

R Bootcamp - How to use R for Data Science

Yigit Aydede

2022-10-13

Contents

About	7
Why R?	8
License	9
1 Introduction	11
1.1 Installation of R, RStudio and R Packages	11
1.2 RStudio	13
1.3 Packages	17
1.4 Working directory	18
1.5 Hints	19
1.6 Console or Script?	20
1.7 R as a calculator	20
1.8 Data & Object Types	20
1.9 R-Style Guide	21
2 Vectors	23
2.1 One type, same type	23
2.2 Patterns	25
2.3 Attributes	26
2.4 Character operators	27
2.5 Sort, rank, and order	28
2.6 Simple descriptive measures	28
2.7 Subsetting Vectors	30
2.8 Vectorization or vector operations	33
2.9 Set operations	34
2.10 Missing values	34
2.11 Factors	35
3 Other R Objects	37
3.1 Matrices	37
3.2 Data Frames	44
3.3 Lists	52
3.4 Array	56

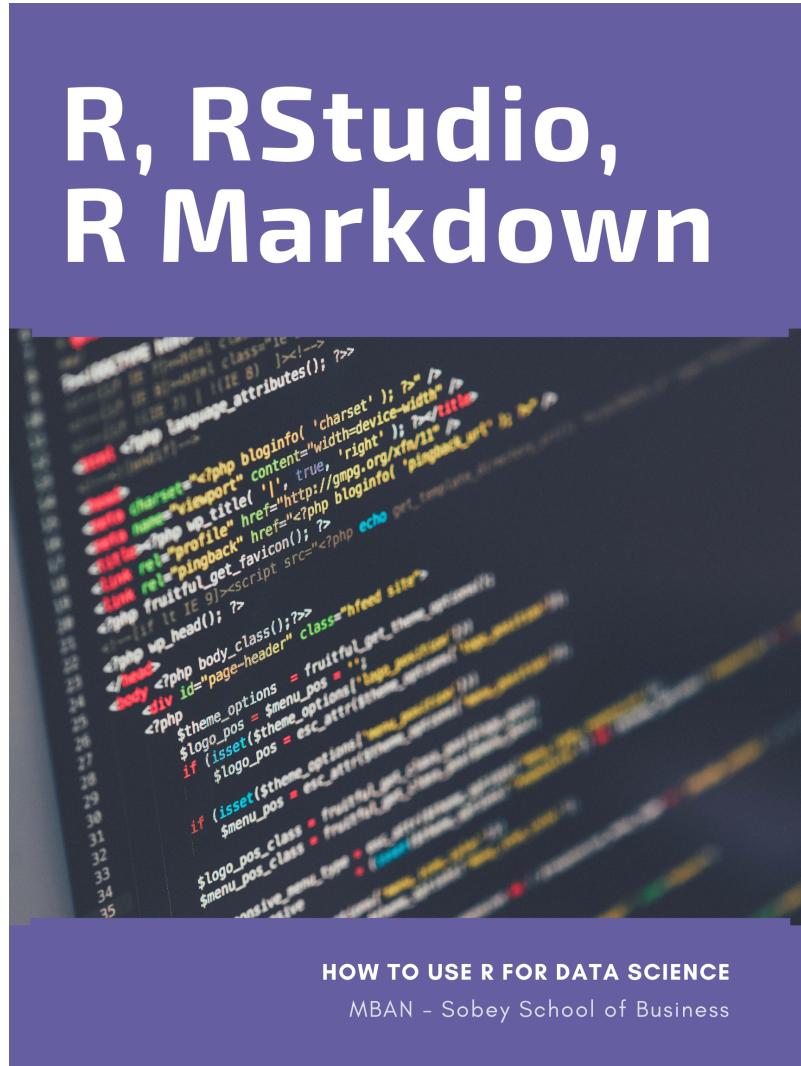
4 Reading and writting data files	61
4.1 Reading (importing)	61
4.2 Writing (exporting)	64
5 Data visualisation with R	65
5.1 Scatterplots	71
5.2 Smoothlines	78
5.3 Histograms & Density	82
5.4 Multiple plots	83
5.5 Add lines	85
5.6 Pairwise relationship	88
6 Data Management	93
6.1 Filter	93
6.2 Arrange	94
6.3 Pipe	96
6.4 Select	96
6.5 Create & <code>group_by()</code>	97
6.6 More tools	98
7 Programming basics	101
7.1 Conditional flows	101
7.2 Loops	104
7.3 The <code>apply()</code> family	109
7.4 Functions	112
7.5 <code>source()</code>	113
8 Simulation in R	115
8.1 Sampling in R: <code>sample()</code>	115
8.2 Random number generating with probability distributions	118
8.3 Simulation for statistical inference	120
8.4 Creating data with a Data Generating Model (DGM)	125
8.5 Bootstrapping	130
8.6 Monty Hall - Fun example	133
9 Parts	139
10 Footnotes and citations	141
10.1 Footnotes	141
10.2 Citations	141
11 Blocks	143
11.1 Equations	143
11.2 Theorems and proofs	143
11.3 Callout blocks	143
12 Sharing your book	145

CONTENTS	5
12.1 Publishing	145
12.2 404 pages	145
12.3 Metadata for sharing	145
13 Cross-references	147
13.1 Chapters and sub-chapters	147
13.2 Captioned figures and tables	147

About

This book covers basics to learn R for Data Science. It is designed for MBAN students.

We also have a companion R package named `RBootcamp`, containing the data sets used as well as interactive exercises for each part.



Why R?

R is both a programming language and software environment for statistical computing, which is free and open-source.

With ever increasing availability of large amounts of data, it is critical to have the ability to analyze the data and learn from it for making informed decisions. Familiarity with software such as R allows users to visualize data, run statistical tests, and apply machine learning algorithms. Even if you already know other software, there are still good reasons to learn R:

1. **R is free.** If your future employer does not already have R installed, you can always download it for free, unlike other proprietary software packages that require expensive licenses. You can always have access to R on your computer.
2. **R gives you access to cutting-edge technology.** Top researchers develop statistical learning methods in R, and new algorithms are constantly added to the list of packages you can download.
3. **R is a useful skill.** Employers that value analytics recognize R as useful and important. If for no other reason, learning R is worthwhile to help **boost your resume**.

Here is a very good article about R and Programming that everybody should read: [7 Reasons for policy professionals to get into R programming in 2019 \[?\]](#).

License



Figure 1: This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Chapter 1

Introduction

The following sections will serve as an introduction to the R basics that could be used in data analytics. At the beginning, these introductory R subjects may feel like an overwhelming amount of information. The leaning curve will be steeper as practice more. You should try all of the codes from these examples and solve the practice exercises.

R is used both for software development and data analysis. We will not use it for software development but apply some concepts in that area. Our main goal will be to analyze data, but we will also perform programming exercises that help illustrate certain algorithmic concepts.

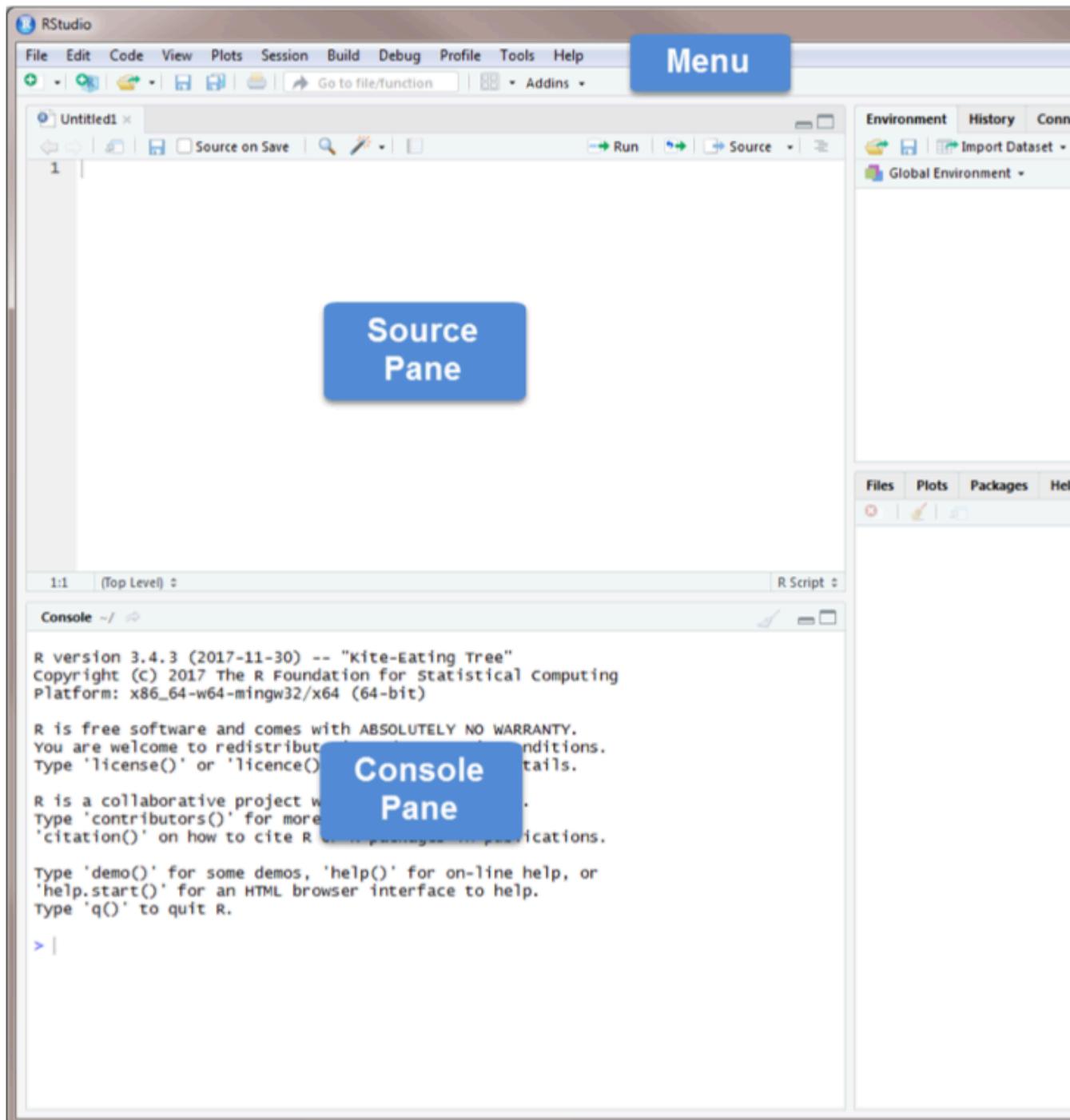
1.1 Installation of R, RStudio and R Packages

To get started, you will need to install two pieces of software:

R, the actual programming language: [Download it from here](#). – Chose your operating system, and select the most recent version.

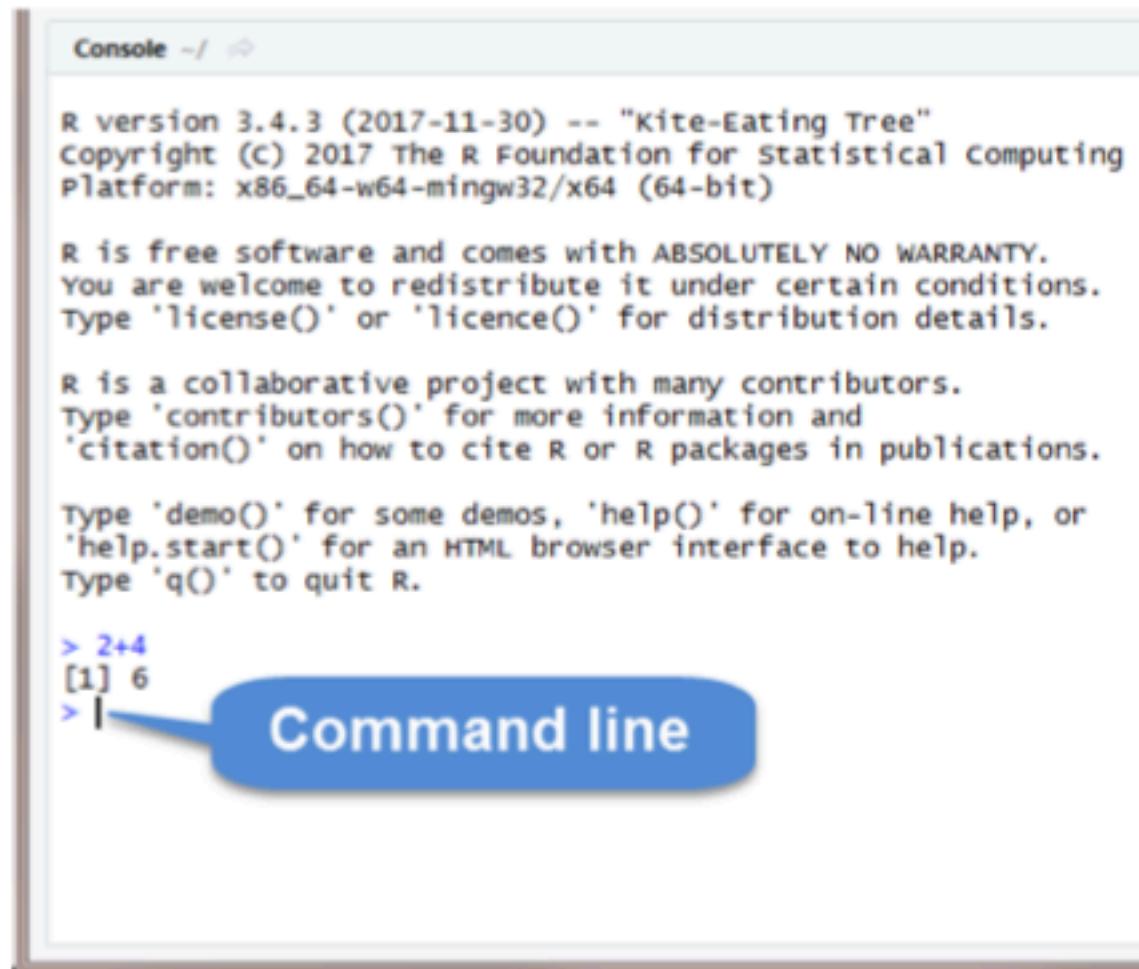
RStudio, an excellent integrated development environment (IDE) for working with R, an interface used to interact with R: [Download it from here](#). Throughout this book, you will use RStudio instead of R to learn R programming.

1.2 RStudio



Source Pane, click on the plus sign in the top left corner. From the drop-down menu, select **R Script**. As shown in that dropdown menu, you can also open an R Script by pressing **Ctrl+Shift+N**. You should now see the screen above.

The **Console Pane** is the interface to R. If you opened R directly instead of opening RStudio, you would see just this console. You can type commands directly in the console. The console displays the results of any command you run. For example, type `2+4` in the command line and press enter. You should see the command you typed, the result of the command, and a new command line.



The screenshot shows the RStudio interface with the 'Console' tab selected. The console window displays the standard R startup message, including the version (R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"), copyright information, and a note about the lack of warranty. Below this, it provides information about contributors and citation details. It also suggests using 'demo()', 'help()', or 'help.start()' for help and 'q()' to quit R. At the bottom of the console, there is a command line input field with the text `> 2+4` and the result `[1] 6` displayed below it. A blue callout bubble with the text "Command line" points to the command line input field.

To clear the console, you press **Ctrl+L** or type `cat("\014")` in the command line.

R code can be entered into the command line directly (in Console Pane) or

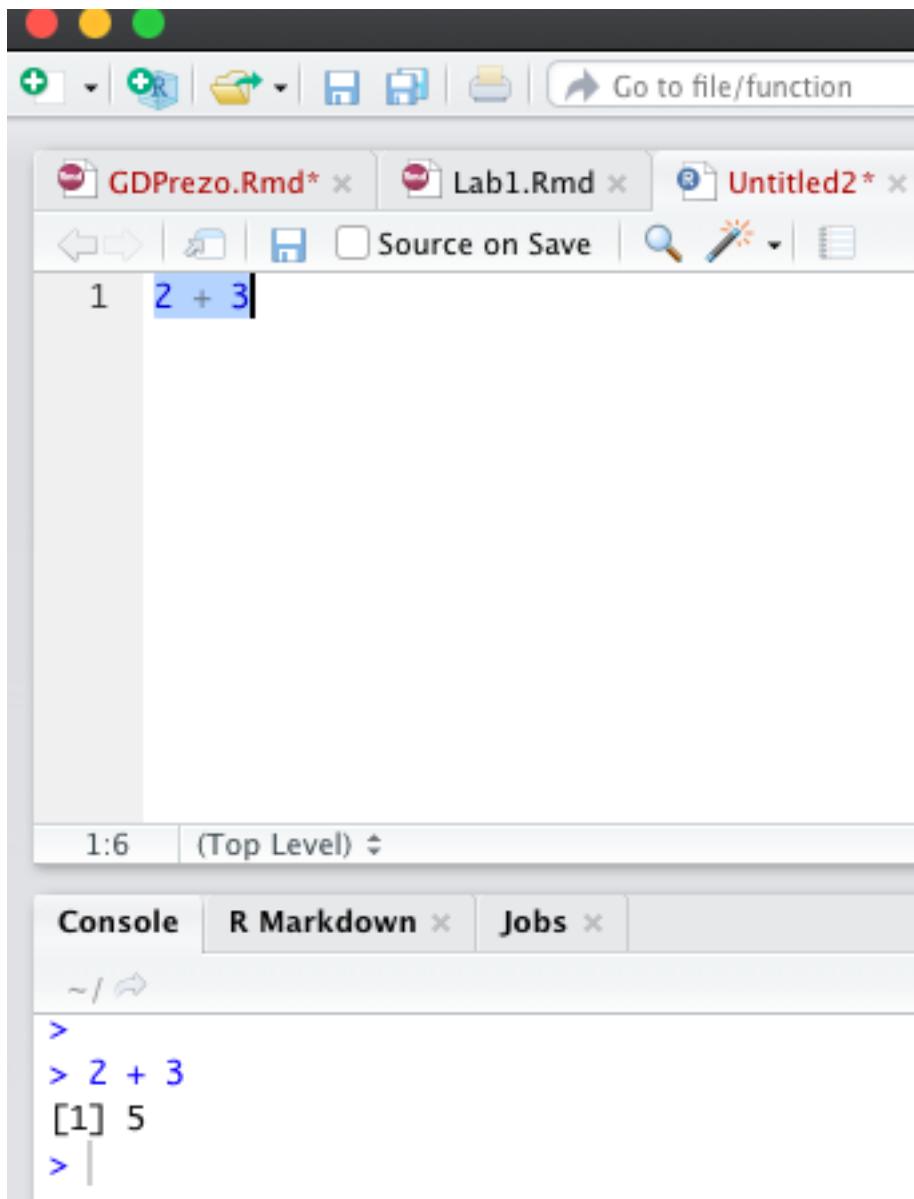
saved to a script (Source Pane).

Let's try some coding.

2 + 3 #write this on the command line and hit Enter

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

Now write the same line into the script in Source Pane and run it



The **Source Pane** is a text editor where you can type your code before running it. You can save your code in a text file called a script. Scripts have typically file names with the extension **.R**. Any text shown in green is a comment in the script. You write a comment by adding a **#** to an RScript. Anything to the right of a **#** is considered a comment and is thus ignored by R when running code. Place your cursor anywhere on the first few lines of code and click **Run**. You can also run code by pressing **Ctrl+Enter**.

The screenshot shows the RStudio interface with the Environment tab selected. The Global Environment pane displays three variables: **x**, **y**, and **z**. The values for **x** are **num [1:4] 2 5 3 1**, for **y** are **num [1:4] 10 2 4 7**, and for **z** are **num [1:4] 1 4 2 NA**. A blue speech bubble points to the Connections tab with the text "Only in RStudio 1.1 or later".

The **Environment Pane** includes an Environment, a History tab, and a Connections tab. The Connections tab makes it easy to connect to any data source on your system.

The Environment tab displays any objects that you have created during your R session. For example, if we create three variables: *x*, *y*, and *z*, R stored those variables as objects so that you can see them in the Environment pane.

To do object assignments, you need to assign value(s) to a name via the assignment operator, which will create a new object with a name.

```
x <- 5
y <- x*1.5
z <- x - y*3
ls()
```

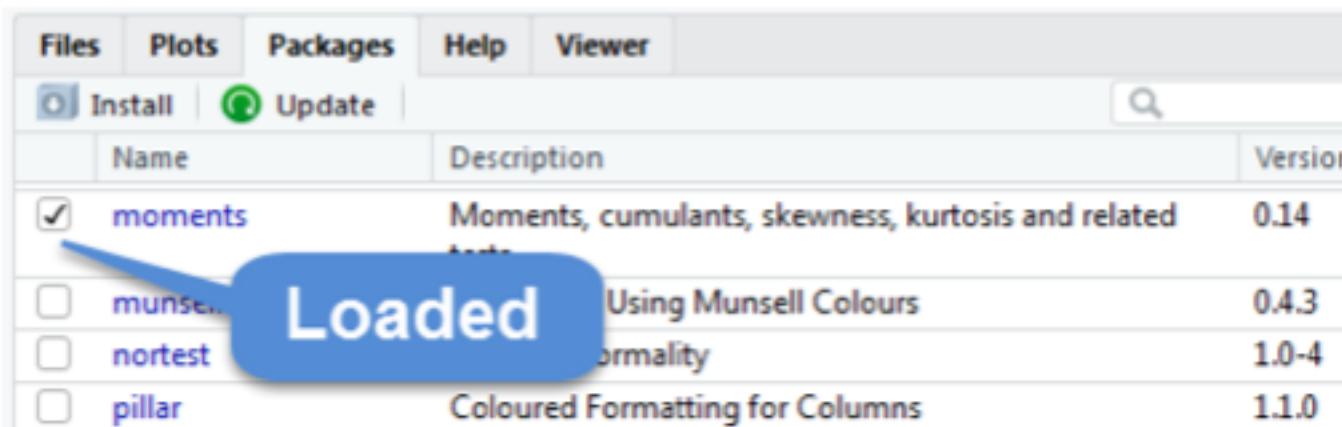
```
## [1] "x" "y" "z"
```

We will discuss R objects in more detail later. If you want to see a list of all objects in the current session, type `ls()` in the command line. You can remove an individual object from the environment with the `rm()` command. For example, remove *x* by typing `rm(x)` in the command line. You can remove all objects from the environment by clicking or typing `rm(list=ls())` in the command line. The History tab keeps a record of all the commands you have run. To copy a command from the history into the console, select the command and press Enter.

The **Files Pane** includes several tabs that provide useful information. The Files tab displays the contents of your working directory. The Plot tab shows all graphs that you have created. The Packages tab displays the R packages that you have installed in your System Library.

1.3 Packages

An R package typically includes code, data, documentation for the package and functions inside, and tests to check everything works as it should. Check to see if the package `moments` has been installed. If you cannot find it, you need to install it by using the command `install.packages("moments")`. Once you have installed the package, you need to load it using the command `library(moments)`. Or you can use install tab and follow the instructions and the go to package to check it to activate as shown below.



Files	Plots	Packages	Help	Viewer
		Install	Update	
		Name	Description	Version
<input checked="" type="checkbox"/>	<code>moments</code>	Moments, cumulants, skewness, kurtosis and related tests		0.14
<input type="checkbox"/>	<code>munse</code>	Using Munsell Colours		0.4.3
<input type="checkbox"/>	<code>nortest</code>	Normality		1.0-4
<input type="checkbox"/>	<code>pillar</code>	Coloured Formatting for Columns		1.1.0

The help tab has built-in documentation for packages and functions in R. The help is automatically available for any loaded packages. You can access that file by typing `help(mean)` or `?mean` in the command line. You can also use the search bar in the help tab.

The packages can be installed from sources other than CRAN. For example, in this book we will use `RBootcamp` which is not located on CRAN

```
#install.packages("remotes")
#remotes::install_github("yayedede/RBootcamp")
```

One of the most difficult things to do when learning R is to know how to find help. Your very first helper should be **Google** where you post your error message or a short description of your issue. The ability to solve problems using this method is quickly becoming an extremely valuable skill.

Do not be discouraged by running into errors and difficulties when learning R. It is simply part of the learning process.

The Viewer tab displays HTML output. R packages such as R Markdown and Shiny create HTML outputs that you can view in the Viewer tab. We'll see it later.

1.4 Working directory

Without further specification, files will be loaded from and saved to the working directory. The functions `getwd()` and `setwd()` will get and set the working directory, respectively.

```
getwd()

## [1] "/Users/yigitaydede/Dropbox/Documents/Courses/MBAN/RBootcamps/Bootcamp_book"
#setwd("Book2022")

#List all the objects in your local workspace using
ls()

## [1] "x" "y" "z"
#List all the files in your working directory using list.files() or
dir()

##  [1] "_book"                  "_bookdown_files"
##  [3] "_bookdown.yml"          "_main_files"
##  [5] "_output.yml"            "01-intro.Rmd"
##  [7] "02-cross-refs_files"    "02-Others_files"
##  [9] "02-Others.Rmd"          "03-ReadWrite.Rmd"
## [11] "04-visual_files"         "04-visual.Rmd"
## [13] "05-DataMan_files"        "05-DataMan.Rmd"
## [15] "06-Programmaringbasics.Rmd" "07-Sim_cache"
## [17] "07-Sim_files"           "07-Sim.Rmd"
## [19] "08-cross-refs_files"    "09-parts.Rmd"
## [21] "10-citations.Rmd"       "11-blocks.Rmd"
## [23] "12-share.Rmd"           "13-references.Rmd"
## [25] "14-cross-refs_files"    "14-cross-refs.Rmd"
## [27] "Book_for_RBootcamp.log" "Book_for_RBootcamp.rds"
## [29] "book.bib"                "Bootcamp_book.Rproj"
## [31] "docs"                    "index.md"
## [33] "index.Rmd"               "packages.bib"
## [35] "png"                     "preamble.tex"
## [37] "README.md"                "render83655cf6217a.rds"
## [39] "some_functions.R"        "style.css"
## [41] "table1.text"

#As we go through this lesson, you should be examining the help page
#for each new function. Check out the help page for list.files with the
```

```

#command
?list.files
#or
help("list.files")

#Using the args() function on a function name is also a handy way to
#see what arguments a function can take.
args(list.files)

## function (path = ".", pattern = NULL, all.files = FALSE, full.names = FALSE,
##   recursive = FALSE, ignore.case = FALSE, include.dirs = FALSE,
##   no.. = FALSE)
## NULL

```

For this bootcamp, I would suggest to create a RStudio Project in your local driver. RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.

RStudio projects are associated with R working directories. You can create an RStudio project:

- In a brand new directory
- In an existing directory where you already have R code and data
- By cloning a version control (Git or Subversion) repository

To create a new project in the RStudio, use the Create Project command (available on the Projects menu and on the global toolbar)

1.5 Hints

- R distinguishes upper case from lower case letters. Thus a variable named `X` differs from another variable named `x`.
- The way to learn programming is through practice. The learning curve to R is not bad. You may struggle a bit in the process, but the skills learned will be invaluable for you in the future.
- There are many ways to write a code to solve the same thing. You can develop your own style. But, if you see nother and better code, try to learn from others!
- There are many option that you can customize R Studio. Check out Tools > Global Options > General tab in the menu bar of RStudio.
- Before you start coding, draft a plan to address the question at hand.
- There is convention in writing codes. Try to adhere these accepted styles.
- Comment your code properly (using a `#` sign at the beginning of each line). Good documentation is a great reminder what you have done. Believe me you will forget later the lines in the your own script.

- The character > in the R Console indicates that R is ready for you to enter a command.
- Do not overwrite the original data set and variables. Create new data sets just to be sure, especially when taking a subset from that data set.

1.6 Console or Script?

The Script Window is the place to enter and run your code so that it is easily edited and saved for future use. You create new R Script by clicking on File > New File > R Script in the RStudio menu bar.

To execute your code in the R script, you can either highlight the code and click on Run, or you can highlight the code and press CTRL + Enter on your keyboard.

If you prefer, you can enter code directly in the Console Window and click Enter. The commands that you run will be shown in the History Window on the top right of RStudio. You can save these commands for future use.

Or, to find the older commands in the console, use the upper arrow to get them again ...

Make sure you save your code. With your code, you can always regenerate your workspace but it could take a little time. Or you can save your workspace which allows you to start where you left off, with all of the variables you created and renamed saved.

Your code (in your script window) is saved by clicking the SAVE button in the RStudio menu bar. The code will be saved in the working directory. See 2.1 for setting and getting the working directory.

1.7 R as a calculator

At a very basic level, we can use R as a calculator.

See [Lesson1](#) in our package [Rbootcamp](#):

```
library(RBootcamp)
#learnr::run_tutorial("Lesson1", "RBootcamp")
```

1.8 Data & Object Types

R has a number of basic data types.

Numeric: Also known as Double. The default type when dealing with numbers.
1,1.0,42.5

Integer: 1L,2L,42L

Complex: $4 + 2i$

Logical: Two possible values: TRUE and FALSE. NA is also considered logical.

Character: "a", "Statistics", "1plus2."

R also has a number of basic data structures. A data structure is either **homogeneous** (all elements are of the same data type) or **heterogeneous** (elements can be of more than one data type): You can think each data structure as **data container** (object types) where your data is stored. Here are the main "container" or data structures. Think it as Stata or Excel spread-sheets.

Vector: 1 dimension (column OR row) and homogeneous. That is every element of the vector has to be the same type. Each vector can be thought of as a variable.

Matrix: 2 dimensions (column AND row) and homogeneous. That is every element of the matrix has to be the same type.

Data Frame: 2 dimensions (column AND row) and heterogeneous. That is every element of the data frame doesn't have to be the same type. This is the main difference between a matrix and a data frame. Data frames are the most common data structure in any data analysis.

List: 1 dimension and heterogeneous. Data can be multiple data structures.

Array: 3+ dimensions and homogeneous.

1.9 R-Style Guide

The idea is simple: your R code, or any other code in different languages, should be written in a readable and maintainable style. Here is a [blog](#) by Roman Pahl that may help you develop a better styling in your codes. (You may find in some chapters and labs that my codes are not following the "good" styling practices. I am trying to improve!)

Chapter 2

Vectors

Many operations in R make heavy use of vectors. Possibly the most common way to create a vector in R is using the `c()` function, which is short for “combine.” As the name suggests, it combines a list of elements separated by commas.

```
c(1, 5, 0, -1)  
## [1] 1 5 0 -1
```

If we would like to store this vector in a **variable** we can do so with the assignment operator `<-` or `=`. But the convention is `<-`

```
x <- c(1, 5, 0, -1)  
z = c(1, 5, 0, -1)  
x  
  
## [1] 1 5 0 -1  
z  
  
## [1] 1 5 0 -1
```

Note that scalars do not exists in R. They are simply vectors of length 1.

```
y <- 24 #this a vector with 1 element, 24
```

2.1 One type, same type

Because vectors must contain elements that are all the same type, R will automatically coerce to a single type when attempting to create a vector that combines multiple types.

```
c(10, "Machine Learning", FALSE)  
  
## [1] "10"           "Machine Learning" "FALSE"
```

```

c(10, FALSE)

## [1] 10 0
c(10, TRUE)

## [1] 10 1
x <- c(10, "Machine Learning", FALSE)
str(x) #this tells us the structure of the object

##  chr [1:3] "10" "Machine Learning" "FALSE"
class(x)

## [1] "character"
y <- c(10, FALSE)
str(y)

##  num [1:2] 10 0
class(y)

## [1] "numeric"

```

We know that vectors are objects that have values of the same type. If you combine them into a vector, R will unify all values into the most complex one, which is usually called the coercion rule.

```

m <- c(TRUE, 5, -2, FALSE)
m

## [1] 1 5 -2 0
class(m)

## [1] "numeric"

And,
```

```

m_2 <- c(8, "Joe", 21, "Mustang")
m_2

## [1] "8"      "Joe"     "21"     "Mustang"
class(m_2)

## [1] "character"

```

You can also manually convert the vectors

```

n <- c(8, 3, 21, 2)
n

```

```
## [1] 8 3 21 2
nc <- as.character(n)
nc

## [1] "8"  "3"  "21" "2"
n <- as.numeric(nc)
n

## [1] 8 3 21 2

And be careful:
```

```
m_2 <- c(8, "Joe", 21, "Mustang")
as.numeric(m_2)

## Warning: NAs introduced by coercion
## [1] 8 NA 21 NA
```

2.2 Patterns

If you want to create a vector based on a sequence of numbers, you can do it easily with an operator, which creates a sequence of integers between two specified integers.

```
y <- c(1:15)
y

## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

#or
y <- 1:15
y

## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

If you want to create a vector based on a specific sequence of numbers increasing or decreasing, you can use seq()
y <- seq(from = 1.5, to = 13, by = 0.9) #increasing
y

## [1] 1.5 2.4 3.3 4.2 5.1 6.0 6.9 7.8 8.7 9.6 10.5 11.4 12.3
y <- seq(1.5, -13, -0.9) #decreasing. Note that you can ignore the argument labels
y

## [1] 1.5 0.6 -0.3 -1.2 -2.1 -3.0 -3.9 -4.8 -5.7 -6.6 -7.5 -8.4
## [13] -9.3 -10.2 -11.1 -12.0 -12.9
```

The other useful tool is `rep()`

```
rep("ML", times = 10)

## [1] "ML" "ML"
#or

x <- c(1, 5, 0, -1)
rep(x, times = 2)

## [1] 1 5 0 -1 1 5 0 -1
```

And we can use them as follows.

```
wow <- c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
wow
```

```
## [1] 1 5 0 -1 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 2 3 42 2 3
## [26] 4
```

Another one, which can be used to create equal intervals.

```
g <- seq(6, 60, length = 4)
g
```

```
## [1] 6 24 42 60
```

See this

```
unique(wow)
```

```
## [1] 1 5 0 -1 3 7 9 2 42 4
```

2.3 Attributes

We can calculate the number of elements in a vector:

```
length(wow)
```

```
## [1] 26
```

There is set of functions starting with `is.***()`. For example: `is.numeric()`, which checks whether a vector is of numeric type,

```
is.numeric(g)
```

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

```
is.character(g)
```

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

In addition to storing the values of a vector, you can also create named vectors.

```
x <- c(165, 60, 22)
x

## [1] 165 60 22
x_n <- c(height = 125, weight = 56, BMI = 21)
x_n

## height weight     BMI
##    125      56      21
```

And,

```
attributes(x_n)

## $names
## [1] "height" "weight" "BMI"
```

2.4 Character operators

```
animals <- c("dog", "cat", "donkey")
nchar(animals)
```

```
## [1] 3 3 6
```

We can concatenate several strings into a single string.

```
wrong <- c("we have", "dogs", "cats", "and, donkey")
wrong

## [1] "we have"      "dogs"        "cats"        "and, donkey"
right <- paste("we have ", "dogs, ", "cats, ", "and, donkey")
right
```

```
## [1] "we have dogs, cats, and, donkey"
```

You can check `paste0()`

```
hah <- toupper(right)
hah

## [1] "WE HAVE DOGS, CATS, AND, DONKEY"
haha <- tolower(hah)
haha

## [1] "we have dogs, cats, and, donkey"
```

2.5 Sort, rank, and order

```
x <- c(2, 3, 2, 0, 4, 7)
x
```

```
## [1] 2 3 2 0 4 7
```

By default, the `sort()` function sorts elements in vector in the increasing order.

```
sort(x)
```

```
## [1] 0 2 2 3 4 7
```

```
sort(x, decreasing = TRUE)
```

```
## [1] 7 4 3 2 2 0
```

The `rank()` function gives the corresponding positions in the ascending order.

```
rank(x)
```

```
## [1] 2.5 4.0 2.5 1.0 5.0 6.0
```

You can see that the smallest value of `x` is 0, which corresponds to the fourth element. Thus, the fourth element has rank 1.

As for the `order()` function, it is confusing and a very different function from `sort()`.

```
order(x)
```

```
## [1] 4 1 3 2 5 6
```

We can see that the `order()` function returns indices for the elements in the ascending order.

2.6 Simple descriptive measures

Let's have a numeric vector:

```
h <- c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
h
```

```
## [1] 2 3 2 0 4 7 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 2 3 42
## [26] 2 3 4
```

```
summary(h)
```

```
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## 0.000   2.000   3.000   5.357   7.000  42.000
```

The set of statistical measures:

```

min(h)

## [1] 0

max(h)

## [1] 42

mean(h)

## [1] 5.357143

median(h)

## [1] 3

sd(h)

## [1] 7.650777

range(h)

## [1] 0 42

sum(h)

## [1] 150

cumsum(h)

## [1] 2 5 7 7 11 18 19 22 27 34 43 44 47 52 59 68 69 72 77
## [20] 84 93 94 96 99 141 143 146 150

prod(h)

## [1] 0

quantile(h)

## 0% 25% 50% 75% 100%
## 0 2 3 7 42

IQR(h) #interquartile range

## [1] 5

We can also find the index number of maximum and minimum numbers

which.max(h)

## [1] 25

which.min(h)

## [1] 4

```

2.7 Subsetting Vectors

One of the most confusing subjects in R is subsetting the data containers. It's an important part in data management and if it is done in 2 steps, the whole operation becomes quite easy:

1. Identifying the index of the element that satisfies the required condition,
2. Calling the index to subset the vector.

But before we start, lets see a simple subsetting. (Note the square brackets)

```
#Suppose we have the following vector
myvector <- c(1, 2, 3, 4, 5, 8, 4, 10, 12)

#I can call each element with its index number:
myvector[c(1,6)]

## [1] 1 8
myvector[4:7]

## [1] 4 5 8 4
myvector[-6]

## [1] 1 2 3 4 5 4 10 12
```

Okay, let's see commonly used operators for doing comparisons:

```
x <- 3

x < 2      #less

## [1] FALSE
x <= 2      #less or equal to

## [1] FALSE
x > 1      #bigger

## [1] TRUE
x >= 1      #bigger or equal to

## [1] TRUE
x == 3      #equal to

## [1] TRUE
#x = 3      #Note that this an assignment operator
x != 3      #not equal to

## [1] FALSE
```

```

#Let's look at this vector
myvector <- c(1, 2, 3, 4, 5, 8, 4, 10, 12)

#We want to subset only those less than 5

#Step 1: use a logical operator to identify the elements
#meeting the condition.
logi <- myvector < 5
logi

## [1] TRUE TRUE TRUE TRUE FALSE FALSE TRUE FALSE FALSE
#logi is a logical vector
class(logi)

## [1] "logical"
#Step 2: use it for subsetting
newvector <- myvector[logi==TRUE]
newvector

## [1] 1 2 3 4 4

```

or better:

```

newvector <- myvector[logi]
newvector

## [1] 1 2 3 4 4

```

This is good as it shows those 2 steps. Perhaps, we can combine these 2 steps as follows:

```

newvector <- myvector[myvector < 5]
newvector

```

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

Another way to do this is to use of `which()`, which gives us the index of each element that satisfies the condition.

```

ind <- which(myvector < 5) # Step 1
ind

```

```
## [1] 1 2 3 4 7
newvector <- myvector[ind] # Step 2
newvector

```

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

Or we can combine these 2 steps:

```
newvector <- myvector[which(myvector < 5)]
newvector
```

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

Last one: find the 4's in `myvector` make them 8 (I know hard, but after a couple of tries it will seem easier):

```
myvector <- c(1, 2, 3, 4, 5, 8, 4, 10, 12)
#I'll show you 3 ways to do that.
```

```
#1st way to show the steps
ind <- which(myvector==4) #identifying the index with 4
newvector <- myvector[ind] + 4 # adding them 4
myvector[ind] <- newvector #replacing those with the new values
myvector
```

```
## [1] 1 2 3 8 5 8 8 10 12
```

```
#2nd and easier way
myvector[which(myvector==4)] <- myvector[which(myvector==4)] + 4
myvector
```

```
## [1] 1 2 3 8 5 8 8 10 12
```

```
#3rd and easiest way
myvector[myvector==4] <- myvector[myvector==4] + 4
myvector
```

```
## [1] 1 2 3 8 5 8 8 10 12
```

What happens if the vector is a character vector? How can we subset it? We can use `grep()` as shown below:

```
m <- c("about", "aboard", "board", "bus", "cat", "abandon")
#Now suppose that we need to pick the elements that contain "ab"
#Same steps again
a <- grep("ab", m) #similar to which() that gives us index numbers
a
```

```
## [1] 1 2 6
```

```
newvector <- m[a]
newvector
```

```
## [1] "about"    "aboard"   "abandon"
```

2.8 Vectorization or vector operations

One of the biggest strengths of R is its use of vectorized operations. Lets see it in action!

```
x <- 1:10
x

## [1] 1 2 3 4 5 6 7 8 9 10
x+1

## [1] 2 3 4 5 6 7 8 9 10 11
2 * x

## [1] 2 4 6 8 10 12 14 16 18 20
2 ^ x

## [1] 2 4 8 16 32 64 128 256 512 1024
x ^ 2

## [1] 1 4 9 16 25 36 49 64 81 100
sqrt(x)

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
## [9] 3.000000 3.162278
log(x)

## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
## [8] 2.0794415 2.1972246 2.3025851
```

Its like a calculator!

```
y <- 1:10
y

## [1] 1 2 3 4 5 6 7 8 9 10
x + y

## [1] 2 4 6 8 10 12 14 16 18 20
```

How about this:

```
y <- 1:11
x + y

## Warning in x + y: longer object length is not a multiple of shorter object
## length
## [1] 2 4 6 8 10 12 14 16 18 20 12
```

OK, the warning is self-explanatory. But what's "12" at the end? It's the sum of the first element of `x`, which is 1 and the last element of `y`, which is 11.

2.9 Set operations

```
x <- c(1, 2, 1, 3, 1)
y <- c(1, 1, 3, 6, 6, 5)
```

You can use the `intersect()` function to get values in both `x` and `y`:

```
intersect(x, y)
```

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

To get values in either `x` or `y`

```
union(x, y)
```

```
## [1] 1 2 3 6 5
```

To get values in `x` but not in `y`

```
setdiff(x, y)
```

```
## [1] 2
```

```
setdiff(y, x)
```

```
## [1] 6 5
```

To check whether each element of `x` is inside `y`

```
is.element(x, y)
```

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

or

```
x %in% y
```

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

2.10 Missing values

R uses `NA` to represent missing values indicating they are not available. In a data file, `NA`'s are very common and have to be dealt with properly. Why?

```
x <- c(1, NA, 2, NA, 3)
mean(x)
```

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

```
sum(x)
```

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

And it's contagious

```
y <- 1:5
```

```
x+y
```

```
## [1] 2 NA 5 NA 8
```

To deal with NA's, we need to know how to find indices with missing values

```
# Do we have any NA?
```

```
anyNA(x)
```

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

```
# Which ones?
```

```
is.na(x)
```

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

```
# Or
```

```
which(is.na(x))
```

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

How to remove? But before removing them:

```
mean(x, na.rm = TRUE)
```

```
## [1] 2
```

So we may skip removing them from the data as many functions have built-in arguments to deal with NA's.

```
x2 <- x[!is.na(x)]
```

```
x2
```

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

```
# Or
```

```
x[complete.cases(x)]
```

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

2.11 Factors

Factor type is known as an “indicator” variable.

```

set.seed((123))
anim <- sample(animals, 100, replace = TRUE)
anim

## [1] "donkey" "donkey" "donkey" "cat"      "donkey" "cat"      "cat"      "cat"
## [9] "donkey" "dog"     "cat"      "cat"      "dog"     "cat"      "donkey" "dog"
## [17] "donkey" "donkey" "dog"     "dog"     "dog"     "dog"     "donkey" "cat"
## [25] "donkey" "cat"     "dog"     "cat"      "donkey" "cat"      "dog"     "donkey"
## [33] "donkey" "dog"     "donkey" "cat"      "dog"     "donkey" "dog"     "dog"
## [41] "cat"     "donkey" "donkey" "dog"      "donkey" "dog"     "donkey" "cat"
## [49] "dog"     "cat"     "dog"     "dog"      "donkey" "dog"     "cat"     "dog"
## [57] "dog"     "donkey" "dog"     "cat"      "dog"     "donkey" "dog"     "donkey"
## [65] "cat"     "donkey" "cat"     "cat"      "donkey" "cat"     "cat"     "donkey"
## [73] "donkey" "dog"     "cat"     "cat"      "dog"     "cat"     "dog"     "dog"
## [81] "cat"     "donkey" "donkey" "dog"      "cat"     "dog"     "cat"     "dog"
## [89] "donkey" "donkey" "cat"     "donkey" "dog"     "cat"     "cat"     "donkey"
## [97] "cat"     "dog"     "donkey" "donkey" "donkey" "donkey" "cat"     "donkey"

table(anim)

## anim
##   cat     dog donkey
##   32     33    35

```

Let's define `anim` vector as factor variable:

```

animf <- as.factor(anim)
levels(animf)

## [1] "cat"     "dog"     "donkey"

```

Chapter 3

Other R Objects

This chapter introduces other types of R objects: matrix, data frame, list, and array.

3.1 Matrices

R stores matrices (and arrays) in a similar way as vectors, but with the attribute called dimension. A matrix is an array that has two dimensions. Data in a matrix are organized into rows and columns. Matrices are commonly used while arrays are rare. We will not see arrays in this book in detail. Matrices are **homogeneous** data structures, just like atomic vectors, but they can have 2 dimensions, rows and columns, unlike vectors.

Matrices can be created using the **matrix** function. In the **matrix()** function, after the data vector, **nrow** and **ncol** specify the desired numbers of rows and columns of the matrix.

```
#Let's create 5 x 4 numeric matrix containing numbers from 1 to 20
mymatrix <- matrix(1:20, nrow = 5, ncol = 4) #Here we order the number by columns
mymatrix

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

class(mymatrix)

## [1] "matrix" "array"
```

```
dim(mymatrix)
```

```
## [1] 5 4
```

Notice that the matrix is created by filling in the columns. If you want to fill the rows instead of columns, you can add the argument `byrow = TRUE`.

```
mymatrix <- matrix(1:20, nrow = 5, ncol = 4, byrow = TRUE)
mymatrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
```

We will be using two different variables. Following the usual mathematical convention, lower-case `x` (or any other letter), which stores a vector and capital `X`, which stores a matrix. Don't forget: we can do this because R is case sensitive.

If the length of the supplied vector is not equal to the number of rows multiplied by the number of columns, R will use the recycling rule on the vector to fill in the matrix, which is usefull:

```
matrix(6, 3, 3)
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]    6    6    6
## [3,]    6    6    6
```

After defining a matrix, we can apply various functions on it.

```
x <- matrix(1:12, nrow = 4)
```

```
dim(x)          #the dimension of a matrix
```

```
## [1] 4 3
```

```
nrow(x)         #the number of row of a matrix
```

```
## [1] 4
```

```
ncol(x)         #the number of column of a matrix
```

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

3.1.1 Matrix Operations

Now some key matrix operations:

```

X <- matrix(1:9, nrow = 3, ncol = 3)
Y <- matrix(11:19, nrow = 3, ncol = 3)

A <- X + Y
A

##      [,1] [,2] [,3]
## [1,]    12   18   24
## [2,]    14   20   26
## [3,]    16   22   28

B <- X * Y
B

##      [,1] [,2] [,3]
## [1,]    11   56  119
## [2,]    24   75  144
## [3,]    39   96  171

#The symbol %*% is called pipe operator.
#And it carries out a matrix multiplication
#different than a simple multiplication.

C <- X%*%Y
C

##      [,1] [,2] [,3]
## [1,]   150   186   222
## [2,]   186   231   276
## [3,]   222   276   330

```

Note that `X * Y` is not a matrix multiplication. It is element by element multiplication. (Same for `X / Y`). Instead, matrix multiplication uses `%*%`. Other matrix functions include `t()` which gives the transpose of a matrix and `solve()` which returns the inverse of a square matrix if it is invertible.

Here are some operations very useful when using matrices:

```

rowMeans(A)

## [1] 18 20 22

colMeans(B)

## [1] 24.66667 75.66667 144.66667

rowSums(B)

## [1] 186 243 306

```

```
colSums(A)
```

```
## [1] 42 60 78
```

Last thing: When vectors are coerced to become matrices, they are column vectors. So a vector of length n becomes an $n \times 1$ matrix after coercion.

```
x <- 1:5
X <- as.matrix(x)
X
```

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

3.1.2 Combine vectors or matrices into a matrix

The `matrix()` function is not the only way to create a matrix. Matrices can also be created by combining vectors as columns, using `cbind()`, or combining vectors as rows, using `rbind()`. Look at this:

#Let's create 2 vectors.

```
x <- rev(c(1:9))  #this can be done by c(9:1). I wanted to show rev()
x
```

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

```
y <- rep(2, 9)
y
```

```
## [1] 2 2 2 2 2 2 2 2 2
```

```
A <- rbind(x, y)
A
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## x     9     8     7     6     5     4     3     2     1
## y     2     2     2     2     2     2     2     2     2
```

```
B <- cbind(x, y)
B
```

```
##      x y
## [1,] 9 2
## [2,] 8 2
## [3,] 7 2
## [4,] 6 2
## [5,] 5 2
```

```

##  [6,] 4 2
##  [7,] 3 2
##  [8,] 2 2
##  [9,] 1 2
#You can label each column and row
colnames(B) <- c("column1", "column2")
B

##      column1 column2
##  [1,]      9      2
##  [2,]      8      2
##  [3,]      7      2
##  [4,]      6      2
##  [5,]      5      2
##  [6,]      4      2
##  [7,]      3      2
##  [8,]      2      2
##  [9,]      1      2

```

Also we can append or merge several matrices

```

m1 <- matrix(1:6, 2, 3)
m1

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

m2 <- matrix(5:10, 2, 3)
m2

##      [,1] [,2] [,3]
## [1,]    5    7    9
## [2,]    6    8   10

rbind(m1, m2) # Append

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

cbind(m1, m2) # Merge

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

```

3.1.3 Subsetting Matrix

Like vectors, matrices can be subsetted using square brackets, []. However, since matrices are two-dimensional, we need to specify both row and column indices when subsetting.

```

Y

##      [,1] [,2] [,3]
## [1,]    11   14   17
## [2,]    12   15   18
## [3,]    13   16   19

Y[1,3]

## [1] 17
Y[,3]

## [1] 17 18 19
Y[2,]

## [1] 12 15 18
Y[2, c(1, 3)] # If we need more than a column (row), we use c()

## [1] 12 18
rownames(Y) <- c("a", "b", "c")
colnames(Y) <- c("x", "y", "z")
Yn <- Y[, c(1, 3)]
Yn

##      x  z
## a 11 17
## b 12 18
## c 13 19

```

If you need to keep the result as a matrix, you can add a third dimension drop = FALSE in the subsetting operation.

Conditional subsetting is the same as before in vectors. Let's solve this problem: what's the number in column 1 in Y when the number in column 3 is 18?

```

Y

##      x  y  z
## a 11 14 17
## b 12 15 18
## c 13 16 19

Y[Y[,3]==18, 1]

```

```

## [1] 12
#What are the numbers in a row when the number in column 3 is 18?
Y[Y[,3]==19, ]

## x y z
## 13 16 19
#Print the rows in Y when the number in column 3 is more than 17?
Y[Y[,3] > 17, ]

## x y z
## b 12 15 18
## c 13 16 19

```

We will see later how these conditional subsetting can be done much smoother with data frames.

3.1.4 `apply()` function

We will see the `apply` family later in more detail. The `apply()` function is very handy for matrices if we may want to apply certain function on each row or column. It takes three arguments by default. The first argument is the object, the second argument is the dimension(s) to apply the function on, and the third argument is the function.

```

apply(Y, 2, mean)

## x y z
## 12 15 18

apply(Y, 1, mean)

## a b c
## 14 15 16

apply(Y, 2, sum)

## x y z
## 36 45 54

apply(Y, 2, sd)

## x y z
## 1 1 1

apply(Y, 2, function(g) g^2)

## x y z
## a 121 196 289
## b 144 225 324
## c 169 256 361

```

3.2 Data Frames

We have seen vectors and matrices for storing data. We will now introduce a data frame that is the most common way to store and interact with data. Data sets for statistical analysis are typically stored in data frames in R. Unlike a matrix, **a data frame can have different data types for each elements (columns)**. A data frame is a list of vectors (columns - you can think of them as “variables”). So, each vector (column) must contain the same data type, but the different vectors (columns) can store different data types.

However, unlike a list, the columns (elements) of a data frame must all be vectors and have the same length (number of observations)

Name	Weight	Height	Age
John	185	69	34.5
Emily	150	62	55.6
Mary	120	65	21.1
Dan	225	72	51.1

Each column vector
is a variable

Data frames combine the features of matrices and lists.

Like matrices, data frames are **rectangular**, where the columns are variables and the rows are observations of those variables. like lists, data frames can have elements (column vectors) of **different data types** (some double, some character, etc.) – but they **must be equal length**. Real data sets usually combine variables of different types, so data frames are well suited for storage.

```
#One way to do that
mydata <- data.frame(diabetic = c(TRUE, FALSE, TRUE, FALSE),
                      height = c(65, 69, 71, 73))
mydata

##   diabetic height
## 1      TRUE     65
## 2     FALSE     69
## 3      TRUE     71
## 4     FALSE     73
str(mydata)

## 'data.frame': 4 obs. of 2 variables:
## $ diabetic: logi  TRUE FALSE TRUE FALSE
## $ height  : num  65 69 71 73
dim(mydata)

## [1] 4 2
#Or create vectors for each column
diabetic = c(TRUE, FALSE, TRUE, FALSE)
height = c(65, 69, 71, 73)

#And include them in a data frame as follows
mydata <- data.frame(diabetic, height)
mydata

##   diabetic height
## 1      TRUE     65
## 2     FALSE     69
## 3      TRUE     71
## 4     FALSE     73
str(mydata)

## 'data.frame': 4 obs. of 2 variables:
## $ diabetic: logi  TRUE FALSE TRUE FALSE
## $ height  : num  65 69 71 73
```

```
dim(mydata)
## [1] 4 2
#And more importantly, you can extend it by adding more columns
weight = c(103, 45, 98.4, 70.5)
mydata <- data.frame(mydata, weight)
mydata

##   diabetic height weight
## 1      TRUE     65 103.0
## 2     FALSE     69  45.0
## 3      TRUE     71  98.4
## 4     FALSE     73  70.5
```

You will have the following mistake a lot. Let's see it now so you can avoid it later.

#Try running the code below separately without the comment # and see what happens

```
#mydata <- data.frame(diabetic = c(TRUE, FALSE, TRUE, FALSE, FALSE),
                      #height = c(65, 69, 71, 73))
```

The problem in the example above is that there are a different number of rows and columns. Here are some useful tools for diagnosing this problem:

```
#Number of columns
ncol(mydata)

## [1] 3

nrow(mydata)

## [1] 4
```

Often data you're working with has abstract column names, such as (x1, x2, x3...). The `cars` is data from the 1920s on “Speed and Stopping Distances of Cars”. There is only 2 columns shown below.

```
colnames(datasets::cars)

## [1] "speed" "dist"
#Using Base r:
colnames(cars)[1:2] <- c("Speed (mph)", "Stopping Distance (ft)")
colnames(cars)

## [1] "Speed (mph)"           "Stopping Distance (ft)"
#Using GREP:
colnames(cars)[grep("dist", colnames(cars))] <- "Stopping Distance (ft)"
colnames(cars)
```

```
## [1] "Speed (mph)"           "Stopping Distance (ft)"
```

Using `summary()` on a data frame, you get the summary statistics for each variable.

```
summary(cars)
```

```
##   Speed (mph)  Stopping Distance (ft)
##   Min.   : 4.0  Min.   : 2.00
##   1st Qu.:12.0  1st Qu.: 26.00
##   Median :15.0  Median : 36.00
##   Mean   :15.4  Mean   : 42.98
##   3rd Qu.:19.0  3rd Qu.: 56.00
##   Max.   :25.0  Max.   :120.00
```

3.2.1 Subsetting Data Frames

Subsetting data frames can work much like subsetting matrices using square brackets, `[,]`. Let's use another data given in the `ggplot2` library.

```
library(ggplot2)
head(mpg, n = 10)
```

```
## # A tibble: 10 x 11
##   manufacturer model      displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>        <chr>     <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi         a4         1.8   1999     4 auto~ f     18    29 p   comp~
## 2 audi         a4         1.8   1999     4 manu~ f     21    29 p   comp~
## 3 audi         a4          2    2008     4 manu~ f     20    31 p   comp~
## 4 audi         a4          2    2008     4 auto~ f     21    30 p   comp~
## 5 audi         a4         2.8   1999     6 auto~ f     16    26 p   comp~
## 6 audi         a4         2.8   1999     6 manu~ f     18    26 p   comp~
## 7 audi         a4         3.1   2008     6 auto~ f     18    27 p   comp~
## 8 audi         a4 quattro  1.8   1999     4 manu~ 4    18    26 p   comp~
## 9 audi         a4 quattro  1.8   1999     4 auto~ 4    16    25 p   comp~
## 10 audi        a4 quattro  2    2008     4 manu~ 4   20    28 p   comp~
```

And we need to see the cars with highway mpg over 35:

```
mpg[mpg$hwy > 35, c("manufacturer", "model", "year")]
```

```
## # A tibble: 6 x 3
##   manufacturer model      year
##   <chr>        <chr>     <int>
## 1 honda         civic     2008
## 2 honda         civic     2008
## 3 toyota        corolla   2008
## 4 volkswagen    jetta    1999
## 5 volkswagen    new beetle 1999
```

```
## 6 volkswagen new beetle 1999
```

An alternative would be to use the `subset()` function, which has a much more readable syntax.

```
subset(mpg, subset = hwy > 35, select = c("manufacturer", "model", "year"))

## # A tibble: 6 x 3
##   manufacturer model      year
##   <chr>        <chr>    <int>
## 1 honda        civic     2008
## 2 honda        civic     2008
## 3 toyota       corolla   2008
## 4 volkswagen   jetta    1999
## 5 volkswagen   new beetle 1999
## 6 volkswagen   new beetle 1999
```

Lastly, we could use the `filter` and `select` functions from the `dplyr` package which introduces the `%>%` operator from the `magrittr` package. This is not necessary for this book, however the `dplyr` package is something you should be aware of as it is becoming a popular tool in the R world.

```
library(dplyr)
mpg %>% filter(hwy > 35) %>% select(manufacturer, model, year)

## # A tibble: 6 x 3
##   manufacturer model      year
##   <chr>        <chr>    <int>
## 1 honda        civic     2008
## 2 honda        civic     2008
## 3 toyota       corolla   2008
## 4 volkswagen   jetta    1999
## 5 volkswagen   new beetle 1999
## 6 volkswagen   new beetle 1999
```

We will see `dplyr` later.

3.2.2 Tibble

Tibbles are data frames, but change some behaviors of data frames to make coding easier. To use the tibble class, you need to install the `tibble` package, which is part of the `tidyverse` package.

```
library(tibble)
animal <- rep(c("sheep", "pig"), c(3,3))
year <- rep(2019:2021, 2)
healthy <- c(rep(TRUE, 5), FALSE)
my_tibble <- tibble(animal, year, healthy)
my_tibble
```

```
## # A tibble: 6 x 3
##   animal  year healthy
##   <chr>   <int> <lgl>
## 1 sheep    2019 TRUE
## 2 sheep    2020 TRUE
## 3 sheep    2021 TRUE
## 4 pig      2019 TRUE
## 5 pig      2020 TRUE
## 6 pig      2021 FALSE
```

You can convert a tibble to a data frame or data frame to tibble.

```
my_data_frame <- data.frame(animal, year, healthy)
tt <- as_tibble(my_data_frame)
tt
```

```
## # A tibble: 6 x 3
##   animal  year healthy
##   <chr>   <int> <lgl>
## 1 sheep    2019 TRUE
## 2 sheep    2020 TRUE
## 3 sheep    2021 TRUE
## 4 pig      2019 TRUE
## 5 pig      2020 TRUE
## 6 pig      2021 FALSE
```

Or

```
bck <- as.data.frame(tt)
bck

##   animal year healthy
## 1 sheep  2019    TRUE
## 2 sheep  2020    TRUE
## 3 sheep  2021    TRUE
## 4 pig    2019    TRUE
## 5 pig    2020    TRUE
## 6 pig    2021 FALSE
```

In some aspects tibbles are useful, data frames are more common.

3.2.3 Plotting from data frame

There are many good ways and packages for plotting. I'll show you one here. Visualizing the relationship between multiple variables can get messy very quickly. Here is the `ggpairs()` function in the **GGally** package [?].

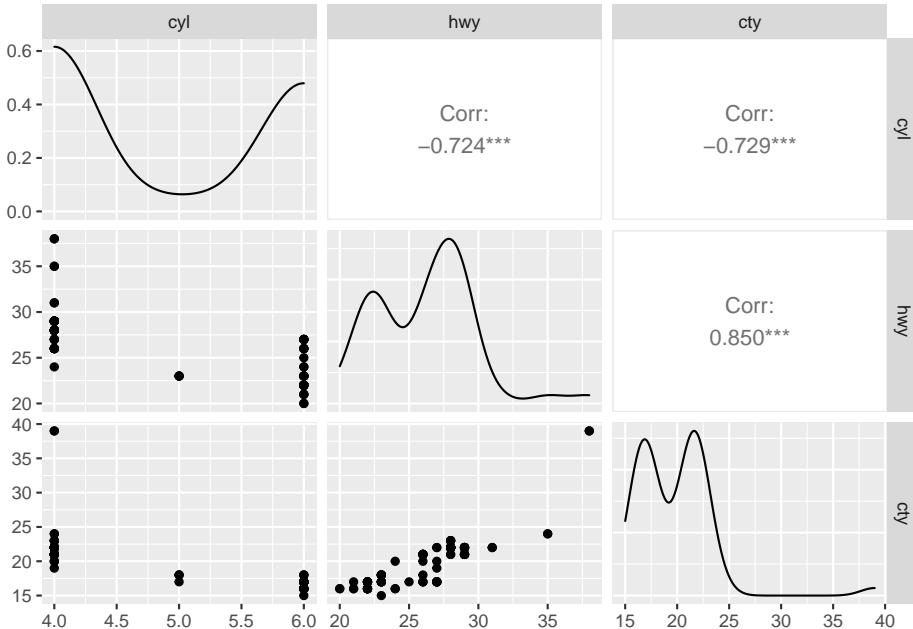
```
library(fueleconomy)  #install.packages("fueleconomy")
data(vehicles)
```

```
df <- vehicles[1:100, ]
str(df)

## # tibble [100 x 12] (S3: tbl_df/tbl/data.frame)
## # id   : num [1:100] 13309 13310 13311 14038 14039 ...
## # make : chr [1:100] "Acura" "Acura" "Acura" "Acura" ...
## # model: chr [1:100] "2.2CL/3.0CL" "2.2CL/3.0CL" "2.2CL/3.0CL" "2.3CL/3.0CL" ...
## # year  : num [1:100] 1997 1997 1997 1998 1998 ...
## # class  : chr [1:100] "Subcompact Cars" "Subcompact Cars" "Subcompact Cars" "Subcompact Cars"
## # trans  : chr [1:100] "Automatic 4-spd" "Manual 5-spd" "Automatic 4-spd" "Automatic 4-spd" ...
## # drive  : chr [1:100] "Front-Wheel Drive" "Front-Wheel Drive" "Front-Wheel Drive" "Front-Wheel
## # cyl    : num [1:100] 4 4 6 4 4 6 4 4 6 5 ...
## # displ  : num [1:100] 2.2 2.2 3 2.3 2.3 3 2.3 2.3 3 2.5 ...
## # fuel   : chr [1:100] "Regular" "Regular" "Regular" "Regular" ...
## # hwy   : num [1:100] 26 28 26 27 29 26 27 29 26 23 ...
## # cty   : num [1:100] 20 22 18 19 21 17 20 21 17 18 ...
```

Let's see how `GGally::ggpairs()` visualizes relationships between quantitative variables:

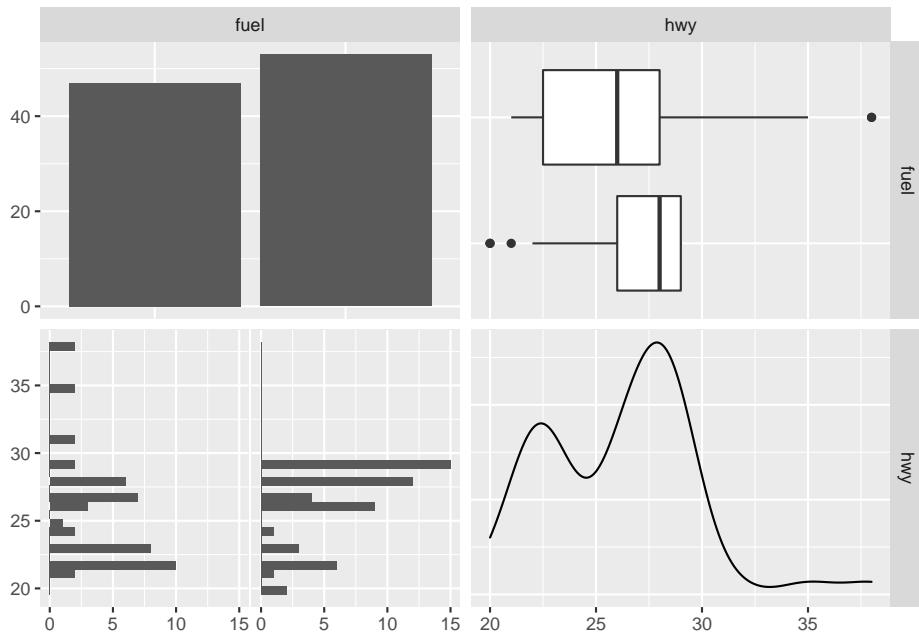
```
library(GGally) #install.packages("GGally")
new_df <- df[, c("cyl", "hwy", "cty")]
ggpairs(new_df)
```



The visualization changes a little when we have a mix of quantitative and categorical variables. Below, fuel is a categorical variable while hwy is a quantitative

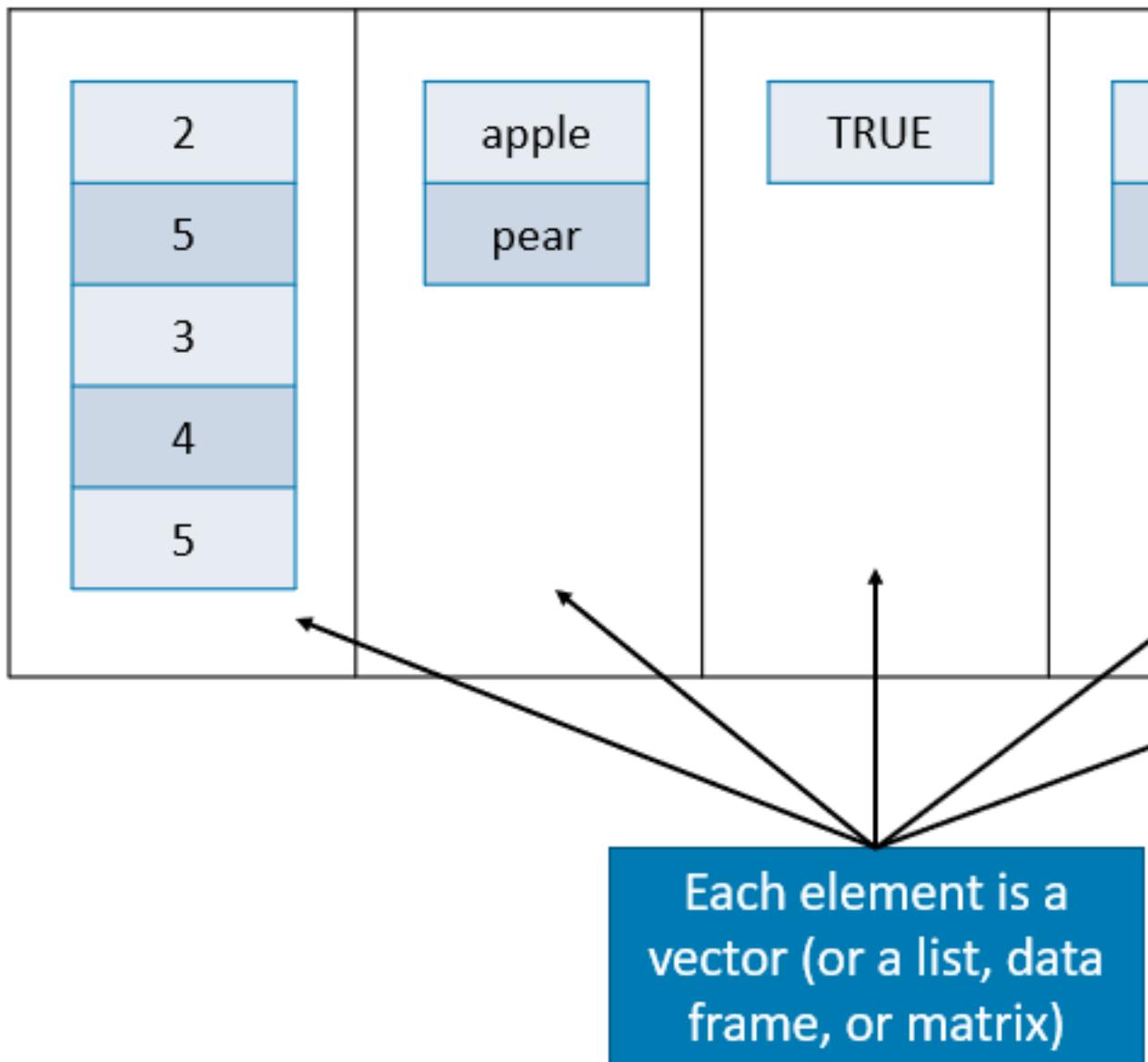
variable.

```
mixed_df <- df[, c("fuel", "hwy")]
ggpairs(mixed_df)
```



3.3 Lists

A list is a one-dimensional heterogeneous data structure. So it is indexed like a vector with a single integer value, but each element can contain an element of any type.



Lets look at some examples of working with them:

```
# creation
A <- list(42, "Hello", TRUE)
```

```

dim(A)

## NULL

str(A)

## List of 3
## $ : num 42
## $ : chr "Hello"
## $ : logi TRUE
class(A)

## [1] "list"
# Another one
B <- list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Hello!",
  d = function(arg = 1) {print("Hello World!")},
  X = matrix(0, 4, 4)
)
B

## $a
## [1] 1 2 3 4
##
## $b
## [1] TRUE
##
## $c
## [1] "Hello!"
##
## $d
## function(arg = 1) {print("Hello World!"})
##
## $X
##      [,1] [,2] [,3] [,4]
## [1,]     0     0     0     0
## [2,]     0     0     0     0
## [3,]     0     0     0     0
## [4,]     0     0     0     0
dim(B)

## NULL

```

```

dim(B$X)

## [1] 4 4

str(B)

## List of 5
## $ a: num [1:4] 1 2 3 4
## $ b: logi TRUE
## $ c: chr "Hello!"
## $ d:function (arg = 1)
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 12 15 12 55 15 55 12 12
##   ... ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7fa2883fe248>
## $ X: num [1:4, 1:4] 0 0 0 0 0 0 0 0 0 0 ...
class(B)

## [1] "list"

```

Lists can be subset using two types of syntax, the \$ operator, and square brackets []. The \$ operator returns a named element of a list. The [] syntax returns a list, while the [[]] returns an element of a list.

#For example to get the matrix in our list

B\$X

```

##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0

```

#or

B[5]

```

## $X
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0

```

#or

B[[5]]

```

##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0

```

```
#And to get the (1,3) element of matrix X in list B
B[[5]][1,3]
```

```
## [1] 0
```

What's the difference between the results of `B[[5]]` and `B[5]`? The former is the third element of `my_list` which is a matrix, while the latter is a list containing a single matrix element. Let's confirm this by looking at their structures.

```
str(B[[5]])
```

```
##  num [1:4, 1:4] 0 0 0 0 0 0 0 0 0 0 ...
str(B[5])
```

```
## List of 1
## $ X: num [1:4, 1:4] 0 0 0 0 0 0 0 0 0 0 ...
```

3.4 Array

Array can be viewed as an extension of vector and matrix to a higher dimensional space, and still **only contains elements of the same type**.

```
A <- array(1:24, c(2,3,4))
```

```
A
```

```
## , , 1
##
## [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
## [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
## [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
## [,1] [,2] [,3]
## [1,]   19   21   23
```

```
## [2,] 20 22 24
dim(A)
## [1] 2 3 4
```

The first argument is the data input (1:241). The second argument is the dimension of the array: 2 is the number of rows, 3 is the number of columns, and 4 is how many matrices we will have. Here is an example with a higher dimension:

```
y <- array(0, c(2,3,4,5))
y

## , , 1, 1
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 2, 1
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 3, 1
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 4, 1
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 1, 2
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 2, 2
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
```

```
## [2,]    0    0    0
##
## , , 3, 2
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 4, 2
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 1, 3
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 2, 3
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 3, 3
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 4, 3
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 1, 4
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 2, 4
##
```

```

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 3, 4
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 4, 4
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 1, 5
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 2, 5
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 3, 5
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
##
## , , 4, 5
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
dim(y)

## [1] 2 3 4 5
A[1, 2, 3]      #the 3rd matrix with [1,2]
## [1] 15

```

```

A[, , 2]      #the 2nd matrix

## [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12

A[2, , 4]      #the 4th matrix, 2nd row

## [1] 20 22 24
A[-2, 3, -3]   #Expect the 3rd matrix, get the 3rd columns without the 2nd rows

## [1] 5 11 23
apply(A, 1, mean)      #calculate the mean all rows (we have 2 rows)

## [1] 12 13
# Verify them
mean(A[1, , ])        #mean of all 1st rows

## [1] 12
mean(A[2, , ])        #mean of all 2nd rows

## [1] 13
apply(A, 2, sum)       #calculate the sum of all columns

## [1] 84 100 116
apply(A, 3, sd)        #calculate the sd each matrix

## [1] 1.870829 1.870829 1.870829 1.870829

```

Chapter 4

Reading and writting data files

4.1 Reading (importing)

For any data analysis, we need data. Data come in all different formats. The data could be readable, sometimes called ascii format. Or the data could be unreadable without the original program, like an Excel workbook (.xlsx) or other statistical software formats like Stata (.dta) or SAS (.sas7bdat).

There are many ways of bringing data into your workspace. A more flexible way to import data is to use **Import Dataset** on the Environment tab in the upper right window of RStudio . Multiple file type options are shown, such as text, Excel, SPSS, SAS, and Stata.

The screenshot shows the RStudio interface with the 'Environment' tab selected. In the top right, there is a 'Import Dataset' button with a dropdown arrow. A context menu is open, listing several options: 'From Text (base)...', 'From Text (readr)...', 'From Excel...', 'From SPSS...', 'From SAS...', and 'From Stata...'. The 'From SAS...' option is highlighted with a blue selection bar. To the right of the menu, a list of variables is visible, including 'mydat', 'mymat', 'predi', 'table', 'X', and 'Y'. Below this, a 'Values' section shows two entries: 'a' and 'diabetic'. The 'a' entry is of type 'int [1:3]' with values 1, 2, 6. The 'diabetic' entry is of type 'lolo [1:4]' with values TRUE, FALSE, TRUE, FALSE.

When you read a data in other formats, they may also be imported as a data frame.

If the data is a .csv file, for example, we would also use the `read_csv()` function from the `readr` package. Note that R has a built in function `read.csv()` that operates very similarly. The `readr` function `read_csv()` has a number of advantages. For example, it is much faster reading larger data. It also uses the `tibble` package to read the data as a `tibble`.

```
library(readr)
library(RCurl)
x <- getURL("https://raw.githubusercontent.com/tidyverse/readr/main/inst/extdata/mtcars.csv")
example_csv = read_csv(x, show_col_types = FALSE)
head(example_csv)

## # A tibble: 6 x 11
##   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 21      6   160   110  3.9   2.62  16.5     0     1     4     4
## 2 21      6   160   110  3.9   2.88  17.0     0     1     4     4
## 3 22.8    4   108   93   3.85  2.32  18.6     1     1     4     1
## 4 21.4    6   258   110  3.08  3.22  19.4     1     0     3     1
```

```

## 5 18.7      8   360   175  3.15  3.44 17.0      0      0   3   2
## 6 18.1      6   225   105  2.76  3.46 20.2      1      0   3   1
str(example_csv)

## spec_tbl_df [32 x 11] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ mpg : num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
## $ cyl : num [1:32] 6 6 4 6 8 6 8 4 4 6 ...
## $ disp: num [1:32] 160 160 108 258 360 ...
## $ hp  : num [1:32] 110 110 93 110 175 105 245 62 95 123 ...
## $ drat: num [1:32] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
## $ wt  : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
## $ qsec: num [1:32] 16.5 17 18.6 19.4 17 ...
## $ vs  : num [1:32] 0 0 1 1 0 1 0 1 1 1 ...
## $ am  : num [1:32] 1 1 1 0 0 0 0 0 0 0 ...
## $ gear: num [1:32] 4 4 4 3 3 3 3 4 4 4 ...
## $ carb: num [1:32] 4 4 1 1 2 1 4 2 2 4 ...
## - attr(*, "spec")=
##   .. cols(
##     ..  mpg = col_double(),
##     ..  cyl = col_double(),
##     ..  disp = col_double(),
##     ..  hp = col_double(),
##     ..  drat = col_double(),
##     ..  wt = col_double(),
##     ..  qsec = col_double(),
##     ..  vs = col_double(),
##     ..  am = col_double(),
##     ..  gear = col_double(),
##     ..  carb = col_double()
##   .. )
## - attr(*, "problems")=<externalptr>

```

A tibble is simply a data frame that prints with sanity. Notice in the output above that we are given additional information such as dimension and variable type.

To understand more about the data set, we use the `?` operator to pull up the documentation for the data. (You can use `??` to search the Internet for more info)

```

?mtcars
??mtcars
#View(mpg)
head(mtcars)

##          mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1    4    4

```

```

## Mazda RX4 Wag    21.0   6 160 110 3.90 2.875 17.02  0 1 4 4
## Datsun 710      22.8   4 108 93 3.85 2.320 18.61  1 1 4 1
## Hornet 4 Drive  21.4   6 258 110 3.08 3.215 19.44  1 0 3 1
## Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0 0 3 2
## Valiant         18.1   6 225 105 2.76 3.460 20.22  1 0 3 1

tail(mtcars)

##          mpg cyl disp hp drat    wt qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3 91 4.43 2.140 16.7  0 1 5 2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1 1 5 2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0 1 5 4
## Ferrari Dino  19.7   6 145.0 175 3.62 2.770 15.5  0 1 5 6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0 1 5 8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1 1 4 2

```

After importing our data, a quick glance at the dataset can often tell us if the data were read in correctly. Use `head()` and `tail()` to look at a specified number of rows at the beginning or end of a dataset, respectively. Use `View()` on a dataset to open a spreadsheet-style view of a dataset. In RStudio, clicking on a dataset in the Environment pane will `View()` it.

4.2 Writing (exporting)

We can export our data in a number of formats, including text, Excel .xlsx, and in other statistical software formats like Stata .dta, using `write_functions` that reverse the operations of the `read_functions`.

Multiple objects can be stored in an R binary file (usually extension “.Rdata”) with `save()` and then later loaded with `load()`.

I did not specify realistic path names below.

- Excel .csv file: `write_csv(dat_csv, file = "path/to/save/filename.csv")`
- Stata .dta file: `write_dta(dat_csv, file = "path/to/save/filename.dta")`
- save these objects to an .Rdata file: `save(dat_csv, mydata, file="path/to/save/filename.Rdata")`

One last thing: if you want to save the entire workspace, `save.image()` is just a short-cut for “save my current workspace”, i.e., `save(list = ls(all.names = TRUE), file = ".RData", envir = .GlobalEnv")`. It is also what happens with `q("yes")`.

Chapter 5

Data visualisation with R

Let's first introduce the data set that will be used throughout this chapter. The data set is a part of the Ames Housing Price data, containing 165 observations and 12 features including the sale date and price.

```
library(AmesHousing)
amesdata <- make_ames()
str(amesdata)

## # tibble [2,930 x 81] (S3: tbl_df/tbl/data.frame)
## # $ MS_SubClass      : Factor w/ 16 levels "One_Story_1946_and_Newer_All_Styles",...: 1 1 1 1 1 1 1 1 1 1 ...
## # $ MS_Zoning        : Factor w/ 7 levels "Floating_Village_Residential",...: 3 2 3 3 3 3 3 3 3 3 ...
## # $ Lot_Frontage     : num [1:2930] 141 80 81 93 74 78 41 43 39 60 ...
## # $ Lot_Area         : int [1:2930] 31770 11622 14267 11160 13830 9978 4920 5005 5389 7500 ...
## # $ Street            : Factor w/ 2 levels "Grvl","Pave": 2 2 2 2 2 2 2 2 2 2 ...
## # $ Alley              : Factor w/ 3 levels "Gravel","No_Alley_Access",...: 2 2 2 2 2 2 2 2 2 2 ...
## # $ Lot_Shape          : Factor w/ 4 levels "Regular","Slightly_Irregular",...: 2 1 2 1 2 2 1 2 2 ...
## # $ Land_Contour       : Factor w/ 4 levels "Bnk","HLS","Low",...: 4 4 4 4 4 4 4 4 2 4 4 ...
## # $ Utilities          : Factor w/ 3 levels "AllPub","NoSeWa",...: 1 1 1 1 1 1 1 1 1 1 ...
## # $ Lot_Config          : Factor w/ 5 levels "Corner","CulDSac",...: 1 5 1 1 5 5 5 5 5 5 ...
## # $ Land_Slope          : Factor w/ 3 levels "Gtl","Mod","Sev": 1 1 1 1 1 1 1 1 1 1 ...
## # $ Neighborhood        : Factor w/ 29 levels "North_Ames","College_Creek",...: 1 1 1 1 1 7 7 7 17 17 ...
## # $ Condition_1         : Factor w/ 9 levels "Artery","Feedr",...: 3 2 3 3 3 3 3 3 3 3 ...
## # $ Condition_2         : Factor w/ 8 levels "Artery","Feedr",...: 3 3 3 3 3 3 3 3 3 ...
## # $ Bldg_Type           : Factor w/ 5 levels "OneFam","TwoFmCon",...: 1 1 1 1 1 1 5 5 5 1 ...
## # $ House_Style          : Factor w/ 8 levels "One_and_Half_Fin",...: 3 3 3 3 8 8 3 3 3 8 ...
## # $ Overall_Qual        : Factor w/ 10 levels "Very_Poor","Poor",...: 6 5 6 7 5 6 8 8 8 7 ...
## # $ Overall_Cond        : Factor w/ 10 levels "Very_Poor","Poor",...: 5 6 6 5 5 6 5 5 5 5 ...
## # $ Year_Built          : int [1:2930] 1960 1961 1958 1968 1997 1998 2001 1992 1995 1999 ...
## # $ Year_Remod_Add      : int [1:2930] 1960 1961 1958 1968 1998 1998 2001 1992 1996 1999 ...
## # $ Roof_Style           : Factor w/ 6 levels "Flat","Gable",...: 4 2 4 4 2 2 2 2 2 2 ...
```



```

## ... .-$ Condition 2      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Bldg Type       : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ House Style     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Overall Qual    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Overall Cond    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Year Built      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Year Remod/Add : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Roof Style      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Roof Matl       : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Exterior 1st     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Exterior 2nd     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Mas Vnr Type    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Mas Vnr Area    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Exter Qual      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Exter Cond      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Foundation      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Bsmt Qual       : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Bsmt Cond       : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Bsmt Exposure   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ BsmtFin Type 1  : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ BsmtFin SF 1    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ BsmtFin Type 2  : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ BsmtFin SF 2    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"

```

```

## ... .-$ Bsmt Unf SF      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Total Bsmt SF   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Heating        : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ Heating QC     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ Central Air    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ Electrical     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ 1st Flr SF     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ 2nd Flr SF     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Low Qual Fin SF: list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Gr Liv Area   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Bsmt Full Bath : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Bsmt Half Bath : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Full Bath      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Half Bath      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Bedroom AbvGr   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Kitchen AbvGr   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Kitchen Qual    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ TotRms AbvGrd   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Functional      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ Fireplaces      : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... .-$ Fireplace Qu    : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ Garage Type     : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... .-$ Garage Yr Blt   : list()
## ... .-$ attr(*, "class")= chr [1:2] "collector_integer" "collector"

```

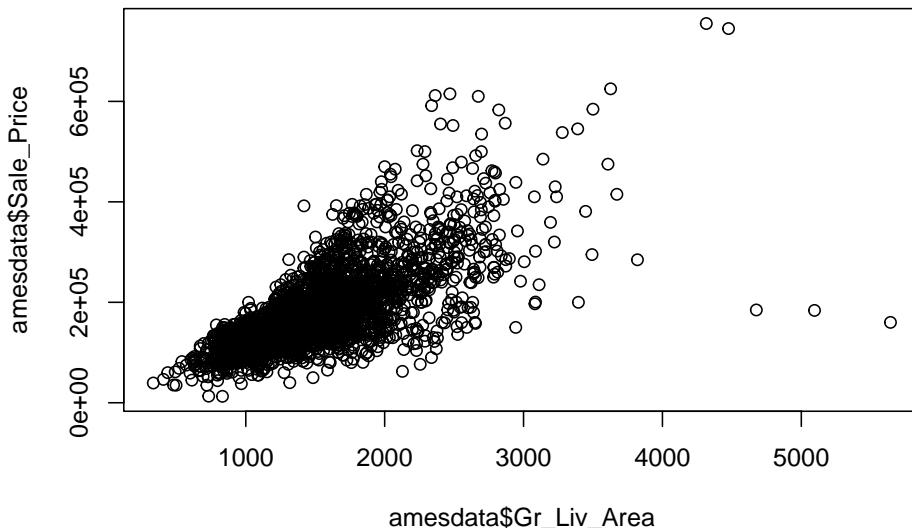
```

## ... .-$ Garage Finish : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Garage Cars : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Garage Area : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Garage Qual : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Garage Cond : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Paved Drive : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Wood Deck SF : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Open Porch SF : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Enclosed Porch : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ 3Ssn Porch : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Screen Porch : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Pool Area : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Pool QC : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Fence : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Misc Feature : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Misc Val : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Mo Sold : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Yr Sold : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ Sale Type : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ Sale Condition : list()
## ... .- attr(*, "class")= chr [1:2] "collector_character" "collector"
## ... $ SalePrice : list()
## ... .- attr(*, "class")= chr [1:2] "collector_integer" "collector"
## ... $ default: list()
## ... - attr(*, "class")= chr [1:2] "collector_guess" "collector"
## ... - attr(*, "class")= chr "col_spec"

```

5.1 Scatterplots

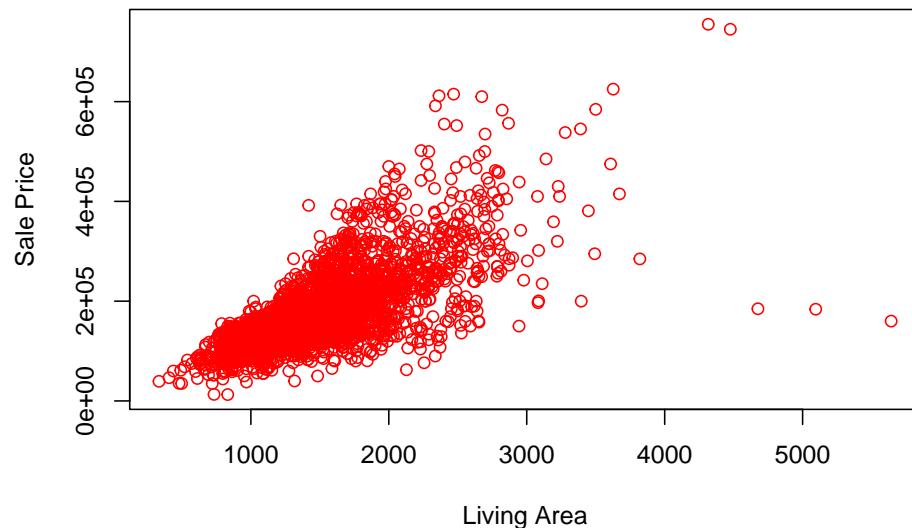
Are the bigger houses more expensive? In base R, we can use the `plot()` function to generate this scatterplot with the first argument being the variable on the x-axis and the second argument being the variable on the y-axis.



It's OK but not nice. The `plot()` function provides a rich capability of customization by setting the graphical parameters.

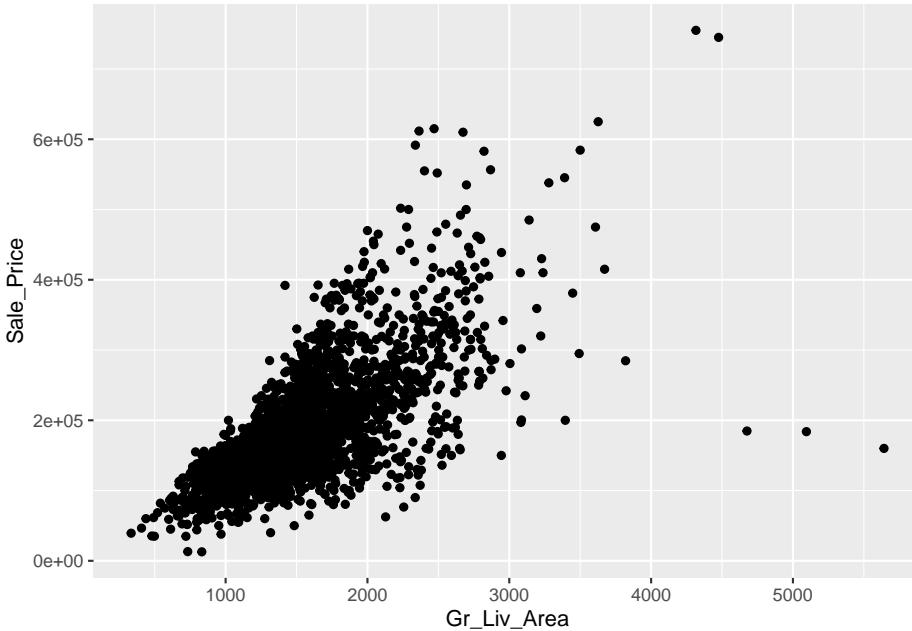
```
plot(amesdata$Gr_Liv_Area, amesdata$Sale_Price,
  col = "red",
  xlab = "Living Area",
  ylab = "Sale Price",
  main = "Sale Price vs. Living Area",
  pch = 1, #Shape of the points
  cex = 1) #Size of text and symbols
```

Sale Price vs. Living Area



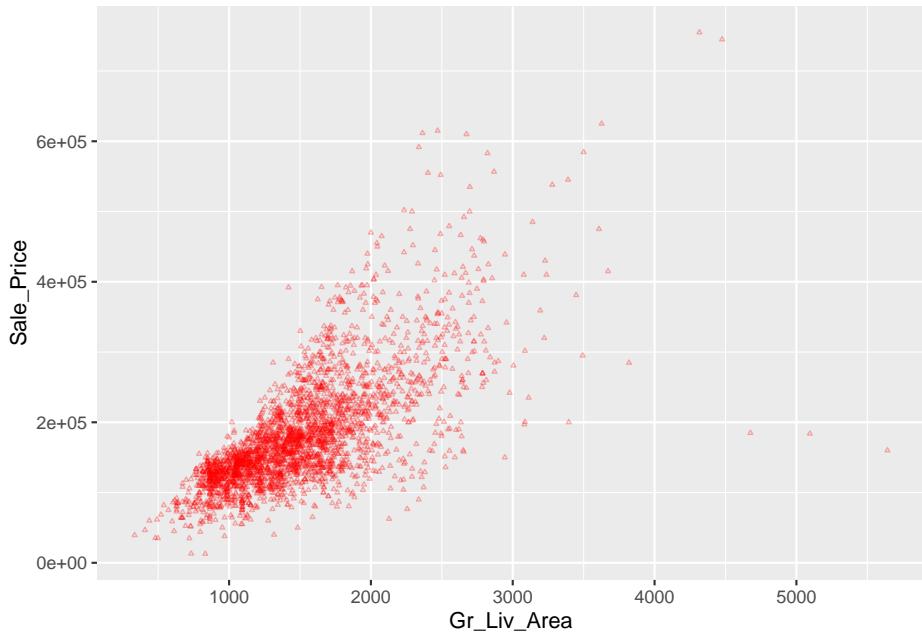
Although the `plot()` function gets the work done, the `ggplot2` package provides a superior user experience. It's a member of the `tidyverse` package, you don't need to install it separately if `tidyverse` was already installed.

```
library(ggplot2)
ggplot(data = amesdata) +
  geom_point(mapping = aes(x = Gr_Liv_Area, y = Sale_Price))
```



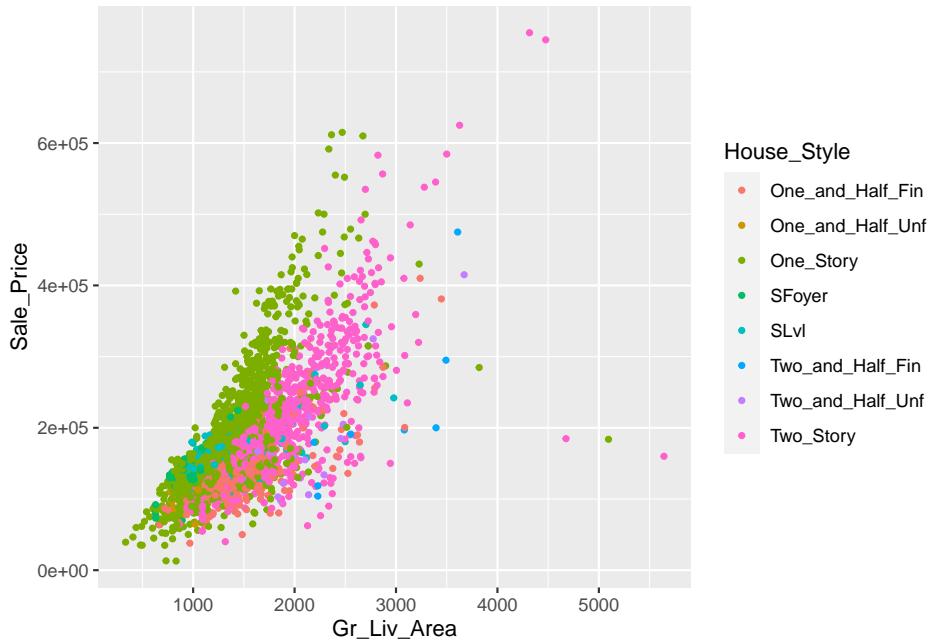
In a nutshell, `ggplot2` implements the grammar of graphics, a coherent system for describing and building graphs. We always start with the function `ggplot()` with a data frame or tibble as its argument. To generate a scatterplot, you can add a layer using the `+` operator followed by the `geom_point()` function, which is one of the many available geoms in `ggplot`. Inside `geom_point()`, you need to set the value of the mapping argument. The mapping argument takes a functional form as `mapping = aes()`, where the `aes` is short for aesthetics. For example, you can use `aes()` to tell `ggplot` to use which variable on the x-axis, which variable on the y-axis.

```
ggplot(data = amesdata) +
  geom_point(mapping = aes(Gr_Liv_Area, y = Sale_Price),
             color = "red",
             shape = 2,
             size = 0.5,
             alpha = 0.3) #transparency level of the points
```



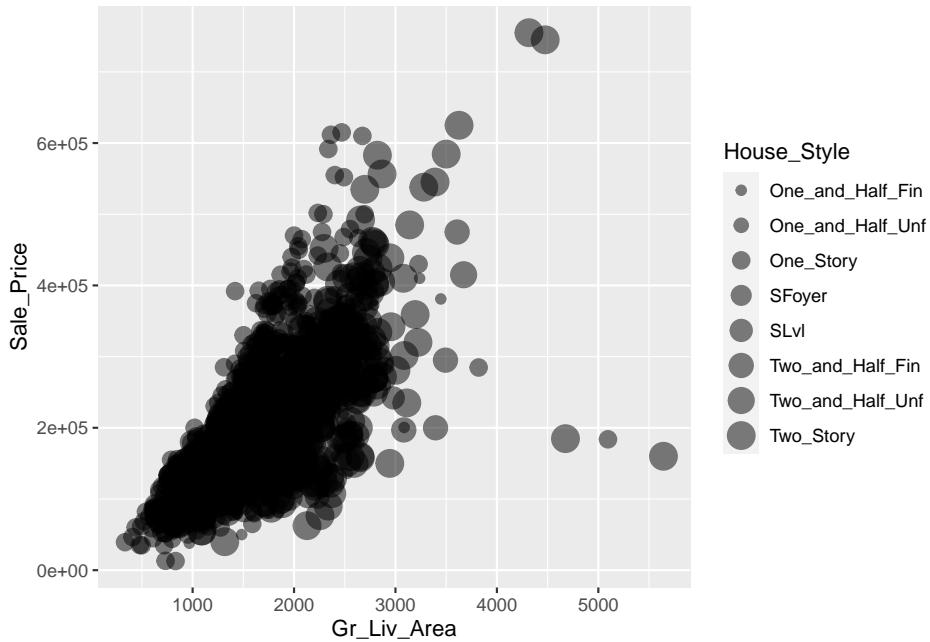
Suppose we want to use different colors according to the different `House_Style` in the scatterplot.

```
ggplot(data = amesdata) +
  geom_point(mapping = aes(x = Gr_Liv_Area,
                           y = Sale_Price,
                           color = House_Style),
             size = 1)
```



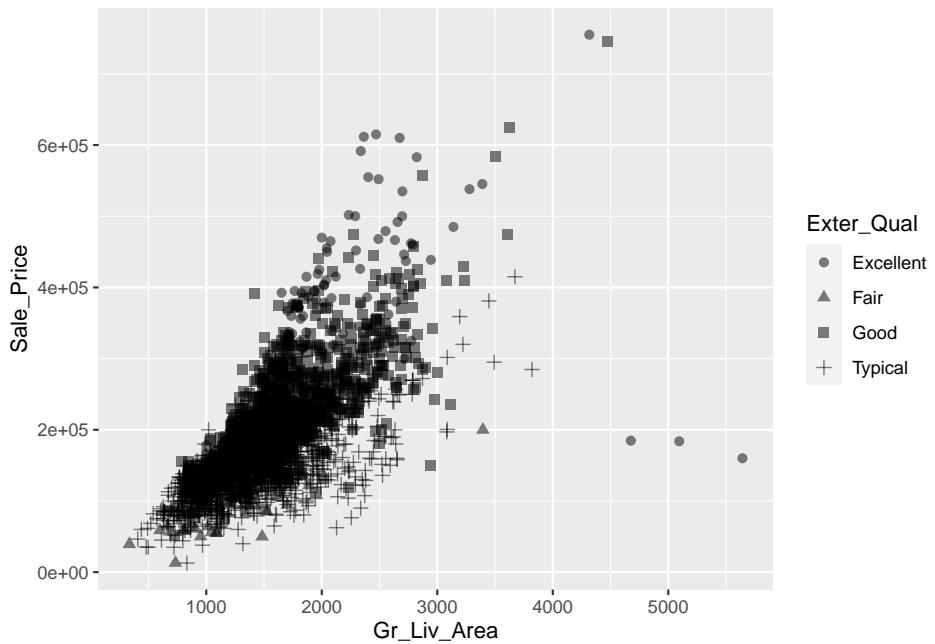
In addition to color, you can also map a discrete variable to the size aesthetic.

```
ggplot(data = amesdata) +
  geom_point(mapping = aes(x = Gr_Liv_Area,
                           y = Sale_Price,
                           size = House_Style),
             alpha = 0.5)
```



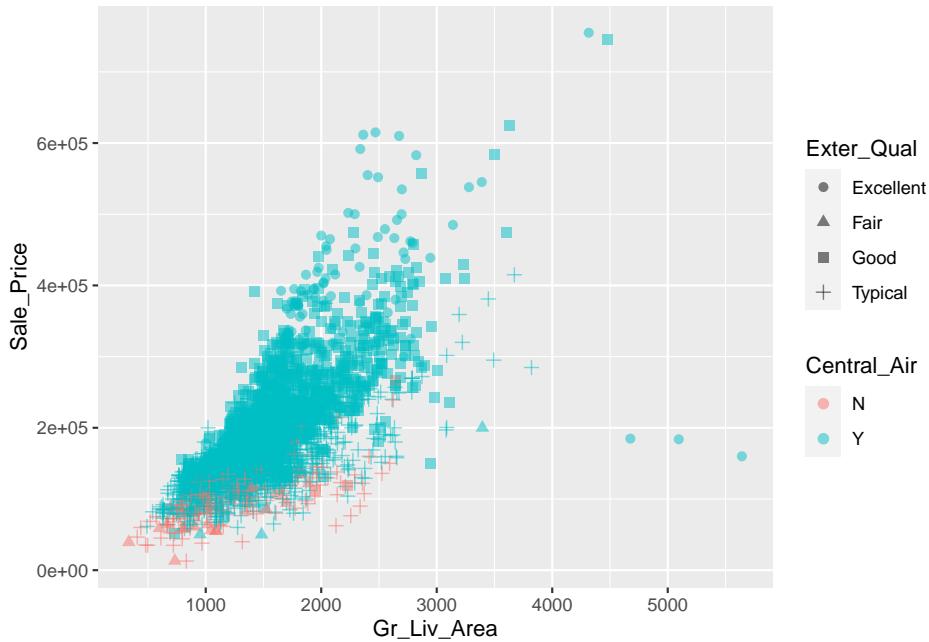
Or

```
ggplot(data = amesdata) +  
  geom_point(mapping = aes(x = Gr_Liv_Area,  
                           y = Sale_Price,  
                           shape = Exter_Qual),  
             alpha = 0.5,  
             size = 2)
```



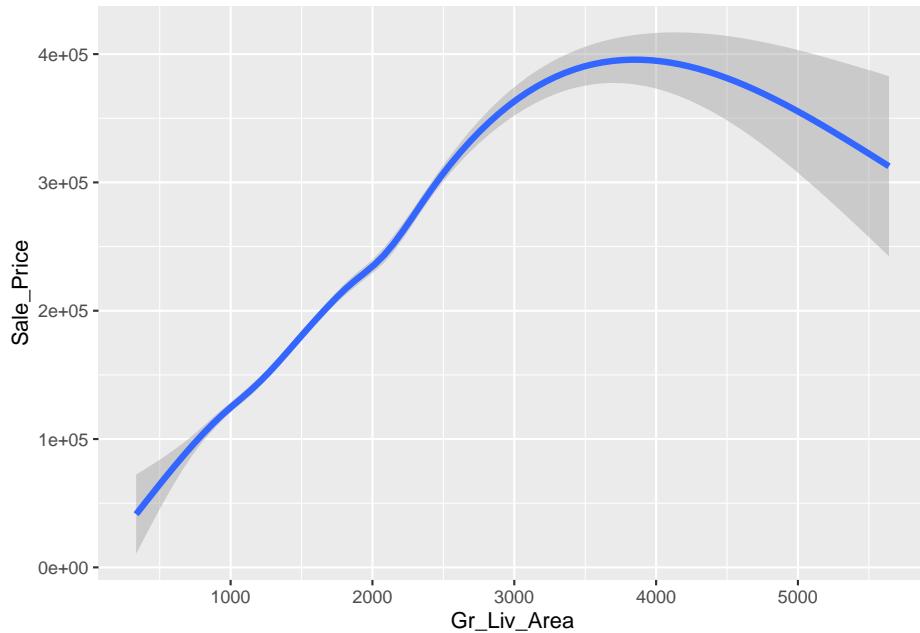
and multiple mapping:

```
ggplot(data = amesdata) +
  geom_point(mapping = aes(x = Gr_Liv_Area,
                           y = Sale_Price,
                           shape = Exter_Qual,
                           color = Central_Air),
             alpha = 0.5,
             size = 2)
```



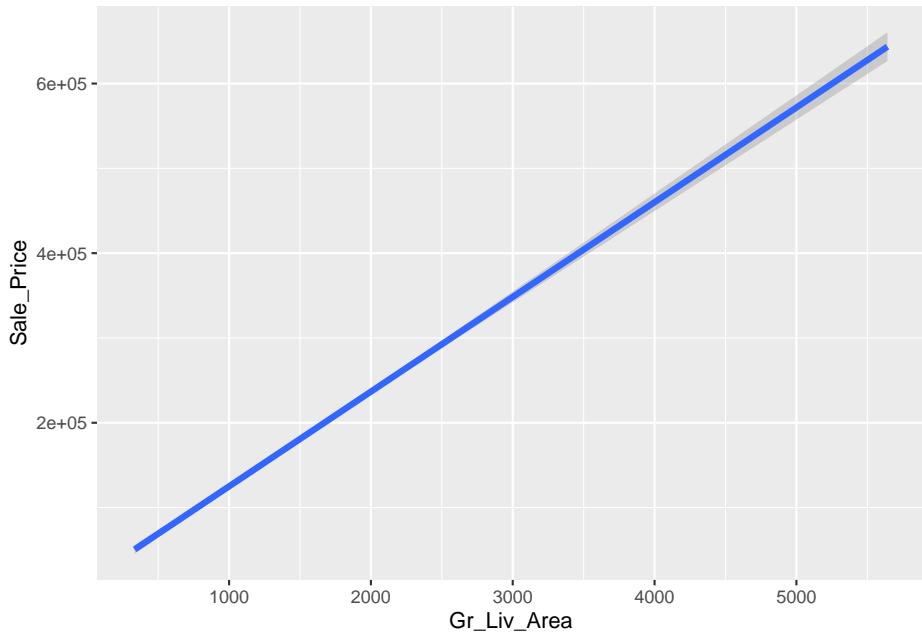
```
ggplot(data = amesdata) +
  geom_smooth(mapping = aes(x = Gr_Liv_Area,
                            y = Sale_Price),
              size = 1.5)
```

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



```
ggplot(data = amesdata) +  
  geom_smooth(mapping = aes(x = Gr_Liv_Area,  
                            y = Sale_Price),  
              size = 1.5,  
              method = "lm")
```

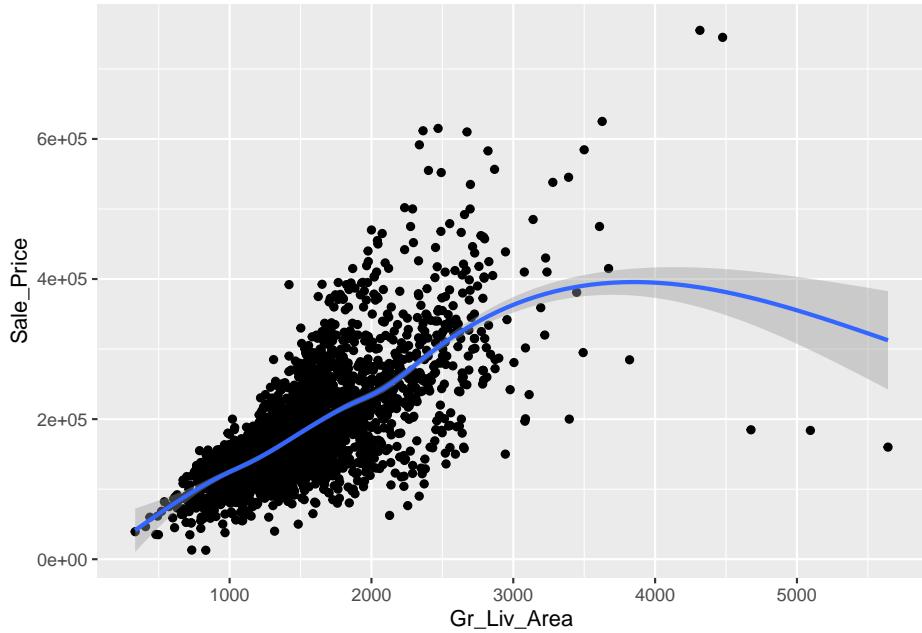
```
## `geom_smooth()` using formula 'y ~ x'
```



And

```
ggplot(data = amesdata) +
  geom_point(mapping = aes(x = Gr_Liv_Area,
                           y = Sale_Price)) +
  geom_smooth(mapping = aes(x = Gr_Liv_Area,
                           y = Sale_Price))
```

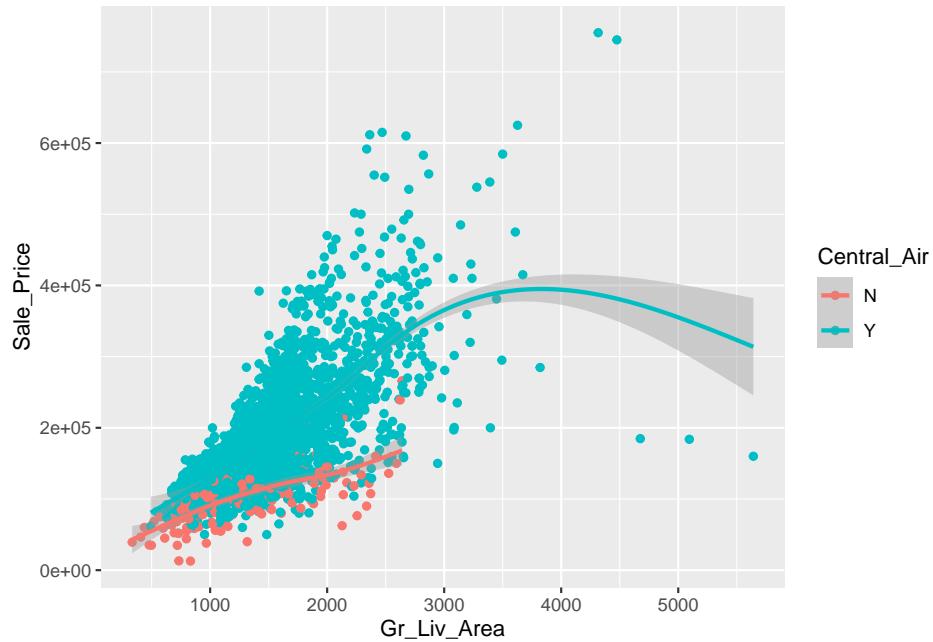
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Or with global mapping

```
ggplot(data = amesdata,
        mapping = aes(x = Gr_Liv_Area,
                      y = Sale_Price,
                      color = Central_Air)) +
  geom_point() +
  geom_smooth()
```

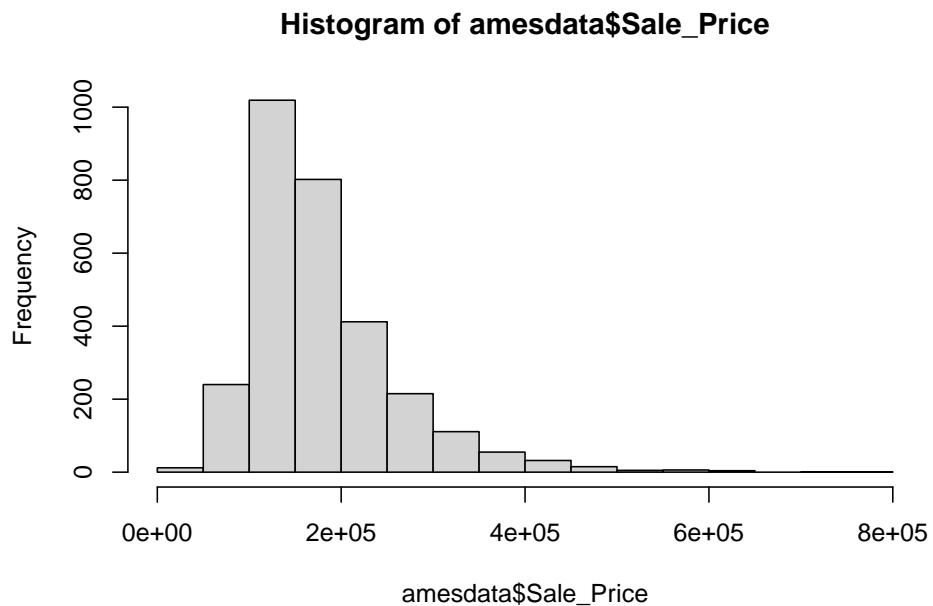
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



5.3 Histograms & Density

How do we visualize continuous variables? One popular plot is called histograms.

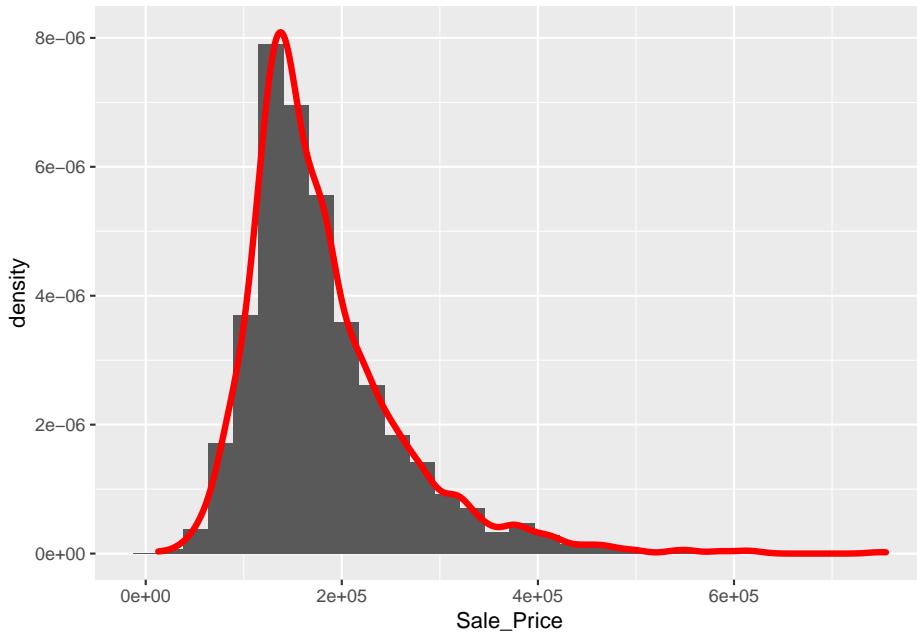
```
hist(amesdata$Sale_Price)
```



And density with ggplot

```
ggplot(data = amesdata, aes(x = Sale_Price)) +
  geom_histogram(aes(y = ..density..)) +
  geom_density(color = "red",
               size = 1.5)
```

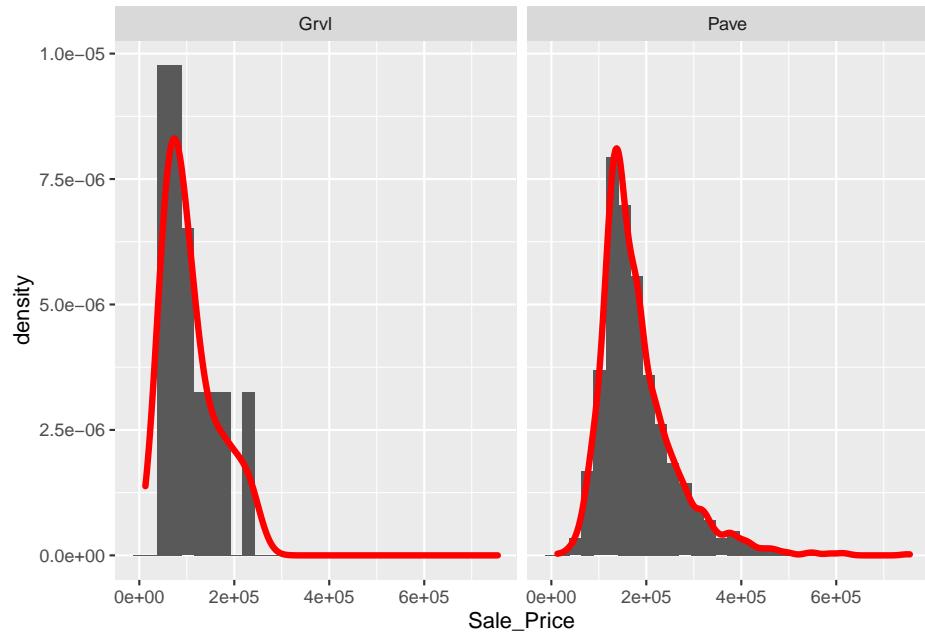
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



5.4 Multiple plots

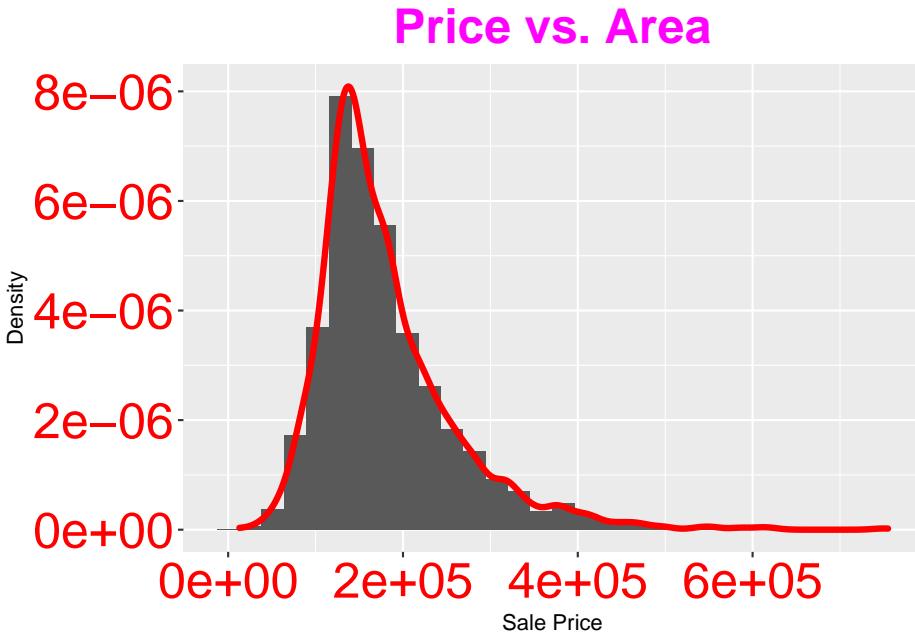
```
fc <- ggplot(data = amesdata, aes(x = Sale_Price)) +
  geom_histogram(aes(y = ..density..)) +
  geom_density(color = "red",
               size = 1.5)
fc + facet_wrap("Street")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



```
## Lables etc.
fc +
  xlab("Sale Price") +
  ylab("Density") +
  ggtitle("Price vs. Area") +
  theme(axis.text = element_text(size = 25, color = "red")) +
  theme(plot.title = element_text(size = 24,
                                    color = "magenta",
                                    face = "bold",
                                    hjust = 0.5))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

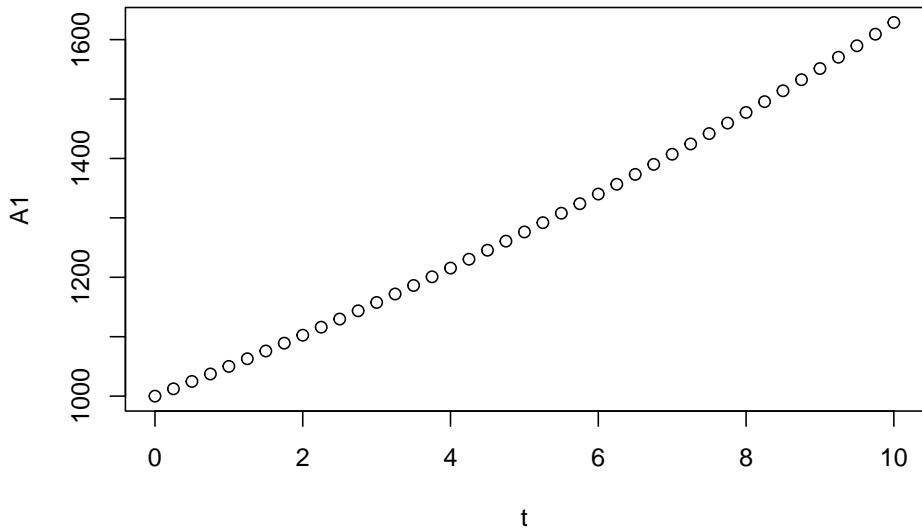


5.5 Add lines

Let's create some series (compounding series) of \$1000:

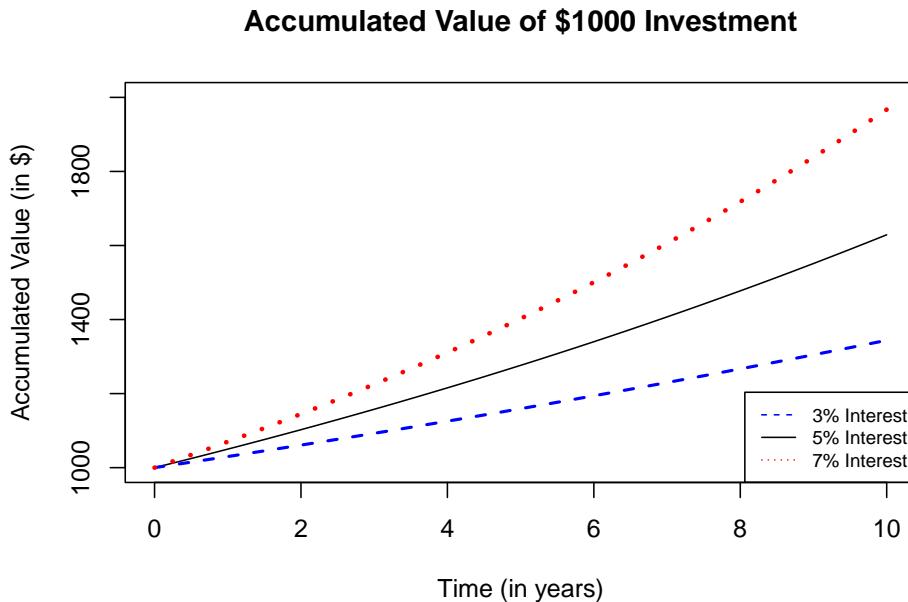
```
t <- seq(0, 10, 1/4)
A1 <- 1000*(1+0.05)^t
A2 <- 1000*(1+0.03)^t
A3 <- 1000*(1+0.07)^t
A4 <- 1000*(1+0.1)^t

plot(t, A1)
```



Here is a simple but beautiful plot ... So you may not want to use ggplot all the time:

```
plot(t, A1, ylim = c(1000,2000),
  type = "l", xlab = "Time (in years)",
  ylab = "Accumulated Value (in $)",
  main = "Accumulated Value of $1000 Investment", col = "black")
lines(t, A2, type = "l", col = "blue", lty = 2, lwd = 2)
lines(t, A3, type = "l", col = "red", lty = 3, lwd = 3)
legend("bottomright",
  legend = c("3% Interest", "5% Interest", "7% Interest"),
  col = c("blue", "black", "red"),
  lty = c(2, 1, 3), bty = "o", cex = 0.75)
```



Or we can put them next to each other:

```
par(mfrow=c(2,2), oma = c(0,0,2,0)) # puts 4 plots in one window (2x2)

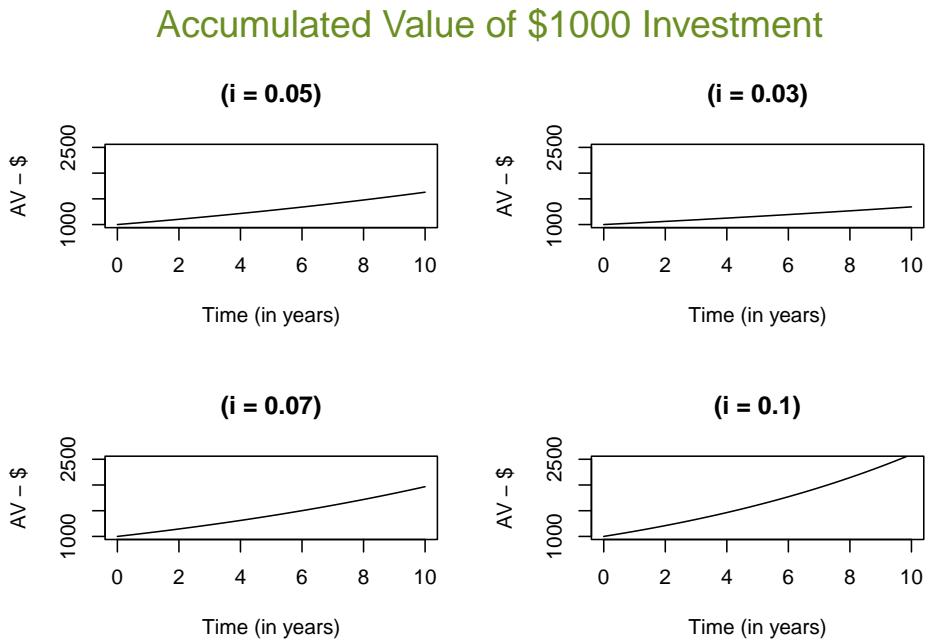
plot(t, A1, type = "l", xlab = "Time (in years)",
      ylab = "AV - $", ylim = c(1000, 2500),
      main = "(i = 0.05)")

plot(t, A2, type = "l", xlab = "Time (in years)",
      ylab = "AV - $", ylim = c(1000, 2500),
      main = "(i = 0.03)")

plot(t, A3, type = "l", xlab = "Time (in years)",
      ylab = "AV - $", ylim = c(1000, 2500),
      main = "(i = 0.07)")

plot(t, A4, type = "l", xlab = "Time (in years)",
      ylab = "AV - $", ylim = c(1000, 2500),
      main = "(i = 0.1)")

mtext("Accumulated Value of $1000 Investment",
      outer=TRUE, cex = 1.5, col="olivedrab")
```

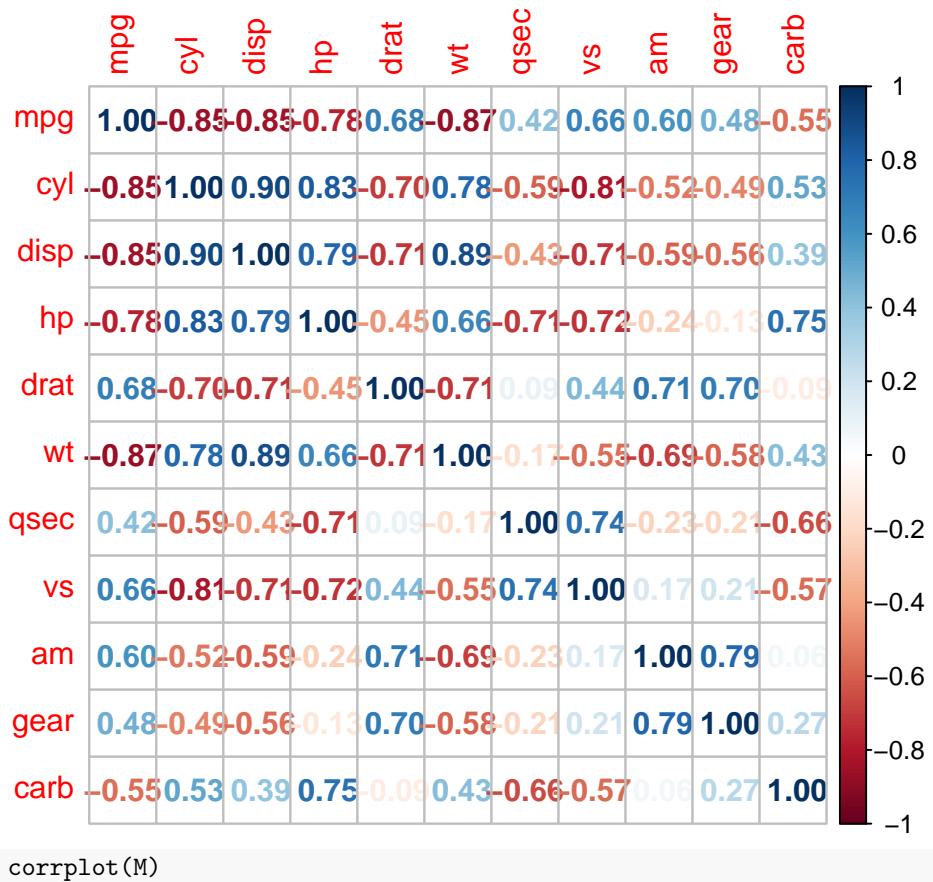


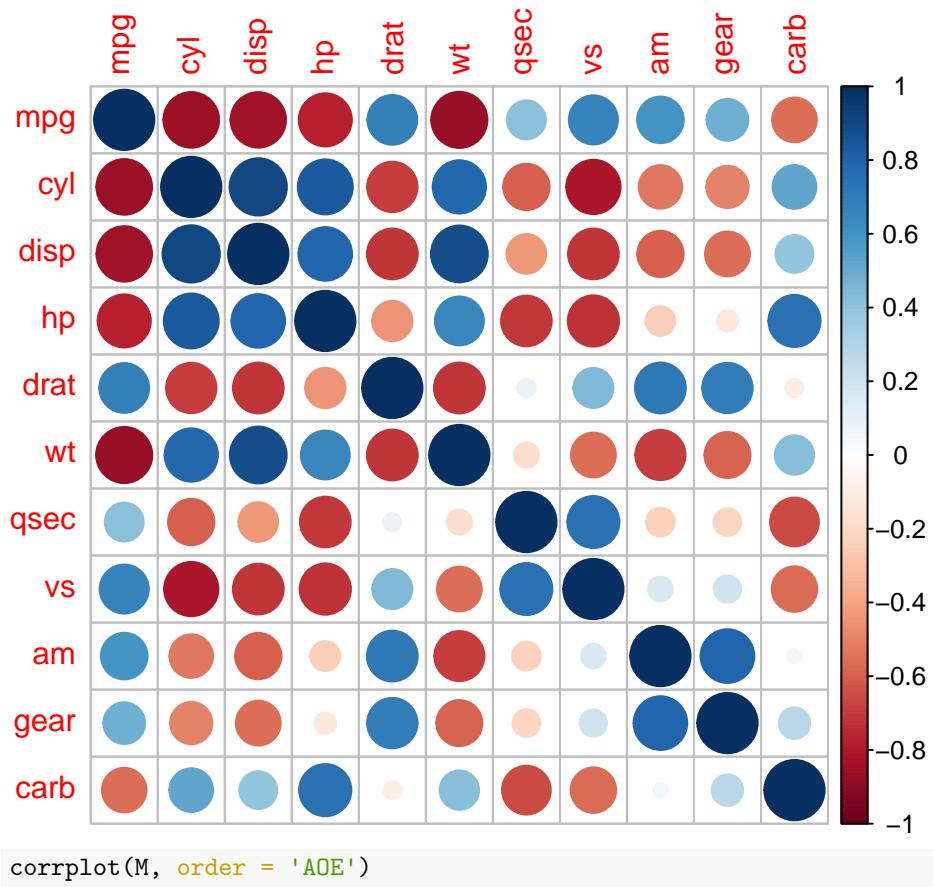
5.6 Pairwise relationship

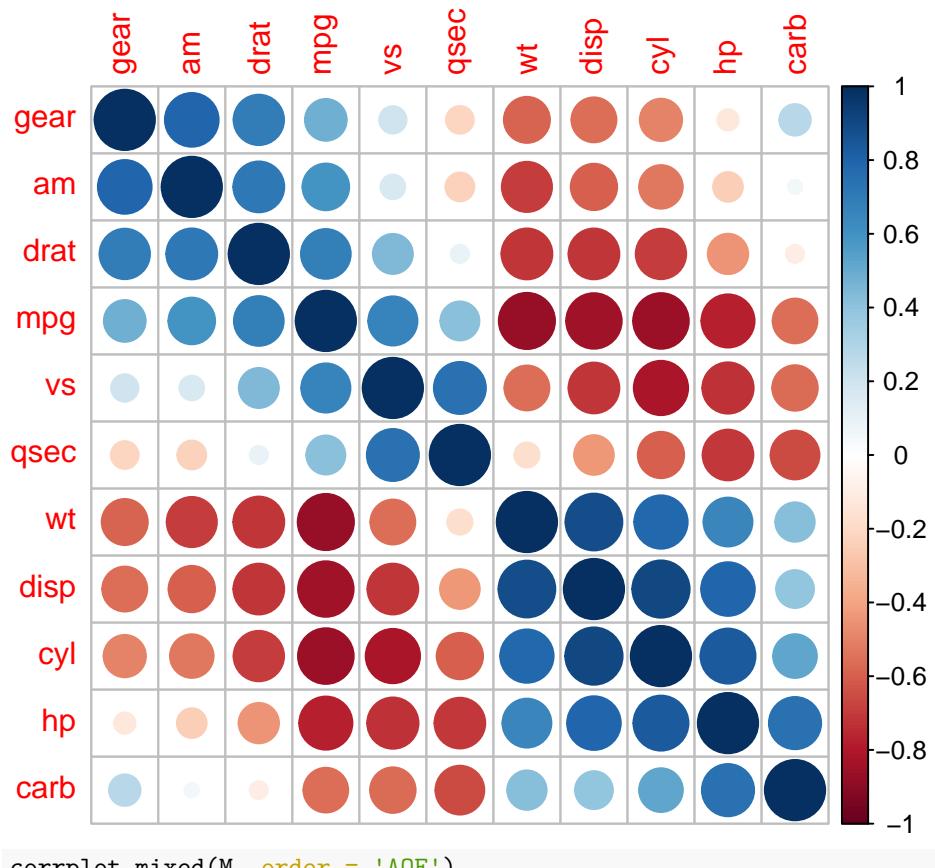
R package `corrplot` provides a visual exploratory tool on correlation matrix that supports automatic variable reordering to help detect hidden patterns among variables. See more details [here](#)

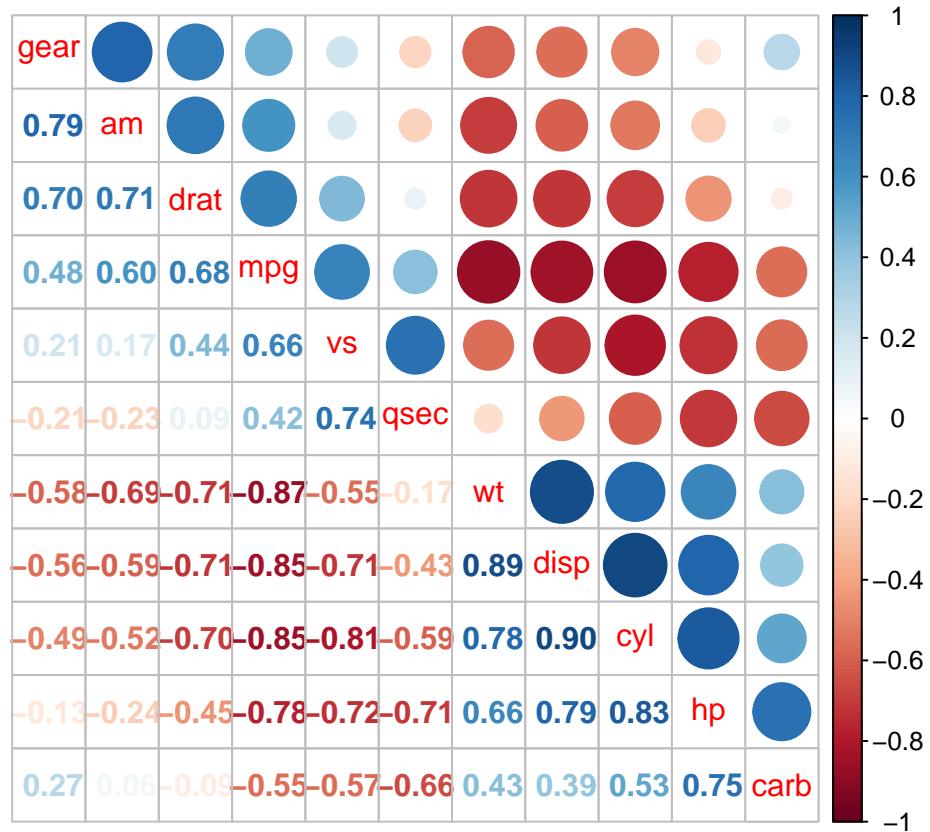
```
library(corrplot)
```

```
## corrplot 0.92 loaded
M = cor(mtcars)
corrplot(M, method = 'number') # colorful number
```









Chapter 6

Data Management

We will use the same data set, Ames Housing Price data, containing 165 observations and 12 features including the sale date and price. Suppose we want to find the houses that are sold in Jan 2010. You can use the function `filter()` in the `dplyr` package, a member of the tidyverse package.

See more about `dplyr` [here](#)

```
library(AmesHousing)
amesdata <- make_ames()
```

6.1 Filter

We can use subsetting operations

```
amesdata[amesdata$Year_Sold == 2010 & amesdata$Mo_Sold == 1, ]
```



```
## # A tibble: 26 x 81
##   MS_Sub~1 MS_Zo~2 Lot_F~3 Lot_A~4 Street Alley Lot_S~5 Land_~6 Utili~7 Lot_C~8
##   <fct>    <fct>    <dbl>   <int> <fct>   <fct>   <fct>   <fct>   <fct>
## 1 One_Sto~ Reside~     43     5005 Pave   No_A~ Slight~ HLS    AllPub  Inside
## 2 One_Sto~ Reside~     105    11751 Pave   No_A~ Slight~ Lvl    AllPub  Inside
## 3 Split_F~ Reside~     85    10625 Pave   No_A~ Regular Lvl   AllPub  Inside
## 4 Two_Sto~ Floati~      0     7500 Pave   No_A~ Regular Lvl   AllPub  Inside
## 5 Two_Sto~ Reside~     102    12858 Pave   No_A~ Slight~ Lvl   AllPub  Inside
## 6 One_Sto~ Reside~     100    18494 Pave   No_A~ Slight~ Lvl   AllPub  Corner
## 7 One_Sto~ Reside~     43     3203 Pave   No_A~ Regular Lvl   AllPub  Inside
## 8 Two_Sto~ Reside~     60     17433 Pave  No_A~ Modera~ Lvl   AllPub  CulDSac
## 9 Two_Sto~ Reside~     76     10142 Pave  No_A~ Slight~ Lvl   AllPub  Inside
## 10 Two_Sto~ Floati~     39     3515 Pave   Paved Regular Lvl  AllPub Inside
## # ... with 16 more rows, 71 more variables: Land_Slope <fct>,
```

```
## #  Neighborhood <fct>, Condition_1 <fct>, Condition_2 <fct>, Bldg_Type <fct>,
## #  House_Style <fct>, Overall_Qual <fct>, Overall_Cond <fct>,
## #  Year_Built <int>, Year_Remod_Add <int>, Roof_Style <fct>, Roof_Matl <fct>,
## #  Exterior_1st <fct>, Exterior_2nd <fct>, Mas_Vnr_Type <fct>,
## #  Mas_Vnr_Area <dbl>, Exter_Qual <fct>, Exter_Cond <fct>, Foundation <fct>,
## #  Bsmt_Qual <fct>, Bsmt_Cond <fct>, Bsmt_Exposure <fct>, ...
```

Or we can use `filter()`:

```
library(dplyr)

## 
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
## 
##     filter, lag

## The following objects are masked from 'package:base':
## 
##     intersect, setdiff, setequal, union

dplyr::filter(amesdata, Year_Sold == 2010, Mo_Sold == 1)

## # A tibble: 26 x 81
##   MS_Sub~1 MS_Zo~2 Lot_F~3 Lot_A~4 Street Alley Lot_S~5 Land_~6 Utili~7 Lot_C~8
##   <fct>     <fct>     <dbl>   <int> <fct>   <fct>   <fct>   <fct>   <fct>
## 1 One_Sto~ Reside~     43     5005  Pave   No_A~ Slight~ HLS   AllPub  Inside
## 2 One_Sto~ Reside~    105    11751  Pave   No_A~ Slight~ Lvl   AllPub  Inside
## 3 Split_F~ Reside~     85    10625  Pave   No_A~ Regular Lvl  AllPub  Inside
## 4 Two_Sto~ Floati~      0     7500  Pave   No_A~ Regular Lvl  AllPub  Inside
## 5 Two_Sto~ Reside~    102    12858  Pave   No_A~ Slight~ Lvl  AllPub  Inside
## 6 One_Sto~ Reside~    100    18494  Pave   No_A~ Slight~ Lvl  AllPub  Corner
## 7 One_Sto~ Reside~     43     3203  Pave   No_A~ Regular Lvl  AllPub  Inside
## 8 Two_Sto~ Reside~     60    17433  Pave   No_A~ Modera~ Lvl  AllPub  CulDSac
## 9 Two_Sto~ Reside~     76    10142  Pave   No_A~ Slight~ Lvl  AllPub  Inside
## 10 Two_Sto~ Floati~    39     3515  Pave   Paved  Regular Lvl AllPub  Inside
## # ... with 16 more rows, 71 more variables: Land_Slope <fct>,
## #  Neighborhood <fct>, Condition_1 <fct>, Condition_2 <fct>, Bldg_Type <fct>,
## #  House_Style <fct>, Overall_Qual <fct>, Overall_Cond <fct>,
## #  Year_Built <int>, Year_Remod_Add <int>, Roof_Style <fct>, Roof_Matl <fct>,
## #  Exterior_1st <fct>, Exterior_2nd <fct>, Mas_Vnr_Type <fct>,
## #  Mas_Vnr_Area <dbl>, Exter_Qual <fct>, Exter_Cond <fct>, Foundation <fct>,
## #  Bsmt_Qual <fct>, Bsmt_Cond <fct>, Bsmt_Exposure <fct>, ...
```

6.2 Arrange

Let's find the 10 houses with the highest sale prices by year

```

ar <- arrange(amesdata, Year_Sold, desc(Sale_Price))
ar

## # A tibble: 2,930 x 81
##   MS_Sub~1 MS_Zo~2 Lot_F~3 Lot_A~4 Street Alley Lot_S~5 Land_~6 Utili~7 Lot_C~8
##   <fct>   <fct>   <dbl>   <int> <fct>   <fct>   <fct>   <fct>   <fct>
## 1 Two_Sto~ Reside~    118    35760 Pave   No_A~ Slight~ Lvl    AllPub  CulDSac
## 2 Two_Sto~ Reside~    114    17242 Pave   No_A~ Slight~ Lvl    AllPub  Inside
## 3 Two_Sto~ Reside~     85    16056 Pave   No_A~ Slight~ Lvl    AllPub  Inside
## 4 Two_Sto~ Reside~     60    18062 Pave   No_A~ Slight~ HLS    AllPub  CulDSac
## 5 Two_Sto~ Reside~     82    16052 Pave   No_A~ Slight~ Lvl    AllPub  CulDSac
## 6 Two_and~ Reside~     90    22950 Pave   No_A~ Modera~ Lvl    AllPub  Inside
## 7 One_Sto~ Reside~    90    18261 Pave   No_A~ Slight~ HLS    AllPub  Inside
## 8 One_Sto~ Reside~    107   13891 Pave   No_A~ Regular Lvl   AllPub  Inside
## 9 Two_Sto~ Reside~     59    16023 Pave   No_A~ Slight~ HLS    AllPub  CulDSac
## 10 Two_Sto~ Reside~    66   13682 Pave   No_A~ Modera~ HLS   AllPub  CulDSac
## # ... with 2,920 more rows, 71 more variables: Land_Slope <fct>,
## #   Neighborhood <fct>, Condition_1 <fct>, Condition_2 <fct>, Bldg_Type <fct>,
## #   House_Style <fct>, Overall_Qual <fct>, Overall_Cond <fct>,
## #   Year_Built <int>, Year_Remod_Add <int>, Roof_Style <fct>, Roof_Matl <fct>,
## #   Exterior_1st <fct>, Exterior_2nd <fct>, Mas_Vnr_Type <fct>,
## #   Mas_Vnr_Area <dbl>, Exter_Qual <fct>, Exter_Cond <fct>, Foundation <fct>,
## #   Bsmt_Qual <fct>, Bsmt_Cond <fct>, Bsmt_Exposure <fct>, ...

```

Or

```

arr <- amesdata[order(amesdata$Year_Sold, desc(amesdata$Sale_Price)), ]
head(arr)

```

```

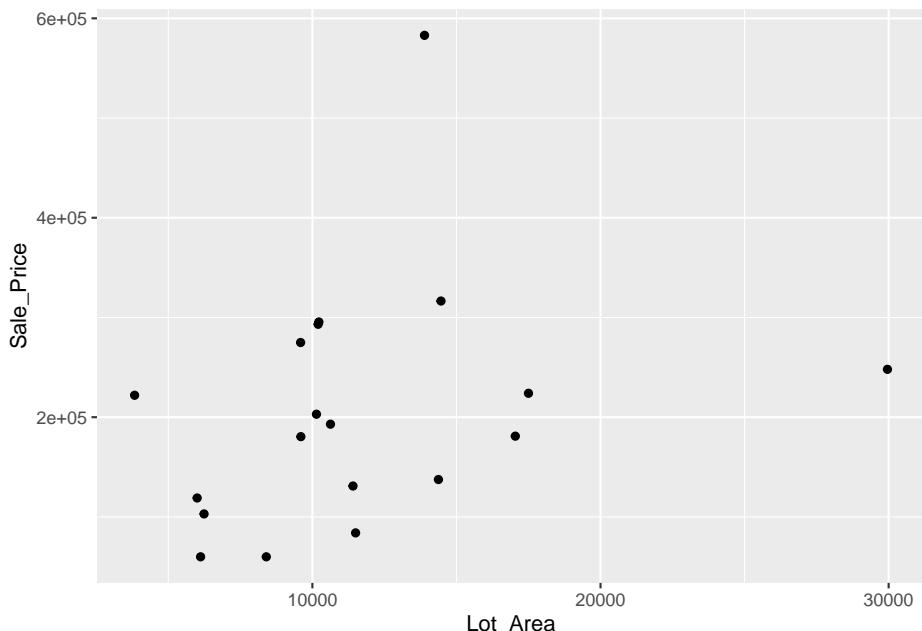
## # A tibble: 6 x 81
##   MS_SubC~1 MS_Zo~2 Lot_F~3 Lot_A~4 Street Alley Lot_S~5 Land_~6 Utili~7 Lot_C~8
##   <fct>   <fct>   <dbl>   <int> <fct>   <fct>   <fct>   <fct>   <fct>
## 1 Two_Stor~ Reside~    118    35760 Pave   No_A~ Slight~ Lvl    AllPub  CulDSac
## 2 Two_Stor~ Reside~    114    17242 Pave   No_A~ Slight~ Lvl    AllPub  Inside
## 3 Two_Stor~ Reside~     85    16056 Pave   No_A~ Slight~ Lvl    AllPub  Inside
## 4 Two_Stor~ Reside~     60    18062 Pave   No_A~ Slight~ HLS    AllPub  CulDSac
## 5 Two_Stor~ Reside~     82    16052 Pave   No_A~ Slight~ Lvl    AllPub  CulDSac
## 6 Two_and_~ Reside~     90    22950 Pave   No_A~ Modera~ Lvl    AllPub  Inside
## # ... with 71 more variables: Land_Slope <fct>, Neighborhood <fct>,
## #   Condition_1 <fct>, Condition_2 <fct>, Bldg_Type <fct>, House_Style <fct>,
## #   Overall_Qual <fct>, Overall_Cond <fct>, Year_Built <int>,
## #   Year_Remod_Add <int>, Roof_Style <fct>, Roof_Matl <fct>,
## #   Exterior_1st <fct>, Exterior_2nd <fct>, Mas_Vnr_Type <fct>,
## #   Mas_Vnr_Area <dbl>, Exter_Qual <fct>, Exter_Cond <fct>, Foundation <fct>,
## #   Bsmt_Qual <fct>, Bsmt_Cond <fct>, Bsmt_Exposure <fct>, ...

```

6.3 Pipe

Pipes help us apply multiple operations sequentially on a given data.

```
library(ggplot2)
amesdata %>%
  filter(Year_Sold == 2009, Mo_Sold == 1) %>%
  arrange(Year_Built) %>%
  ggplot(mapping = aes(x = Lot_Area, y = Sale_Price)) +
  geom_point()
```



6.4 Select

How do we select variables based on some characteristics

```
amesdata %>% select(starts_with("Year"), Sale_Price)

## # A tibble: 2,930 x 4
##   Year_Built Year_Remod_Add Year_Sold Sale_Price
##       <int>        <int>     <int>     <int>
## 1     1960        1960     2010    215000
## 2     1961        1961     2010    105000
## 3     1958        1958     2010    172000
## 4     1968        1968     2010    244000
## 5     1997        1998     2010    189900
## 6     1998        1998     2010    195500
```

```

## 7      2001      2001      2010    213500
## 8      1992      1992      2010    191500
## 9      1995      1996      2010    236500
## 10     1999      1999      2010    189000
## # ... with 2,920 more rows
amesdata %>% select(contains("Mo"))

## # A tibble: 2,930 x 2
##   Year_Remod_Add Mo_Sold
##   <int>      <int>
## 1 1960          5
## 2 1961          6
## 3 1958          6
## 4 1968          4
## 5 1998          3
## 6 1998          6
## 7 2001          4
## 8 1992          1
## 9 1996          3
## 10 1999          6
## # ... with 2,920 more rows

```

6.5 Create & group_by()

We may want to create new variables as functions of the existing ones by `mutate()`:

```

library(r02pro)
library(tidyverse)
amesdata %>%
  select(Overall_Qual, Lot_Area, Sale_Price) %>%
  mutate(ave_price = Sale_Price/Lot_Area)

## # A tibble: 2,930 x 4
##   Overall_Qual  Lot_Area Sale_Price ave_price
##   <fct>        <int>      <int>      <dbl>
## 1 Above_Average  31770     215000      6.77
## 2 Average        11622     105000      9.03
## 3 Above_Average  14267     172000     12.1
## 4 Good           11160     244000     21.9
## 5 Average        13830     189900     13.7
## 6 Above_Average  9978      195500     19.6
## 7 Very_Good      4920      213500     43.4
## 8 Very_Good      5005      191500     38.3
## 9 Very_Good      5389      236500     43.9
## 10 Good          7500      189000     25.2

```

```
## # ... with 2,920 more rows
```

Can we summarize by groups? First let's see an example for `summarize`:

```
amesdata %>%
  summarize(n_houses = n(),
            ave_liv_area = mean(Lot_Area),
            prob = c(0.25, 0.75),
            q_price = quantile(Sale_Price, c(0.25, 0.75),
                                na.rm = TRUE))

## # A tibble: 2 x 4
##   n_houses  ave_liv_area  prob  q_price
##       <int>        <dbl> <dbl>    <dbl>
## 1     2930        10148.  0.25  129500
## 2     2930        10148.  0.75  213500
```

Another

```
amesdata %>%
  group_by(Overall_Qual) %>%
  summarize(n_houses = n(),
            ave_liv_area = mean(Lot_Area),
            ave_price = mean(Sale_Price),
            na.rm = TRUE)

## # A tibble: 10 x 5
##   Overall_Qual  n_houses  ave_liv_area  ave_price na.rm
##   <fct>        <int>        <dbl>        <dbl> <lgl>
## 1 Very_Poor      4        15214.      48725  TRUE
## 2 Poor           13        9326.      52325  TRUE
## 3 Fair            40        9439.      83186  TRUE
## 4 Below_Average  226        8464.     106485  TRUE
## 5 Average         825        9996.     134753  TRUE
## 6 Above_Average  732        9788.     162130  TRUE
## 7 Good            602        10309.    205026  TRUE
## 8 Very_Good       350        10618.    270914  TRUE
## 9 Excellent        107        12777.    368337  TRUE
## 10 Very_Excellent  31        18071.    450217 TRUE
```

6.6 More tools

6.6.1 `subset()`

```
any(is.na(amesdata))
```

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

```

#Pay attention to subset(). This will be a time-saver
sub <- subset(amesdata, amesdata$Overall_Qual != "Fair")
dim(sub)

## [1] 2890 81
dim(amesdata)

## [1] 2930 81

#You can drop columns (variables) as well
amesless = subset(amesdata, select = c("Sale_Type", "Mo_Sold"))
head(amesless)

## # A tibble: 6 x 2
##   Sale_Type Mo_Sold
##   <fct>      <int>
## 1 "WD "        5
## 2 "WD "        6
## 3 "WD "        6
## 4 "WD "        4
## 5 "WD "        3
## 6 "WD "        6

```

However, look at the `help(subset)`: “This is a convenience function intended for use interactively. *For programming it is better to use the standard subsetting functions like `[]`, and in particular the non-standard evaluation of argument `subset` can have unanticipated consequences*”.

6.6.2 `within()` & `with()`

Here is an example to use `within()`:

```

ana <- within(amesdata, Sale_Price[Fence != 2] <- 0)

#which is a short cut of

amesdata$Sale_Price[amesdata$Fence != 2] <- 0

```

And `with()`

```

mean(with(amesdata, Sale_Price[Mo_Sold == 5 & Overall_Qual == "Good"]))
## [1] 0

```

6.6.3 `aggregate()`

The `aggregate()` function in R can be used to calculate summary statistics for a dataset.

```

# create data frame
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B', 'B'),
                  position=c('G', 'G', 'F', 'G', 'F', 'F'),
                  points=c(99, 90, 86, 88, 95, 99),
                  assists=c(33, 28, 31, 39, 34, 23),
                  rebounds=c(30, 28, 24, 24, 28, 33))

df

##   team position points assists rebounds
## 1   A         G    99     33      30
## 2   A         G    90     28      28
## 3   A         F    86     31      24
## 4   B         G    88     39      24
## 5   B         F    95     34      28
## 6   B         F    99     23      33

# find mean points by team
aggregate(df$points, by=list(df$team), FUN=mean)

##   Group.1      x
## 1       A 91.66667
## 2       B 94.00000

aggregate(df$points, by=list(df$team, df$position), FUN=mean)

##   Group.1 Group.2      x
## 1       A       F 86.0
## 2       B       F 97.0
## 3       A       G 94.5
## 4       B       G 88.0

```

We can also define our own functions

```

mine <- function(x){
  return(sort(x))
}

aggregate(df$points, by=list(df$assists), FUN=mine)

##   Group.1 x
## 1       23 99
## 2       28 90
## 3       31 86
## 4       33 99
## 5       34 95
## 6       39 88

```

Chapter 7

Programming basics

In this section we see three main applications: conditional flows, loops, and functions, that are main pillars of any type of programming.

7.1 Conditional flows

7.1.1 if/Else

The main syntax is as follows

```
if (condition) {  
  some R code  
} else {  
  more R code  
}
```

Here is a simple example:

```
x <- c("what", "is", "truth")  
  
if("Truth" %in% x) {  
  print("Truth is found")  
} else {  
  print("Truth is not found")  
}  
  
## [1] "Truth is not found"
```

How about this:

```
x <- c(1, 4, 4)  
a <- 3
```

```
#Here is a nice if/Else
if(length(x[x == a]) > 0) {
  print(paste("x has", length(x[x==a]), a))
} else {
  print(paste("x doesn't have any", a))
}

## [1] "x doesn't have any 3"

#Another one with piping
a <- 4
if(a %in% x) {
  print(paste("x has", length(x[x==a]), a))
} else {
  print(paste("x doesn't have any", a))
}

## [1] "x has 2 4"
```

7.1.2 Nested conditions

```
#Change the numbers to see all conditions
x <- 0
y <- 4
if (x == 0 & y != 0) {
  print("a number cannot be divided by zero")
} else if (x == 0 & y == 0) {
  print("a zero cannot be divided by zero")
} else {
  a <- y/x
  print(paste("y/x = ", a))
}
```

```
## [1] "a number cannot be divided by zero"
```

Building multiple conditions without else (it's a silly example!):

```
z <- 0
w <- 4
x <- 5
y <- 3
if(z > w) print("z is bigger than w")
if(w > z) print("w is bigger than z")

## [1] "w is bigger than z"
if(x > y) print("x is bigger than y")
```

```
## [1] "x is bigger than y"
if(y > x) print("y is bigger than x")
if(z > x) print("z is bigger than x")
if(x > z) print("x is bigger than z")

## [1] "x is bigger than z"
if(w > y) print("w is bigger than y")

## [1] "w is bigger than y"
if(y > w) print("y is bigger than w")
```

7.1.3 Simpler `ifelse`

A simpler, one-line `ifelse`:

```
#Change the numbers
x <- 0
y <- 4
ifelse (x > y, "x is bigger than y", "y is bigger than x")

## [1] "y is bigger than x"

#Better (ifelse will fail if x = y. Try it!)
ifelse (x == y, "x is the same as y",
       ifelse(x > y, "x is bigger than y", "y is bigger than x"))

## [1] "y is bigger than x"
```

A simpler, without else!

```
z <- 0
w <- 4
if(z > w) print("w is bigger than z")

#Change the numbers
x <- 5
y <- 3
if(x > y) print("x is bigger than y")

## [1] "x is bigger than y"

#See that both of them moves to the next line.
```

The `ifelse()` function only allows for one “if” statement, two cases. You could add nested “if” statements, but that’s just a pain, especially if the 3+ conditions you want to use are all on the same level, conceptually. Is there a way to specify multiple conditions at the same time?

```
#Let's create a data frame:
df <- data.frame("name"=c("Kaija", "Ella", "Andis"), "test1" = c(FALSE, TRUE, TRUE),
                  "test2" = c(FALSE, FALSE, TRUE))
df

##      name test1 test2
## 1  Kaija FALSE FALSE
## 2   Ella  TRUE FALSE
## 3 Andis  TRUE  TRUE
```

Suppose we want separate the people into three groups:

- People who passed both tests: Group A
- People who passed one test: Group B
- People who passed neither test: Group C

dplyr has a function for exactly this purpose: `case_when()`.

```
library(dplyr)
df <- df %>%
  mutate(group = case_when(test1 & test2 ~ "A", # both tests: group A
                           xor(test1, test2) ~ "B", # one test: group B
                           !test1 & !test2 ~ "C" # neither test: group C
  ))
df

##      name test1 test2 group
## 1  Kaija FALSE FALSE     C
## 2   Ella  TRUE FALSE     B
## 3 Andis  TRUE  TRUE     A
```

7.2 Loops

What would you do if you needed to execute a block of code multiple times? In general, statements are executed sequentially. A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most programming languages. There are 3 main loop types: `while()`, `for()`, `repeat()`.

Here are some examples for `for()` loop:

```
x <- c(3, -1, 4, 2, 10, 5)

for (i in 1:length(x)) {
  x[i] <- x[i] * 2
}
```

```
x
## [1] 6 -2 8 4 20 10
```

Note that this just for an example. If we want to multiply each element of a vector by 2, a loop isn't the best way. Although it is very normal in many programming languages, we would simply use a vectorized operation in R.

```
x <- c(3, -1, 4, 2, 10, 5)
x <- x * 2
x
```

```
## [1] 6 -2 8 4 20 10
```

7.2.1 Conditional loops

But sometimes it would be very handy. If the element in $x > 3$, multiply it with the subsequent element:

```
x <- c(3, -1, 0, 2, 10, 5)

x_new <- c() #empty container
for (i in 1:(length(x)-1)) {
  ifelse(x[i] > 3, x_new[i] <- x[i] * x[i + 1], x_new[i] <- 0)
}

x
## [1] 3 -1 0 2 10 5
x_new
```

```
## [1] 0 0 0 0 50
```

Inside the `if` and `else` clause, you can use `next` and `break` to further control the flow. The `next` function goes directly to the next loop cycle, while `break` jumped out of the current loop.

```
x <- c(9, -1, 0, 5, -7, 16, 22)
zn <- c()

for(i in 1:length(x)){
  if(x[i] < 0){
    next
  }
  zn <- c(zn, sqrt(x[i]))
}

zn
## [1] 3.000000 0.000000 2.236068 4.000000 4.690416
```

Inside the `if` and `else` clause, you can use `next` and `break` to further control the flow. The `next` function goes directly to the next loop cycle, while `break` jumped out of the current loop.

```
x <- c(9, 1, 0, 5, 7, 16, 22)
bn <- c()

for(i in 1:length(x)){
  if(x[i] > 10){
    break
  }
  bn <- c(bn,  sqrt(x[i]))
}

bn

## [1] 3.000000 1.000000 0.000000 2.236068 2.645751
```

7.2.2 `while()` and `repeat()`

Here are some examples for `while()` loop:

```
# Let's use our first example

x <- 3
cnt <- 1

while (cnt < 11) {
  x = x * 2
  cnt = cnt + 1
}
x

## [1] 3072
```

Here are some examples for `repeat()` loop:

```
# Let's use our first example

x <- 3
cnt <- 1

repeat {
  x = x * 2
  cnt = cnt + 1

  if(cnt > 10) break
}
x
```

```
## [1] 3072
```

7.2.3 Nested loops

It is also common to put one loop inside another one. Let's say we want to create a 5x5 matrix where each element $A_{ij} = i + j$

```
A <- matrix(0, 5, 5) #initialize the matrix A

for (i in 1:5)      #loop over index i
  for (j in 1:5){    #loop over index j
    A[i, j] <- i + j #set the (i, j)-th element of A
  }
A

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

7.2.4 outer()

`outer()` takes two vectors and a function (that itself takes two arguments) and builds a **matrix** by calling the given function for each combination of the elements in the two vectors.

```
x <- c(0, 1, 2)
y <- c(0, 1, 2, 3, 4)

m <- outer (
  y,      # First dimension: the columns (y)
  x,      # Second dimension: the rows      (x)
  function (x, y)  x+2*y
)

m

##      [,1] [,2] [,3]
## [1,]    0    2    4
## [2,]    1    3    5
## [3,]    2    4    6
## [4,]    3    5    7
## [5,]    4    6    8
```

In place of the function, an operator can be given, which makes it easy to create a matrix with simple calculations (such as multiplying):

```
m <- outer(c(10, 20, 30, 40), c(2, 4, 6), "*")
m

##      [,1] [,2] [,3]
## [1,]    20   40   60
## [2,]    40   80  120
## [3,]    60  120  180
## [4,]    80  160  240
```

It becomes very handy when we build a polynomial model:

```
x <- sample(0:20, 10, replace = TRUE)
x
```

```
## [1] 2 8 11 19 4 12 4 19 9 7
```

```
m <- outer(x, 1:4, "^")
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2     4     8    16
## [2,]    8    64   512  4096
## [3,]   11   121  1331 14641
## [4,]   19   361  6859 130321
## [5,]    4    16    64    256
## [6,]   12   144  1728  20736
## [7,]    4    16    64    256
## [8,]   19   361  6859 130321
## [9,]    9    81   729   6561
## [10,]   7    49   343   2401
```

We can also use `outer()` for this example

```
outer(1:5, 1:5, "+")
```

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

Or

```
outer(1:4, 1:4, function(i, j){0.5^{|abs(i-j)|}})
```

```
##      [,1] [,2] [,3] [,4]
## [1,] 1.000 0.50 0.25 0.125
## [2,] 0.500 1.00 0.50 0.250
## [3,] 0.250 0.50 1.00 0.500
```

```
## [4,] 0.125 0.25 0.50 1.000
```

7.3 The apply() family

The `apply()` family is one of the R base packages and is populated with functions to manipulate slices of data from matrices, arrays, lists and data frames in a repetitive way. These functions allow crossing the data in a number of ways and avoid explicit use of loop constructs. They act on an input list, matrix or array and apply a named function with one or several optional arguments. The family is made up of the `apply()`, `lapply()`, `sapply()`, `vapply()`, `mapply()`, `rapply()`, and `tapply()` functions.

7.3.1 `apply()`

The R base manual tells you that it's called as follows: `apply(X, MARGIN, FUN, ...)`, where, X is an array or a matrix if the dimension of the array is 2; MARGIN is a variable defining how the function is applied: when `MARGIN=1`, it applies over rows, whereas with `MARGIN=2`, it works over columns. Note that when you use the construct `MARGIN=c(1,2)`, it applies to both rows and columns; and FUN, which is the function that you want to apply to the data. It can be any R function, including a User Defined Function (UDF).

```
# Construct a 5x6 matrix
X <- matrix(rnorm(30), nrow=5, ncol=6)

# Sum the values of each column with `apply()`
apply(X, 2, sum)

## [1] -1.4530773  0.3055281 -1.2713307 -1.7703674  0.7358674  0.7410579
apply(X, 2, length)

## [1] 5 5 5 5 5 5
apply(X, 1, length)

## [1] 6 6 6 6 6
apply(X, 2, function (x) length(x)-1)

## [1] 4 4 4 4 4 4
#If you don't want to write a function inside of the arguments
len <- function(x){
  length(x)-1
}
apply(X, 2, len)

## [1] 4 4 4 4 4 4
```

```
#It can also be used to repeat a function on cells within a matrix
X_new <- apply(X[1:2,], 1, function(x) x+1)
X_new

##          [,1]      [,2]
## [1,]  2.5439088 -2.1169675
## [2,]  0.8758525  2.2703538
## [3,]  0.6992624  0.6585331
## [4,] -0.7017633  1.3695946
## [5,]  0.5572144  2.3454425
## [6,]  2.0342029  0.6654779
```

Since `apply()` is used only for matrices, if you apply `apply()` to a data frame, it first coerces your `data.frame` to an array which means all the columns must have the same type. Depending on your context, this could have unintended consequences. For a safer practice in data frames, we can use `lapply()` and `sapply()`:

7.3.2 `lapply()`

You want to apply a given function to every element of a list and obtain a list as a result. When you execute `?lapply`, you see that the syntax looks like the `apply()` function. The difference is that it can be used for other objects like data frames, lists or vectors. And the output returned is a list (which explains the “l” in the function name), which has the same number of elements as the object passed to it. `lapply()` function does not need MARGIN.

```
A<-c(1:9)
B<-c(1:12)
C<-c(1:15)
my.lst<-list(A,B,C)
lapply(my.lst, sum)

## [[1]]
## [1] 45
##
## [[2]]
## [1] 78
##
## [[3]]
## [1] 120
```

7.3.3 `sapply()`

`sapply` works just like `lapply`, but will simplify the output if possible. This means that instead of returning a list like `lapply`, it will return a vector instead if the data is simplifiable.

```
A<-c(1:9)
B<-c(1:12)
C<-c(1:15)
my.lst<-list(A,B,C)
sapply(my.lst, sum)
```

```
## [1] 45 78 120
```

7.3.4 tapply()

Sometimes you may want to perform the apply function on some data, but have it separated by factor. In that case, you should use `tapply`. Let's take a look at the information for `tapply`.

```
X <- matrix(c(1:10, 11:20, 21:30), nrow = 10, ncol = 3)
tdata <- as.data.frame(cbind(c(1,1,1,1,1,2,2,2,2,2), X))
tdata
```

```
##      V1 V2 V3 V4
## 1     1 11 21
## 2     1 12 22
## 3     1 13 23
## 4     1 14 24
## 5     1 15 25
## 6     2 16 26
## 7     2 17 27
## 8     2 18 28
## 9     2 19 29
## 10    2 20 30
```

```
tapply(tdata$V2, tdata$V1, mean)
```

```
## 1 2
## 3 8
```

What we have here is an important tool: We have a conditional mean of column 2 (V2) with respect to column 1 (V1). You can use `tapply` to do some quick summary statistics on a variable split by condition.

```
summary <- tapply(tdata$V2, tdata$V1, function(x) c(mean(x), sd(x)))
summary
```

```
## $`1`
## [1] 3.000000 1.581139
##
## $`2`
## [1] 8.000000 1.581139
```

7.3.5 mapply()

`mapply()` would be used to create a new variable. For example, using dataset `tdata`, we could divide one column by another column to create a new value. This would be useful for creating a ratio of two variables as shown in the example below.

```
tdata$V5 <- mapply(function(x, y) x/y, tdata$V2, tdata$V4)
tdata$V5

## [1] 0.04761905 0.09090909 0.13043478 0.16666667 0.20000000 0.23076923
## [7] 0.25925926 0.28571429 0.31034483 0.33333333
```

7.4 Functions

An R function is created by using the keyword `function`. Let's write our first function:

```
first <- function(a){
  b <- a ^ 2
  return(b)
}

first(1675)

## [1] 2805625
```

Let's have a function that find the z-score (standardization). That's subtracting the sample mean, and dividing by the sample standard deviation.

$$\frac{x - \bar{x}}{s}$$

```
z_score <- function(x){
  return((x - mean(x))/sd(x))
}

set.seed(1)
x <- rnorm(10, 3, 30)
z <- z_score(x)
z

## [1] -0.97190653  0.06589991 -1.23987805  1.87433300  0.25276523 -1.22045645
## [7]  0.45507643  0.77649606  0.56826358 -0.56059319
```

Lets create a function that prints the factorials:

```
fact <- function(a){
  b <- 1
```

```

for (i in 1:(a-1)) {
  b <- b*(i+1)
}
b
}

fact(5)

## [1] 120

```

Creating loops is an act of art and requires very careful thinking. The same loop can be done by many different structures. And it always takes more time to understand somebody else's loop than your own!

7.5 `source()`

You can use the `source()` function in R to reuse functions that you create in another R script. The function uses the following basic syntax:

```
source("path/to/some/file.R")
```

Suppose we have the following R script called `some_functions.R` that contains two simple user-defined functions:

```

divide_by_two <- function(x) { return(x/2) }

multiply_by_three <- function(x) { return(x*3) }

```

Now suppose we're currently working with some R script called `main_script.R`. Assuming `some_functions.R` and `main_script.R` are located within the same folder, we can use `source` at the top of our `main_script.R` to allow us to use the functions we defined in the `some_functions.R` script:

```

source("some_functions.R")

df <- data.frame(team=c('A', 'B', 'C', 'D', 'E', 'F'),
                  points=c(14, 19, 22, 15, 30, 40))

df$half_points <- divide_by_two(df$points)

df$triple_points <- multiply_by_three(df$points)
df

##   team points half_points triple_points
## 1   A      14       7.0       42
## 2   B      19       9.5       57
## 3   C      22      11.0       66
## 4   D      15       7.5       45
## 5   E      30      15.0       90

```

```
## 6      F      40      20.0      120
```

We can use as many source functions as we'd like if we want to reuse functions defined in several different scripts.

Chapter 8

Simulation in R

In this chapter, we will learn how to simulate data and illustrate their use in several examples. More specifically we'll cover the following subjects:

1. Sampling in R: `sample()`,
2. Random number generating with probability distributions,
3. Simulation for statistical inference,
4. Creating data with a DGM,
5. Bootstrapping,
6. Power of simulation - A fun example.

Why would we want to simulate data? Why not just use real data? Because with real data, we don't know what the right answer is. Suppose we use real data and we apply a method to extract information, how do we know that we applied the method correctly? Now suppose we create artificial data by simulating a "Data Generating Model". Since we can know the correct answer, we can check whether or not our methods work to extract the information we wish to have. If our method is correct, than we can apply it to real data.

8.1 Sampling in R: `sample()`

Let's play with `sample()` for simple random sampling. We will see the arguments of `sample()` function.

```
sample(c("H", "T"), size = 8, replace = TRUE) # fair coin
## [1] "H" "T" "H" "T" "T" "T" "H" "T"
sample(1:6, size = 2, replace = TRUE, prob=c(3,3,3,4,4,4))
## [1] 5 4
```

```
#let's do it again
sample(c("H","T"), size = 8, replace = TRUE) # fair coin

## [1] "H" "H" "H" "T" "T" "T" "T" "H"
sample(1:6, size = 2, replace = TRUE, prob=c(3,3,3,4,4,4))

## [1] 1 5
```

The results are different. If we use `set.seed()` then we can get the same results each time. Lets try now:

```
set.seed(123)
sample(c("H","T"), size = 8, replace = TRUE) # fair coin

## [1] "H" "H" "H" "T" "H" "T" "T" "T"
sample(1:6, size = 2, replace = TRUE, prob=c(3,3,3,4,4,4))

## [1] 4 4
#let's do it again

set.seed(123)
sample(c("H","T"), size = 8, replace = TRUE) # fair coin

## [1] "H" "H" "H" "T" "H" "T" "T" "T"
sample(1:6, size = 2, replace = TRUE, prob=c(3,3,3,4,4,4))

## [1] 4 4
```

We use `replace=TRUE` to override the default sample without replacement. This means the same thing can get selected from the population multiple times. And, `prob=` to sample elements with different probabilities, e.g. over sample based on some factor. The `set.seed()` function allow you to make a reproducible set of random numbers. Let's see the difference.

```
x <- 1:12
# a random permutation

set.seed(123)
sample(x)

## [1] 3 12 10 2 6 11 5 4 9 8 1 7
# This shuffles the numbers
set.seed(123)
sample(x, replace = TRUE)

## [1] 3 3 10 2 6 11 5 4 6 9 10 11
```

```
# This shuffles the numbers and replacing them
```

More...

```
# sample()'s surprise -- example
x <- 1:10
  sample(x[x > 3]) # length 2
```

```
## [1] 8 6 9 4 7 10 5
sample(x[x > 9]) # oops -- length 10! So this doesn't work
```

```
## [1] 3 8 2 7 9 1 6 10 4 5
sample(x[x > 10]) # length 0
```

```
## integer(0)
```

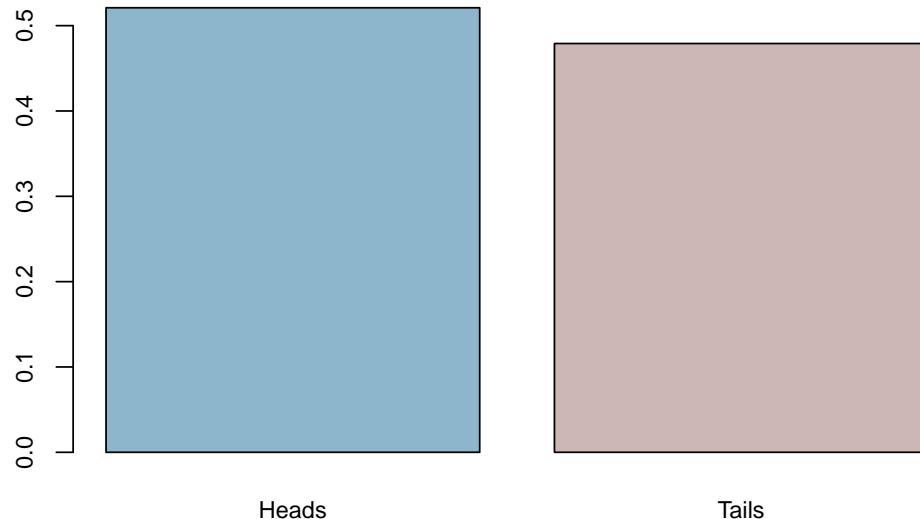
Here is an example: let's generate 501 coin flips. In the true model, this should generate heads half of the time, and tails half of the time.

```
set.seed(123)
coins <- sample(c("Heads", "Tails"), 501, replace = TRUE)
```

Now let's take that data as given and analyze it in our standard way! The proportion of heads is:

```
mean(coins == 'Heads')
```

```
## [1] 0.5209581
barplot(prop.table(table(coins)),
  col = c("lightskyblue3", "mistyrose3"),
  cex.axis = 0.8, cex.names = 0.8)
```



So what's our conclusion? We came to the conclusion that the true model generates heads 0.493014 of the time. But it is NOT 0.50, so pretty close, but not exact. Did this whole thing work or not? What if it always errs on the same side? In other words, what if it's always bias towards *heads* in every sample with 501 flips? We will do our first simulation to answer it momentarily.

One more useful application:

```
sample(letters, 10, replace = TRUE)
##  [1] "p" "z" "o" "s" "c" "n" "a" "x" "a" "p"
```

8.2 Random number generating with probability distributions

Here are the common probability distributions in R. Search help in R for more detail.

```
beta(shape1, shape2, ncp),
binom(size, prob),
chisq(df, ncp),
exp(rate),
gamma(shape, scale),
logis(location, scale),
norm(mean, sd),
pois(lambda),
t(df, ncp),
unif(min, max),
```

`dnorm(x,)` returns the density or the value on the y-axis of a probability distri-

bution for a discrete value of x,

`pnorm(q,)` returns the cumulative density function (CDF) or the area under the curve to the left of an x value on a probability distribution curve, `qnorm(p,)` returns the quantile value, i.e. the standardized z value for x, `rnorm(n,)` returns a random simulation of size n

```

rnorm(6) # 6 std nrml distribution values

## [1] -0.2645952 -0.9472983  0.7395213  0.8967787 -0.3460009 -1.7820571

rnorm(10, mean = 50, sd = 19) # set parameters

## [1] 58.83389 12.93042 40.19385 59.29253 67.13847 62.16690 68.07297 38.61666
## [9] 24.71680 38.74801

runif(n = 10, min = 0, max = 1) #uniform distribution

## [1] 0.96415257 0.08146656 0.85436475 0.80223822 0.38517360 0.32759740
## [7] 0.20493870 0.56938266 0.88805519 0.52971409

rpois(n = 10, lambda = 15) # Poisson distribution

## [1] 15 15  6 17 16 13 15 15 15 21
# toss coin 8 times using binomial distribution
rbinom(n = 8, size = 1, p = 0.5)

## [1] 0 0 1 0 1 1 1 0
rbinom(8,1,.5) # args correct order

## [1] 1 0 0 1 1 1 0
# 18 trials, sample size 10, prob success =.2
rbinom(18, 10, 0.2)

## [1] 5 1 0 4 2 1 4 0 1 3 1 1 1 3 1 3 1 1

```

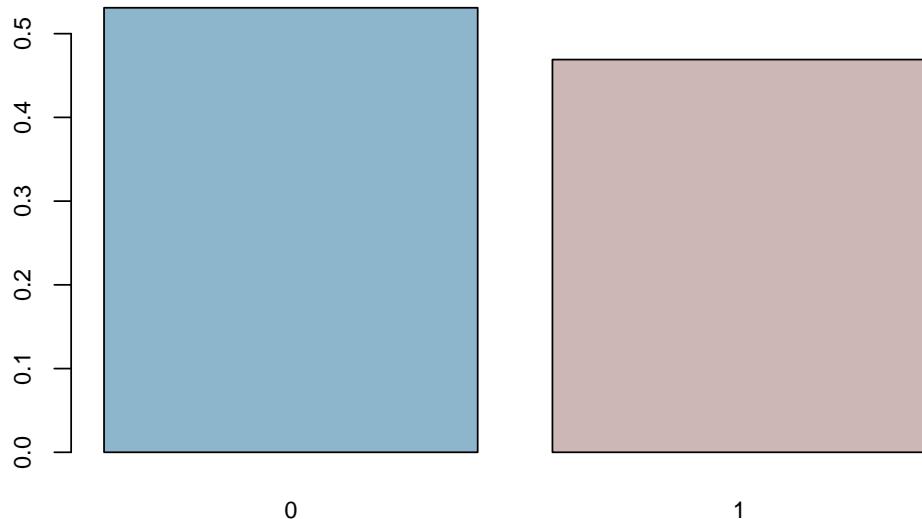
Can we replicate our coin-flip example here with probability distributions? Yes, we can!

```

set.seed(123)
coins <- rbinom(n = 501, size = 1, p = 0.5)
mean(coins==0)

## [1] 0.5309381
barplot(prop.table(table(coins)),
        col = c("lightskyblue3","mistyrose3"),
        cex.axis = 0.8, cex.names = 0.8)

```



8.3 Simulation for statistical inference

Let's predict number of girls in 400 births, where probability of female birth is 48.8%

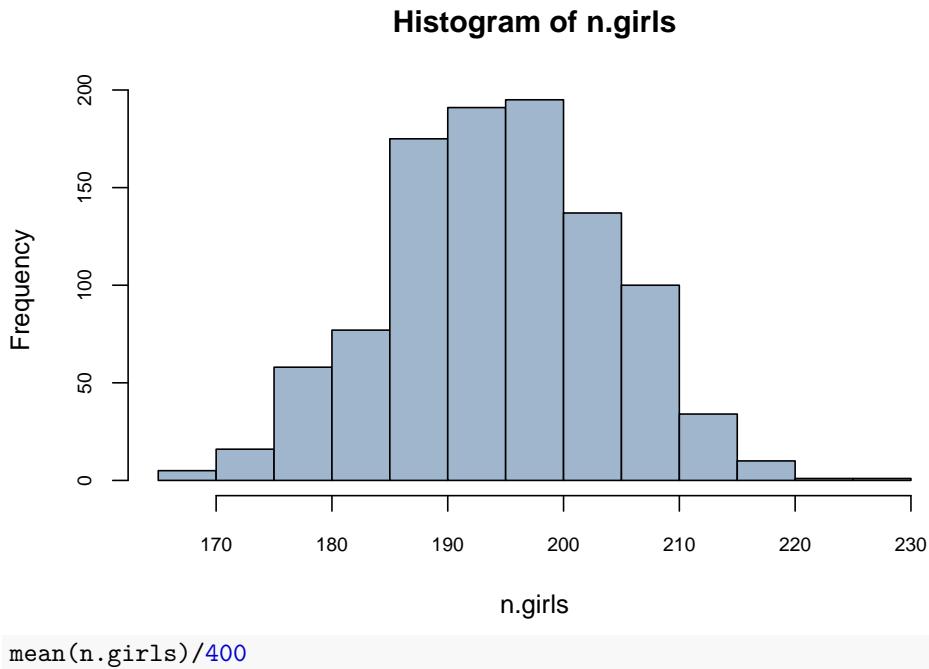
```
n.girls <- rbinom(1, 400, 0.488)
n.girls
```

```
## [1] 201
n.girls/400
```

```
## [1] 0.5025
```

Now, to get distribution of the simulations, repeat the simulation many times.

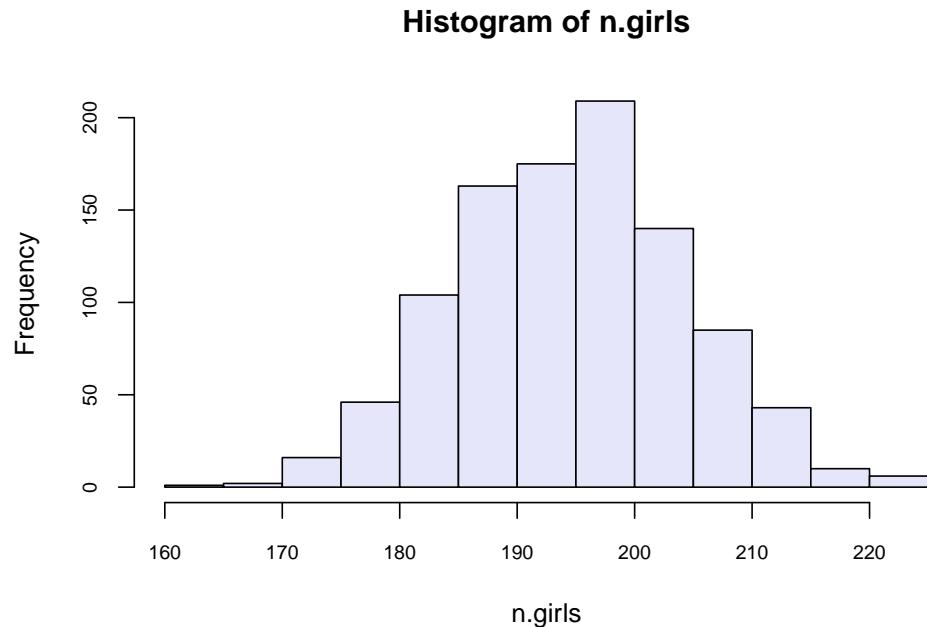
```
n.sims <- 1000
n.girls <- rbinom(n.sims, 400, .488)
hist(n.girls, col = "slategray3", cex.axis = 0.75)
```



```
## [1] 0.4872775
```

This is called as *sampling distribution*. Can we do same thing with a loop?

```
n.sims <- 1000
n.girls <- rep(NA, n.sims) # create vector to store simulations
for (i in 1:n.sims){
  n.girls[i] <- rbinom(1, 400, 0.488)
}
hist(n.girls, col = "lavender", cex.axis = 0.75)
```

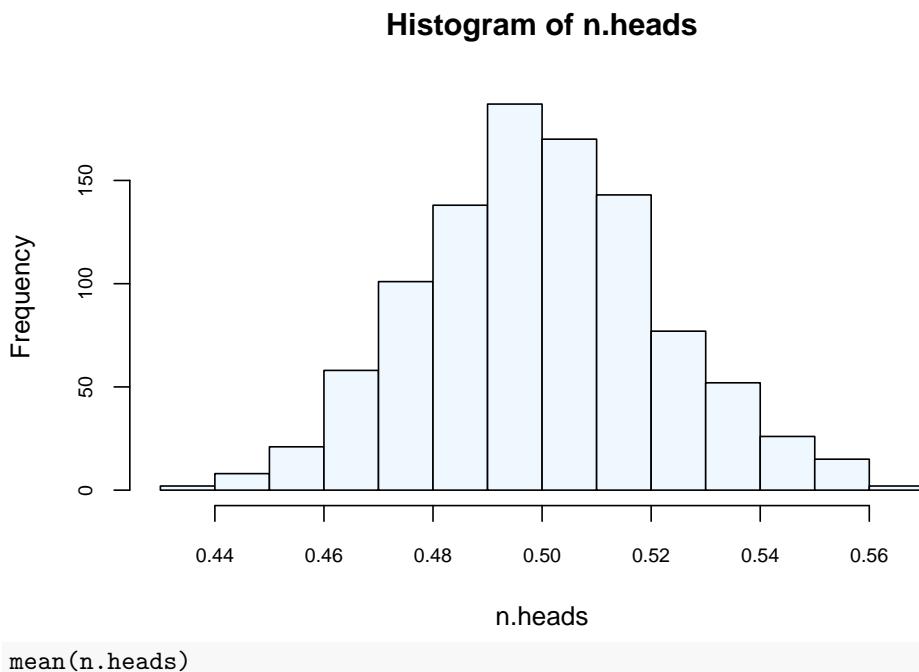


Let's apply a similar simulation to our coin flipping.

```

n.sims <- 1000
n.heads <- rep(NA, n.sims) # create vector to store simulations
for (i in 1:n.sims){
  n.heads[i] <- mean(rbinom(n = 501, size = 1, p = 0.5))
}
hist(n.heads, col="aliceblue", cex.axis = 0.75)

```

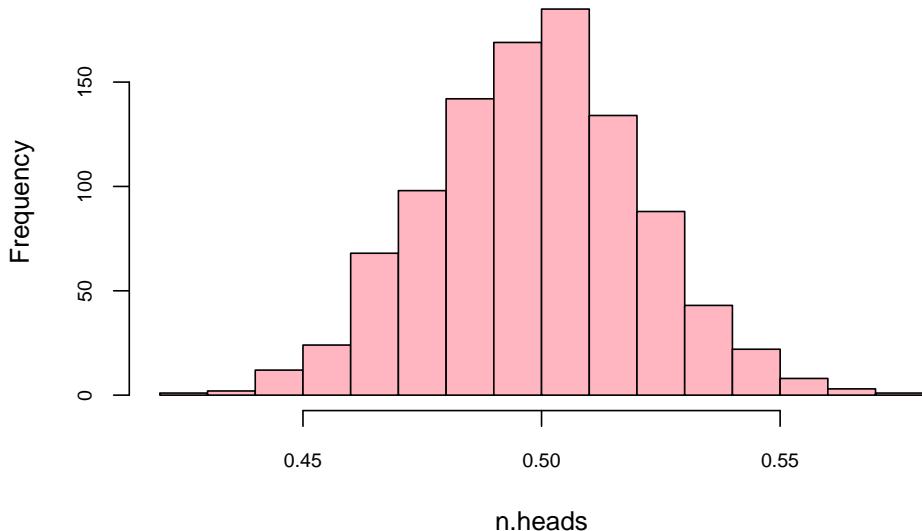


```
## [1] 0.4997705
```

Here is another way for the same simulation:

```
n.heads <- replicate(1000, mean(rbinom(n = 501, size = 1, p = 0.5)))
hist(n.heads, col="lightpink", cex.axis = 0.75)
```

Histogram of n.heads



```
mean(n.heads)
```

```
## [1] 0.4987265
```

What's the 95% confidence interval for the mean?

```
sd <- sd(n.heads)
CI95 <- c(-2*sd+mean(n.heads), 2*sd+mean(n.heads))
CI95
```

```
## [1] 0.4538446 0.5436085
```

What happens if we use a “wrong” estimator for the mean, like `sum(heads)/300`?

```
n.sims <- 1000
n.heads <- rep(NA, n.sims) # create vector to store simulations
for (i in 1:n.sims){
  n.heads[i] <- sum(rbinom(n = 501, size = 1, p = 0.5))/300
}
mean(n.heads)
```

```
## [1] 0.83496
```

Because we are working with a simulation, identifying that the result from this incorrect estimator is wrong becomes easy.

8.4 Creating data with a Data Generating Model (DGM)

One of the major tasks of statistics is to obtain information about populations. In most of cases, the population is unknown and the only thing that is known for the researcher is a finite subset of observations drawn from the population. The main aim of the statistical analysis is to obtain information about the population through analysis of the sample. Since very little information is known about the population characteristics, one has to establish some assumptions about the behavior of this unknown population. For example, for a regression analysis, we can state that the whole population regression function (PRF) is a linear function of the different values of X . One important issue related to the PRF is the error term (u_i) in the regression equation. For a pair of realizations (x_i, y_i) from the random variables (X, Y) , we can write the following equalities:

$$y_i = E(Y|X = x_i) + u_i = \alpha + \beta x_i + u_i$$

and

$$E(u|X = x_i) = 0$$

This result implies that for $X = x_i$, the divergences of all values of Y with respect to the conditional expectation $E(Y|X = x_i)$ are averaged out. There are several reasons for the existence of the error term in the regression: (1) the error term is taking into account variables which are not in the model; (2) we do not have great confidence about the correctness of the model; and (3) we do not know if there are measurement errors in the variables.

In a regression analysis, the PRF is a Data Generating Model for y_i , which is unknown to us. Because it is unknown, we must try to learn about it from a sample since that is the only available data for us. If we assume that there is a specific PRF that generates the data, then given any estimator of α and β , namely $\hat{\beta}$ and $\hat{\alpha}$, we can estimate them from our sample with the sample regression function (SRF):

$$\hat{y}_i = \hat{\alpha} + \hat{\beta} x_i, \quad i = 1, \dots, n$$

The relationship between the PRF and SRF is:

$$y_i = \hat{y}_i + \hat{u}_i, \quad i = 1, \dots, n$$

where \hat{u}_i is denoted the residuals from SRF.

With a data generating process (DGP) at hand, it is possible to create new simulated data. With α , β and the vector of exogenous variables X (fixed), a

sample of size n can be used to obtain N values of Y with random variable u . This yields one complete **population** of size N . Note that this artificially generated set of data could be viewed as an example of real-world data that a researcher would be faced with when dealing with the kind of estimation problem this model represents. Note especially that the set of data obtained depends crucially on the particular set of error terms drawn. A different set of error terms would create a different data set of Y for the same problem.

With the artificial data we generated, DGM is now known and the whole population is accessible. That is, we can test many models on different samples drawn from this population in order to see whether their inferential properties are in line with DGM. We'll have several examples below.

Here is our DGM:

$$Y_i = \beta_1 + \beta_2 X_{2i} + \beta_3 X_{3i} + \beta_4 X_{2i} X_{3i} + \beta_5 X_{5i},$$

with the following coefficient vector: $\beta = (12, -0.7, 34, -0.17, 5.4)$. Moreover x_2 is binary variable with values of 0 and 1 and x_5 and x_3 are highly correlated with $\rho = 0.65$. When we add the error term, u , which is independently and identically (i.i.d) distributed with $N(0, 1)$, we can get the whole *population* of 10,000 observations. DGM plus the error term is called the data generating process (DGP)

```
library(MASS)
N <- 10000
x_2 <- sample(c(0,1), N, replace = TRUE) #Dummy variable

#mvrnorm() creates a matrix of correlated variables
X_corr <- mvrnorm(N, mu = c(0,0), Sigma = matrix(c(1,0.65,0.65,1), ncol = 2),
                   empirical = TRUE)

#We can check their correlation
cor(X_corr)

##      [,1] [,2]
## [1,] 1.00 0.65
## [2,] 0.65 1.00

#Each column is one of our variables
x_3 <- X_corr[,1]
x_5 <- X_corr[,2]

#interaction
x_23 <- x_2*x_3

# Now DGM
beta <- c(12, -0.7, 34, -0.17, 5.4)
```

```

dgm <- beta[1] + beta[2]*x_2 + beta[3]*x_3 + beta[4]*x_23 + beta[5]*x_5

#And our Yi
y <- dgm + rnorm(N,0,1)
pop <- data.frame(y, x_2, x_3, x_23, x_5)
str(pop)

## 'data.frame': 10000 obs. of 5 variables:
## $ y : num -37.09 8.41 12.84 44.55 31.87 ...
## $ x_2 : num 0 0 1 1 0 1 1 1 0 0 ...
## $ x_3 : num -1.3163 -0.1002 0.0558 0.7737 0.6297 ...
## $ x_23: num 0 0 0.0558 0.7737 0 ...
## $ x_5 : num -0.6134 -0.0465 -0.0857 1.5022 -0.3612 ...

#Here is new thing/trick to learn
#for better looking tables install.packages("stargazer")
library(stargazer)
stargazer(pop, type = "text", title = "Descriptive Statistics",
          digits = 1, out = "table1.text")

## 
## Descriptive Statistics
## =====
## Statistic N Mean St. Dev. Min Max
## -----
## y 10,000 11.7 37.7 -168.8 164.3
## x_2 10,000 0.5 0.5 0 1
## x_3 10,000 0.0 1.0 -4.7 3.9
## x_23 10,000 0.004 0.7 -4.7 3.9
## x_5 10,000 -0.0 1.0 -4.4 3.8
## 

#The table will be saved in the working directory
#with whatever name you write in the out option.
#You can open this file with any word processor

```

Now we are going to sample this population and run a SRF.

```

library(stargazer)

n <- 500 #sample size
ind <- sample(nrow(pop), n, replace = FALSE)
sample <- pop[ind, ]
str(sample)

## 'data.frame': 500 obs. of 5 variables:
## $ y : num 10.7 44.7 -47.1 -30.6 24 ...
## $ x_2 : num 0 1 0 0 1 0 1 1 1 0 ...

```

```

##  $ x_3 : num  -0.111 0.705 -1.62 -1.153 0.358 ...
##  $ x_23: num  0 0.705 0 0 0.358 ...
##  $ x_5 : num  0.627 1.528 -0.889 -0.722 0.281 ...
model <- lm(y ~ ., data = sample)
stargazer(model, type = "text", title = "G O O D - M O D E L",
           dep.var.labels = "Y",
           digits = 3)

##
## G O O D - M O D E L
## =====
##             Dependent variable:
## -----
##                               Y
## -----
## x_2                  -0.718***  

##                               (0.087)
## 
## x_3                  34.025***  

##                               (0.068)
## 
## x_23                 -0.162*  

##                               (0.085)
## 
## x_5                  5.357***  

##                               (0.058)
## 
## Constant             12.075***  

##                               (0.060)
## 
## -----
## Observations          500
## R2                   0.999
## Adjusted R2          0.999
## Residual Std. Error  0.974 (df = 495)
## F Statistic          196,907.500*** (df = 4; 495)
## =====
## Note:                *p<0.1; **p<0.05; ***p<0.01

```

As you can see the coefficients are very close to our “true” coefficients specified in DGM. Now we can test what happens if we omit x_5 in our SRF and estimate it.

```

library(stargazer)

n <- 500 #sample size

```

```

sample <- pop[sample(nrow(pop), n, replace = FALSE), ]
str(sample)

## 'data.frame': 500 obs. of 5 variables:
## $ y : num -21.42 35.02 -2.79 -12.15 -10.37 ...
## $ x_2 : num 0 1 1 0 0 0 1 0 1 ...
## $ x_3 : num -0.828 0.65 -0.387 -0.907 -0.423 ...
## $ x_23: num 0 0.65 -0.387 0 0 ...
## $ x_5 : num -1.035 0.194 -0.298 1.311 -1.467 ...

model_bad <- lm(y ~ x_2 + x_3 + x_23, data = sample)
stargazer(model_bad, type = "text", title = "B A D - M O D E L",
           dep.var.labels = "Y",
           digits = 3)

## 
## B A D - M O D E L
## =====
##                               Dependent variable:
##                               -----
##                               Y
## -----
## x_2                      -0.717*
##                           (0.379)
## 
## x_3                      37.534*** 
##                           (0.275)
## 
## x_23                     -0.581
##                           (0.406)
## 
## Constant                  11.861*** 
##                           (0.266)
## 
## -----
## Observations               500
## R2                        0.986
## Adjusted R2                0.986
## Residual Std. Error      4.231 (df = 496)
## F Statistic                11,326.660*** (df = 3; 496)
## =====
## Note:                      *p<0.1; **p<0.05; ***p<0.01

```

Now it seems that none of the coefficients are as good as before, except for the intercept. This is a so-called omitted variable bias (OVB) problem, also known as a model underfitting or specification error. Would be the case that that this

is a problem for only one sample? We can simulate the results many times and see whether **on average** $\hat{\beta}_3$ is biased or not.

```
n.sims <- 500
n <- 500 #sample size
beta_3 <- c(NA, n.sims)
for (i in 1:n.sims){
  sample <- pop[sample(nrow(pop), n, replace = FALSE), ]
  model_bad <- lm(y ~ x_2 + x_3 + x_23, data = sample)
  beta_3[i] <- model_bad$coefficients["x_3"]
}
summary(beta_3)

##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  36.56   37.33   37.49   37.50   37.68   38.19
```

As we can see the OVB problem is not a problem in one sample. We withdrew a sample and estimated the same underfitting model 500 times with a simulation. Therefore we collected 500 $\hat{\beta}_3$. The average is 37.47. If we do the same simulation with a model that is correctly specified, you can see the results: the average of 500 $\hat{\beta}_3$ is 34, which is the “correct” true coefficient in our DGM.

```
n.sims <- 500
n <- 500 #sample size
beta_3 <- c(NA, n.sims)
for (i in 1:n.sims){
  sample <- pop[sample(nrow(pop), n, replace = FALSE), ]
  model_good <- lm(y ~ x_2 + x_3 + x_23 + x_5, data = sample)
  beta_3[i] <- model_good$coefficients["x_3"]
}
summary(beta_3)

##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  33.81   33.96   34.02   34.01   34.06   34.23
```

8.5 Bootstrapping

Bootstrapping is the process of resampling with replacement (all values in the sample have an equal probability of being selected, including multiple times, so a value could have duplicates). Resample, calculate a statistic (e.g. the mean), repeat this hundreds or thousands of times and you are able to estimate a precise/accurate uncertainty of the mean (confidence interval) of the data’s distribution. There are less assumptions about the underlying distribution using bootstrap compared to calculating the standard error directly.

Generally bootstrapping follows the same basic steps:

- Resample a given data set a specified number of times,

- Calculate a specific statistic from each sample,
- Find the standard deviation of the distribution of that statistic.

In the following bootstrapping example we would like to obtain a standard error for the estimate of the mean. We will be using the `lapply()`, `sapply()` functions in combination with the `sample` function. (see <https://stats.idre.ucla.edu/r/library/r-library-introduction-to-bootstrapping/>)[?]

```
#creating the data set by taking 100 observations
#from a normal distribution with mean 5 and stdev 3

set.seed(123)
data <- rnorm(100, 5, 3) #rounding each observation to nearest integer
data[1:10]

## [1] 3.318573 4.309468 9.676125 5.211525 5.387863 10.145195 6.382749
## [8] 1.204816 2.939441 3.663014

#obtaining 20 bootstrap samples and storing in a list
resamples <- lapply(1:20, function(i) sample(data, replace = T))
#display the first of the bootstrap samples
resamples[1]

## [[1]]
## [1] 8.76144476 3.11628177 4.02220524 10.36073941 6.30554447 9.10580685
## [7] 2.93597415 3.60003394 3.58162578 6.34462934 5.71619521 7.06592076
## [13] 4.91435973 4.34607526 7.33989536 4.37624817 5.37156273 6.93312965
## [19] 8.67224539 4.32268704 1.20481630 1.63067425 4.33854031 5.91058592
## [25] 4.14568098 1.63067425 11.15025406 -1.92750663 11.50686790 4.11478555
## [31] 7.06592076 8.62388599 5.33204815 10.36073941 8.29051704 7.68537698
## [37] 3.85858700 3.85858700 3.66301409 4.02220524 -1.92750663 6.15584120
## [43] 2.93944144 6.38274862 6.38274862 6.75384125 6.13891845 2.87239771
## [49] 2.81332631 4.00037785 9.10580685 1.92198666 -0.06007993 7.68537698
## [55] 0.35374159 1.58558919 3.66301409 4.87138863 9.10580685 4.14568098
## [61] 8.67224539 3.12488220 4.91435973 -0.06007993 6.38274862 3.12488220
## [67] 8.29051704 8.44642286 4.11478555 6.93312965 2.81332631 0.35374159
## [73] 8.01721557 1.92198666 5.33204815 10.14519496 7.98051157 3.31857306
## [79] 8.44642286 6.75384125 5.01729256 1.58558919 -0.06007993 7.51336113
## [85] 4.33854031 6.38274862 5.64782471 -0.06007993 2.91587906 6.93312965
## [91] 10.14519496 3.11628177 6.27939266 5.71619521 6.49355143 1.94427385
## [97] 6.66175296 0.35374159 3.58162578 7.46474324
```

Here is another way to do the same thing:

```
set.seed(123)
data <- rnorm(100, 5, 3)
resamples_2 <- matrix(NA, nrow = 100, ncol = 20)
```

```

for (i in 1:20) {
  resamples_2[,i] <- sample(data, 100, replace = TRUE)
}
str(resamples_2)

##  num [1:100, 1:20] 8.76 3.12 4.02 10.36 6.31 ...
##display the first of the bootstrap samples
resamples_2[, 1]

## [1] 8.76144476 3.11628177 4.02220524 10.36073941 6.30554447 9.10580685
## [7] 2.93597415 3.60003394 3.58162578 6.34462934 5.71619521 7.06592076
## [13] 4.91435973 4.34607526 7.33989536 4.37624817 5.37156273 6.93312965
## [19] 8.67224539 4.32268704 1.20481630 1.63067425 4.33854031 5.91058592
## [25] 4.14568098 1.63067425 11.15025406 -1.92750663 11.50686790 4.11478555
## [31] 7.06592076 8.62388599 5.33204815 10.36073941 8.29051704 7.68537698
## [37] 3.85858700 3.85858700 3.66301409 4.02220524 -1.92750663 6.15584120
## [43] 2.93944144 6.38274862 6.38274862 6.75384125 6.13891845 2.87239771
## [49] 2.81332631 4.00037785 9.10580685 1.92198666 -0.06007993 7.68537698
## [55] 0.35374159 1.58558919 3.66301409 4.87138863 9.10580685 4.14568098
## [61] 8.67224539 3.12488220 4.91435973 -0.06007993 6.38274862 3.12488220
## [67] 8.29051704 8.44642286 4.11478555 6.93312965 2.81332631 0.35374159
## [73] 8.01721557 1.92198666 5.33204815 10.14519496 7.98051157 3.31857306
## [79] 8.44642286 6.75384125 5.01729256 1.58558919 -0.06007993 7.51336113
## [85] 4.33854031 6.38274862 5.64782471 -0.06007993 2.91587906 6.93312965
## [91] 10.14519496 3.11628177 6.27939266 5.71619521 6.49355143 1.94427385
## [97] 6.66175296 0.35374159 3.58162578 7.46474324

```

Calculating the mean for each bootstrap sample:

```
colMeans(resamples_2)
```

```

## [1] 5.095470 5.611315 5.283893 4.930731 4.804722 5.187125 4.946582 4.952693
## [9] 5.470162 5.058354 4.790996 5.357154 5.479364 5.366046 5.454458 5.474732
## [17] 5.566421 5.229395 5.111966 5.262666
##and the mean of all means
mean(colMeans(resamples_2))

```

```
## [1] 5.221712
```

Calculating the standard deviation of the distribution of means:

```
sqrt(var(colMeans(resamples_2)))
```

```
## [1] 0.2523254
```

8.6 Monty Hall - Fun example

The Monty Hall problem is a brain teaser, in the form of a probability puzzle, loosely based on the American television game show Let's Make a Deal and named after its original host, Monty Hall. The problem was originally posed (and solved) in a letter by Steve Selvin to the American Statistician in 1975 (Selvin 1975a), (Selvin 1975b). It became famous as a question from a reader's letter quoted in Marilyn vos Savant's "Ask Marilyn" column in Parade magazine in 1990:

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?

Vos Savant's response was that the contestant should switch to the other door (vos Savant 1990a). Under the standard assumptions, contestants who switch have a 2/3 chance of winning the car, while contestants who stick to their initial choice have only a 1/3 chance.

Many readers of vos Savant's column refused to believe switching is beneficial despite her explanation. After the problem appeared in Parade, approximately 10,000 readers, **including nearly 1,000 with PhDs**, wrote to the magazine, most of them claiming vos Savant was wrong. Even when given explanations, simulations, and formal mathematical proofs, many people still do not accept that switching is the best strategy. **Paul Erdős, one of the most prolific mathematicians in history, remained unconvinced until he was shown a computer simulation demonstrating the predicted result.**

The given probabilities depend on specific assumptions about how the host and contestant choose their doors. A key insight is that, under these standard conditions, there is more information about doors 2 and 3 that was not available at the beginning of the game, when door 1 was chosen by the player: the host's deliberate action adds value to the door he did not choose to eliminate, but not to the one chosen by the contestant originally. Another insight is that switching doors is a different action than choosing between the two remaining doors at random, as the first action uses the previous information and the latter does not. Other possible behaviors than the one described can reveal different additional information, or none at all, and yield different probabilities.

Here is the simple Bayes rule: $Pr(A|B) = Pr(B|A)Pr(A)/Pr(B)$.

Let's play it: The player picks Door 1, Monty Hall opens Door 3. My question is this:

$$Pr(CAR = 1|Open = 3) < Pr(CAR = 2|Open = 3)?$$

If this is true the player should always switch. Here is the Bayesian answer:

$$\begin{aligned} Pr(Car = 1|Open = 3) &= Pr(Open = 3|Car = 1)Pr(Car = 1)/Pr(Open = 3) \\ &= 1/2 \times (1/3) / (1/2) = 1/3 \end{aligned}$$

Let's see each number. Given that the player picks Door 1, if the car is behind Door 1, Monty should be indifferent between opening Doors 2 and 3. So the first term is 1/2. The second term is easy: Probability that the car is behind Door 1 is 1/3. The third term is also simple and usually overlooked. This is not a conditional probability. If the car were behind Door 2, the probability that Monty opens Door 3 would be 1. And this explains why the second option is different, below:

$$\begin{aligned} Pr(Car = 2|Open = 3) &= Pr(Open = 3|Car = 2)Pr(Car = 2)/Pr(Open = 3) \\ &= 1 \times (1/3) / (1/2) = 2/3 \end{aligned}$$



Image taken from http://media.graytvinc.com/images/690*388/mon+tyhall.jpg

Simulation to prove it

Step 1: Decide the number of plays

```
n <- 100000
```

Step 2: Define all possible door combinations

3 doors, the first one has the car. All possible outcomes for the game:

```
outcomes <- c(123,132,213,231,312,321)
```

Step 3: Create empty containers where you store the outcomes from each game

```
car <- rep(0, n)
goat1 <- rep(0, n)
goat2 <- rep(0, n)
choice <- rep(0, n)
monty <- rep(0, n)
winner <- rep(0, n)
```

Step 4: Loop

```
for (i in 1:n){
  doors <- sample(outcomes,1) #The game's door combination
  car[i] <- substring(doors, first = c(1,2,3), last = c(1,2,3))[1] #the right door
  goat1[i] <- substring(doors, first = c(1,2,3), last = c(1,2,3))[2] #The first wrong door
  goat2[i] <- substring(doors, first = c(1,2,3), last = c(1,2,3))[3] #The second wrong door

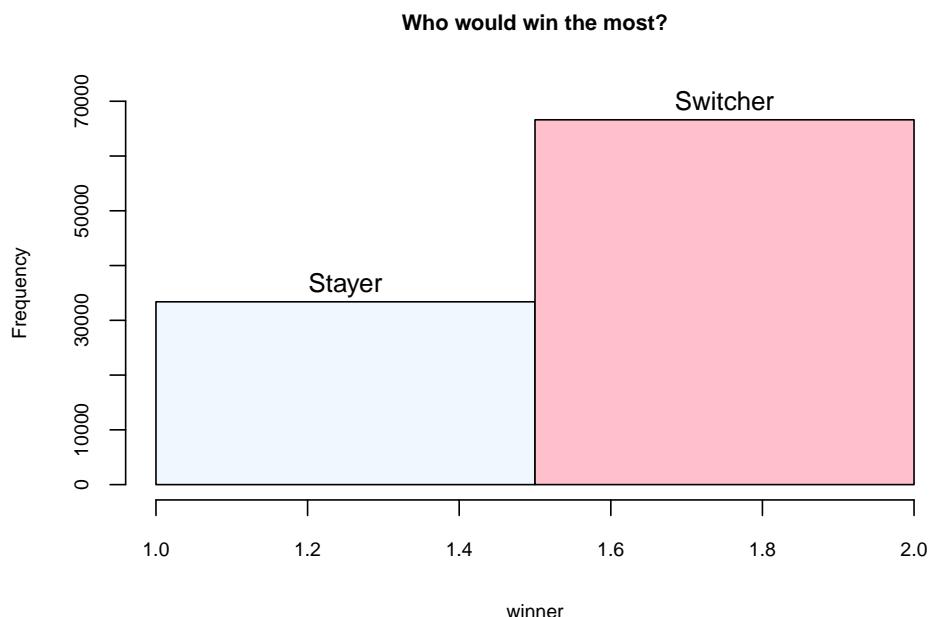
  #Person selects a random door
  choice[i] <- sample(1:3,1)

  #Now Monty opens a door
  if (choice[i] == car[i])
    {monty[i] = sample(c(goat1[i],goat2[i]),1)}
  else if (choice[i] == goat1[i])
    {monty[i] = goat2[i]}
  else
    {monty[i] = goat1[i]}

  # 1 represents the stayer who remains by her initial choice
  # 2 represents the switcher who changes her initial choice
  if (choice[i] == car[i]) {winner[i] = 1} else {winner[i] = 2}
}
```

Step 5: Chart

```
hist(winner, breaks = 2, main = "Who would win the most?",
      ylim = c(0,70000), labels = c("Stayer", "Switcher"),
      col = c("aliceblue", "pink"),
      cex.axis = 0.75, cex.lab = 0.75, cex.main = 0.85)
```



The simulation is inspired by <https://theressomethingaboutr.wordpress.com/2019/02/12/in-memory-of-monty-hall/> [?]

Chapter 9

Parts

You can add parts to organize one or more book chapters together. Parts can be inserted at the top of an .Rmd file, before the first-level chapter heading in that same file.

Add a numbered part: `# (PART) Act one {-} (followed by # A chapter)`

Add an unnumbered part: `# (PART*) Act one {-} (followed by # A chapter)`

Add an appendix as a special kind of un-numbered part: `# (APPENDIX) Other stuff {-} (followed by # A chapter)`. Chapters in an appendix are prepended with letters instead of numbers.

Chapter 10

Footnotes and citations

10.1 Footnotes

Footnotes are put inside the square brackets after a caret `^[]`. Like this one ¹.

10.2 Citations

Reference items in your bibliography file(s) using `@key`.

For example, we are using the **bookdown** package [Xie, 2022] (check out the last code chunk in index.Rmd to see how this citation key was added) in this sample book, which was built on top of R Markdown and **knitr** [Xie, 2015] (this citation was added manually in an external file book.bib). Note that the `.bib` files need to be listed in the index.Rmd with the YAML `bibliography` key.

The RStudio Visual Markdown Editor can also make it easier to insert citations:
<https://rstudio.github.io/visual-markdown-editing/#/citations>

¹This is a footnote.

Chapter 11

Blocks

11.1 Equations

Here is an equation.

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k} \quad (11.1)$$

You may refer to using `\@ref(eq:binom)`, like see Equation (11.1).

11.2 Theorems and proofs

Labeled theorems can be referenced in text using `\@ref(thm:tri)`, for example, check out this smart theorem 11.1.

Theorem 11.1. *For a right triangle, if c denotes the length of the hypotenuse and a and b denote the lengths of the **other** two sides, we have*

$$a^2 + b^2 = c^2$$

Read more here <https://bookdown.org/yihui/bookdown/markdown-extensions-by-bookdown.html>.

11.3 Callout blocks

The R Markdown Cookbook provides more help on how to use custom blocks to design your own callouts: <https://bookdown.org/yihui/rmarkdown-cookbook/custom-blocks.html>

Chapter 12

Sharing your book

12.1 Publishing

HTML books can be published online, see: <https://bookdown.org/yihui/bookdown/publishing.html>

12.2 404 pages

By default, users will be directed to a 404 page if they try to access a webpage that cannot be found. If you'd like to customize your 404 page instead of using the default, you may add either a `_404.Rmd` or `_404.md` file to your project root and use code and/or Markdown syntax.

12.3 Metadata for sharing

Bookdown HTML books will provide HTML metadata for social sharing on platforms like Twitter, Facebook, and LinkedIn, using information you provide in the `index.Rmd` YAML. To setup, set the `url` for your book and the path to your `cover-image` file. Your book's `title` and `description` are also used.

This `gitbook` uses the same social sharing data across all chapters in your book—all links shared will look the same.

Specify your book's source repository on GitHub using the `edit` key under the configuration options in the `_output.yml` file, which allows users to suggest an edit by linking to a chapter's source file.

Read more about the features of this output format here:

<https://pkgs.rstudio.com/bookdown/reference/gitbook.html>

Or use:

```
?bookdown::gitbook
```

Chapter 13

Cross-references

Cross-references make it easier for your readers to find and link to elements in your book.

13.1 Chapters and sub-chapters

There are two steps to cross-reference any heading:

1. Label the heading: `# Hello world {#nice-label}`.
 - Leave the label off if you like the automated heading generated based on your heading title: for example, `# Hello world = # Hello world {#hello-world}`.
 - To label an un-numbered heading, use: `# Hello world {-#nice-label}` or `{# Hello world .unnumbered}`.
2. Next, reference the labeled heading anywhere in the text using `\@ref(nice-label)`; for example, please see Chapter 13.
 - If you prefer text as the link instead of a numbered reference use: `any text you want can go here`.

13.2 Captioned figures and tables

Figures and tables *with captions* can also be cross-referenced from elsewhere in your book using `\@ref(fig:chunk-label)` and `\@ref(tab:chunk-label)`, respectively.

See Figure 13.1.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Don't miss Table 13.1.

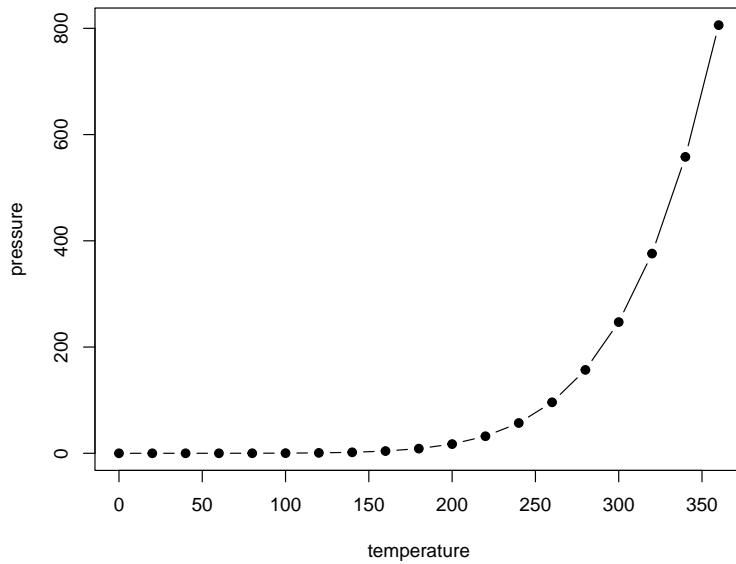


Figure 13.1: Here is a nice figure!

```
knitr::kable(  
  head(pressure, 10), caption = 'Here is a nice table!',  
  booktabs = TRUE  
)
```

Table 13.1: Here is a nice table!

temperature	pressure
0	0.0002
20	0.0012
40	0.0060
60	0.0300
80	0.0900
100	0.2700
120	0.7500
140	1.8500
160	4.2000
180	8.8000

Bibliography

Yihui Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. URL <http://yihui.org/knitr/>. ISBN 978-1498716963.

Yihui Xie. *bookdown: Authoring Books and Technical Documents with R Markdown*, 2022. URL <https://CRAN.R-project.org/package=bookdown>. R package version 0.28.