATIO), format = "e", digits = 2),
  MeanTChol = mean(LBXTC),
  sdTChol = sd(LBXTC),
  minTChol = min(LBXTC),
```

```
maxTChol = max(LBXTC)
)
```

`summarise()` has grouped output by 'RIAGENDR'. You can override using the `.groups` argument.

Print out the results

Example2

```
# A tibble: 10 x 10
# Groups:   RIAGENDR [2]
  RIAGENDR AGEGROUP    Men Women MeanPFASR sdPFASR MeanTChol sdTChol minTChol
  <dbl>   <chr>    <int> <int> <chr>      <chr>      <dbl>    <dbl>    <dbl>
1      1 G0T018     175     0 8.47e-07 8.68e-07    152.    28.6     94
2      1 G19T035    235     0 1.68e-06 1.88e-06    180.    35.6     98
3      1 G36T065    382     0 3.24e-06 5.99e-06    198.    46.7    111
4      1 G66T079    117     0 3.95e-06 3.92e-06    169.    35.1     90
5      1 G80         44     0 5.12e-06 6.42e-06    165.    35.0    101
6      2 G0T018       0    141 7.70e-07 7.88e-07    158.    29.5    103
7      2 G19T035       0    238 9.68e-07 1.31e-06    176.    38.5    100
8      2 G36T065       0    459 2.35e-06 3.80e-06    201.    36.3    106
9      2 G66T079       0    124 4.58e-06 6.32e-06    202.    48.5     84
10     2 G80          0     53 5.86e-06 6.29e-06    187.    34.1    126
# i 1 more variable: maxTChol <dbl>
```

The print-out might be truncated (depending on the format of this document.)



As expected the output is a tibble. A subtle but important note is in the second line of the output: # Groups: RIAGENDR [2]. It may be important at a later stage to use the dplyr function `ungroup()` to modify *e.g.* the column name or the values within that column.

The highest cholesterol value is **545** and is for a person of gender code **1** in age group **G36T065**.

(See section 13.2.4 later to learn how these numbers were embedded within the text automatically without copy/paste!)

10.7.2.1 Sorting / arranging

The `Example2` is a summary table (in tibble / data frame format) and can be further sorted with the `arrange()` function. For example to see a table with the cholesterol levels:

```
Example2 %>%
  select(RIAGENDR, AGEGROUP, maxTChol) %>%
  arrange(-maxTChol) %>%
  head(2)
```

```
# A tibble: 2 x 3
# Groups:   RIAGENDR [2]
  RIAGENDR AGEGROUP maxTChol
    <dbl>   <chr>      <dbl>
1         1 G36T065      545
2         2 G66T079      358
```

10.7.3 Base R Bar plot

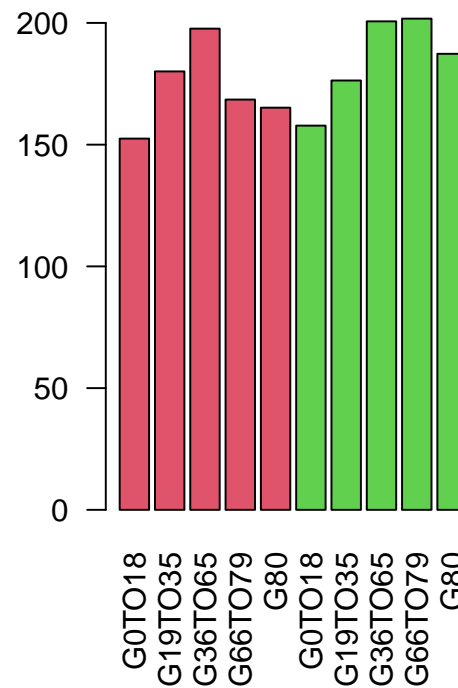
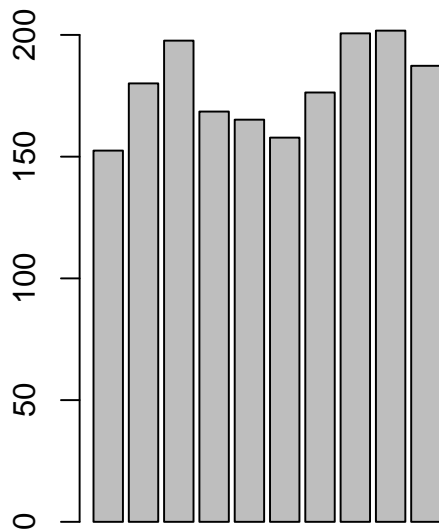
This summary table can be used with the base R function `barplot()` to create a representation of the mean cholesterol data. The default plot yields just gray bars and not horizontal label. Here are examples plotted together on a single graph that show the effect of some options.

As a reminder:

- `las=2` is rotating the labels (seen in section 6.3.)
- `col = RIAGENDR + 1` selects two of the nine colors in the base R color palette (see colors in section 5.5.) Since RIAGENDR values are 1 and 2 the resulting colors would be black and red which is not very appealing. Adding + 1 will choose red and green. Other numbers will select the next colors in the list.
- `names.arg=` specifies the source of the horizontal axis names displayed.

These commands use the `with()` function but could also be written with the `$` method, for example with `Example2$AGEGROUP`.

```
par(mfrow = c(1,2))  
# Plain version  
with(Example2, barplot(MeanTChol))  
# Add white/gray alternate colors, rotated horiz labels  
with(Example2, barplot(MeanTChol,  
                        col = RIAGENDR + 1,  
                        names.arg=AGEGROUP, las =2 ))
```



```
par(mfrow = c(1,1))
```

Error bars? There are methods in base R to add error bars and various examples can be found online. However, the methods are all quirky and the best is now to use `ggplot` to create such graphs.

10.7.4 ggplot2 versions

Example plots have been moved to section [11.2](#).

Chapter 11

ggplot2

Basic R has multiple, separate functions, each used for creating a specific type of representation: boxplot, histogram, scatter plot etc. ggplot2 is an R package for creating elegant data visualization using the conceptual philosophy that views a plot as the assembly of different fundamental parts:

$$Plot = Data + Aesthetics + Geometry$$

- **Plot:** the final graphics
- **Data:** tabular data in *tibble* or a *data frame*
- **Aesthetics:** Describe visual characteristics that represent data (position, size, color, shape, transparency, fill, scales)
- **Geometry:** defines the graphical representation: histogram, boxplot, scatter plot. Defines the type of geometric objects that represent data (points, lines, polygons.)

Each element is built as a *layer* based on a “**grammar of graphics**” all assembled into a final plot.

The “grammar” contains more definitions for graphics elements

- *coordinate system:* e.g. Cartesian, polar, map projections

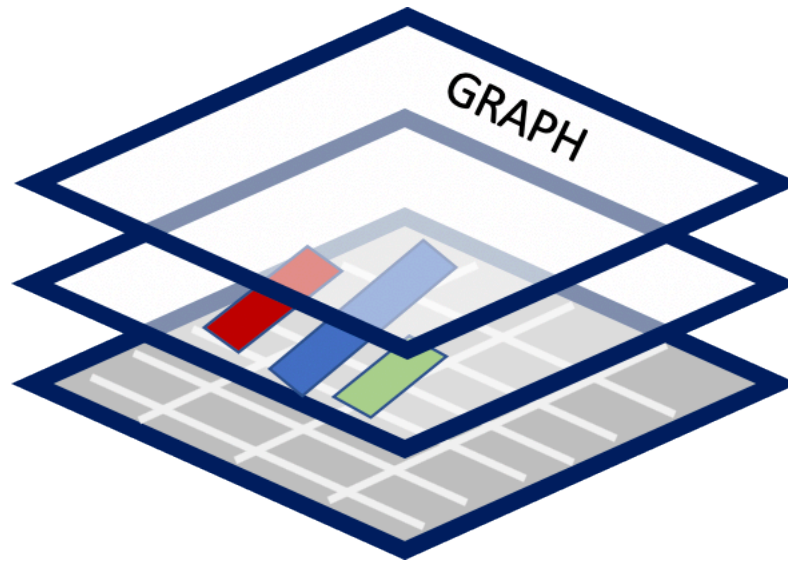


Figure 11.1: ggplot2 constructs graphs in layers using a grammar of graphics.

- *geoms*: describe type of geometric objects that represent data (points, lines, polygons)
- *aesthetics*: describe visual characteristics that represent data (position, size, color, shape, transparency, fill.)
- *scales*: for each aesthetic: log scales, color scales, size scales, shape scales.
- *stats* : describe statistical transformations that typically summarize data: counts, means, medians, regression lines.
- *facets*: describe how data is split into subsets and displayed as multiple, separate small graphs.
- *Theme*: controls appearance of non-data elements



Exerpts from Hadley Wickham’s “ggplot2: Elegant Graphics for Data Analysis” ([Wickham and Sievert \(2016\)](#).) (The most revised version of the book

is also available free online: ggplot2-book.org/)

ggplot2 is an R package for producing statistical, or data, graphics, but it is unlike most other graphics packages because it has a deep underlying grammar. This grammar, based on the Grammar of Graphics (Wilkinson, 2005), is made up of a set of independent components that can be composed in many different ways. This makes ggplot2 very powerful because you are not limited to a set of pre-specified graphics, but you can create new graphics that are precisely tailored for your problem.

Without the grammar, there is no underlying theory, so most graphics packages are just a big collection of special cases.

In his 2017 presentation Hadley Wickham mentions that `ggplot` was created before Tidyverse and lacks the Tidyverse philosophy on the ideas of distinguishing and separating **command** (action) and **query** (computation) functions. (See references in 8.) However it is well integrated within the Tidyverse and can be placed at the end of a `%>%` pipeline as the last **command**.

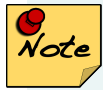
11.1 Tutorials

There are many tutorials online to learn how to use `ggplot`. See Appendix G for a table of just a few that seem useful based on the number of examples with `ggplot` code. There are many more to be found with a simple web search.

Readers are encouraged to learn how to use `ggplot2()` on some of the provided links in Appendix G before continuing with the examples in the next section 11.2.

One suggestion is [The Complete ggplot2 Tutorial](http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html)¹ split over multiple documents.

¹<http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html>



IMPORTANT CONSIDERATION: Regardless of the chosen online tutorial, **your data** may not be in the same *shape* (rows/columns) or have the same *attributes* (numerical, continuous, **categorical**) that may make converting online examples to fit your data challenging and frustrating. Being aware of that fact may certainly help!

If the data you are working with is not “**tidy**” watching the lesson on the `tidyr` package might be helpful - see data wrangling section 9.

Perseverance is always rewarded.

A personal example:

- a bar chart can be created by two types of `geom`: `geom_bar()` and `geom_col()`. This simple knowledge can save you hours of frustration (see help with `?geom_bar()`.)
- Categorical variables are usually recognized automatically, but numerical and continuous variables have to be “made” into categories (or “levels”) by using `as.factor()` but in some cases `as.character()` might also work depending on the variable in question.

11.2 ggplot2 using dplyr chapter results

The `dplyr` chapter ended with the creation putting together a pipeline to create a summary data table. The story will continue here as that chapter ended.

11.2.1 Barplot with `qplot` / `ggplot`

Splitting the pipeline above is most useful for using `qplot` or `ggplot`.

Example 2 pipeline at midpoint before summarization, saved in object Mid. It is the same code as above but stopped where the midpoint was suggested.

```
Mid <- Master4 %>%
# select columns
select(SEQN, RIAGENDR, RIDAGEYR, LBXMFOS, URXUCR, LBXTC ) %>%
# filter all rows to remove NAs
drop_na() %>%
# Creatinine adjustment
mutate (RATIO = (LBXMFOS/URXUCR)*10^-4) %>%
# categorize ages in 5 groups:
# Children: G0T018, younger adults: G19T035,
# and older adults: G36T065, seniors: G66T079,
# and 80 and older: G80.
mutate(AGEGROUP = ifelse(RIDAGEYR %in% 0:18, "G0T018",
                        ifelse(RIDAGEYR %in% 19:35, "G19T035",
                              ifelse(RIDAGEYR %in% 36:65, "G36T065",
                                    ifelse(RIDAGEYR %in% 66:79,
                                            "G66T079", "G80")))))
```

Below are some plot examples using Mid. The addition of facet_grid splits the data “as a function of” (~) gender in RIAGENDR.

```
# Qplot
qplot(AGEGROUP, data = Mid, geom="bar")
```

It would be useful to visualize based on gender.

```
# Qplot
qplot(AGEGROUP, data = Mid, geom="bar") +
  facet_grid(~RIAGENDR)
```

To add color we need to use geom_bar instead of geom = "bar" so that we can add an aesthetics (aes) request to color, as a factor of the values in RIAGENDR.

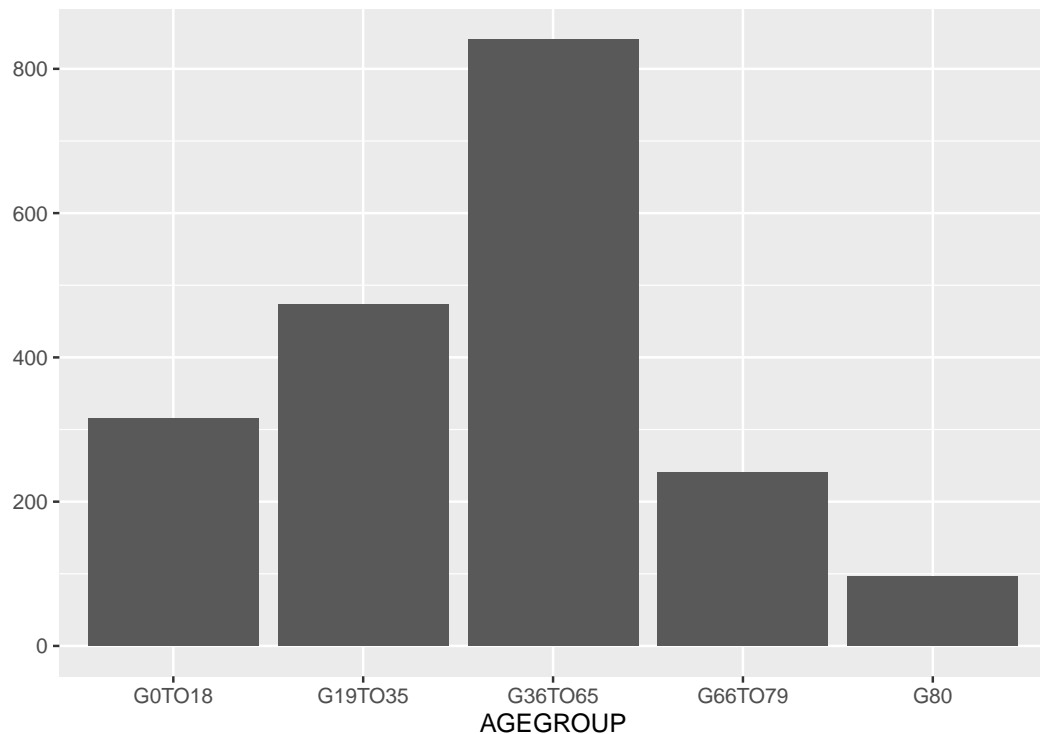


Figure 11.2: Bar plot showing total count by age group without gender distinction.

```
# Qplot
qplot(x=AGEGROUP, data=Mid) +
  facet_grid(~RIAGENDR) +
  geom_bar(aes(fill = as.factor(RIAGENDR)))
```

A similar plot but with stacked bars can be achieved with `ggplot`.

We can avoid using `as.factor` that is necessary since `RIAGENDR` is coded as a number that `ggplot` considers a *numerical* (perhaps continuous) rather than a *categorical* variable. We could avoid this problem by “recoding” the values of 1 and 2 to words such as `male` and `female` or `Men` and `Women` on a short pipeline before the plot is done. (Review `recode()` in section 10.6.)

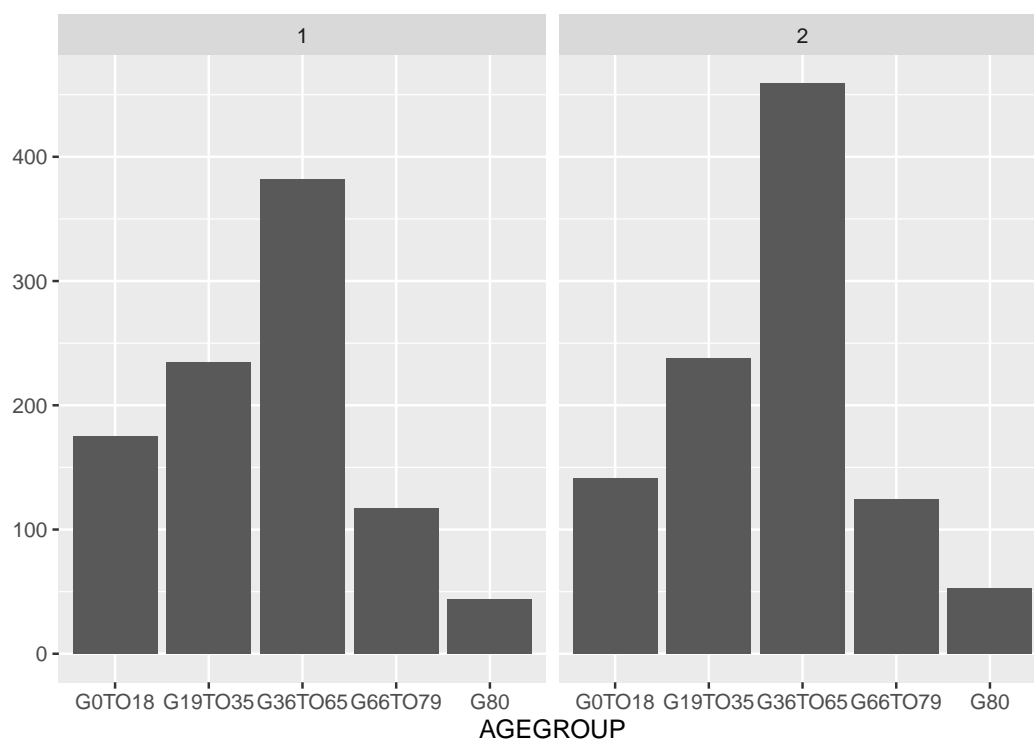


Figure 11.3: With `facet_grid()` the age distribution by gender is on two separate graphs.

```
Mid %>%
  mutate(RIAGENDR =
    recode(RIAGENDR,
      `1` = "Men",
      `2` = "Women")) %>%
  ggplot(aes(x = AGEGROUP)) +
  geom_bar(aes(fill = RIAGENDR))
```

We now also have a better description, avoiding 1 and 2 as well as `as.factor` in the legend.

A final touch could be to rename the column `RIAGENDR` to simply `Gender` and `AGEGROUP` to `Age_group` by using the `rename()` function (section 10.6.)

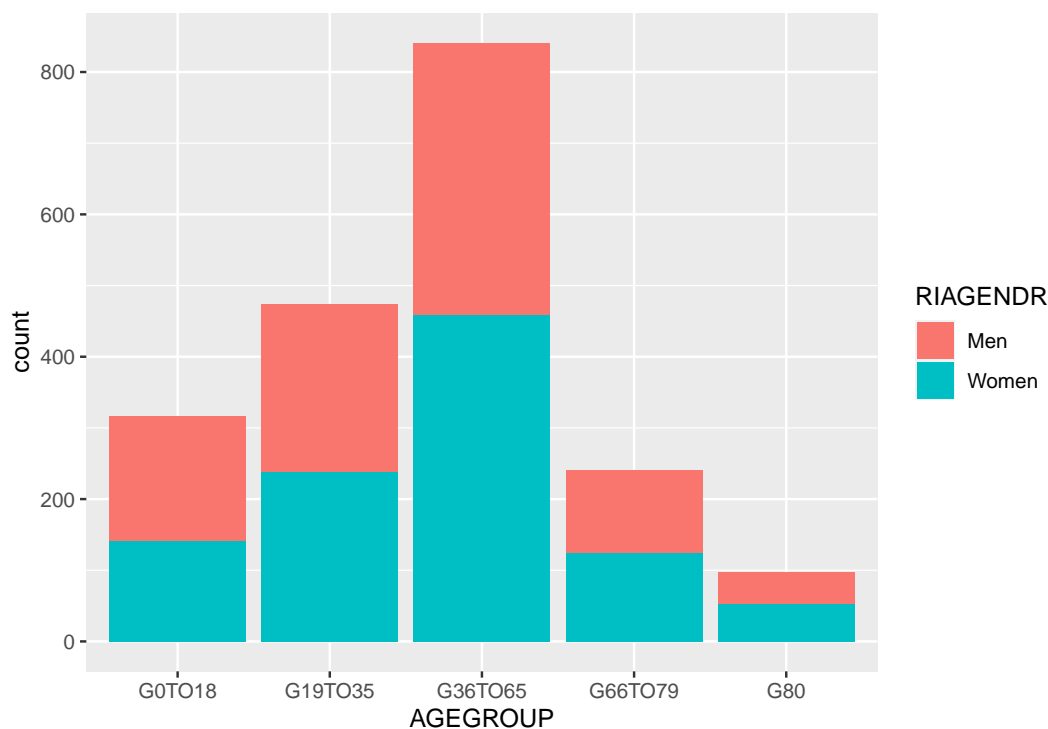


Figure 11.4: Bar plot showing age group distribution by gender. Stack bars is the default.

We can save this in Mid2. Note the need of quote for Age group to take care of the blank space.

```
Mid2 <- Mid %>%
  mutate(RIAGENDR =
    recode(RIAGENDR,
      `1` = "Men",
      `2` = "Women")) %>%
  rename(Gender = RIAGENDR, "Age group" = AGEGROUP)
```



Warning: Using blank spaces in columns or in data in general is a source of trouble.

Note that to use the `Age group` column in a `ggplot` command it is required to use backticks ``` to have it considered a single entity in a similar way that was used in the `recode()` function with numbers.

To have the bars side by side for each age group the additional `position =` option is introduced with option `"dodge"` (bars touch) or `"dodge2"` (white space between bars.)

```
#
Mid2 %>% ggplot(aes(x = `Age group`)) +
  geom_bar(aes(fill = Gender), position = "dodge2")
```

It is possible to combine options:

```
Mid2 %>%
  ggplot(aes(x = `Age group`)) +
  geom_bar(aes(fill = Gender), position = "dodge2") +
  facet_wrap(~ `Age group`)
```

11.2.2 Error bars and meanTChol

Example derived from info at [Plotting with ggplot: bar plots with error bars](#) (See also Appendix G.)

We need to use `ungroup()` as data were grouped when creating `Example2`. (Section 10.7.2.)

```
Example2 %>%
  # ungroup to allow changes for mutate and rename
  ungroup() %>%
```

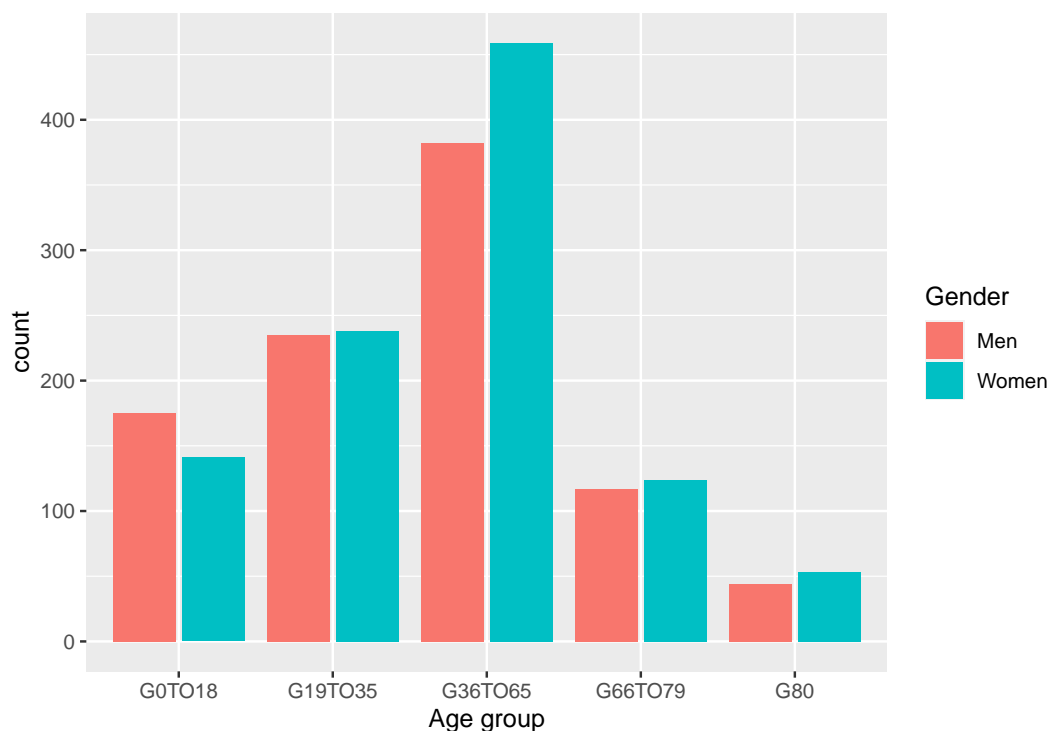


Figure 11.5: Side by side bar of gender count by age group is possible with the `dodge` or `dodge2` options.

```
mutate(RIAGENDR =
  recode(RIAGENDR,
    `1` = "Men",
    `2` = "Women")) %>%
  rename(Gender = RIAGENDR) %>%
# start ggplot commands
ggplot(aes(AGEGROUP, MeanTChol)) +
  geom_col(aes(fill = Gender)) +
  geom_errorbar(aes(ymin = MeanTChol - sdTChol,
    ymax = MeanTChol + sdTChol),
    width=0.3) +
  facet_wrap(~Gender) +
```

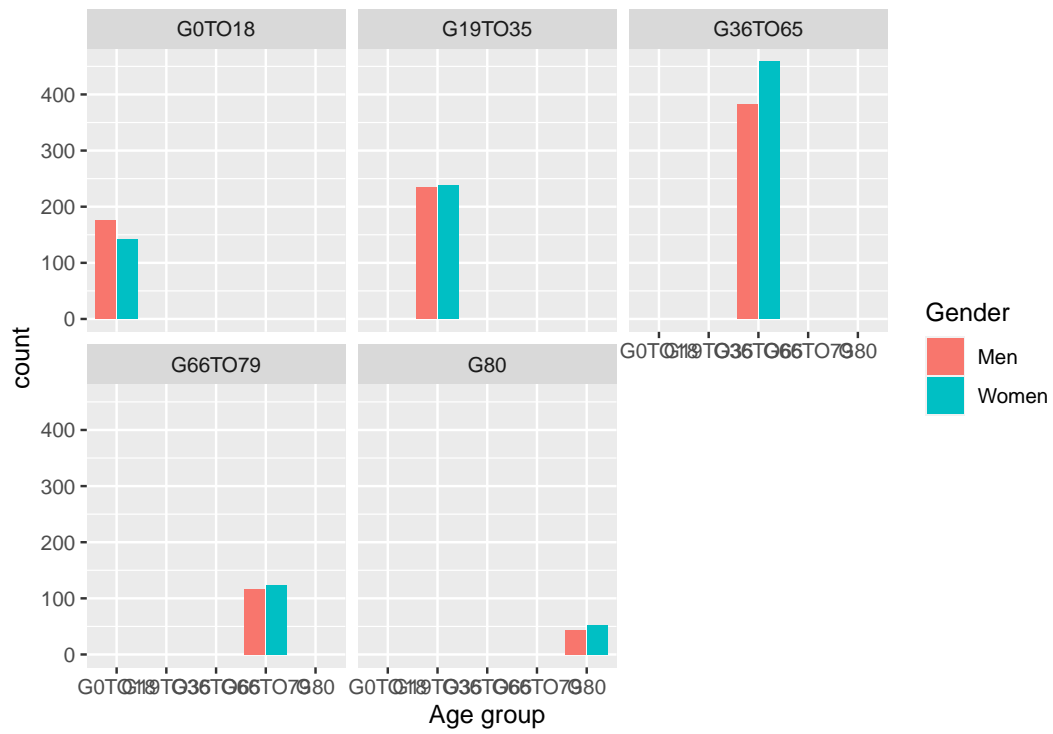
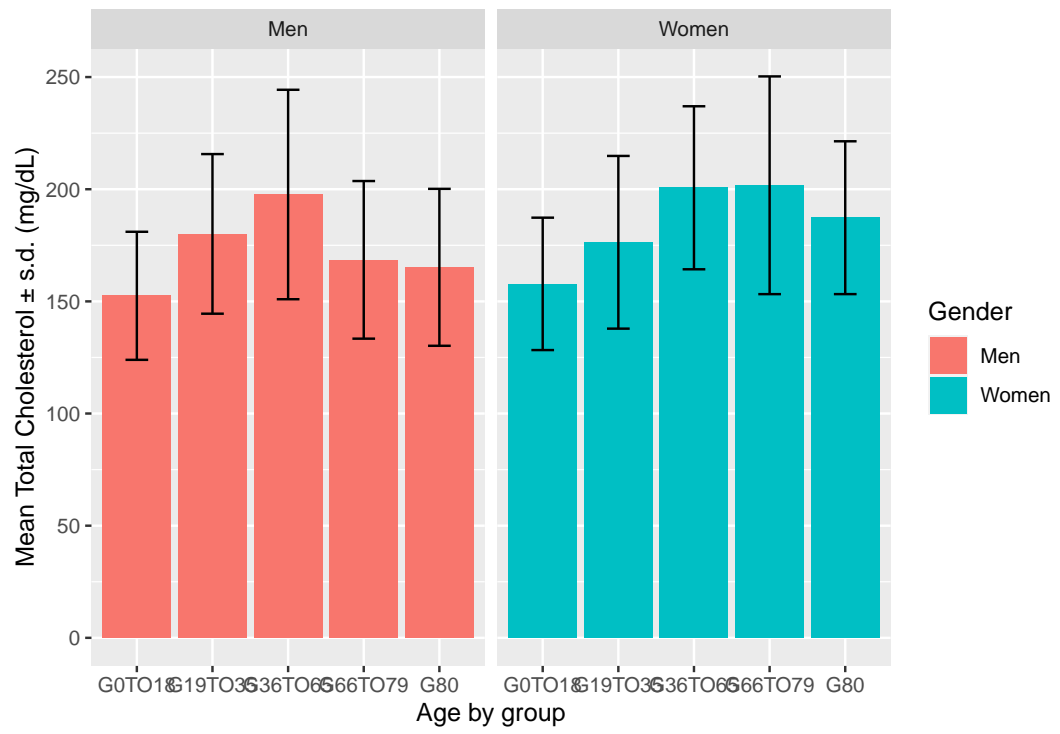


Figure 11.6: Side by side in each facet.

```
labs(y="Mean Total Cholesterol ± s.d. (mg/dL)", x = "Age by group")
```



Chapter 12

Using NHANES weights



Using weights

An excellent demonstration of incorporating NHANES provided weights as a commented R code page is available on this blog post: [How to Use Survey Weights in R¹](#) by Mike Burke.

The code has a few base R commands but most of the code is a perfect demonstration of the usefulness of the `dplyr` package and how to combine commands in streams on small pipelines. It is also worth noting how the comments within the code facilitates the understanding of the successive steps.

The code contains **all** the relevant and necessary information, including the activation of packages with the `library()` function. If all packages are already installed the code can be “Copied/Pasted” in its entirety and proceed without errors.

To review the code we’ll cut it in smaller portion and perhaps add a few commands the check the content or status of the R objects that are defined along the way.

12.1 Header comments and packages

The top of the code contains well defined titles for each section informing the purpose of the program and the needed packages.

When it is run typical information on the loading of tidyverse is displayed.

```
#####
# General Information #
#####

# This is an RScript for showing how to use survey weights with NHANES data. You
# can find a narrative to this script at:
# https://stylizeddata.com/how-to-use-survey-weights-in-r/

#####
# Packages #
#####

# For reading SAS XPT file from NHANES website
# haven::read_xpt

library(haven)

# For using survey weights
# survey::svydesign, svymean, svyglm

library(survey)

# For data wrangling
# dplyr::select, mutate, select, recode

library(dplyr)
```

12.2 Acquiring NHANES data

This chunk uses the haven package that has an easier method to download the data directly from the web site without the need of an intermediate file as we saw in section 6.2.1.

```
#####
# Load the dataset #
#####

# Import NHANES demographic data

nhanesDemo <- read_xpt(url(
  "https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/DEMO_I.XPT"))
```

12.3 Data wrangling: renaming and selecting data

This section of the code reorganizes and renames for easier understanding.

12.3.1 Renaming columns

This code portion has a goal to make the data easier for humans to understand by changing the name of the data columns. It is performed with base R subsetting with \$ and overwriting the specified column. However, this could also have been done with Tidyverse method by using the rename() function as we have seen in section 10.6.

Only columns that will be selected in the next step are altered. The only data specification is for INDFMPIR others can be found on the [DEMO_I web page](#) and listed below:

Table 12.1: Chosen columns and their description

Code	Description	range
INDFMPIR	Ratio of family income to poverty	0 - 5
RIDAGEYR	Age in years at screening	0 - 80
RIAGENDR	Gender	1 -2
WTINT2YR	Full sample 2 year interview weight	3293.928267 - 233755.84185
SDMVPSU	Masked variance pseudo-PSU	1 to 2
SDMVSTRA	Masked variance pseudo-stratum	119 to 133

NHANES data uses the method described in article [Primary sampling unit \(PSU\) masking and variance estimation in complex surveys](#) available online².

```
#####
# Data Wrangling #
#####

# Copy and rename variables so they are more intuitive. "fpl" is percent
# of the federal poverty level. It ranges from 0 to 5.

nhanesDemo$fpl      <- nhanesDemo$INDFMPIR

nhanesDemo$age      <- nhanesDemo$RIDAGEYR

nhanesDemo$gender   <- nhanesDemo$RIAGENDR

nhanesDemo$persWeight <- nhanesDemo$WTINT2YR

nhanesDemo$psu      <- nhanesDemo$SDMVPSU

nhanesDemo$strata   <- nhanesDemo$SDMVSTRA
```

²<https://www150.statcan.gc.ca/n1/en/catalogue/12-001-X200800210759>

12.3.2 Selecting columns

This chunk selects only the columns that are desired with a pipeline using the `select()` function.

```
# Since there are 47 variables, we will select only the variables we will use in  
# this analysis.  
  
nhanesAnalysis <- nhanesDemo %>%  
  select(fpl,  
         age,  
         gender,  
         persWeight,  
         psu,  
         strata)
```

12.3.3 Changing variable status to a factor

We encountered this problem before (section 10.6) when we needed to convert a variable from “continuous” to a “factor” so that the data would be seen as a “category” with just a few “options”. Just changing Integer numbers to characters could accomplish that.

Here the author chooses to keep the code as integers but change the value for men from 1 to 0 and for women from 2 to 1. The cryptic L in 0L and 1L is a special code that means “Long” and therefore **does not** represent the letter L as a character but is a form of “coercion” “forcing” the number to be an integer in its forms recorded by the computer.



What does L mean?

There are multiple entries in Stack Overflow for this, including:

The author of this in R never explained why he chose the notation, but it is shorter than `as.integer(10)`, and **more efficient as the coercion is done at parse time.**

See more info in links:

- [Clarification of L in R](#)
- [What's the difference between 1L and 1?](#)

Also in the **R Language Definition** book ([R Core Team \(2020b\)](#))^{3, 4} it is stated, in reference to the number 1 (as in: `class(c(1))`)

“Perhaps unexpectedly, the number returned from the expression 1 is a numeric. In most cases, the difference between an integer and a numeric value will be unimportant as R will do the right thing when using the numbers. There are, however, times when we would like to explicitly create an integer value for a constant. We can do this by calling the function `as.integer` or using various other techniques. But perhaps the simplest approach is to qualify our constant with the suffix character L.”

“We can use the L suffix to qualify any number with the intent of making it an explicit integer.”

Finally, internally this is related also to the amount of memory (RAM) used by the computer to hold an integer and that may depend if it is a 32 bit or a 64 bit system. To know the larger **Integer** that R can hold can be revealed by command `.Machine$integer.max` (starts with a dot.) For a 32 bit computer this will be exactly 2147483647. Using 64 bit may be extremely beneficial for very large dataset. A 64 bit version of R can be downloaded from info at <http://r.research.att.com/> or <https://mac.r-project.org/>

Note that the first command mutating the gender column `nhanesAnalysis` is in tidyverse format using the pipe, while the last command is in base R format.

```
# Recode gender

nhanesAnalysis <- nhanesAnalysis %>%
  mutate(gender = recode(gender, `1` = 0L,
                           `2` = 1L))

# Convert "gender" to a factor variable. We need to do this so it isn't treated
# as a continuous variable in our analyses

nhanesAnalysis$gender <- as.factor(nhanesAnalysis$gender)
```

At this point we can add commands to get a better understanding of the data format and content. For example:

```
head(nhanesAnalysis$gender)
```

```
[1] 0 0 0 1 1 1
Levels: 0 1
```

```
class(nhanesAnalysis)
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

```
dim(nhanesAnalysis)
```

```
[1] 9971      6
```

```
str( nhanesAnalysis)
```

```
tibble [9,971 x 6] (S3: tbl_df/tbl/data.frame)
 $ fpl      : num [1:9971] 4.39 1.32 1.51 5 1.23 2.82 1.18 4.22 NA 2.08 ...
 ..- attr(*, "label")= chr "Ratio of family income to poverty"
 $ age      : num [1:9971] 62 53 78 56 42 72 11 4 1 22 ...
 ..- attr(*, "label")= chr "Age in years at screening"
 $ gender   : Factor w/ 2 levels "0","1": 1 1 1 2 2 2 2 1 1 1 ...
 $ persWeight: num [1:9971] 134671 24329 12400 102718 17628 ...
 ..- attr(*, "label")= chr "Full sample 2 year interview weight"
 $ psu      : num [1:9971] 1 1 1 1 2 1 1 2 1 2 ...
 ..- attr(*, "label")= chr "Masked variance pseudo-PSU"
 $ strata   : num [1:9971] 125 125 131 131 126 128 120 124 119 128 ...
 ..- attr(*, "label")= chr "Masked variance pseudo-stratum"
```

```
head(nhanesAnalysis)
```

```
# A tibble: 6 x 6
   fpl    age gender persWeight    psu strata
  <dbl> <dbl> <fct>      <dbl> <dbl>  <dbl>
1  4.39   62  0          134671.    1    125
2  1.32   53  0           24329.    1    125
3  1.51   78  0           12400.    1    131
4  5       56  1          102718.    1    131
5  1.23   42  1           17628.    2    126
6  2.82   72  1           11252.    1    128
```

The gender assignment can be confusing as the **values** are still 1 and 2 but the **levels** are now 0 and 1 as shown in the short table showing gender as <fct> meaning **factor**, and as can be deciphered from the `str()` output for the gender line:

```
$ gender      : Factor w/ 2 levels "0","1": 1 1 1 2 2 2 2 1 1 1 ...
```


12.3.4 Adding the weight information

```
#####
# Survey Weights #
#####

# Here we use "svydesign" to assign the weights. We will use this new design
# variable "nhanesDesign" when running our analyses.

nhanesDesign <- svydesign(id      = ~psu,
                        strata   = ~strata,
                        weights  = ~persWeight,
                        nest     = TRUE,
                        data     = nhanesAnalysis)

# Here we use "subset" to tell "nhanesDesign" that we want to only look at a
# specific subpopulation (i.e., those age between 18-79 years). This is
# important to do. If you don't do this and just restrict it in a different way
# your estimates won't have correct SEs.

ageDesign <- subset(nhanesDesign, age > 17 &
                    age < 80)
```

We can printout the content of these:

```
nhanesDesign
```

```
Stratified 1 - level Cluster Sampling design (with replacement)
With (30) clusters.
svydesign(id = ~psu, strata = ~strata, weights = ~persWeight,
          nest = TRUE, data = nhanesAnalysis)
```

```
ageDesign
```

```
Stratified 1 - level Cluster Sampling design (with replacement)
With (30) clusters.
subset(nhanesDesign, age > 17 & age < 80)
```

12.3.5 Statistics

```
#####
# Statistics #
#####

# We will use "svymean" to calculate the population mean for age. The na.rm
# argument "TRUE" excludes missing values from the calculation. We see that
# the mean age is 45.648 and the standard error is 0.5131.

svymean(~age, ageDesign, na.rm = TRUE)
```

```
      mean      SE
age 45.648 0.5131
```

```
# Since gender is a factor variable, "svymean" will treat it as such and give us
# the proportion of women. We see that men are 48.601% and woman are 51.399% of
# the population in this age of 18 to 79.
```

```
svymean(~gender, ageDesign, na.rm = TRUE)
```

```
      mean      SE
gender0 0.48601 0.006
gender1 0.51399 0.006
```

```
# Now we will run a general linear model (glm) with a gaussian link function.
# We tell svyglm that nhanesAnalysis is the dataset to use and to apply the
# "svydesign" object "ageDesign." I won't dive into the results here, but you
# can see that age is positively correlated with FPL and that women are
# predicted to have a lower FPL than men.
```

```
output <- svyglm(fpl ~ age +
                 gender,
                 family = gaussian(),
                 data = nhanesAnalysis,
                 design = ageDesign)
```

We can add the following to show output results:

```
output
```

```
Stratified 1 - level Cluster Sampling design (with replacement)
With (30) clusters.
```

```
subset(nhanesDesign, age > 17 & age < 80)
```

```
Call: svyglm(formula = fpl ~ age + gender, design = ageDesign, family = gaussian(),
             data = nhanesAnalysis)
```

```
Coefficients:
```

```
(Intercept)      age      gender1
    2.44519    0.01488   -0.21055
```

```
Degrees of Freedom: 5013 Total (i.e. Null);  13 Residual
(602 observations deleted due to missingness)
```

```
Null Deviance:      14170
```

```
Residual Deviance: 13820    AIC: 20920
```

In the comments it is said “...you can see that age is positively correlated with FPL and that women are predicted to have a lower FPL than men.”

The definition of the function `svyglm()` in help is: “Fit a generalised linear model to data from a complex survey design, with inverse-probability weighting and design-based standard errors.”

The final data are located in `output$coefficients`:

```
output$coefficients
```

(Intercept)	age	gender1
2.4451935	0.0148763	-0.2105470

Most likely from the conclusion the following is done:

1. Compute the regression line for age
2. Compute a correlation coefficient for gender

From the printed results we can see:

- The intercept 2.4451935 is where the regression line crosses the vertical y axis.
- The slope defining the relationship with age is 0.0148763
- The third item would be the correlation coefficient with gender, and is negative at -0.210547. Since it is labeled as `gender1` that should mean “gender level 1” which is the code for women after the changes added with the `mutate()` function to change the label numbers.

Chapter 13

Markdown and Reproducible research

Reproducible research is becoming a vast field. This chapter is to provide a flavor of what's possible in creating a “live” document for data analysis. There are many sources online, here is one from a 6-hour workshop from the “Monash Bioinformatics Platform”: [Reproducible Research in R¹](#) (2019-07-25).

What is Reproducible Research?^a

Research is considered to be reproducible when the exact results can be reproduced if given access to the original data, software, or code. Reproducible research is sometimes known as reproducibility, reproducible statistical analysis, reproducible data analysis, reproducible reporting, and literate programming.

Literate programming is simply telling a “story” with the embedded code which is “rendered” in the final output.

^a<https://www.displayr.com/what-is-reproducible-research/>

Reproducible research usually refers more to the **analysis** of the data, while research that is **replicable** is the idea that **research results** can be reproduced by in-

¹<https://monashdatafluency.github.io/r-rep-res/>

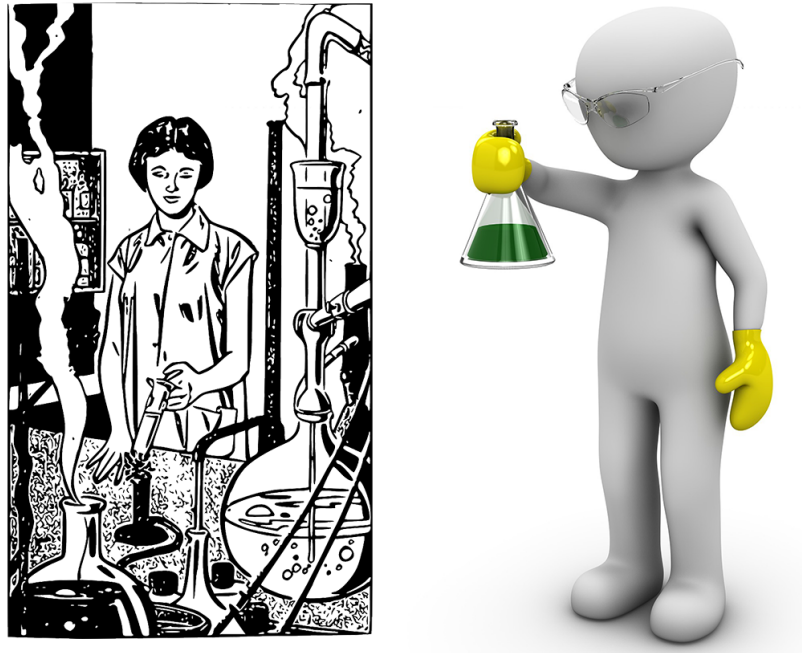


Figure 13.1: Reproducible research is more about computer analysis, replicable research is about reproducing research results.

dependent researchers using different methods.

Table 13.1: A course on reproducible research using R

Name	Course Web site
R for Reproducible Research	https://annakrystalli.me/rrresearch/index.html

13.1 Markdown



What is markdown? Markdown is a lightweight markup language with plain-text-formatting syntax, created in 2004 by John Gruber with Aaron Swartz.^a (Note the play on words between *markdown* and *markup*!)

The philosophy of markdown is described by John Gruber on his web site: “DARING FIREBALL”^b.

^a<https://en.wikipedia.org/wiki/Markdown>

^b<https://daringfireball.net/projects/markdown/>

At its origin, John Gruber created markdown to *easily* create HTML pages with an easy syntax. The markdown document is a plain text file that in the end is used as a *source* to create an HTML page.



This very document is being written with the help of markdown!

A web page is written in **HTML** or “Hyper Text *Markup* Language” and its syntax requires a lot of characters to specify a format. The name “markdown” is a play on word and its syntax is very easy. Here is an example to make a word **bold**:

- HTML: `word`
- Markdown: `**word**`

Another more remarkable example would be the “heading” as it is used on the web but also in MSWord as a section title:

- HTML: `<h1>heading1</h1>` -> requires 9 characters on both sides of heading1
- Markdown: `# heading1` -> requires a single character!

The result is that text files that are formatted in markdown can be read “as is” very easily, while a page of HTML code would be much harder for a human to read “as is”. In fact that was a key design goal: *readability*.

13.1.1 Markdown syntax

The basic syntax is illustrated on this page: www.markdownguide.org/basic-syntax/²

The basic markdown syntax can be summarized in a short table from <https://www.markdownguide.org/cheat-sheet/>³.

²<https://www.markdownguide.org/basic-syntax/>

³<https://www.markdownguide.org/cheat-sheet/>

Table 13.2: Basic Markdown Syntax

Element	Markdown Syntax
Heading	# H1 ## H2 ### H3
Bold	**bold text**
Italic	<i>*italicized text*</i>
Blockquote	> blockquote
Ordered List	1. First item 2. Second item 3. Third item
Unordered List	- First item - Second item - Third item
Code	code
Horizontal Rule	- - -
Link	[title] (https://www.example.com)
Image	![alt text] (image.jpg)

Extended syntax can be useful for making tables (such as the table describing basic markdown) or footnotes and listed further down on the same guide page.

Basic and most extended markdown syntax are included in RStudio.



Interactive tutorial

One easy way to learn how to use markdown is to go through the very easy interactive exercises dynamically rendered in the **free interactive tutorial** at www.markdowntutorial.com/ available in English, Spanish, French, Korean, and Japanese.

In turn RStudio created a method to add code within a markdown file which is then called an “R markdown” file.

Regular markdown can easily be learned from the above links, the next section will provide details on R markdown.

13.2 R markdown magic



Before experiencing the *Magic* of R markdown it is necessary to have an even rudimentary understanding of “plain” markdown - see previous section [13.1.1](#).

Markdown allows a document to be formatted easily but *Rmarkdown* provides the means to create a *dynamic document* that makes it easy to maintain both the *narrative* (text, story, information) and the *analysis* in the form of computer code that is *woven* within the file and can automatically embed data, tables and even plots and graphs automatically. Since this is all automated, if the original data is changed, converting the Rmarkdown document once more to a final output format (HTML, PDF, MSWord) will *recompute* and update everything, literally with ***one click!***

This is a valuable tool in the context of *Reproducible research* as a paper could be completely self-contained within an Rmarkdown document: the story, the analysis code, and the figures (automatically generated by the analysis code.)



The free online book [R Markdown: The Definitive Guide](#)⁴ by Yihui Xie, J. J. Allaire, Garrett Golemund (2020-04-26) should prove a very valuable reference.

See more resources in Appendix [H](#).

13.2.1 Before your start



Some packages are needed to create output from R markdown documents which you can install in advance, for example with:

```
install.packages(c("knitr", "rmarkdown", "markdown"))
```

However, the newest versions of RStudio will prompt you if you want to install a package that is necessary but not yet installed.



The knitr package is used to transform the R markdown .Rmd file into a beautifully rendered document in various formats. The knitr package name reflects the “*knitting together*” (weaving together) the text and the embedded literal programming code and at the same time makes things look a lot more “*neater*.”

13.2.2 How to create an R markdown file



TASK: open an R markdown template

To follow these exercises create a new R markdown file with the menu cascade:

File -> New File -> R Markdown...

In the new window replace "Untitled" with a title for your document.

Keep HTML selected as the "Default output format"

Press **OK**

Save the file now (or later) and provide a name for the file.

The new file will have a filename extension of .Rmd

The top of the file will look something like this:

```
---  
title: "Test1"  
author: "My Name"  
date: "7/22/2020"  
output: html_document  
---
```

**WARNING! DO NOT TOUCH THIS SECTION YET!**

This section is a special header that provides instructions on how to export the final document (output: `html_document`) and can be changed with further instructions. This is formatted in a simple language called YAML⁵.

The rest of the page is meant to write text with or without (regular) markdown formatting, but also can contain R code that can be shown or hidden, executed or inert. It is worth pointing out that RStudio supports many more languages that just R and are called “engines” in that context⁶.

13.2.3 Adding R code

The whole purpose of an `.Rmd` file is to tell a story with markdown and perform the analysis at the same time when it is rendered. This is accomplished by adding R code “chunks” within the file that will be evaluated when the weaving/knitting of the file output is done.

To add R code we can use the “Insert” button on Rstudio bar, or simply write the code between special characters that specify that it is code and not just text in this way:

⁶Command `names(knitr::knit_engines$get())` will print supported languages (‘engines’). Install `knitr` package first.

```

```{r}
Here goes the R code
V <- c(1:10)
```

```

A name can be given to the “chunk” and a various number of options that can modify the results of what happens when the final document is knitted. For example the code could be running but not shown in the final document by adding `echo = FALSE`. ([Complete chunk options list\(PDF\)⁷](#).) It is easier to see an example:

```

---
title: "Tiny Rmd"
output: html_document
---
In R it is possible to tell a story and weave computer code
to perform an analysis at the same time by adding "chunks" of code.
This code will create a vector and take the log10 of each value

```{r mychunk, eval=TRUE}
This chunk will be computed
V <- c(1:10)
log10(V)
```

```

The values are automatically printed

When the **knit** button is pressed the rendering in HTML will look like this:

Exercise 13.1. Exercise

You can try to Copy/Paste the text for Tiny Rmd file above and paste it within a new .Rmd file (details in section 13.2.2,) replacing all of the demo content with the pasted text of Tiny Rmd. Then press the knit button and see the result!

⁷<https://rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf>

Tiny Rmd

In R it is possible to tell a story and weave computer code to perform an analysis at the same time by adding “chunks” of code. This code will create a vector and take the log10 of each value

```
# This chunk will be computed
V <- c(1:10)
log10(V)

## [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513 0.8450980
## [8] 0.9030900 0.9542425 1.0000000
```

The values are automatically printed

Figure 13.2: HTML output of Tiny Rmd as knit output.

13.2.4 Very tiny Rmd file: Inline code

Here is one of the most useful and somewhat **advanced** ways of using R code to avoid “Copy/Paste” of information that may be unstable and could change over time. For example the size (length, dimensions, etc.) of the provided data for R to analyze may be updated with new information.

Here is an example of a very small file that shows how R code can be embedded within the text and **rendered** in the context of reporting.

- The YAML is very minimal
- the first line prints out 5 letters from the English alphabet (LETTERS is pre-defined in R.)
- The second line embeds **two** commands separated by a semi-colon ; that first defines a vector of numbers, and then computes the sum of the numbers.
- In both cases the results are shown in bold.

```
---
title: "Tinyest Rmd"
output: html_document
---

Some random letters: **`r sample(LETTERS, 5)`**

Let's make a vector and add all its numbers:
**`r vec <- c(1:10); sum(vec)`**.
Only the results will show on the final print.
```

Pressing the RStudio Knit button will convert this .Rmd file into an HTML document.

Tinyest Rmd

Some random letters: O, A, G, U, L
Let's make a vector and add all its numbers: 55 - but only the results will show on the final print.

Figure 13.3: HTML output for Tinyest R markdown conversion with Knit button.

Exercise 13.2. Exercise: The story of vector V

You can read the “magical story of vector V” from the text in Appendix I that you can Copy/Paste into a new .Rmd file.

This is a way to learn by example about R code chunks and the very useful *inline* R code.

The *magic* is perhaps in the story, but more importantly it is also the demonstration of weaving text and code together in a single rendered document.

13.3 Other formats

The two formats that should work by default are HTML and Word. Most people would be interested in creating a PDF but that requires the installation of a *typesetting* engine called \LaTeX “LaTeX” (pronounced “lay tek.”) In the early days this required the installation of software independent of RStudio that was heavy in size in the multiple *Gigabytes* (most are 5Gb or more.)

TinyTex for PDF



Fortunately there is now a special package called TinyTex that is much easier to install and much smaller in size at about 150Mb only. Information on the package and installation instructions can be found on yihui.org/tinytex/ (Yihui Xie is a software engineer at RStudio and author of knitr and Tinytex among others.)



Optional Installation TinyTex

The **tinytex** R package (written as bold, lower case) is used to install *TinyTeX*, its distribution version of “Latex” \LaTeX (pronounce “la-tek.”)

The installation is simple and requires 2 easy steps:

1. install the **tinytex** package.
2. use **tinytex** to install the *TinyTeX* distribution.

Here are the 2 commands to accomplish this⁸ plus a third, commented command to uninstall if necessary.

```
install.packages('tinytex')
tinytex::install_tinytex()
# to uninstall TinyTeX, run tinytex::uninstall_tinytex()
```

13.4 A word on YAML

YAML is a *language* and therefore can be overwhelming, confusing and offer too many “options” (as most computer languages do.)

However, as the language of the **header** of the .Rmd files there are just a few things that are of real importance.

13.4.1 Limits

The header is limited by three dashes at the top and at the bottom. Beyond this limit it become the realm of R markdown.

13.4.2 Indentation and White space

White space is part of YAML's formatting. Unless otherwise indicated, newlines indicate the end of a field.

Indentations:

- * used to structure a YAML document.
- * only use white space, never Tabs.
- * in .Rmd indentation is 0, 2 or 4 spaces *exactly*.

⁸<https://yihui.org/tinytex/>

13.4.3 Automatic modifications

Parts of the YAML header may change automatically depending on actions. For example, suddenly decided to knit a simple document to a new format will modify the output statement.

```

---
title: "Tiny Rmd"
output: html_document
---
```

In the original version the keyword `output :` line contains a colon (`:`) followed the expected document format.

After requesting a different format, the output will automatically be changed, each time. The new `output :` line is now ending with a newline and the now multiple formats are each on a separate line ***indented by exactly 2 spaces*** (not 1, 3, or 4, or tab all of which would cause an error later.) The last document format requested will **always** be the one shown on top in the first indented line, updated each time the document is knitted.

```

---
title: "Tiny Rmd"
output:
  word_document: default
  html_document: default
  pdf_document: default
---
```

13.4.4 Quotes

Text should prudently be placed within double quotes, for example `title: "Tiny Rmd"` even though `title: Tiny Rmd` would also work. Adding the quotes as it is done by default prevents text with special characters to cause an error.

13.4.5 Date

When a new `.Rmd` file is created it is given the date true on that moment and would not change later.

It is possible to use code so that the date is updated each time the document is knitted into a final format. Here are options to format the date at that moment:

- `date: "Last Updated:" `r Sys.Date()` ` "`
- `date: ' `r Sys.Date()' ` '`
- `date: " `r format(Sys.time(), '%d %B, %Y')` "`
- `date: " `r format(Sys.time(), '%Y, %B %d')` "`

Which would result in the following formats:

- `date: "Last Updated: 2023-06-30"`
- `date: '2023-06-30'`
- `date: "30 June, 2023"`
- `date: "2023, June 30"`

13.4.6 YAML resources

For further reference see the online book [R Markdown: The Definitive Guide](#) that details advanced options for YAML headers:

- HTML content: <https://bookdown.org/yihui/rmarkdown/html-document.html>
- PDF content: <https://bookdown.org/yihui/rmarkdown/pdf-document.html>
- MSWord: <https://bookdown.org/yihui/rmarkdown/word-document.html>
- General output formats: <https://bookdown.org/yihui/rmarkdown/output-formats.html>

An interesting way to see if your YAML header has any errors:

- YAML validator: <http://www.yamllint.com/>

Chapter 14

Report-template

Here are a few suggestions to help in the writing of a “report” for the analysis of your NHANES chemical data that can be created as an R markdown document containing all of the report narrative (story,) analysis code (shown or hidden,) graphs and illustrations.

A single file with most of the code from this chapter as a “template” can be found online as a plain text file but with either `.Rmd` or `.Rmd.txt` filename extensions. The content is identical in both files. Depending on the settings of your browser most likely the `.txt` version will appear within the browser. For just `.Rmd` it may appear within the browser or be downloaded automatically in your default Downloads folder.



Sample report template in R markdown for download:

- [SampleReport.Rmd.txt¹](#)
- [SampleReport.Rmd²](#)

14.1 Overall template format

The report should be in the R markdown format with a YAML header and a body with markdown and R code. A minimal outline could be:

```
---  
title: "NHANES report"  
author: "your name"  
output: html_document  
---
```

```
# Preface {-}
```

Some background on something if wanted. Or remove

```
# Introduction
```

Some useful into.

```
# Chemical info
```

The chemical studied and why

```
# NHANES data
```

What is it, where to find

```
## Download
```

In this section the download code an optionally be hidden

```
## Selected data
```


What are the columns of data to be used.

Analysis

Some kind of analysis. With R code shown or not.
May include tables, graphics etc.

Results

This may be a summary of some of the analysis

Conclusion

Is there a general conclusion that can be drawn from the analysis and the results?

14.2 YAML example

This part may be simple or more complex, for example requesting figure caption or requesting the automatic creation of a table of contents with a specified number of levels. For HTML the table can be “floating” as shown on the left hand side for easier navigation.

Here is an example YAML for all 3 major output formats with these options. The line `fontsize: 12pt` is most useful for PDF output avoiding the 10pt default.

```
title: "NHANES Report Example"
author: "Your Name Here"
date: " 30 June, 2023 "
output:
  word_document:
    toc: true
```

```

    toc_depth: 2
    fig_caption: true
pdf_document:
  toc: true
  toc_depth: 2
  fig_caption: true
  number_sections: true
html_document:
  toc: true
  toc_depth: 2
  toc_float: true
  fig_caption: true
  number_sections: true
fontsize: 12pt
---
```

14.3 General chunk options

The default R markdown template in RStudio automatically adds this general option chunk that can be expanded. For example, to make **all R code hidden** change `echo = TRUE` to `FALSE`. The code in these options apply to all code in the document but can be overridden by placing the opposite or desired option within the individual `{r}` tags in each chunk.

```

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

14.4 Preamble, Preface and Introduction

“Preamble” or a “Preface” are often optional but usually not numbered as the rest of the sections. To prevent numbering we add {-} so that the numbering would start at the next heading tag # for example for # Introduction.

14.5 Activating packages

It may be useful to activate some packages at the beginning to make sure they are “up” for later. Hiding the code is useful for a generic report. The default RStudio options of `echo=FALSE` and `warning=FALSE` are not enough to suppress all messages. The following code should load `tidyverse` quietly:

```
```{r eval=TRUE, echo=FALSE, warning=FALSE}
Load quietly here and add a code below with eval=FALSE
options(tidyverse.quiet = TRUE)
library(tidyverse)
```
```

14.6 Live web links

NHANES or other web references can be created as “live” links in all document types by using the format [Name in Square brackets] (<http://web.site.address.here>)
For example: [NHANES web site] (<https://www.cdc.gov/nchs/nhanes/>).

14.7 Embedding graphs

Graphs can be embedded with optional legends. The age distribution histogram without showing R code. Since age ranges from 0 to 80 there are 81 “slots” all

represented individually by specifying `breaks = 81`. Alignment can be specified. Optionally width and height are added and expressed in inches by adding `fig.width=7, fig.height=5`.

```
```{r echo = FALSE, fig.cap="Histogram of age distribution", fig.align='center'}
with(nhanesDemo , hist(RIDAGEYR, breaks = 81))
```
```

14.8 Inline code

Inline code is the **secret** that can help make your report precise and useful as it allows you to access and print information in the report that you do not have to know and most of all that is not necessary to copy/paste.

Inline code can be fancy and contain more than just a simple computation such as ``r 1+1``. Indeed it can even be a pipeline as shown in this example:

```
There are `r dim(nhanesDemo)[1]` observation for `r select(nhanesDemo,
RIDAGEYR) %>% filter(RIDAGEYR < 18 ) %>% count()` children
participants less than 18, `r select(nhanesDemo, RIDAGEYR) %>%
filter(RIDAGEYR > 18 & RIDAGEYR < 80) %>% count()` adult par-
ticipants between 18 and 79 and `r select(nhanesDemo, RIDAGEYR) %>%
filter(RIDAGEYR >= 80) %>% count()` adults over the age of 80.
```

This will be rendered in the final text as:

There are 9971 observation for 3979 children participants less than 18, 5478 adult participants between 18 and 79 and 376 adults over the age of 80.

14.9 Math formula

Examples of math formulas can be found at:

- [Mathematics in R Markdown R Pruim³](#) and
- [An Example R Markdown⁴](#)

One \$ sign keep the formula in line. Two \$\$ make the formula displayed on a different line. For example:

The creatinine adjustment requires a division and a multiplication by 10^{-4} . The final formula is

$$ratio = \frac{Analyte}{Creatinine} * 10^{-4}$$

14.10 Addendum

In a technical report is it customary to also report how your R session was at the moment of computation. This is accomplished by adding the command `sessionInfo()` In this example the `eval=FALSE` makes that the code is not run. Update as needed depending on the type of report that you write and who it is for.

```
sessionInfo()
```

³<https://www.calvin.edu/~rpruim/courses/s341/S17/from-class/MathinRmd.html>

⁴<http://www.math.mcgill.ca/yyang/regression/RMarkdown/example.html>

Chapter 15

Report resources

Here are a few useful resources that can be useful when creating a report.

15.1 Illustrations

It is on occasion desirable to add an illustration to a report that may either highlight an idea or a concept. However, it is not always possible to use just any image found on the Internet due to copyright issues.

The following web site provide free images, illustrations, line art at no charge and without the need of citation. In fact many illustrations were used from one of these in this document. Illustrations are marked

- Free to use.
- No attribution required.

Table 15.1: Free online images and illustrations

| Name | Web site |
|---------|---|
| Pixabay | https://pixabay.com/ |
| Pexels | https://www.pexels.com/ |

15.1.1 Adding and sizing images

Images can be too large and especially for HTML smaller physical and file sizes are desirable and in that case it is better to use an image editor to reduce the file size to start with.

The standard markdown code to add an image in .png, .gif, or .jpg format is simple, for example using the image `myimage.png` in the current directory or `mydir` directory. The square bracket can be left empty or can contain “Alt text” which is shown in the web browser if the mouse is “hovered” over the included image.

- `![] (myimage.png)`
- `![] (mydir/myimage.png)`
- `![Alt text here] (myimage.png)`

By default this notation will present the image “as it is” in its dimensions and there is no figure legend possible. Fortunately it is possible to control the width (only but the height is deduced) by adding a specification in pixels inside curly brackets connected to the last parenthesis. For example:

- `![] (myimage.png){width=100px}`

However, in R markdown it may be best to use the `knitr::include_graphics()` function. For example:


```
```{recho=FALSE, fig.cap="Caption here"}  
Include image name
knitr::include_graphics("images/myimage.png")
```
```

15.2 Markdown tables

Creating tables in markdown is not very difficult but it can be time consuming if the table is a bit complicated. [Tableconvert](#) offers conversion between various table formats from files or pasted text and [Table Generator](#) can create empty table to fill.

Table 15.2: Table format conversions including Excel and markdown.

| Name | Web site |
|------------------|--|
| Table convert | tableconvert.com |
| Tables Generator | tablesgenerator.com |

Appendix A

The story of R



Note: appendices are labeled with letters.

A very complete history of R was written by Roger D. Peng that includes a history of its ancestors S and S-Plus and available on the web version¹ of his book (Peng (2016)) as well as a [16min video](#)² detailing the creation of S and then R.

S is a statistical programming language developed primarily by [John Chambers](#) and (in earlier versions) Rick Becker and Allan Wilks of [Bell Laboratories](#). The aim of the language, as expressed by John Chambers, is “*to turn ideas into software, quickly and faithfully*”.

R is a programming language and free software environment for statistical computing and graphics. R was developed by [Ross Ihaka](#) and [Robert Gentleman](#). They reported their experience developing R in 1996 ([Ihaka and Gentleman \(1996\)](#).)

¹<https://bookdown.org/rdpeng/rprogdatascience/history-and-overview-of-r.html>

²<https://youtu.be/STihTnVSZnI>

Appendix B

Simple math

It is assumed that students are familiar with basic mathematics and their related symbols or *arithmetic operators*. However, some symbols may be different. For example the *multiplication* symbol within R is `*` rather than `x` or `.` when writing it by hand.

Here are a few reminders of mathematical operators and their symbols for arithmetic and logical operations.

The first 3 minutes of this 7 minutes video *Arithmetic, Rational, Logical Operators - Introduction to R Programming - Part 4*¹ summarizes the tables below.

B.1 Arithmetic operators

Here is a table defining the arithmetic operators represented by the symbols used within R. These operators are used on numbers or groups of numbers.

¹https://youtu.be/wX_ArwIiRxs

Table B.1: Arithmetic operators and their symbols in R

| Operator | Description |
|----------|----------------|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ^ or ^ | Exponentiation |
| %% | Modulo |

Depending on the complexity of the calculation it may be necessary to use parenthesis (()) to separate values.

It is important to remember the notion of *precedence* as detailed below from Wikipedia.²

In mathematics and computer programming, the order of operations (or operator precedence) is a collection of rules that reflect conventions about which procedures to perform first in order to evaluate a given mathematical expression.*

The order is: **P**arentheses, **E**xponents, **M**ultiplication, **D**ivision, **A**ddition, **S**ubtraction. In the United States, the acronym **PEMDAS** is common*. (See Wiki reference for other countries.)

Misinterpreting any of the above rules to mean “addition first, subtraction afterward” would incorrectly evaluate the expression

$$10 - 3 + 2.$$

The correct value is 9 (not 5, as would be the case if you added the 3 and the 2 before subtracting from the 10).

²https://en.wikipedia.org/wiki/Order_of_operations

B.2 Boolean values

A Boolean value is either **true** or **false**. These values can be the result of a logical operator (see below) or a statement within an R function, for example stating that there is (T) or there isn't (F) a header in a table of numbers.

Table B.2: Boolean values

| Value | Notation |
|--------------|-------------------------|
| true | TRUE or T in uppercase |
| false | FALSE or F in uppercase |

B.3 Rational operators

Table B.3: Rational operators

| Operator | Description |
|----------|--------------------------|
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| == | Exactly equal to |
| != | Not equal to |

B.4 Logical operators

Logical operators can be used to create *conditional statements* as they result in Boolean values of **true** or **false**.

The symbols used imply the boolean operators “AND”, “OR” and “NOT”.

| Operator | Description |
|------------------------|----------------------------------|
| <code>x & y</code> | x AND y |
| <code>x y</code> | x OR y |
| <code>!x</code> | NOT x |
| <code>isTRUE(x)</code> | Test if x has Boolean value TRUE |

Appendix C

Import NHANES sample code

The online NHANES tutorials provide the following sample code to import data into R.

- Sample code tutorial page page:
 - <https://wwwn.cdc.gov/nchs/nhanes/tutorials/SampleCode.aspx>
- R code to import SAS .XPT transfer data files:
 - https://wwwn.cdc.gov/nchs/data/tutorials/file_download_import_R.R

This code is reproduced below:

```
# Code from page:
# https://wwwn.cdc.gov/nchs/data/tutorials/file_download_import_R.R

#####
# Example code to download/import NHANES data files (SAS transport .XPT files) as a dataset #
# For R #
#####

## Note to tutorial users: you must update some lines of code (e.g. file paths)
## to run this code yourself. Search for comments labeled "TutorialUser"

# Include Foreign Package To Read SAS Transport Files
```

```

library(foreign)

#####
## Example 1: import SAS transport file that is saved on your hard drive ##
#####

# First, download the NHANES 2015-2016 Demographics file and save it to your hard drive #
# from: https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Demographics&CycleBeginYear=2015 #
# You may need to right-click the link to the data file and select "Save target as..." #

# Create data frame from saved XPT file
# TutorialUser: update the file path here
# for Windows users, be sure to change the slashes between directories to a forward slash / (as on Mac or Unix)
# or to double backslashes \\

DEMO_I <- read.xport("C:\\NHANES\\DATA\\DEMO_I.xpt")
DEMO_I2 <- read.xport("C:/NHANES/DATA/DEMO_I.xpt")

# this code with typical Windows single backslashes between directories will throw an error
#DEMO_I <- read.xport("C:\\NHANES\\DATA\\DEMO_I.xpt")

# save as an R data frame
# TutorialUser: update the file path here to a directory where you want to save the data frame
saveRDS(DEMO_I, file="C:\\NHANES\\DATA\\DEMO_I.rds")

#####
## Example 2: Download and import the transport file through R ##
#####

# Download NHANES 2015-2016 to temporary file
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/DEMO_I.XPT", tf <- tempfile(), mode="wb")

# Create Data Frame From Temporary File
DEMO_I3 <- foreign::read.xport(tf)

# save as an R data frame
# TutorialUser: update the file path here to a directory where you want to save the data frame
saveRDS(DEMO_I3, file="C:\\NHANES\\DATA\\DEMO_I.rds")

```

Appendix D

Merge Downloads into a Master file

Examples for downloading and merging NHANES .XPT data files from the following data:

Documentation links:

- DEMO_I: https://wwwn.cdc.gov/nchs/nhanes/2015-2016/DEMO_I.htm
- BMI_I: https://wwwn.cdc.gov/nchs/nhanes/2015-2016/BMX_I.htm
- PFAS_I: https://wwwn.cdc.gov/nchs/nhanes/2015-2016/PFAS_I.htm
- TCHOL_I: https://wwwn.cdc.gov/nchs/nhanes/2015-2016/TCHOL_I.htm
- ALB_CR_I: https://wwwn.cdc.gov/nchs/nhanes/2015-2016/ALB_CR_I.htm

D.1 Download into R object with NHANES code

Downloaded data is saved in a temporary file in a temporary directory and `tf` simply holds the name/path to that data. Example on a Mac:

```
> tf  
[1] "/var/folders/zg/9hl9fx_n7b970gcj51t8tkq1xx62d5/T//RtmpL6j3Wp/file5d87195178e"
```

```

# Download NHANES 2015-2016 to temporary file: DEMO_I
download.file(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/DEMO_I.XPT",
  tf <- tempfile(), mode="wb")
# Create Data Frame From Temporary File
DEMO_I <- foreign::read.xport(tf)
#####
# REPEAT For:
# BMX_I
download.file(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/BMX_I.XPT",
  tf2 <- tempfile(), mode="wb")
BMX_I <- foreign::read.xport(tf2) # TMP file
# PFAS_I #####
download.file(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/PFAS_I.XPT",
  tf3 <- tempfile(), mode="wb")
PFAS_I <- foreign::read.xport(tf3) # TMP file
# TCHOL_I #####
download.file(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/TCHOL_I.XPT",
  tf4 <- tempfile(), mode="wb")
TCHOL_I <- foreign::read.xport(tf4) # TMP file
# ALB_CR_I #####
download.file(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/ALB_CR_I.XPT",
  tf5 <- tempfile(), mode="wb")
ALB_CR_I <- foreign::read.xport(tf5) # TMP file

```

D.2 Combine files into Master

Use `merge()` function. The `SEQN` column is common to all and `merge()` will automatically identify it. `all.x=TRUE` will keep all rows and fill non existent data with `NA` so that all data are kept.

```
Master1 <- merge(DEMO_I, BMX_I, all.x=TRUE)
Master2 <- merge(Master1, PFAS_I, all.x=TRUE)
Master3 <- merge(Master2, TCHOL_I, all.x=TRUE)
Master4 <- merge(Master3, ALB_CR_I, all.x=TRUE)
```

D.3 Save/Write Master file to disk

If available use `write_csv()` function (`dplyr` package) which is “twice as fast as `write.csv()`”, and never writes row names. For example to export the data as `.csv` within the current directory:

```
library(dplyr)
write_csv(Master4, "Master4.csv")
```

Base R version:

```
write.csv(Master4, "Master4.csv")
```

D.4 Alternate download to R object with haven

Using the `haven` package the code may look and feel easier as it only requires one line per file .

```
library(haven)
#
DEMO_I <- read_xpt(url(
  "https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/DEMO_I.XPT"))
#
BMX_I <- read_xpt(url(
  "https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/BMX_I.XPT"))
#
PFAS_I <- read_xpt(url(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/PFAS_I.XPT"))
#
TCHOL_I <- read_xpt(url(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/TCHOL_I.XPT"))
#
ALB_CR_I <- read_xpt(url(
  "https://wwwn.cdc.gov/nchs/nhanes/2015-2016/ALB_CR_I.XPT"))
```

D.5 Download, save XPT files to hard drive

To just download the .XPT files on your hard drive:

```
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/DEMO_I.XPT",
  "DEMO_I.XPT")
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/BMX_I.XPT",
  "BMX_I.XPT")
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/PFAS_I.XPT",
  "PFAS_I.XPT")
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/ALB_CR_I.XPT",
  "ALB_CR_I.XPT")
download.file("https://wwwn.cdc.gov/nchs/nhanes/2015-2016/TCHOL_I.XPT",
  "TCHOL_I.XPT")
```

Appendix E

PFAS_I codes

Source: web documentation at https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/PFAS_I.htm

Table E.1: PFAS_I analysis code

| Code | Description |
|----------|--|
| SEQN | Respondent sequence number |
| WTSB2YR | Subsample B weights |
| LBXPFDE | Perfluorodecanoic acid (ng/mL) |
| LBDPFDEL | Perfluorodecanoic acid Comment Code |
| LBXPFHS | Perfluorohexane sulfonic acid (ng/mL) |
| LBDPFHSL | Perfluorohexane sulfonic acid Comt Code |
| LBXMPAH | 2-(N-methyl-PFOSA)acetic acid (ng/mL) |
| LBDMPAHL | 2-(N-methyl-PFOSA) acetic acid Comt Code |
| LBXPFNA | Perfluorononanoic acid (ng/mL) |
| LBDPFNAL | Perfluorononanoic acid Comment Code |
| LBXPFUA | Perfluoroundecanoic acid (ng/mL) |
| LBDPFUAL | Perfluoroundecanoic acid Comment Code |
| LBXPFDO | Perfluorododecanoic acid (ng/mL) |
| LBDPFDOL | Perfluorododecanoic acid comment |

| Code | Description |
|----------|--|
| LBXNFOA | n-perfluorooctanoic acid (ng/mL) |
| LBDNFOAL | n-perfluorooctanoic acid Comment Code |
| LBXBFOA | Br. perfluorooctanoic acid iso (ng/mL) |
| LBDBFOAL | Br. perfluorooctanoic acid iso Comt Code |
| LBXNFOS | n-perfluorooctane sulfonic acid (ng/mL) |
| LBDNFOSL | n-perfluorooctane sulfonic Comt Code |
| LBXMFOS | Sm-PFOS (ng/mL) |
| LBDMFOSL | Sm-PFOS Comment Code |

Appendix F

Perfluoroalkyl and polyfluoroalkyl

Table 1. From [Buck et al. \(2011\)](#): *Examples of the correct and incorrect (or undesirable) uses of the proposed nomenclature for perfluoroalkyl and polyfluoroalkyl substances (PFASs).*

Table 1. Examples of the correct and incorrect (or undesirable) uses of the proposed nomenclature for perfluoroalkyl and polyfluoroalkyl substances (PFASs)

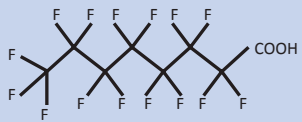
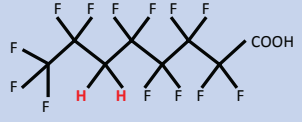
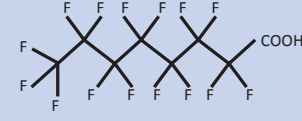
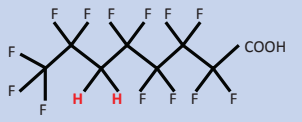
| Example | Example statements | |
|---|---|--|
| | Correct | Incorrect or undesirable |
|  | <ul style="list-style-type: none"> Both are PFASs, within the family of perfluoroalkyl and polyfluoroalkyl substances | <ul style="list-style-type: none"> Both are: <ul style="list-style-type: none"> Perfluoroalkyl substances, chemicals, compounds Perfluorinated substances, chemicals, compounds Polyfluoroalkyl substances Polyfluorinated substances Fluorocarbons Perfluorocarbons Fluorinated substances, chemicals, compounds Perfluorochemicals Perfluorinated chemicals Both contain fluorocarbons |
|  | <ul style="list-style-type: none"> Both are carboxylic acids | |
|  | <ul style="list-style-type: none"> All H atoms on all C atoms in the alkyl chain attached to the carboxylic acid functional group are replaced by F This is a: PFAS, perfluoroalkyl acid (PFAA), perfluoroalkyl carboxylic acid (PFCA) Specifically, this is perfluorooctanoic acid, CAS number 335-67-1 | <ul style="list-style-type: none"> This is a: <ul style="list-style-type: none"> Perfluorinated substance, chemical, compound Fluorinated substance, chemical, compound Fluorocarbon Perfluorocarbon |
|  | <ul style="list-style-type: none"> The alkyl chain attached to the carboxylic acid functional group is polyfluorinated This is a: PFAS, polyfluoroalkyl acid, polyfluoroalkyl carboxylic acid Specifically, this is 2,2,3,3,4,4,5,5,7,7,8,8,8-tridecafluorooctanoic acid | <ul style="list-style-type: none"> This is a: <ul style="list-style-type: none"> Polyfluorinated substance, chemical, compound Fluorinated substance, chemical, compound Perfluorinated substance, chemical, compound A portion of this compound is perfluorinated |

Figure F.1: proposed nomenclature for perfluoroalkyl and polyfluoroalkyl substances

Appendix G

ggplot2 tutorials online

Note: The “official” ggplot2 book is available online and contains tutorial materials: ggplot2-book.org/)

Table G.1: ggplot2 tutorials online

| Name | Web site |
|--|---|
| Quick Introduction to ggplot2 | https://bookdown.org/agrogankaylor/quick-intro-to-ggplot2/quick-intro-to-ggplot2.html |
| The Complete ggplot2 Tutorial | http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html |
| Introduction to GGPlot2 | https://www.datanovia.com/en/lessons/introduction-to-ggplot2/ |
| Data Visualisation with ggplot2 | https://datacarpentry.org/r-socialsci/04-ggplot2/index.html |
| R graphics with ggplot2 workshop notes | http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html |
| Introduction to ggplot2 | https://opr.princeton.edu/workshops/Downloads/2015Jan_ggplot2Koffman.pdf |

Specialized tutorials on bar charts:

Table G.2: Tutorials on bar graph

| Name | Web site |
|---|---|
| Grouped barchart - ggplot2 | https://www.r-graph-gallery.com/48-grouped-barplot-with-ggplot2 |
| Grouped barplot in R with error bars | http://environmentalcomputing.net/plotting-with-ggplot-bar-plots-with-error-bars/ |
| Plotting with ggplot: bar plots with error bars | http://environmentalcomputing.net/plotting-with-ggplot-bar-plots-with-error-bars/ |
| Tutorial: Turning a Table into a Horizontal Bar Graph using ggplot2 | https://rstudio-pubs-static.s3.amazonaws.com/4305_8df3611f69fa48c2ba6bbca9a8367895.html |
| 8 tips to make better barplots with ggplot2 in R | https://cmdlinetips.com/2019/10/barplots-with-ggplot2-in-r/ |
| Grouped, stacked and percent stacked barplot in base R | https://www.r-graph-gallery.com/211-basic-grouped-or-stacked-barplot.htm |

Appendix H

Rmarkdown resources

A web engine search will provide a lot of possible references for R markdown. Here are a few that I found useful:

Table H.1: Rmarkdown tutorials online

| Name | Web site |
|--|---|
| Getting Used to R, RStudio, and R Markdown
Chester Ismay and Patrick C. Kennedy
2019-11-12 | https://ismayc.github.io/rbasics-book/ |
| R Markdown Quick Tour - Overview (Video) | https://rmarkdown.rstudio.com/authoring_quick_tour.html |
| Introduction to R Markdown | https://rmarkdown.rstudio.com/articles_intro.html |
| Knitr with R Markdown | https://kbroman.org/knitr_knutshell/pages/Rmarkdown.html |
| R Markdown and Publishing (R Cookbook, 2nd Edition) | https://rc2e.com/rmarkdown |

| Name | Web site |
|-----------------------------------|---|
| Writing documents with R Markdown | https://monashbioinformaticsplatform.github.io/2017-11-16-open-science-training/topics/rmarkdown.html |
| R markdown document | https://mgimond.github.io/ES218/Misco1.html |
| R Markdown (for Data Science) | https://r4ds.had.co.nz/r-markdown.html |
| Getting started in R markdown | https://www.statsandr.com/blog/getting-started-in-r-markdown/ |

Templates are also available online, for example:

Table H.2: Rmarkdown templates

| Name | Web site |
|----------------------|---|
| R Markdown TEMPLATES | https://rmarkdown.rstudio.com/gallery.html |

Appendix I

The Story of Vector V: an R markdown example

This is an example of R markdown text that can be cut and pasted in a new R markdown document (section 13.2.2) and then knitted into an HTML or other type of document. This illustrates the use of R code in chunks (shown or hidden) and as inline commands.

Note: Depending on the format (HTML/PDF) of this document some elements might have a background color. However, **ALL** these apparent parts constitute the .Rmd file.

```
---
title: "The story of vector V"
output: html_document
---
```

```
# `V` learns numbers
```

Once upon a time there was a vector named ``V`` that was feeling empty and was trying to learn numbers from ``0`` through ``9``. One day ``V`` met the **magical** combine function ``c()`` that was able to add the

```
numbers ***inside*** `V`, like this:
```

```
`r`{r}
V <- c(1,2,3,4,5,6,7,8,9)
`r`
```

`V` was very happy, and `V` was now spending its time enumerating the numbers: ``r V``. Sometimes it would pick one at random: ``r sample(V, 1)`` and it was pleased that it was not always the same number coming up.

```
## `V` wonders about itself
```

Wanting to know itself better `V` asked:

```
* what is my class? And the answer was: `r class(V)`
* how long am I? And the answer came as `r length(V)`
```

```
# `V` wants more
```

But then `V` wanted more: it wanted to add these numbers but not `in` the open like this, it wanted to do that `"in its head"` so it could be done like this: `**`r sum(V)`` (*and the value will be printed here directly as calculate by ``R`.*`)

```
## `V` meets the Math Wizard
```

But how to describe that to `*Math Wizards*`?

He asked his `*fairy*` friend `*Equation*` who gave `V` the `*magic*` codes:

```
$$\sum_{n=1}^{\{9\}} n = sum(V)$$
```

which is still `**`r sum(V)``.


```
## `V` in the land of vectorization
```

But it wanted more again... `V` wanted to be 10 times more. So `V` went on a journey across the land to know what to do. It was a long and arduous journey, but `V` ended in the "Land of Vectorization" and there, he was augmented 100 times to be like this: ``r 100*V``. But it was cumbersome to feel these big numbers and `*division*` helped one more time to make it just 10 times smaller to be ``r 10*(V)``

```
# `V` and the mental picture
```

This time it wanted to have a mental "picture" of the numbers and it could think of 2 ways but it had to keep the ``R`` code `*secret*` so that it would not be stolen:

```
```{r echo=FALSE, fig.height=3}
par(mfrow = c(1,2))
plot(V)
boxplot(V)
par(mfrow = c(1,1))
```
```

```
# Conclusion
```

It is useful to have friends that help you, and `V` is very grateful of its magical encounter with ``c()`` and other friends along the way.

About the authors

Jean-Yves Sgro, a senior scientist with years of experience in using and teaching computer programs, creates, organizes and teaches hands on workshops.



Jean-Yves Sgro, a senior scientist with years of experience in using and teaching computer programs, creates, organizes and teaches the workshops.

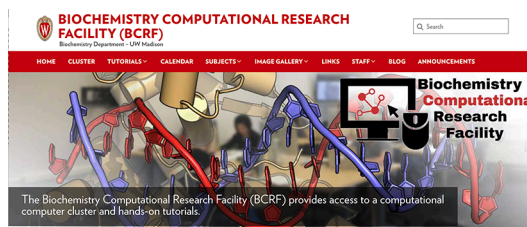
Jean-Yves has been at UW since 1986 after a Master in Physiology and a Ph.D. in Cellular and Molecular Biology from [Joseph Fourier University](#), Grenoble, France, and researched at the European Molecular Biology Laboratory ([EMBL](#)) where he already used large computers for sequence analysis.

In Madison, at the Institute for Molecular Virology ([IMV](#)) he continued developing computer expertise in addition to his wet-lab research – 3D molecular visualization ([virusworld](#)), RNA-folding predictions, sequence and data analysis...

In 1996 he joined the [UW Biotechnology Center](#) to better help Campus biologists analyze and visualize their data while continuing research at IMV until 2014 when this part-time position was transferred to the [Biochemistry Department](#) where

he organizes and teaches hands-on tutorials on molecular graphics, data analysis as a support to the department personnel.

Hist tutorials are available on line from the The Biochemistry Computational Research Facility ([BCRF](#).)



Jean-Yves also volunteers as an [instructor](#) for the [Carpentries](#) global community data science workshops.



Summary: Kristen Malecki is Associate Professor in Population Health Sciences¹ and Director and PI of Survey of the Health of Wisconsin (SHOW)²

Her research interests are: Environmental health, epidemiology, survey research methods, metabolic health and immune function, epigenetics, microbiome and applied public health practice.

Bio: In June 2022 Dr. Kristen Malecki has joined the University of Chicago School of Public Health (Twitter: @uicpublichealth) as the new director of the Division of Environmental and Occupational Health Sciences.

Her [current bio at UIC](#):

Kristen Malecki, PhD, MPH is Professor of Environmental and Occupational Health Sciences (EOHS) at the University of Illinois at Chicago (UIC), School of Public Health, where she also serves as the EOHS Division Director. She has a PhD in Environmental Epidemiology and Health Policy and Masters of Public Health from Johns Hopkins University Bloomberg School of Public Health. She was recently appointed to the National Academies of Sciences standing committee on the “Use of Emerging Science for Environmental Health Decisions.” She uses a multi-omic approach to examine combined chemical (air pollution, water pollution), physical and social stressors, and their influence on adult chronic disease, aging and health disparities.) As a member of the Molecular Environmental Toxicology Center, her transdisciplinary work uses

¹(<https://pophealth.wisc.edu/staff/malecki-kristen/>)

²<https://show.wisc.edu/>

epigenetics, transcriptomics in both human and animals studies of the gut microbiome and to identify interim biomarkers of exposure and response to improve understanding of the biological mechanisms underlying persistent health disparities.

She also serves as the Principal Investigator for a number of community-academic partnerships and is committed to advancing urban and rural health equity. She also maintains longstanding partnerships with colleagues at the Wisconsin Department of Health Services which has facilitated a breadth of applied public health initiatives. Before becoming an academic, she served as the lead epidemiologist for the State of Wisconsin Environmental Public Health Tracking Program. In these roles she gained extensive experience in leading and managing multi-disciplinary teams of researchers, practitioners, and policy makers in development of new approaches to addressing environmental and occupational health challenges.

Previously, Dr. Kristen Malecki was an Associate Professor in the Department of Population Health Sciences at the University of Wisconsin-Madison (Twitter: @uwsmph.) Her previous UW-Madison *bio* stated*:

Dr. Malecki serves as the co-director for the Survey of the Health of Wisconsin (SHOW), overseeing survey implementation efforts and ancillary study development. She has been a leader in the development and evaluation of indicators for environmental health risk assessment and policy. Dr. Malecki also works to bridge applied public health practice with academic research focusing on environmental health and health disparities using a social determinants of health model. She recently served as Principal Investigator for the Wisconsin Groundwater Coordinating Council project addressing vulnerability among private well owners in Wisconsin. Her current research is also focused on developing models to examine combined chemical (air pollution, water pollution), physical and social stressors and influence on adult chronic disease, childhood development and obesity. She is a member of the University of Wisconsin National Institute for Environmental Health Breast Cancer and the Environment Research Program (coordinating center). Her transdisciplinary work includes identification of biomarkers of expression and response using epigenetics and transcriptomics. She also

serves as the Principal Investigator for a number of SHOW ancillary studies involving community-academic partnerships.

Before coming to the UW she served as the lead epidemiologist for the state Environmental Public Health Tracking Program. In these roles she has gained extensive experience in leading and managing multi-disciplinary teams of researchers, practitioners, and policy makers in development of environmental health surveillance and epidemiologic data for addressing chronic diseases and disparities in the State of Wisconsin and the nation.

Her teaching interests and experience spans from environmental health to survey research methods and applied public health practice.

Acknowledgments

I.1 R packages used for the book

R base ([R Core Team \(2020b\)](#)) and other added packages

Book creation: [Xie \(2020a\)](#), [Allaire et al. \(2020\)](#), [Xie \(2020b\)](#).

Book template adapted from rstudio4edu-book:

* <https://rstudio4edu.github.io/rstudio4edu-book/>

Tidyverse: [Wickham et al. \(2019\)](#)

Data import: [R Core Team \(2020a\)](#), [Wickham and Miller \(2020\)](#)

I.2 Extra Icons used:

I.2.1 Exercise / Homework



Icon made by [Prosymbols](#) from [Flaticon](#)

- link: https://www.flaticon.com/free-icon/homework_748646

I.2.2 Study at home:



Icon made from [Icon Fonts](#) is licensed by [CC BY 3.0](#). Recolored version by JYS.

- link: <https://www.onlinewebfonts.com/icon/532202>

References may be placed here or may be found on each page when cited depending on the format (HTML, PDF...) of this document.

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 2.1.
- Beals, K. A. (2008). Nutrition and well-being a to z.
- Buck, R. C., Franklin, J., Berger, U., Conder, J. M., Cousins, I. T., de Voogt, P., Jensen, A. A., Kannan, K., Mabury, S. A., and van Leeuwen, S. P. J. (2011). Per-fluoroalkyl and polyfluoroalkyl substances in the environment: Terminology, classification, and origins. *Integrated Environmental Assessment and Management*, 7:513 – 541.
- Chambers, J. M., Cleaveland, W. S., Kellner, B., and Tukey, P. A. (1985). *Graphical Methods for Data Analysis*. Wadsworth (U.A.).
- Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.
- of the Vice Provost, O. (2013). UW-Madison Strategic Diversity Update. Accessed: 7-29-2020.
- Peng, R. D. (2016). *R Programming for data science*. Leanpub.
- R Core Team (2020a). *foreign: Read Data Stored by 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...* R package version 0.8-76.
- R Core Team (2020b). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

- Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemond, 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 Miller, E. (2020). *haven: Import and Export 'SPSS', 'Stata' and 'SAS' Files*. R package version 2.3.1.
- Wickham, H. and Sievert, C. (2016). *ggplot2: elegant graphics for data analysis*. Springer.
- Wilkinson, L. (2005). *The Grammar of Graphics*. Statistics and Computing. Springer, 2nd edition.
- Xie, Y. (2020a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.18.
- Xie, Y. (2020b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.28.

Index

- airquality, 50
- albumin, 111
- alkyl, 115
- analyte, 111
- assignment operator
 - `<-`, 21
 - `=`, 21
 - R, 21
- BMI
 - distribution, 115
- boxplot, 57
- cholesterol, 101
 - HDL, 101
 - LDL, 101
 - total, 101
 - triglycerides, 101
- Classic R, 79
- code chunk options, 202
- code chunks
 - R markdown, 189
- coefficients: linear regression, 118
- comma delimited, 80
- conditional statement, 43, 144
- correlation factor
 - Pearson, 118
- creatinine, 107
 - adjust. equation, 108
 - adjustment, 107, 108, 131
 - ALB_CR, 109
 - concentration, 108
 - corrected, 108
 - equation, 131
 - excreted, 108
 - lean body mass, 108
 - levels, 108
 - mg/dL, 111
 - ratio, 111, 112, 131
 - urine dilution, 107, 108
 - weight/volume, 111
- cylinder, 125
- data frame, 34
 - tibble, 127
- data stream
 - `..`, 129
 - `data=.`, 129
 - pipe, 125
- data wrangling, 133, 171
- datasets
 - airquality, 50
 - R, 5
 - subsetting, 54

- decimal point
 - Europe, 80
 - USA, 80
- dplyr
 - video course, 135
- dynamic document, 181
 - computer code, 186
 - narrative, 186
 - woven, 186
- equation
 - math notation, 205
- error bars
 - sd, 168
- Europe, 80
- factor, 38, 71
- file type
 - csv, 79
 - R, 14
 - Rmd, 188
 - tab, 80
 - txt, 80
 - XPT, 86
- functions
 - abline(), 67, 119
 - arguments, 26
 - arrange(), 129, 153
 - as.factor(), 60
 - as.integer(), 174
 - as.numeric(), 77
 - barplot(), 153
 - boxplot(), 45
 - by_group(), 129
 - cbind(), 32
 - class(), 27, 28
 - colnames(), 36
 - colSums(), 52
 - cor(), 118
 - data.frame(), 34
 - dim(), 40, 52
 - download.file(), 84
 - drop_na(), 141, 161
 - expand.grid(), 39
 - filter(), 129, 139
 - formatC(), 150
 - getwd(), 29
 - ggarrange(), 73
 - ggplot(), 68
 - gl(), 38
 - head(), 51
 - help(), 29
 - hist(), 57
 - ifelse(), 43, 144
 - include_graphics(), 208
 - is.na(), 52
 - lapply(), 95
 - length(), 29
 - levels(), 61, 71
 - library(), 68
 - lm(), 66
 - log(), 90
 - log10(), 113
 - ls(), 27
 - matrix(), 31
 - mean(), 55
 - merge(), 103, 121

mutate(), 129
names(), 34
palette(), 59
par(), 44
plot(), 44, 63
qplot(), 68
rbind(), 32
read.csv(), 80, 81
read.delim(), 80
read.table(), 80
read.xport(), 85
read_xpt(), 85, 171
recode(), 146, 163
rep(), 37
rnorm(), 40
row.names(), 36
rownames(), 35
sample(), 42
sd(), 150
select(), 129, 138
seq(), 37, 99
sequence(), 37
sessionInfo(), 205
set.seed(), 41
setwd(), 29, 66
stack(), 98
str(), 53
summarise(), 150
summarize(), 129
summary(), 53
summrise(), 129
svyglm(), 180
tail(), 51
ungroup(), 153
with(), 56
write.csv(), 122
write.table(), 80
write_csv(), 122
geom
 ., 129
 boxplot, 73
 ggplot2, 158
 point, 76
 smooth, 76
ggplot2
 aes, 165
 aesthetics, 158
 facet_grid(), 161
 facets, 158
 geom, 158
 geom_bar(), 160
 geom_bar(aes), 161
 geom_col(), 160
 geom_errorbar(), 168
 labs(), 168
 ungroup(), 152
grammar of graphics
 ggplot2, 157
graphics
 bar plot, 160
 barplot, 153
 base R, 57
 boxplot, 57
 boxplot(), 45
 error bars, 168

- ggplot(), 68
- hist(), 57
- par(), 44
- plot(), 44, 63
- qplot(), 68
- scatter plot, 74
- setwd(), 67
- Grolemund, Garrett, 133
- Gruber, John, 183
- Hadley Wickham
 - tidyverse, 123
- header
 - YAML, 189
 - YAML example, 201
- histogram, 57
 - breaks, 95
 - densities, 95
 - frequencies, 95
- HTML, 184
- IF, 43
- ifelse()
 - nested, 144
- illustrations
 - free images, 207
- impute, 66
- inline code
 - R markdown, 192
- jam, 21
- jar, 20
- knit
 - R markdown, 192
- knitr
 - opts_chunk, 202
- label, 71
- LaTeX, 193
- legend
 - qplot(), 77
- LETTERS, 42, 191
- level, 38
- lierate programming, 181
- linear model, 66
- linear regression, 66
 - coefficients, 118
 - qplot(), 76
- loess, 77
 - loess, 77
- magic, 186, 192
- magical, 95
- Magritte, Rene, 127
- markdown, 183
 - add image, 208
 - basic syntax, 184
 - extended syntax, 185
 - image size, 208
 - interactive tutorial, 185
 - markup, 183
 - R markdown, 185
 - readability, 184
 - source, 184
 - table convert, 209
 - table generator, 209
 - tables, 209
 - web link, 203

- math equation, 205
- matrix, 31
- MSWord, 184
- New York, 50
- NHANES
 - ALB_CR, 109
 - BMX, 115
 - combining data, 100
 - DEMO, 84
 - demographics, 84, 145
 - HDL, 101
 - import data, 81, 85
 - merging data, 100
 - PFAS, 83
 - SEQN, 87
 - subsetting, 102
 - TCHOL, 101
 - TRIGLY, 101
 - urine sample, 107
- objects
 - LETTERS, 42
 - R, 25
 - user defined, 9
 - vector, 28
- odd number columns, 99
- omit columns, 90
- Ozone, 56, 63
- parameters
 - all.x, 121
 - breaks, 95
 - by.x, 110
 - by.y, 110
 - graphics, 44
 - las, 92
 - legend.position, 77
 - levels, 71
 - lwd, 67
 - mfrow, 44
 - month.abb, 71
 - names.arg, 154
 - ordered, 71
 - par(), 44
 - pch, 64
 - ylim, 89
- pch, 64
- Pearson
 - correlation factor, 118
- PFAS
 - hydrophilic, 115
 - hydrophobic, 115
 - lipophobic, 115
 - ng/ml, 111
 - oleophobic, 115
 - partitioning, 115
 - Perfluoroalkyl, 83
 - Polyfluoroalkyl, 83
- pipe, 125
 - data stream, 125
 - symbol, 125
 - Unix, 125
- pipeline, 126
- plot
 - characters, 64
 - geometric shapes, 64

- symbols, 64
- programming
 - literate, 181
- qplot
 - theme(), 72
- qplot()
 - geom, 69
 - legend.position, 77
 - linear regression, 76
 - lm, 77
 - loess, 77
 - method, 77
 - scatter plot, 74
- R
 - .Machine\$integer.max, 174
 - arguments, 26
 - assignment operator, 21
 - base, 51, 80
 - base R, 57
 - Classic, 19
 - classic, 79
 - comments, 15
 - datasets(), 5
 - functions, 26
 - graphics, 57
 - L:integer coercion, 173
 - levels, 38
 - objects, 9, 20, 25
 - script, 14
 - workspace, 9
- R markdown, 185
 - code chunk options, 202
 - code chunks, 189
 - convert, 192
 - echo=, 203
 - eval=, 203
 - example, 191, 231
 - graphic size, 208
 - inline code, 192, 204
 - knit, 192
 - magic, 186
 - math formula, 205
 - output formats, 193
 - report template, 199
 - tables, 209
 - warning=, 203
- R package
 - dplyr, 129
 - foreign, 81
 - ggplot2, 68, 157
 - ggpubr, 73
 - haven, 85, 170
 - knitr, 187
 - magrittr, 127
 - survey, 170
 - tibble, 127
 - tidyr, 133
 - tidyverse, 68, 123
 - tinytex, 193
- RAM, 9
- ratio
 - computation, 142
 - creatinine, 111
 - mutate(), 142
- report template

- R markdown, 199
- reproducible research, 181
- research
 - replicable, 182
 - reproducible, 181
- RStudio
 - 2017 conference, 123
 - console, 12
 - environment, 12
 - execute command, 16
 - files, 13
 - history, 12
 - organize data, 13
 - panes, 12
 - project, 13
 - source, 12
 - working directory, 16
 - workspace, 12
- scatter plot, 63, 74
- SEQN
 - unique individual, 100
- Software instal.
 - Packages, 2
 - R, 2
 - RStudio, 2
 - TinyTex, 193
- Star Trek, 128
- statistics
 - generalised linear model, 180
 - mean, 55
 - quartile, 54
 - standard deviation, 150
 - weights, 169
- story
 - vector V, 192
- strawberry, 21
- subsetting, 54, 88
 - [,], 54
 - \$, 56
- Swartz, Aaron, 183
- Symbols
 - +, 72
 - „, 54
 - ., 80, 129
 - .., 16
 - /, 29
 - :, 37
 - ::, 85
 - <-, 21
 - ?, 29
 - [], 54
 - #, 15
 - \$, 56, 71
 - ~, 59
- tab delimited, 80
- tables
 - markdown, 209
- tabular data, 79
- template
 - R markdown, 199
- tidyverse
 - command, 126
 - dplyr, 129
 - philosophy, 124

- query, 126
- tibble, 127
- tidyverse.quiet, 129
- Wickham, Hadley, 123
- transfer mode
 - binary, 84
- Tribble, 128
- triglycerides, 101
- USA, 80
- variable
 - as.factor(), 160
 - categorical, 60
 - categories, 160
 - factor, 60
 - level, 60
 - levels, 160
- vector, 28, 31
- vector V
 - story, 192, 231
- vectorisation, 30
- web link
 - markdown, 203
- Web links
 - CRAN, 20
 - NCHS, 4
 - NHANES, 4
 - NIEHS, 20
 - RStudio, 20
- weights
 - statistics, 169
- Wickham, Hadley, 123
- working directory
 - getwd(), 29
 - RStudio, 16
 - setwd(), 29
- workspace
 - R, 9
 - RStudio, 12
- YAML
 - delimitation, 194
 - header, 189
 - header example, 201
 - indentation, 194
 - language, 194
 - resources, 197
 - validator, 197