Analyzing Data with R – An Introduction

Lecture Notes

Prof. Dr. Stephan Huber
October 20, 2022

HOCHSCHULE
FRESENIUS
UNIVERSITY OF APPLIED SCIENCES

This script aims to support my lecture at the HS Fresenius. It is incomplete and no substitute for taking actively part in class. I am thankful for comments and suggestions for improvement; Tel.: +49 221 973199-523; stephan.huber@hs-fresenius.de.

Contact

Prof. Dr. Stephan Huber
Hochschule Fresenius für Wirtschaft & Medien GmbH
Im MediaPark 4c
50670 Cologne

Office: 4b OG-1 Bü01
www.hs-fresenius.de

+49 221 973199-523
stephan.huber@hs-fresenius.de
www.t1p.de/stephanhuber

Contents

Preface

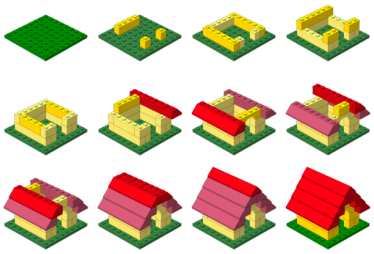
- Questions, comments, suggestions are welcome.
- You can contact (at every time) me in the lecture or via
  - phone (+49 221 973199-523),
  - e-mail (`Stephan.Huber@hs-fresenius.de`), or
  - in office (4b OG-1 Bü01).
- News and files can be found on ILIAS.
- Two necessary conditions for your success:
  1. Believe that you can learn R.
  2. Believe that being able to do R can make a difference for your future life.

Learning objectives

The students who have successfully completed the module are able to...

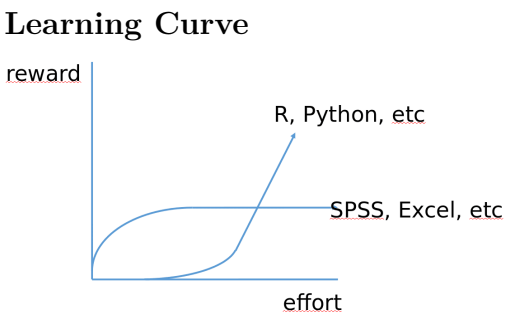
- ...install, code and use R Programming Language in RStudio to perform basic tasks on vectors, matrices and data frames.
- ...load a R script file, run lines from it, edit and save the script file. Set a working directory.
- ...install R packages.
- ...type own R code to import, manipulate, visualize, and analyze data.
- ...identify books, websites, and additional sources for further learning and help.

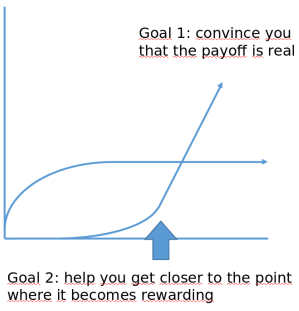
R is Like Building a House



Source:

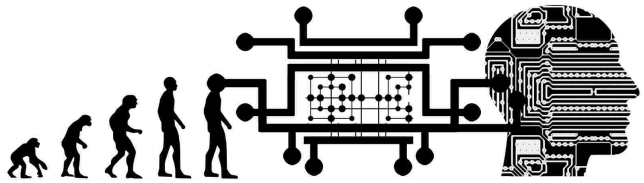
http://anoved.net/tag/lego/page/3/*



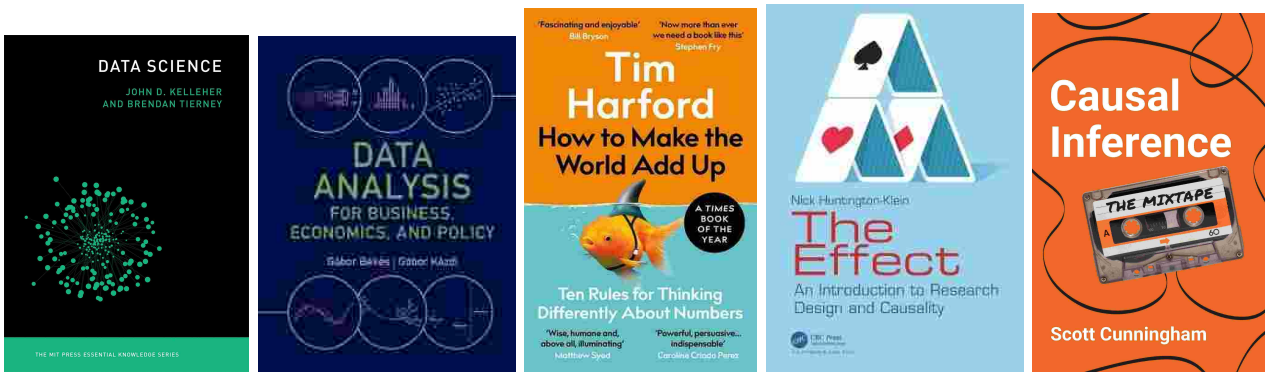


Chapter 1

Introduction: What is data science and data analysis?



1.1 Five books to learn data science and data analysis



- Kelleher, J. D. and Tierney, B. (2018). *Data Science*. MIT Press (Kelleher and Tierney, 2018)
- Harford, T. (2020). *How to Make the World Add Up: Ten Rules for Thinking Differently about Numbers*. The Bridge Street Press (Harford, 2020)
- Békés, G. and Kézdi, G. (2021). *Data Analysis for Business, Economics, and Policy*. Cambridge University Press (Békés and Kézdi, 2021)
- Huntington-Klein, N. (2021). *The Effect: An Introduction to Research Design and Causality*. Chapman and Hall/CRC[1] (Huntington-Klein, 2021)
- Cunningham, S. (2021). *Causal Inference: The Mixtape*. Yale University Press[2] (Cunningham, 2021)

[1] Freely available online: https://theeffectbook.net
[2] Freely available online: https://mixtape.scunning.com

Data science is what data scientists do

Data analysts...

- ...collect, manipulate, visualize, and analyze data.
- ...aim to discover useful information to support decision-making.
- ...should help to understand and optimize processes and to open up new product areas and/or lines of business.
- ...need to interact with customers, i.e., the user of the information as the addressee, and with domain experts.

Data analysts should have...

- ...econometric skills (Regressions, Time Series Analysis, Machine Learning)
- ...knowledge on statistical software packages (R, Phyton, SAS, etc.)
- ...a basic understanding of business operations (Business Intelligence)
- ...technical knowledge in database design, data models, data mining and segmentation techniques

1.2 My first PC

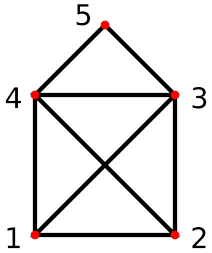
> "Data science is best understood as partnership between a data scientist and a computer."
> — Kelleher and Tierney (2018, p. 97)

Commodore C64, price about **750 Euro**, CPU < 1 MHz, RAM 64 KByte, i.e., 0,064 MB or 0,000064 GB, Graphics 320 x 200 with 16 colors, no HD but 5,25" floppy disks of 166KByte and a max of 144 files.
My main PC today has 4100Mhz at 8 cores and 16 threats with 32GB of RAM and my video card can do 3840 x 2160 at four screens, 1 NVmE of 1 TB, 2 SSD of 500GB each, 2 HD of 4TB each and 3TB in a cloud.
My first external HD in 2002 had 160GB and cost 170 Euro.

Exercise 1.1 — What is the house of Santa Claus? (Solution → p. 7)

The house of Santa Claus is an old German drawing game. It goes like this: You have to draw a house in one line where you (a) must start at bottom left (point 1), (b) you are not allowed to lift your pencil while drawing and (c) it is forbidden to repeat a line. During drawing you say: "Das ist das Haus des Nikolaus".

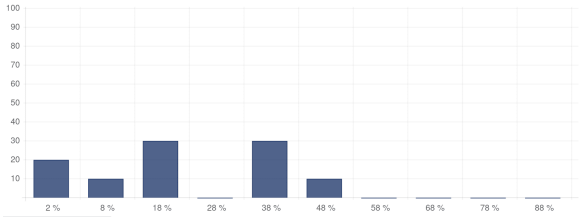
How many ways to fail and succeed, respectively do exist? Can you think of a logical procedure that gives you a complete solution?



Solution to Exercise 1.1 — What is the house of Santa Claus? (Exercise → p. 6)

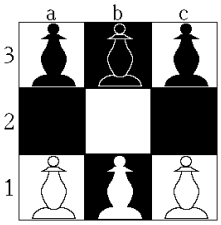
https://de.wikipedia.org/wiki/Haus_vom_Nikolaus

There are 44 solutions and only 10 different ways to fail. Thus, the probability to fail is about 18.5% and hence the probability to succeed is about 81.5%. In the course 10 persons participated in the poll. Here are the answers:



Nobody came close to the correct probability.

Exercise 1.2 — Hexapawn (Solution → p. 8)



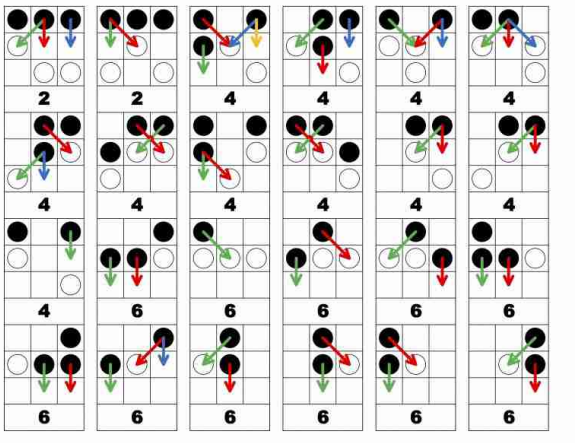
Hexapawn is a simple game with the following rules: As in chess, each pawn may be moved in two different ways: it may be moved one square forward, or it may capture a pawn one square diagonally ahead of it. A pawn may not be moved forward if there is a pawn in the next square. Unlike chess, the first move of a pawn may not advance it by two spaces. A player loses if they have no legal moves or the other player reaches the end of the board with a pawn.

a) Play that game with a fellow student.

b) Think hard: Is there a way how you could improve your winning rate?

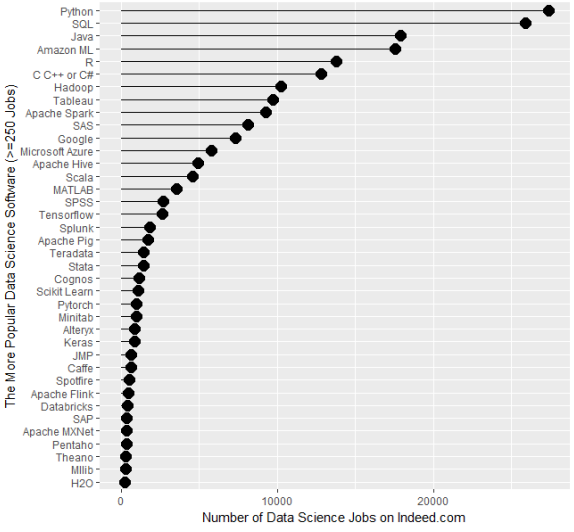
Solution to Exercise 1.2 — Hexapawn (Exercise → p. 7)

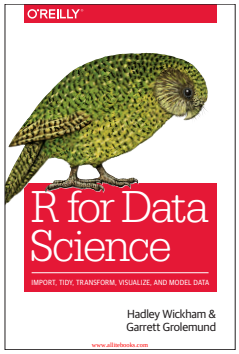
In his article ***How to build a game-learning machine and then teach it to play and to win*** *Gardner (1962) discussed how a computer could be taught to play the game Hexapawn using a relatively small number of training matches. The basic idea was to keep track of the different possible states of the board and the potential for success (i.e., a win) from each state. When a particular move led directly to a loss, the computer forgot the move, thereby causing it to avoid that particular loss in the future. By pruning possible moves in this way, various intermediate game moves could indirectly lead to losses (i.e., to states that previously resulted in losses), and thus those intermediate moves would be pruned out as well. This idea is a very simple version of reinforcement learning which serves as the basis for today's machine learning gaming systems such as Google's Alpha-Go. Gardner's original computer was constructed from matchboxes containing colored beads (this was the 1960's, remember). Each bead corresponded to a potential move, and pruning involved disposing of the last bead played.*



To understand how the game-learning machine works, watch ▶ *The Game That Learns* https://youtu.be/sw7UAZNgGg8 *or watch* ▶ *'Building a MENACE machine', Matthew Scroggs, University College London* https://youtu.be/hK25eXRaBdc

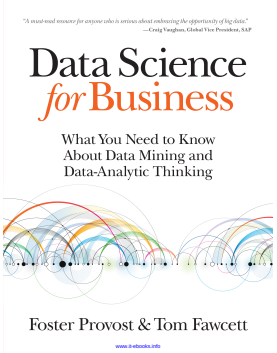
Programming skills





Source: Muenchen (2019)

Data science skills are required in many jobs



'Data scientist' is a weakly defined label

Business analysts...

- ...also focus on data.
- ...rarely use sophisticated methods like machine learning or tools to analyse unstructured data
- ...analyze data and assesses requirements from a business perspective related to an organization's overall system.
- ...are more concerned with the business implications of the data and the actions that should result (investment A vs. B).
- ...should leverage the work of data science teams to communicate an answer.

Stereotypes

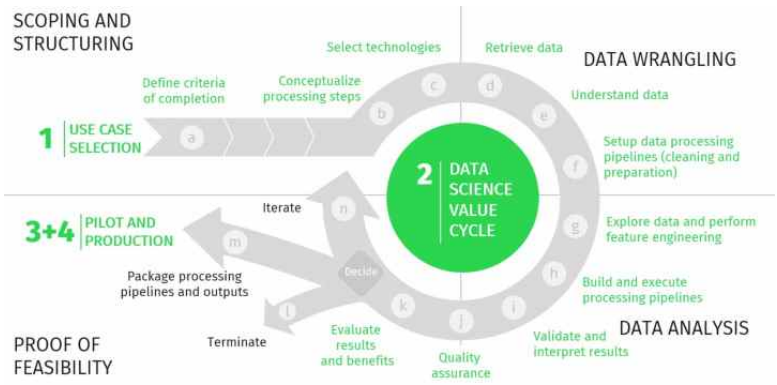
data scientists are nerds

business analysts are networking moneymaker

See http://www.worldofanalytics.be/blog/are-all-data-scientists-nerds for a funny investigation of this topic.

Data Science Approach: from prototype to production

The goal is to use data in a way which creates value for a business.



Source: https://www.onelogic.de/en/data-science-services

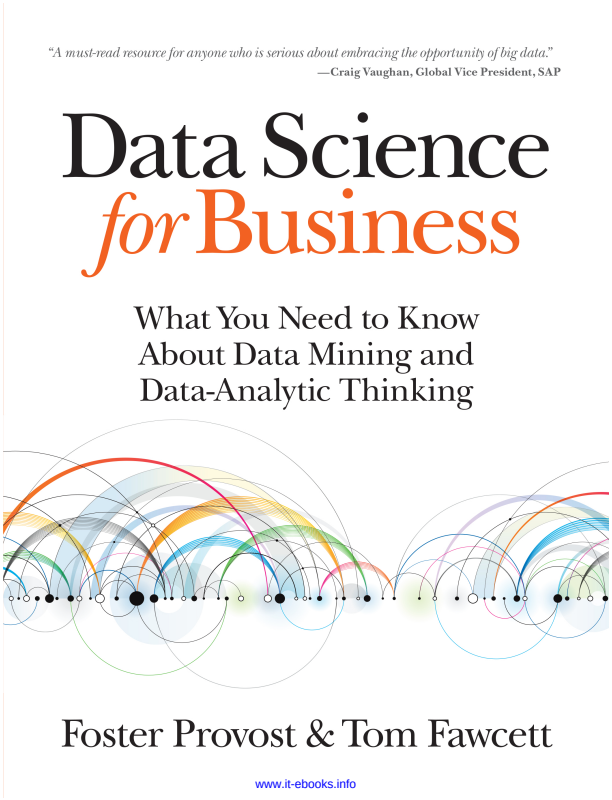
This lecture

The goal is to use data in a way which creates value for a business.

Source: https://www.onelogic.de/en/data-science-services

1.3 Example: I am too lazy to read the book

Provost and Fawcett (2013)



My view as a data scientist on Ch. 1&2 of Provost and Fawcett (2013)

Source: own calculations

The size of the word represents its frequency in the text, e.g., data was counted about 600 times, mining about 140 times and science about 100 times.

This is a **wordcloud**. A nice tool to analyze unstructured data such as the text of two chapters. Unstructured data does not have a pre-defined data model or is not organized in a pre-defined manner.

1.4 Example: Taxi driver in NY

1.4.1 Best and worst job

University Professor is ranked third with the best work environment and less stress, but only a median salary of $76,000.

Source: https://www.careercast.com/jobs-rated/2019-jobs-rated-report

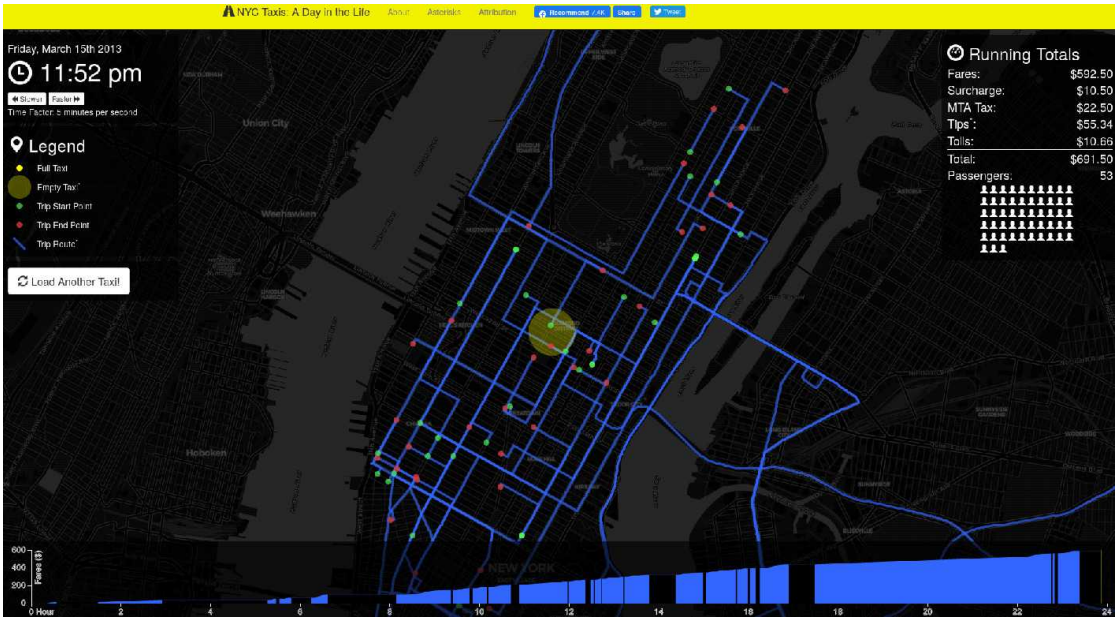
Can data science make taxi drivers happier?

NYC Taxis: A Day in the Life

- Visit: http://chriswhong.github.io/nyctaxi/
- This visualization displays the data for one random NYC yellow taxi on a single day in 2013. See where it operated, how much money it made, and how busy it was over 24 hours.

Get the data ready Create a spreadsheet/dataset that contains all the shown information excluding the particular route taken by the taxi driver.

NYC Taxis: A Day in the Life



How to create and organize data

Identifying Variables dr_id, date, op_id, trip

What is the 'Operation ID' and 'trip'? A Taxi can either be full or empty. With each change of this condition the ID must change. For example: A taxi starts empty (ID=1), then, the first passenger is picked up (ID=2), after that, a new trip begins when the taxi is empty again, ...

Attributes (Informing Variables) Start-Coordinates (latitude/longitude), End-Coordinates (latitude/longitude), Start-Time, End-Time, Travel-Distance (miles), Fares, Surcharege, MTA Tax, Tips, Tolls, Number of Passengers

Excerpt of the (structured) data:

dr_id	date	op_id	trip	GPS_1	GPS_2	time_1	time_2	dist	fares
A	03-15	1	1	40.7127837 -74.0059414	40.7137837 -74.0168515	09:00	09:10	1500	9
A	03-15	2	1	40.7137837 -74.0168515	40.7149827 -74.0157314	09:10	09:12	200	19
A	03-15	1	2	40.7149827 -74.0157314	40.9137837 -74.0168517	09:12	09:55	35000	19
...	...	...	...	...	...	...	...	...	...

Please note: This exemplary table is incomplete. In particular, the route of the trip is excluded. Real time travel information has a high granularity, i.e., tracking the taxi over time is more data intensive and the analysis is a real challenge.

Are these sort of data 'big data'?

For example, the T-Drive trajectory dataset of Zheng (2011) contains a one-week trajectories of 10,357 taxis. The total number of points in this dataset is about 15 million and the total distance of the trajectories reaches 9 million kilometers.

Exercise 1.3 — What can data scientists do for taxi drivers? (Solution → p. ??)

Maybe a driver wants

- more profit (fare-costs+tips)
- more tips
- less time being 'empty' and driving around searching
- to know fuel saving-routes
- to work in areas with less stressful traffic (How to avoid Manhattan?)
- to start at the optimal place and time
- to know when and where to rest without loosing much
- ...

  Discuss how data scientist may help.

Exercise 1.4 — Helping Luigi (Solution → p. 14)

Suppose you are consulting the delivery service of Luigi's Pizza restaurant. So far, Luigi's advertised his service in local television. Now, he commissioned you to use his database for targeted advertising. In particular, he wants find one district to start a poster advertising campaign.
Luigi gives you a large dataset with information on all 13,810 deliveries of the year 2018. Here is an excerpt of the first seven orders:

Order	Time	Meal	Price	Address	District	Time to deliver
1	10:08:00	21,45	17,87	ABC Street 34	A	10
2	10:38:00	23	17,25	Nevermindstreet 3	A	8
3	10:44:00	7	9,99	Summer Blv 43	C	18
4	10:51:00	3,6,32,44,51	45,12	Baker Street 21	B	34
5	10:55:00	45	12,9	Caxton Street 1	C	18
6	11:15:00	24,31,4,3	45,7	Chatham Avenue 33	B	40
7	11:17:00	6,76	19,8	Fitzroy Square 7	D	3

Legend:
Order: Number of order (from 1 to 13,810).
Time: Time of order (24-hour clock).
Meal: Component of the orders. Overall, Luigi offers 100 meals (from 1 to 100).

Price: Total Price of order measured in Euro. Luigi does not charge delivery costs.
Address: Street name and number. All addresses are in the same city.
District: The city consists of four districts (A, B, C, D). All have about the same size.
Time to deliver: The time measured in minutes that it takes to drive the food to the customer.

a) Explain the structure of the dataset. In particular, name the variables that identify the dataset.
b) Suppose you are a data scientist. Name the services you can provide to Luigi.
c) You know little about the effectiveness of poster advertising and Luigi's goals. Thus, in order to make the poster advertising campaign a success. You probably need more information. Luigi is well informed and open for your questions. Name four reasonable and concrete questions that you like to ask Luigi.
d) Luigi heart about the wordclouds and he thinks it is a modern tool that can help him. Explain Luigi what a wordcloud actually is and comment if it could really help him, or not.

Solution to Exercise 1.4 — Helping Luigi (Exercise → p. 13)

a) This is a structured dataset. The identifying variable is ***Order****.*
b) I can manipulate, visualize, and analyze the dataset for him. If necessary, I can collect and merge additional datasets to complement his information. Doing so, I can discover useful information to support his decision making. At best, I can help to understand and optimize his processes and to open up new product areas and/or lines of business.
c) Possible answer:
- *What do you like to improve and maximize, respectively.*
- *Do you like to maximize volume of sales?*
- *Do you like to maximize the profit?*
- *If you like to maximize the profit, can you give information about the profit/contribution margin for each meal?*
- *What are the average costs for a driver per minute?*
- *Can you give me your address? I need it to calculate the travel distances between your restaurant and the address of the customer.*
- *Do you think that a poster campaign help to attract new customers or increase the order frequency?*

d) A wordcloud is a tool to visualize unstructured dataset. Mostly, it is used to identify the importance of specific words in a text. Luigi's dataset only contains numeric variables with the addresses being the exception. Although a wordcloud could be used to visualize the popularity of meals or the frequency of orderings from particular districts, for example, it is probably better to do that in tables that show the frequency of occurrence for each meal and the number of orders by district.

Exercise 1.5 — AlphaGo (Solution → p. ??)

Watch ▶ *AlphaGo - The Movie | Full award-winning documentary* https://youtu.be/WXuK6gekU1Y

1.5 What is data?

Data are values of qualitative or quantitative variables, belonging to a set of items.

set of items: Sometimes called the population; the set of objects you are interested in
variables: A measurement or characteristic of an item.
qualitative: Country of origin, sex, treatment
quantitative: Height, weight, blood pressure

Although the terms '**data**' and '**information**' are often used interchangeably, these terms have distinct meanings. Data is sometimes said to be transformed into information when it is viewed in context or in post-analysis. In academic treatments of the subject, however, data are simply units of information. [...]

Data is measured, collected and reported, and analyzed, whereupon it can be visualized using graphs, images or other analysis tools. Data as a general concept refers to the fact that some existing information or knowledge is represented or coded in some form suitable for better usage or processing.

1.6 Types of data

Raw data ('unprocessed data') is a collection of numbers or characters before it has been 'cleaned' and corrected by researchers. Raw data needs to be corrected to remove outliers or obvious instrument or data entry errors [...]'. Data processing commonly occurs by stages, and the 'processed data' from one stage may be considered the 'raw data' of the next stage.
Field data is raw data that is collected in an uncontrolled environment.
Experimental data is data that is generated within the context of a scientific investigation by observation and recording.
Structured data: Data is stored, processed, and manipulated in a traditional relational database management system (e.g., temperature)
Un-structured data is commonly generated from human activities and does not fit into a structured database format (e.g., text)
Semi-structured data does not fit into a structured database system, but is nonetheless structured by tags that are useful for creating a form of order and hierarchy in the data
Big data *see below*
Dark data *see below*

1.6.1 Structured vs. semi-structured data

Business Intelligency requires analysts to deal with both structured and semi-structured data. The term semi-structured data is used for all data that does not fit neatly into relational or flat files, which is called structured data. We use the term semi-structured (rather than the more common unstructured) to recognize that most data has some structure to it. A survey indicated that 60% of CIOs and CTOs consider semi-structured data as critical for improving operations and creating new business opportunities (Blumberg and Atre, 2003).

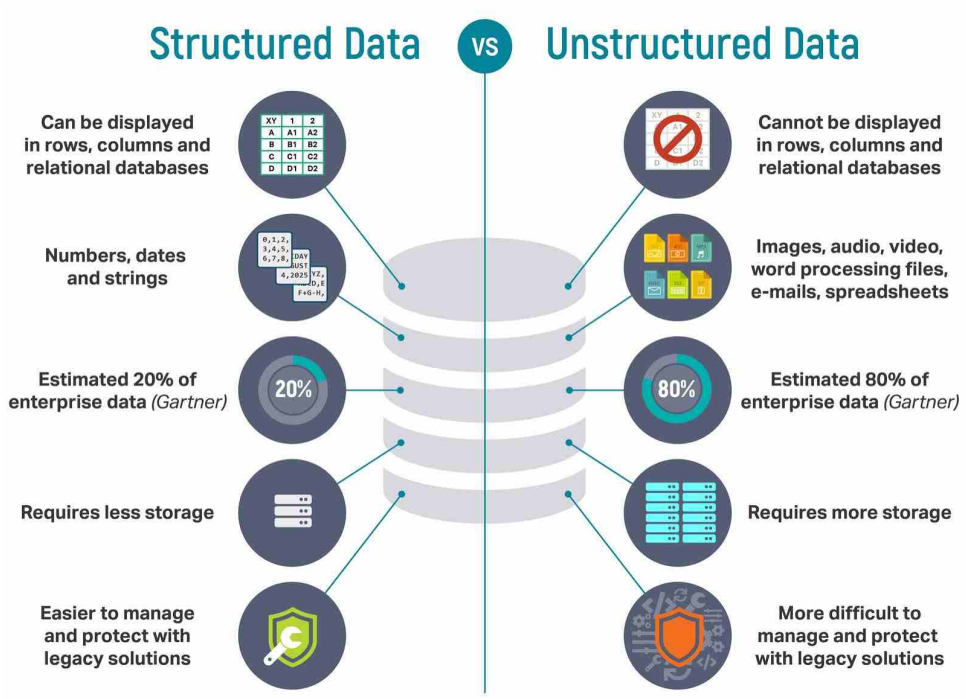


Figure 1.1: Structured vs. semi-structured data

Executive at Fortune 500 telecommunciations provider

"We have between 50,000 and 100,000 conversations with our customers daily, and I don't know what was discussed. I can see only the end point – for example, they changed their calling plan. I'm blind to the content of the conversations." (see Blumberg and Atre, 2003)

1.6.2 Semi-structured data: Examples

For example, e-mail is divided into messages and messages are accumulated into file folders.

Business processes, Chats, E-mails, Graphics, Image files, Letters, Marketing material, Memos, Movies, News items, Phone, conversations, Presentations, Reports, Research, Spreadsheet files, User group files, Video files, Web pages, White papers, Word processing text

Exercise 1.6 — Data Mining (Solution → p. **??**)

- Watch: https://youtu.be/EH3bp5335IU
- Read the Wikipedia page of 'Data Mining'

1.6.3 Big data

Big data is a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software. [...] Big data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source. Big data was originally associated with three key concepts: volume, variety, and velocity. When we handle big data, we may not sample but simply observe and track what happens. Therefore, big data often includes data with sizes that exceed the capacity of traditional software to process within an acceptable time and value.

Current usage of the term big data tends to refer to the use of predictive analytics, user behavior analytics, or certain other advanced data analytics methods that extract value from data, and seldom to a particular size of data set. [...] Analysis of data sets can find new correlations to "spot business trends,

prevent diseases, combat crime and so on." Scientists, business executives, practitioners of medicine, advertising and governments alike regularly meet difficulties with large data-sets in areas including Internet searches, fintech, urban informatics, and business informatics. Scientists encounter limitations in e-Science work, including meteorology, genomics, connectomics, complex physics simulations, biology and environmental research. **(Wikipedia)**

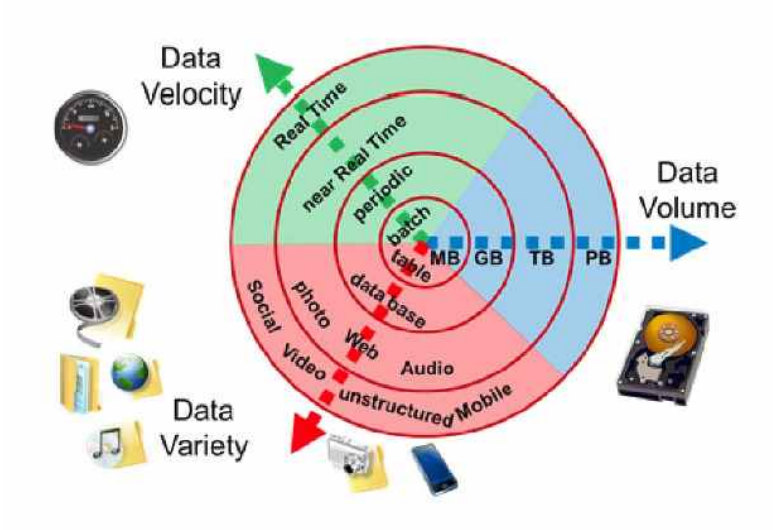
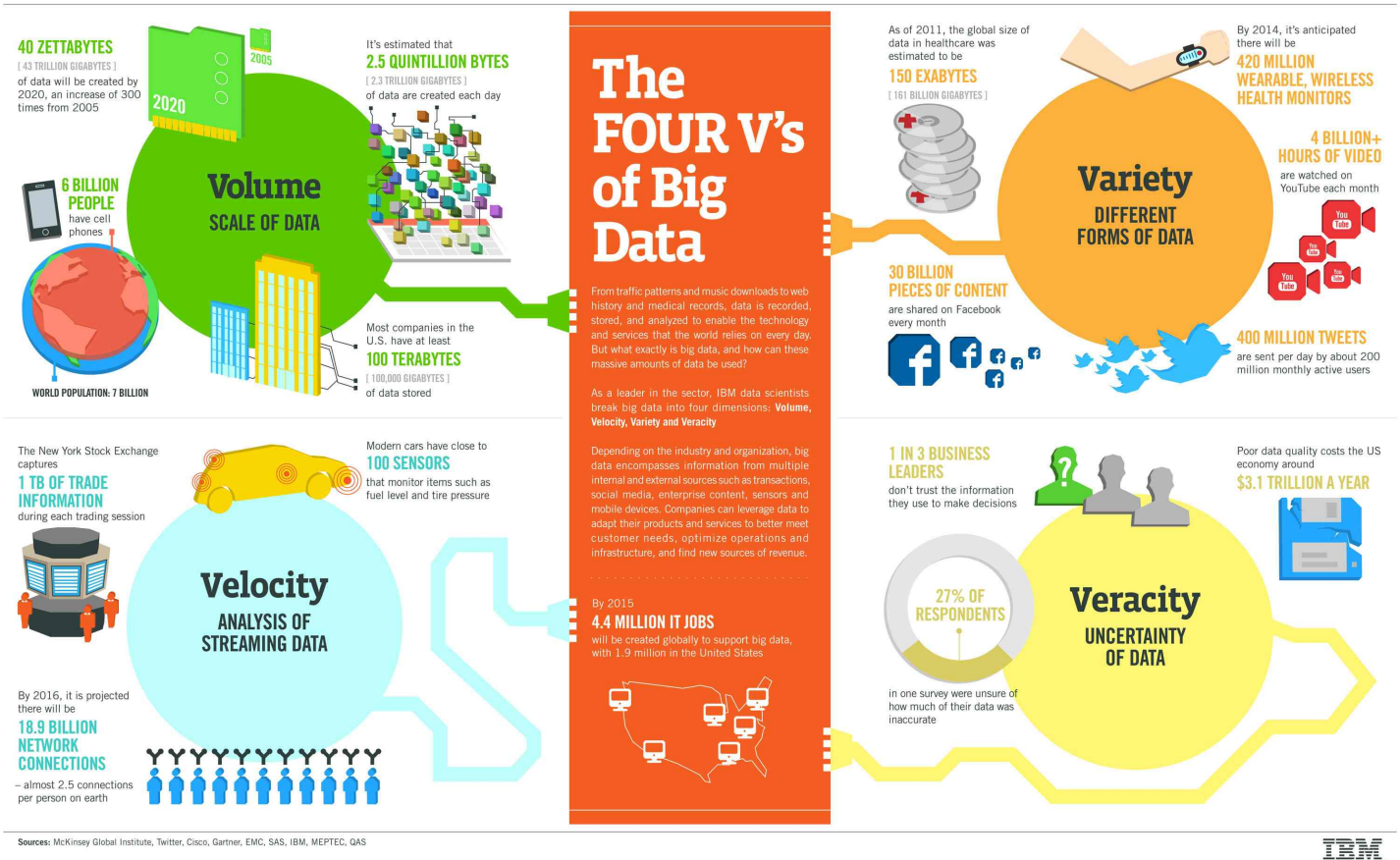


Figure 1.2: Big data characteristics

Volume: The amount of data matters. With big data, you'll have to process high volumes of low-density, unstructured data. This can be data of unknown value, such as Twitter data feeds, clickstreams on a webpage or a mobile app, or sensor-enabled equipment.

Velocity: The speed at which the data is generated and processed to meet the demands and challenges. Normally, the highest velocity of data streams directly into memory versus being written to disk. Some internet-enabled smart products operate in real time or near real time and will require real-time evaluation and action

Variety: Variety refers to the many types of data that are available. Traditional data types were structured and fit neatly in a relational database. With the rise of big data, data comes in new unstructured data types. Unstructured and semistructured data types, such as text, audio, and video, require additional preprocessing to derive meaning and support metadata


40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005
2005
2020
6 BILLION PEOPLE
have cell phones
WORLD POPULATION: 7 BILLION
Volume
SCALE OF DATA
It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day
Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored
The Four V's of Big Data
From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?
As a leader in the sector, IBM data scientists break big data into four dimensions: Volume, Velocity, Variety and Veracity
Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.
By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States
As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]
30 BILLION PIECES OF CONTENT
are shared on Facebook every month
Variety
DIFFERENT FORMS OF DATA
By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS
4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month
400 MILLION TWEETS
are sent per day by about 200 million monthly active users
The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session
Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure
Velocity
ANALYSIS OF STREAMING DATA
By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth
1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions
27% OF RESPONDENTS
in one survey were unsure of how much of their data was inaccurate
Veracity
UNCERTAINTY OF DATA
Poor data quality costs the US economy around
$3.1 TRILLION A YEAR
Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS
IBM


Big Data: What It Is and Why It Matters
What is big data?[1]
Big data sets are too large and complex to be processed by traditional methods. Consider that in a single minute there are:
277,777 Instagram stories posted
511,200 tweets sent
4,500,000 YouTube videos watched
4,497,420 Google searches conducted
18,100,000 texts sent
188,000,000 emails sent
The 3 V's of big data – Plus 2
These are the defining properties or dimensions of big data.
Volume the amount of data
Velocity the speed of data
Variety the types of data
Variability the variety of and changes in data flows
Veracity the quality of data
How do organizations optimize the value of big data?
Regardless of location, size, sources, owners or users, these steps can unleash value from an organization's complex data landscape (data fabric).
Set a big data strategy.
Identify big data sources.
Access, manage and store big data.
Analyze big data.
Make data-driven decisions.

1.6.4 Dark data

Dark data is data that is collected through various computer network operations but is not used in any way to gain insight or make decisions. An organization's ability to collect data may exceed its capacity with which to analyze the data. In some cases, the company is not even aware that the data is being collected. Approximately 90 percent of the data generated by sensors and analog-to-digital converters is never used.

1.7 What is data analytics?

The data is the second most important thing

- The most important thing in data science is the question
- The second most important is the data
- Often the data will limit or enable the questions
- But having data can't save you if you don't have a question

Data science is an inter-disciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from many structural and unstructured data. Data science is related to data mining and big data.
Data science is a "concept to unify statistics, data analysis, machine learning and their related methods" in order to "understand and analyze actual phenomena" with data. It employs techniques and theories drawn from many fields within the context of mathematics, statistics, computer science, and information science.

Data analytics is what data scientists do

- Define the question
- Define the ideal data set
- Determine what data you can access
- Obtain the data
- Clean the data
- Exploratory data analysis
- Statistical prediction/modeling
- Interpret results
- Challenge results
- Synthesize/write up results
- Create reproducible code
- Distribute results to other people

Exercise 1.7 — What is data analysis (Solution → p. ??)

Open the Wikipedia page of 'Data Analysis'.

- Therein, the process of data analysis is described in eight steps. Read and try to memorize these steps.
- Stephen Few described eight types of quantitative messages that users may attempt to understand or communicate from a set of data and the associated graphs used to help communicate the message. Read and try to memorize these steps.
- The work of a data analyst contains many different tasks. Some of these tasks are described in the article. Which of these are new to you?

Data Scientist:
The Sexiest Job of the 21st Century

Meet the people who can coax treasure out of messy, unstructured data.

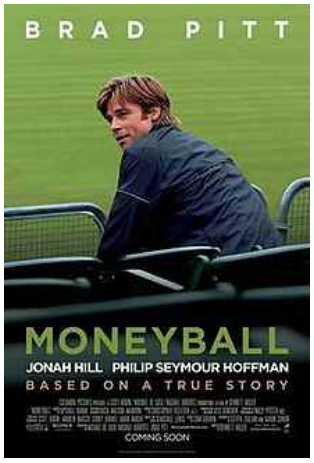
by Thomas H. Davenport and D.J. Patil

When Jonathan Goldman arrived for work in June 2006 at LinkedIn, the business networking site, the place still felt like a start-up. The company had just under 8 million accounts, and the number was growing quickly as existing members invited their friends and colleagues to join. But users weren't seeking out connections with the people who were already on the site at the rate executives had expected. Something was apparently missing in the social experience. As one LinkedIn manager put it, "It was like arriving at a conference reception and realizing you don't know anyone. So you just stand in the corner sipping your drink—and you probably leave early."

70 Harvard Business Review **October 2012**

Figure 1.3: The sexiest job of the 21st century
Source: Davenport and Patil (2012)

Data analysis...



... is a process of inspecting, cleansing, transforming and modeling data with the goal of discovering useful information, informing conclusion and supporting decision-making. Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, and is used in different business, science, and social science domains. In today's business world, data analysis plays a role in making decisions more *scientific* and helping businesses operate more effectively.

- Please watch: https://youtu.be/J36ZfXBsGjs
- Sport Economics[a] is a well-accepted discipline, see: https://journals.sagepub.com/home/jse

[a] It covers both the ways in which economists can study the distinctive institutions of sports, and the ways in which sports can allow economists to research many topics, including discrimination and antitrust law.

1.8 Machine learning, AI, automated decision-making

While **machine learning** (ML) is based on the idea that machines should be able to learn and adapt through experience, artificial intelligence (AI) refers to a broader idea where machines can execute tasks *smartly*. AI applies ML techniques to solve actual problems and to automate decision making.

1.8.1 Machine learning...

...is the study of computer algorithms that improve automatically through experience. In particular, machine learning is a form of artificial intelligence (AI) as it provides machines and systems to automatically learn and improve from experience. Machine learning algorithms build a mathematical model based on sample data, known as 'training data', in order to make predictions or decisions without being explicitly programmed to do so.

...for making predictions: if you want a model to determine future trends; machine learning algorithms are the best bet. This falls under the paradigm of supervised learning. It is called supervised because you already have the data based on which you can train your machines (for

example, a fraud detection model can be trained using a historical record of fraudulent purchases).

. . . for pattern discovery: If you don't have the parameters based on which you can make predictions, then you need to find out the hidden patterns within the dataset to be able to make meaningful predictions. This is nothing but the unsupervised model as you don't have any predefined labels for grouping. The most common algorithm used for pattern discovery is Clustering.

1.8.2 What is automated decision-making?

Automated decision-making is the process of making a decision by automated means without any human involvement. These decisions can be based on factual data, as well as on digitally created profiles or inferred data. Examples of this include:

- an online decision to award a loan; and
- an aptitude test used for recruitment which uses pre-programmed algorithms and criteria.

Automated decision-making often involves **profiling**, but it does not have to.

Demetis and Lee (2018): "Another well-known example comes from Amazon. The vast majority of prices are defined by algorithms in so far as Amazon vendors "use algorithmic pricing to ensure that they can automatically change their product prices based on a competitor"' [39], with the result that vendors are being forced to engage in this practice for fear of losing out to the competition. Meanwhile, the algorithmic interactions between vendors carry the possibility of developing unpredictable consequences. Such algorithmic pricing on Amazon can be found in the example of the book entitled The Making of a Fly by evolutionary biologist Peter Lawrence. This book came to be priced at $23,698,655.93 (plus $3.99 shipping) as two sellers were using algorithms to adjust the price of the book in response to one another. It took 10 days for humans to notice and intervene to bring back the prices to normal levels [43]; ironically, "normal levels" merely indicated a temporary human decision that would allow the continuation of algorithmic pricing."

1.8.3 What is profiling?

Profiling analyzes aspects of an individual's personality, behavior, interests and habits to make predictions or decisions about them. In particular, profiling' means any form of automated processing of personal data consisting of the use of personal data to evaluate certain personal aspects relating to a natural person, in particular to analyze or predict aspects concerning that natural person's performance at work, economic situation, health, personal preferences, interests, reliability, behavior, location or movements.

Watch: https://youtu.be/7-MNbzv8lAA

You are carrying out profiling if you:

- collect and analyse personal data on a large scale, using algorithms, AI or machine-learning;
- identify associations to build links between different behaviours and attributes;
- create profiles that you apply to individuals; or
- predict individuals' behaviour based on their assigned profiles.

Organizations obtain personal information about individuals from a variety of different sources. Internet searches, buying habits, lifestyle and behavior data gathered from mobile phones, social networks, video surveillance systems and the Internet of Things are examples of the types of data organizations might collect.

They analyze this information to classify people into different groups or sectors. This analysis identifies correlations between different behaviors and characteristics to create profiles for individuals. This profile will be new personal data about that individual.

Organizations use profiling to

- find something out about individuals' preferences;
- predict their behavior; and/or
- make decisions about them.

Profiling can use **algorithms**. An algorithm is a sequence of instructions or set of rules designed to complete a task or solve a problem. Profiling uses algorithms to find correlations between separate datasets. These algorithms can then be used to make a wide range of decisions, for example to predict behavior or to control access to a service. Artificial intelligence (AI) systems and machine learning are increasingly used to create and apply algorithms.

Although many people think of marketing as being the most common reason for profiling, this is not the only application.

What are the benefits of profiling and automated decision-making? Profiling and automated decision making can be very useful for organisations and also benefit individuals in many sectors, including healthcare, education, financial services and marketing. They can lead to quicker and more consistent decisions, particularly in cases where a very large volume of data needs to be analysed and decisions made very quickly.

Examples Profiling is used in some medical treatments, by applying machine learning to predict patients' health or the likelihood of a treatment being successful for a particular patient based on certain group characteristics.

Less obvious forms of profiling involve drawing inferences from apparently unrelated aspects of individuals' behavior.

Using social media posts to analyze the personalities of car drivers by using an algorithm to analyze words and phrases which suggest 'safe' and 'unsafe' driving in order to assign a risk level to an individual and set their insurance premium accordingly.

What are the risks? Although these techniques can be useful, there are potential risks:

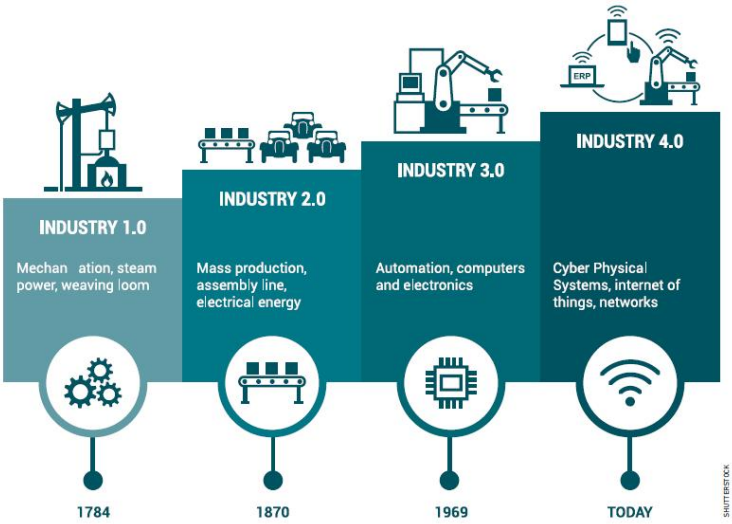
- Profiling is often invisible to individuals.
- People might not expect their personal information to be used in this way.
- People might not understand how the process works or how it can affect them.
- The decisions taken may lead to significant adverse effects for some people.

Just because analysis of the data finds a correlation doesn't mean that this is significant. As the process can only make an assumption about someone's behaviour or characteristics, there will always be a margin of error and a balancing exercise is needed to weigh up the risks of using the results.

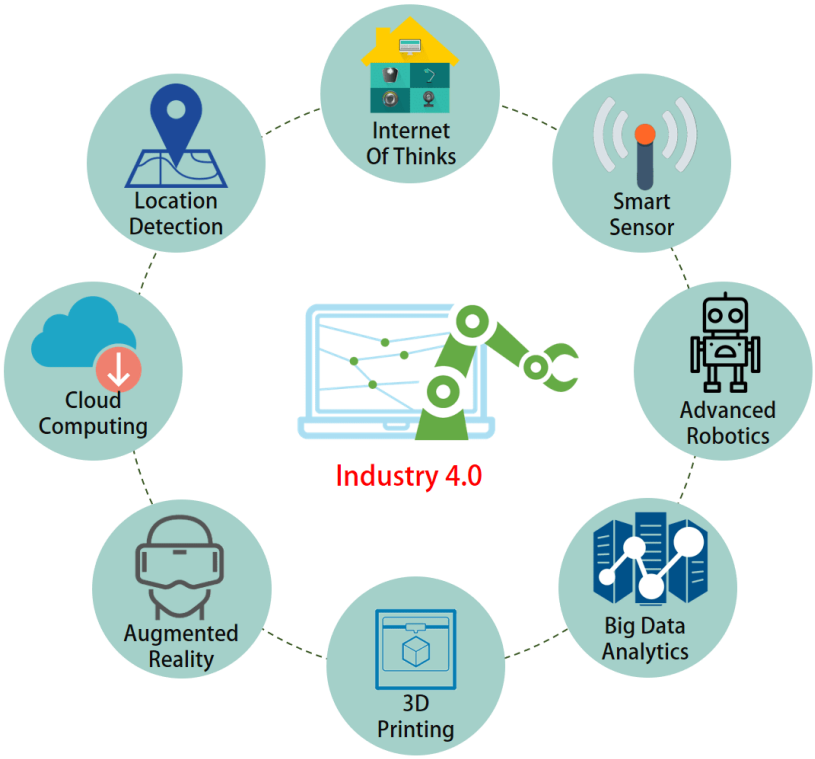
1.8.4 Industry 4.0

Smart industry or *INDUSTRIE 4.0* refers to the technological evolution from embedded systems to cyber-physical system. 'INDUSTRIE 4.0 represents the coming fourth industrial revolution on the way to an Internet of Things, Data and Services. Decentralized intelligence helps create intelligent object

networking and independent process management, with the interaction of the real and virtual worlds representing a crucial new aspect of the manufacturing and production process.



INDUSTRY 4.0 FRAMEWORK – THE DIGITAL TECHNOLOGIES



Chapter 2

Why and how to learn R

There are multiple different approaches on how to learn R. It pretty much depends on your preferences, needs, goals, prerequisites, and constraints. I mean how many time do you want to spend on learning R and what type of learning do you prefer. You have to find your own way. However, I offer this script. It should guide you through a lot of things that I found important to know when working in R and it should help you to dig deeper and learn more if you like to do so. There are thousands of other ressources to learn R: textbooks, online courses, videos, guided tutorials, and so on.
In the following, I give you a list of ressources that are worth a look. Maybe you find there what you are looking for. If not, just continue to read this book. Especially those who participated in one of my courses in person, feel free to contact me whenever you think I can help.

2.1 Why R?

Maybe your answer is "because that's what my stats class uses". Let me work on that by explaining a little why your lecturer and University, respectively, has chosen to use R for the class.

- **R is an artist!** Check out:

  https://www.r-graph-gallery.com/ ,

  http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html , and

  https://www.r-bloggers.com/2020/05/7-useful-interactive-charts-in-r/

  feel impressed by the beautiful graphical visualizations.
- **R is an employment insurance.** If you are good in R programming or if you are good in writing programming code in general, you have plenty of opportunities to earn a decent salary.
- **R uses the computer and computers are great!** Doing statistics on a computer is faster, easier and more powerful than doing it by hand. Computers excel at mindless repetitive tasks. For most people, the only reason to ever do statistical calculations with pencil and paper is for learning purposes. However, I do occasionally suggest doing some calculations by hand, but the only real value to it is pedagogical.
- **Excel is bad!** Doing statistics in a spreadsheet (e.g., Microsoft Excel) is generally a bad idea in the long run. Although many people are likely feel more familiar with them, spreadsheets are very limited in terms of what analyses they allow you to do. You can easily lose the overview and it is hard to keep track of what you have done and in comparison with command line driven programs. In particluar, the ability to make your analysis *replicate-able* is limited.
- **R is good, proprietary software is bad!** Avoiding proprietary software is a very good idea. Commercial software is expensive: usually, the company sells "student versions" (crippled versions of the real thing) very cheaply; they sell full powered "educational versions" at a price

that makes me wince; and they sell commercial licenses with a staggeringly high price tag. The business model here is to suck you in during your student days, and then leave you dependent on their tools when you go out into the real world. It's hard to blame them for trying, but personally I'm not in favor of shelling out thousands of dollars if I can avoid it. And you can avoid it: if you make use of packages like R that are open source and free, you never get trapped having to pay exorbitant licensing fees.

- **R is big!** Something that you might not appreciate now, but will love later on if you do anything involving data analysis, is the fact that R is highly extensible. When you download and install R, you get all the basic "packages", and those are very powerful on their own. However, because R is so open and so widely used, it's become something of a standard tool in statistics, and so lots of people write their own packages that extend the system. And these are freely available too. One of the consequences of this, I've noticed, is that if you open up an advanced textbook (a recent one, that is) rather than introductory textbooks, is that a *lot* of them use R. In other words, if you learn how to do your basic statistics in R, then you're a lot closer to being able to use the state of the art methods than you would be if you'd started out with a "simpler" system: so if you want to become a genuine expert in data analysis, learning R is a very good use of your time.
- **R is the future!** Related to the previous point: R is a real programming language. As you get better at using R for data analysis, you're also learning to program. To some people this might seem like a bad thing, but in truth, **programming is a core skill** in research, economics, and business. R is one of the most widely used programming languages in the world today. It is used in almost every industry such as finance, banking, medicine or manufacturing. R is used for portfolio management, risk analytics in finance and banking industries. It is used for carrying out an analysis of drug discovery and genomic analysis in bioinformatics. R is also used to implement various statistical measures to optimize industrial processes. R is the quasi-standard in data science.

Warning: R is not without its flaws: it's not easy to learn, it has a few very annoying quirks to it that we're all pretty much stuck with, it is slower than other languages (Phyton, MATLAB), and the algorithms and sources of R are spread across many packages (as there is no big company behind that wants you to buy it). This sometimes makes it very hard for beginners to find what they are looking for. In simply words: you can get lost!

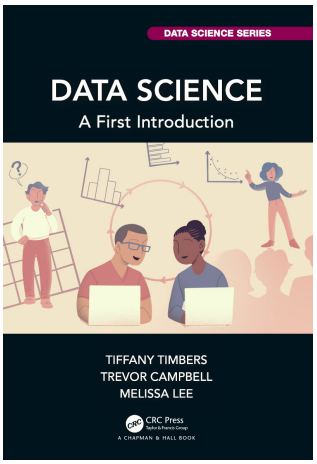
2.2 R learning resources

2.3 Collection of links and ressources

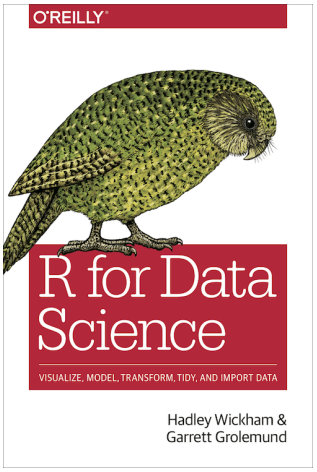
AWESOME
R Learning Resources

https://github.com/iamericfletcher/awesome-r-learning-resources
and
www.bigbookofr.com Probably the biggest collection of R books

2.4 Textbooks



Timbers et al. (2022) *Data Science: A First Introduction* — Very good and up to date book. You can read the web version of the book on https://datasciencebook.ca/ The book has accompanying worksheets providing exercises. All of the worksheets are available at https://github.com/UBC-DSCI/data-science-a-first-intro-worksheets



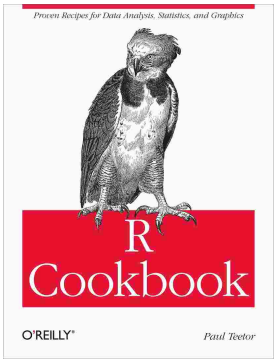
Wickham and Grolemund (2018) *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data* — Maybe the most popular source to learn R. It has a focus on introducing the tidyverse package and all its powerful functions. It is freely available online (https://r4ds.had.co.nz/). Unlike the book, the online version is updated regularly. It will teach you how to do data science with R: You'll learn how to get your data into R, get it into the most useful structure, transform it, visualize it and model it. You will find a practicum of skills for data science. Just as a chemist learns how to clean test tubes and stock a lab, you'll learn how to clean data and draw plots—and many other things besides. These are the skills that allow data science to happen, and here you will find the best practices for doing each of these things with R. You'll learn how to use the grammar of graphics, literate programming, and reproducible research to save time. You'll also learn how to manage cognitive resources to facilitate discoveries when wrangling, visualizing, and exploring data.

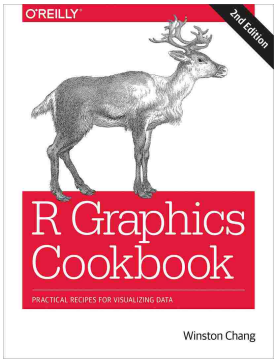
Venables et al. (2022) *An Introduction to R Notes on R: A Programming Environment for Data Analysis and Graphics* — This is a manual from the R Core Development Team shows how to use R without having to install and load additional packages.

Irizarry (2020) *Introduction to Data Science. Data Analysis and Prediction Algorithms with R* — https://rafalab.github.io/dsbook/

Crawley (2013) *The R Book* — Another big book on R.

Teetor (2011) *Cookbook for R* (also see: http://www.cookbook-r.com/) and
Chang (2018) *R Graphics Cookbook: Practical Recipes for Visualizing Data* (also see: https://r-graphics.org/)





Various free textbooks Please find more hints where to find more free textbooks: https://cmdlinetips.com/2018/01/free-online-resources-books-to-learn-r-and-data-science/

https://bookdown.org/ A rather long list of books on R are published here.

2.5 Online tutorials

In RStudio you find in the left panel at the bottom a panel that is called *Help*. There you find a lot of links, manuals, and referrences that offer you tons of resources to learn R for free including: https://education.rstudio.com/] https://support.rstudio.com/hc/en-us/articles/200552336-Getting-Help-with-R

On Youtube you find dozens of good explanations to every topic that relates to R and RStudio. Also, there are hundreds of online courses available. Just Google it. Personally, I like the *two-minute* approach of twotorials.com or https://www.datamentor.io/r-programming/. Also worth to mention are the following: https://www.statmethods.net/ https://www.r-bloggers.com/2015/12/how-to-learn-r-2/ https://data-flair.training/blogs/r-tutorial/ Tutorial that you can simply watch and follow the instructions are usually for free. In any case, I would not recommend to spend a dime on commercial sources that want you to believe that they have found the holy grail how to learn R quick. Open source ressources are as good as commercial offers. Better spend your money to the open source community. They actually invented, build up, and run R.

2.6 Other resources

- The Rseek website (www.rseek.org). One thing that I really find annoying about the R help documentation is that it's hard to search properly. When coupled with the fact that the documentation is dense and highly technical, it's often a better idea to search or ask online for answers to your questions. With that in mind, the Rseek website is great: it's an R specific search engine. I find it really useful, and it's almost always my first port of call when I'm looking around.
- The R-help mailing list (see http://www.r-project.org/mail.html for details). This is the official R help mailing list. It can be very helpful, but it's *very* important that you do your homework before posting a question. The list gets a lot of traffic. While the people on the list try as hard as they can to answer questions, they do so for free, and you *really* don't want to know how much money they could charge on an hourly rate if they wanted to apply market rates. In short, they are doing you a favour, so be polite. Don't waste their time asking questions that can be easily answered by a quick search on Rseek (it's rude), make sure your question is clear, and all of the relevant information is included. In short, read the posting guidelines carefully (http://www.r-project.org/posting-guide.html), and make use of the `help.request()` function that R provides to check that you're actually doing what you're expected.
- Here you find all documentations online: www.rdocumentation.org
- Here is a video that discusses how you should ask for help: https://youtu.be/ZFaWxxzouCY

Chapter 3

Installing R and RStudio

In this chapter I'll talk about how to download and install R and RStudio.

Okay, enough with the sales pitch. Let us set up R on your computer. Let's start by downloading R here:

http://cran.r-project.org/

At the top of the webpage you'll see separate links for Windows users, Mac users, and Linux users. If you follow the relevant link, you'll see that the online instructions are pretty self-explanatory. The version of R changes frequently. Please, just go for the most recent one.

3.1 Installing R on a Windows Computer

The CRAN homepage changes from time to time, and it's not particularly pretty. However, it's not difficult to find what you're after. In general you'll find a link at the top of the page with the text "Download R for Windows". If you click on that, it will take you to a page that offers you a few options. Again, at the very top of the page you'll be told to click on a link that says to click here if you're installing R for the first time. That's probably what you want. This will take you to a page that has a prominent link at the top called "Download R 4.0.4 for Windows". That's the one you want. Click on that and your browser should start downloading a file called `R-4.0.4-win.exe`, or whatever the equivalent most recent version number is by the time you read this. Once you've downloaded the file, double click to install it. As with any software you download online, Windows will ask you some questions about whether you trust the file and so on. After you click through those, it'll ask you where you want to install it, and what components you want to install. The default values should be fine for most people, so again, just click through. Once all that is done, you should have R installed on your system. You can access it from the Start menu, or from the desktop if you asked it to add a shortcut there. You can now open up R in the usual way if you want to, but what I'm going to suggest is that instead of doing that you should now install **R Studio** RStudio.

3.2 Installing R on a Mac

When you click on the Mac OS X link, you should find yourself on a page with the title "R for Mac OS X". There's a fairly prominent link on the page called "R-4.0.4.pkg" (maybe the version number is different), which is the one you want. Click on that link and you'll start downloading the installer file. Once you've downloaded it, all you need to do is open it by double clicking on the package file. The installation should go smoothly from there: just follow all the instructions just like you usually do when you install something. Once it's finished, you'll find a file called `R.app` in the Applications folder. You can now open up R in the usual way if you want to, but what I'm going to suggest is that instead of doing that you should now install Rstudio.

3.3 Installing R on a Linux Computer

The instructions on the website are easy enough. The CRAN site has precompiled binaries for Debian, Red Hat, Suse and Ubuntu and has separate instructions for each. Once you've got R installed, you can run it from the command line just by typing `R`.

3.4 Using R in the cloud

If you don't want to install R on your PC, you can run R in the *cloud*. To play around with the console in your browser, see: https://rdrr.io/snippets/ or https://rextester.com/l/r_online_compiler

For a more advanced experience, you can check out RStudio Cloud (for free for individuals) which is a lightweight, cloud-based solution that allows anyone to do, share, teach and learn data science online, see: https://rstudio.cloud/

3.5 Starting up R

When you start R, the first thing you'll see is a whole lot of text that doesn't make much sense. It should look something like this:

```
When you start \R, the first thing you'll see is a whole lot of text that doesn't make much

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

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

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

>
```

Most of this text is pretty uninteresting, and when doing real data analysis you'll never really pay much attention to it. The important part of it is this...

```
>
```

... which has a flashing cursor next to it. That's the **command prompt**. When you see this, it means that R is waiting for you to do something!

3.6 Downloading and installing RStudio

Regardless of what operating system you're using, the last thing that I told you to do is to download RStudio. To understand why I've suggested this, you need to understand a little bit more about R itself. The term R doesn't really refer to a specific application on your computer. Rather, it refers to the underlying statistical language. You can use this language through lots of different applications. When you install R initially, it comes with one application that lets you do this: it's the R.exe application on a Windows machine, and the R.app application on a Mac. But that's not the only way to do it. There are lots of different applications that you can use that will let you interact with R. One of those is called RStudio, and it's the one I'm going to suggest that you use. RStudio provides a clean, professional interface to R that is much more user-friendly. In particular, RStudio is freely available open-source Integrated Development Environment (IDE). RStudio provides an environment with many features to make using R easier and is a great alternative to working on R in the terminal.

Like R itself, RStudio is free software. Now, download RStudio here:

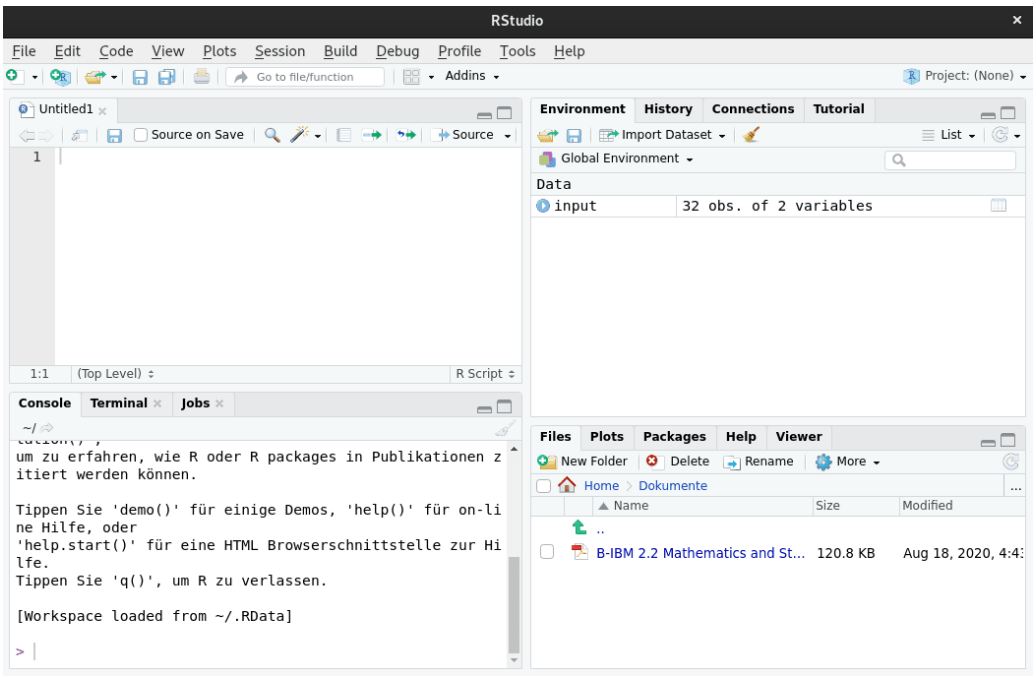


Figure 3.1: An R session in progress running through RStudio.

https://rstudio.com/products/rstudio/download/

On the homepage various versions are offered. You should go for the 'RStudio Desktop' version with an open source license. After choosing the desktop version it will take you to a page that shows several possible downloads: there's a different one for each operating system. Click on the appropriate link, and the RStudio installer file will start downloading. Open the installer file in the usual way to install RStudio. After it's finished installing, you can start R by opening RStudio. You don't need to open R.app or R.exe in order to access R. RStudio will take care of that for you. To illustrate what RStudio looks like, Figure 3.1 shows a screenshot of an R session in progress. RStudio looks almost identical no matter what operating system you have. There are a few minor differences in where things are located in the menus and in the shortcut keys, because RStudio is trying to "feel" like a proper Mac application or a proper Windows application, and this means that it has to change its behavior a little bit depending on what computer it's running on. Please notice, the look can be altered rather easily in the global options. For example, I use a dark mode which is a bit better for the eyes when watching at it for hours.

The RStudio Interface has four main panels:

Console: where you can type commands and see output. The console is all you would see if you ran R in the command line without RStudio.

Script editor: where you can type out commands and save to file. You can also submit the commands to run in the console.

description/History: environment (a.k.a. workspace) shows all active objects and history keeps track of all commands run in console

Files/Plots/Packages/Help

For more information, see: https://rstudio.com/products/rstudio/?wvideo=520zbd3tij

Chapter 4

{swirl}-it

Required exercises:

- Students should go through the following learning modules that are all part of my *swirl-it* course:
  - *huber-intro-1*
  - *huber-intro-2*
  - *huber-data-1*
  - *huber-data-2*
  - *huber-data-3*

4.1 What is {swirl}?

`swirl` teaches you R programming and data science interactively, at your own pace, and right in the R console! Just follow the instructions on https://swirlstats.com/students.html and you will learn R step by step within RStudio itself.

4.2 A short introduction to R and RStudio in two {swirl} modules

The *swirl* R package makes it fun and easy to learn R programming and data science. If you are new to R, have no fear. *swirl* will walk you through each of the steps required to employ RStudio and R for your purpose.

Open RStudio and type in the console the following:

```
install.packages("swirl")
library("swirl")
ls()
rm(list=ls())
```

The above lines of code do the following:

- Install the *swirl* package.
- Load the *swirl* package.
- List the content of the environment.
- Remove everything from the environment.
- Start *swirl*.

Now type in the Console the following:[1]

```
install_course_github("hubchev", "swirl-it")
```

[1] If the course has failed to install, you can try to download the file `swirl-it.swc` from https://github.com/hubchev/swirl-it and install the course with `install_course()`

With the first line, you install my *swirl* course that is hosted on GitHub.

To start swirl and your learning experience type

```
swirl()
```

With `swirl()` you start *swirl*. Please choose the course *swirl-it* and the learning module *huber-intro-1*. You can exit *swirl* at any time by typing `bye()` or by clicking the `Esc` on your keyboard.

After you have successfully finished learning module *huber-intro-1* please go ahead with the learning module *huber-intro-2* that is also part of my swirl course *swirl-it*.

4.3 {swirl} module on data analytical basics

In my swirl modules *huber-data-1*, *huber-data-2*, and *huber-data-3* I introduce some very basic statistical principles on how to analyse data. These modules are part of my *swirl-it* course that can be installed as explained in section 4.2.

4.4 {swirl} module on the *tidyverse* package

I compiled a short *swirl* module to introduce the *tidyverse* universe. This is a powerful collection of packages which I discuss in chapter 7. The learning module is also part of my *swirl-it* course.

4.5 Other {swirl} courses

You can also install some other courses. You find a list of courses here `http://swirlstats.com/scn/index.html` or here `https://github.com/swirldev/swirl_courses`

I can recommend the following:

```
swirl::install_course_github("swirldev", "R_Programming_E")
swirl::install_course_github("matt-dray", "tidyswirl")
%  swirl::install_course("Exploratory Data Analysis")
swirl::install_course("Getting and Cleaning Data")
swirl::install_course_github("sysilviakim", "swirl-tidy")
swirl::install_course("Regression Models")
```

Chapter 5

Getting started with R

In this chapter I will focus on getting you started typing R commands. Our goal in this chapter is not to learn any statistical concepts: we're just trying to learn the basics of how R works and get comfortable interacting with the system. To do this, we'll spend a bit of time using R as a simple calculator and you'll get a bit of a feel for what it's like to work in R. From there I'll introduce some very basic programming ideas: in particular, I'll talk about the idea of defining *variables* to store information, and a few things that you can do with these variables.

5.1 Typing commands at the R console

One of the easiest things you can do with R is use it as a simple calculator, so it's a good place to start. For instance, try typing `10 + 20`, and hitting enter.[1] When you do this, you've entered a **command**, and R will "execute" that command. What you see on screen now will be this:

```
> 10 + 20
[1] 30
```

The `>` symbol is just the R command prompt and isn't part of the actual command. The `[1]` part just means "the answer to the 1st question you asked is 30".

5.1.1 Be very careful to avoid typos

While R is good software, it is still kind of stupid as it can't handle typos. It takes it on faith that you meant to type *exactly* what you did type. For example, suppose that you type `10 = 20` rather than `10 + 20`. Here's what happens:

```
> 10 = 20
Error in 10 = 20 : invalid (do_set) left-hand side to assignment
```

What's happened here is that R has attempted to interpret `10 = 20` as a command, and spits out an error message because the command doesn't make any sense to it. To some extent, I'm stating the obvious here, but it's important. The people who wrote R are smart. You, the user, are smart. But R itself is dumb. And because it's dumb, it has to be mindlessly obedient. It does *exactly* what you ask it to do. There is no equivalent to "autocorrect" in R.

Moreover, consider that R **is case-sensitive**. That means it makes a difference whether you use capital letters or not.

[1] Seriously. If you're in a position to do so, open up R and start typing. The simple act of typing it rather than "just reading" makes a big difference. It makes the concepts more concrete, and it ties the abstract ideas (programming and statistics) to the actual context in which you need to use them. Statistics is something you *do*, not just something you read about in a textbook.

Table 5.1: Basic arithmetic operations in R.

operation	operator	example input	example output
addition	`+`	`10 + 2`	12
subtraction	`-`	`9 - 3`	6
multiplication	`*`	`5 * 5`	25
division	`/`	`10 / 3`	3
power	`^`	`5 ^ 2`	25

5.1.2 R is (a bit) flexible with spacing

Albeit I stated that R is dumb. It is smart enough to ignore redundant spacing. For example, when I typed `10 + 20` before, I could equally have done this

```
> 10    + 20
[1] 30
```

or this

```
> 10+20
[1] 30
```

and I would get exactly the same answer. However, that doesn't mean that you can insert spaces in any place. For example, you cannot insert spaces in the middle of a function:

```
> setw d()
Error: unexpected symbol in "setw d"
```

5.1.3 R sometimes knows that you're not finished yet

If you type `10 +` and then press enter, R is smart enough to realize that you probably wanted to type in another number. So here's what happens:

```
> 10+
+
```

and there's a blinking cursor next to the plus sign. R "thinks" you're still typing your command, so it hasn't tried to execute it yet. Thus, this plus sign is actually another command prompt. It's different from the usual one (i.e., the `>` symbol). If I then go on to type `20` and hit enter, what I get is this:

```
> 10+
+ 20
[1] 30
```

Similarly, when you forgot to close the bracket R reminds you to do so:

```
> getwd(
+
+
+ )
```

5.2 Doing simple calculations with R

5.2.1 Adding, subtracting, multiplying and dividing

We already know R can do any kind of arithmetic calculation. Table 5.1 lists the arithmetic operators that correspond to the basic arithmetic we learned in primary school: addition, subtraction, multiplication and division. As you can see, R uses fairly standard symbols to denote each of the different operations you might want to perform: addition is done using the `+` operator, subtraction is performed by the `-` operator, and so on. So if I wanted to find out what 57 times 61 is (and who wouldn't?), I can use R instead of a calculator, like so:

```
> 57 * 61
[1] 3477
```

Please notice that

```
> 1 + 2 * 4
[1] 9
```

and

```
> 2 * 4 + 1
[1] 9
```

but

```
> (1 + 2) * 4
[1] 12
```

Thus, the **order of operations** plays a role. It's actually the same order that you got taught when you were at school: the "BEDMAS" order. That is, first calculate things inside **B**rackets `()`, then calculate **E**xponents `^`, then **D**ivision `/` and **M**ultiplication `*`, then **A**ddition `+` and **S**ubtraction `-`. Thus, better write:

5.3 Storing a number as a variable

One of the most important things to be able to do in R (or any programming language, for that matter) is to store information in **variables** or so called **objects**. At a conceptual level you can think of a variable as *label* for a certain piece of information, or even several different pieces of information. When doing statistical analysis in R all of your data (the variables you measured in your study) will be stored as variables in R. Let's look at the very basics for how we create variables and work with them.

5.3.1 The assignment operator `<-`

Suppose I'm trying to calculate how much money I'm going to make from this book. I agree, it is an unrealistic example but it will help you to understand R. Let's assume I'm only going to sell 350 copies. To create a variable called `sales` that assign a **value** to my variable `sales`, we need to use the **assignment operator** of R, which is `<-` as follows:

```
> sales <- 350
```

When you hit enter, R doesn't print out any output. (If you are using Rstudio, and the "environment" panel you can see that something happened there, can you?) It just gives you another command prompt. However, behind the scenes R has created a variable called `sales` and given it a value of `350`. You can check that this has happened by asking R to print the variable on screen. And the simplest way to do *that* is to type the name of the variable and hit enter.

```
> sales
[1] 350
```

Okay, so now we know how to assign variables and print out the value(s) of a variable.

Worth a mentioning is the curious features of R that there are several different ways of making assignments. In addition to the `<-` operator, we can also use `->` and `=`. If you want to use `->`, you might expect from just looking at the symbol you should write it like this:

```
> 350 -> sales
```

However, it is common practice to use `<-` and I recommend only to use this one because it is easier to read in scripts.

5.3.2 Doing calculations using variables

In addition to defining a `sales` variable that counts the number of copies I'm going to sell, I can also create a variable called `royalty`, indicating how much money I get per copy. Let's say that my royalties

are about $7 per book:

```
> sales <- 350
> royalty <- 7
```

The nice thing about variables (in fact, the whole point of having variables) is that we can do anything with a variable that we ought to be able to do with the information that it stores. That is, since R allows me to multiply `350` by `7`

```
> 350 * 7
[1] 2450
```

it also allows me to multiply `sales` by `royalty`

```
> sales * royalty
[1] 2450
```

As far as R is concerned, the `sales * royalty` command is the same as the `350 * 7` command. Not surprisingly, I can assign the output of this calculation to a new variable, which I'll call `revenue`. And when we do this, the new variable `revenue` gets the value `2450`. So let's do that, and then get R to print out the value of `revenue` so that we can verify that it's done what we asked:

```
> revenue <- sales * royalty
> revenue
[1] 2450
```

That's fairly straightforward. A slightly more subtle thing we can do is reassign the value of my variable, based on its current value. For instance, suppose that one of my students (no doubt under the influence of psychotropic drugs) loves the book so much that he or she donates me an extra $550. The simplest way to capture this is by a command like this:

```
> revenue <- revenue + 550
> revenue
[1] 3000
```

In this calculation, R has taken the old value of `revenue` (i.e., 2450) and added 550 to that value, producing a value of 3000. This new value is assigned to the `revenue` variable, overwriting its previous value. In any case, we now know that I'm expecting to make $3000 off this.

5.3.3 Rules and conventions for naming variables

Variables can be given almost any name, such as 'x', 'current_temperature', or 'subject_id'. However, there are some rules and suggestions you should keep in mind. Firstly, try to be consistent with the styling of your code (where you put spaces, how you name variable, etc.). In R, two popular style guides are *Hadley Wickham's style guide* (see: http://adv-r.had.co.nz/Style.html) and *Google's* style guide (see: http://web.stanford.edu/class/cs109l/unrestricted/resources/google-style.html). In the examples that we've seen so far, my variable names (`sales` and `revenue`) have just been English-language words written using lowercase letters. However, R allows a lot more flexibility when it comes to naming your variables, as the following list of rules illustrates:

- Make your names explicit and not too long.
- Avoid names starting with a number ('2x' is not valid but 'x2' is)
- Avoid names of fundamental functions in R (e.g., 'if', 'else', 'for', see https://statisticsglobe.com/r-functions-list/ for a complete list. In general, even if it's allowed, it's best to not use other function names (e.g., 'c', 'T', 'mean', 'data') as variable names. When in doubt check the help to see if the name is already in use.
- Avoid dots ('.') within a variable name as in 'my.dataset'. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them.
- Use nouns for object names and verbs for function names
- Keep in mind that R *is case sensitive* (e.g., 'genome_length' is different from 'Genome_length')

- Variable names can only use the upper case alphabetic characters `A-Z` as well as the lower case characters `a-z`. You can also include numeric characters `0-9` in the variable name, as well as the period `.` or underscore `_` character. In other words, you can use `SaL.e_s` as a variable name (though I can't think why you would want to), but you can't use `Sales?`.
- Variable names cannot include spaces: therefore `my sales` is not a valid name, but `my.sales` is.
- Variable names are case sensitive: that is, `Sales` and `sales` are *different* variable names.
- Variable names must start with a letter or a period. You can't use something like `_sales` or `1sales` as a variable name. You can use `.sales` as a variable name if you want, but it's not usually a good idea. By convention, variables starting with a `.` are used for special purposes, so you should avoid doing so.
- Variable names cannot be one of the reserved keywords. These are special names that R needs to keep "safe" from us mere users, so you can't use them as the names of variables. The keywords are: `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, and finally, `NA_character_`. Don't feel especially obliged to memorise these: if you make a mistake and try to use one of the keywords as a variable name, R will complain about it like the whiny little automaton it is.

In addition to those rules that R enforces, there are some informal conventions that people tend to follow when naming variables. One of them you've already seen: i.e., don't use variables that start with a period. But there are several others. You aren't obliged to follow these conventions, and there are many situations in which it's advisable to ignore them, but it's generally a good idea to follow them when you can:

- Use informative variable names. As a general rule, using meaningful names like `sales` and `revenue` is preferred over arbitrary ones like `variable1` and `variable2`. Otherwise it's very hard to remember what the contents of different variables are, and it becomes hard to understand what your commands actually do.
- Use short variable names. Typing is a pain and no-one likes doing it. So we much prefer to use a name like `sales` over a name like `sales.for.this.book.that.you.are.reading`. Obviously there's a bit of a tension between using informative names (which tend to be long) and using short names (which tend to be meaningless), so use a bit of common sense when trading off these two conventions.
- Use one of the conventional naming styles for multi-word variable names. Suppose I want to name a variable that stores "my new salary". Obviously I can't include spaces in the variable name, so how should I do this? There are three different conventions that you sometimes see R users employing. Firstly, you can separate the words using periods, which would give you `my.new.salary` as the variable name. Alternatively, you could separate words using underscores, as in `my_new_salary`. Finally, you could use capital letters at the beginning of each word (except the first one), which gives you `myNewSalary` as the variable name. I don't think there's any strong reason to prefer one over the other, but it's important to be consistent.

5.4 Using functions to do calculations

As we've seen, you can do quite a lot of calculations just by using arithmetic operators such as +, -, * etc. However, to do more advanced calculations, you're going to need to start using **functions**. For example, if you like to calculate

$$\sqrt{25} = 5$$

which is sometimes is written like this $25^{0.5} = 5$. you can use the operator `^`, just like we did earlier:

```
> 225 ^ 0.5
[1] 15
```

However, there's a second way that we can do this, since R also provides a **square root function**, `sqrt()`:

```
> sqrt( 225 )
[1] 15
```

When we use a function to do something, we generally refer to this as **calling** the function, and the values that we type into the function (there can be more than one) are referred to as the **arguments** of that function.

R offers many (more helpful) functions such as the **absolute value function** which converts negative numbers to positive numbers, and leaves positive numbers alone. Mathematically, the absolute value of x is written $|x|$ or sometimes abs(x). Calculating absolute values in R is done with the `abs()` function:

```
> abs( -13 )
[1] 13
```

It's worth noting that – in the same way that R allows us to put multiple operations together into a longer command, it also lets us put functions together and even combine functions with operators. For example:

```
> sqrt( 1 + abs(-8) )
[1] 3
```

When R executes this command, starts out by calculating the value of `abs(-8)`, which produces an intermediate value of `8`. Having done so, the command simplifies to `sqrt( 1 + 8 )`. To solve the square root it first needs to add `1 + 8` to get `9`, at which point it evaluates `sqrt(9)`, and so it finally outputs a value of `3`.

5.4.1 Function arguments, their names and their defaults

There's two more important things that you need to understand about how functions work in R, and that's the use of "named" arguments, and "default values" for arguments. To understand what these two concepts are all about, I'll introduce another function. The `round()` function can be used to round some value to the nearest whole number. For example, I could type this:

```
> round( 3.1415 )
[1] 3
```

However, suppose I only wanted to round it to two decimal places: that is, I want to get `3.14` as the output. The `round()` function supports this, by allowing you to input a second argument to the function that specifies the number of decimal places that you want to round the number to. In other words, I could do this:

```
> round( 3.14165, 2 )
[1] 3.14
```

What's happening here is that I've specified *two* arguments: the first argument is the number that needs to be rounded (i.e., `3.1415`), the second argument is the number of decimal places that it should be rounded to (i.e., `2`), and the two arguments are separated by a comma. In this simple example, it's quite easy to remember which one argument comes first and which one comes second, but for more complicated functions this is not easy. Fortunately, most R functions make use of **argument names**. For the `round()` function, for example the number that needs to be rounded is specified using the `x` argument, and the number of decimal points that you want it rounded to is specified using the `digits` argument. Because we have these names available to us, we can specify the arguments to the function by name. We do so like this:

```
> round( x = 3.1415, digits = 2 )
[1] 3.14
```

Notice that this is kind of similar in spirit to variable assignment (Section 5.3), except that I used `=` here, rather than `<-`. In both cases we're specifying specific values to be associated with a label. However, there are some differences between what I was doing earlier on when creating variables, and what I'm doing here when specifying arguments, and so as a consequence it's important that you use `=` in this context.

As you can see, specifying the arguments by name involves a lot more typing, but it's also a lot easier to read. Because of this, the commands in this book will usually specify arguments by name, since that makes it clearer to you what I'm doing. However, one important thing to note is that when specifying the arguments using their names, it doesn't matter what order you type them in. But if you don't use the argument names, then you have to input the arguments in the correct order. In other words, these three commands all produce the same output...

```
> round( 3.14165, 2 )
> round( x = 3.1415, digits = 2 )
> round( digits = 2, x = 3.1415 )
```

but this one does not...

```
> round( 2, 3.14165 )
```

How do you find out what the correct order is? There's a few different ways, but the easiest one is to look at the help documentation for the function (see Section **??**). However, if you're ever unsure, it's probably best to actually type in the argument name.

Okay, so that's the first thing I said you'd need to know: argument names. The second thing you need to know about is default values. Notice that the first time I called the `round()` function I didn't actually specify the `digits` argument at all, and yet R somehow knew that this meant it should round to the nearest whole number. How did that happen? The answer is that the `digits` argument has a **default value** of `0`, meaning that if you decide not to specify a value for `digits` then R will act as if you had typed `digits = 0`. This is quite handy: the vast majority of the time when you want to round a number you want to round it to the nearest whole number, and it would be pretty annoying to have to specify the `digits` argument every single time. On the other hand, sometimes you actually do want to round to something other than the nearest whole number, and it would be even more annoying if R didn't allow this! Thus, by having `digits = 0` as the default value, we get the best of both worlds.

Exercise 5.1 — Calculate (Solution → p. **??**)

Write the code to...
a) ... ***calculate and store*** the result of the following calculation:

$$\sqrt{8} \cdot \left(\frac{3}{4} + 7^{-2} \right) \cdot 9 : 2$$

b) ...round the result up (three digits) and store the result in an object entitled `result_rounded`

5.5 Letting RStudio help you with your commands

At this stage you know how to type in basic commands, including how to use R functions. And it's probably beginning to dawn on you that there are a *lot* of R functions, all of which have their own arguments. You're probably also worried that you're going to have to remember all of them! Thankfully, it's not that bad. In fact, very few data analysts bother to try to remember all the commands. What they really do is use tricks to make their lives easier. The first (and arguably most important one) is to use the internet. If you don't know how a particular R function works: Google it (or use `www.rseek.org`)! Second, you can look up the R help documentation. Third, you can use the *autocomplete service* of RStudio. Fourth, you can use the *History* privided by RStudio.

Cheatsheets: Check out these **cheatsheets**: https://rstudio.com/resources/cheatsheets/

5.5.1 Autocomplete using "tab"

The first thing I want to call your attention to is the *autocomplete* ability in RStudio. Let's stick to our example above and assume that what you want to do is to round a number. This time around, start

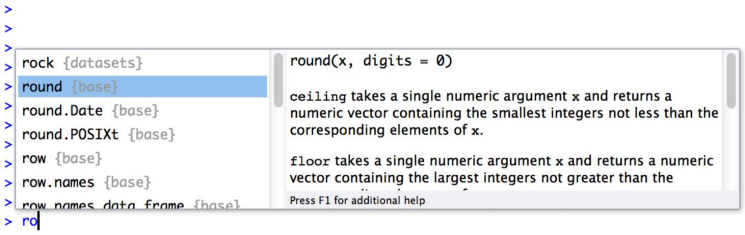


Figure 5.1: Start typing the name of a function or a variable, and hit the "tab" key. RStudio brings up a little dialog box like this one that lets you select the one you want, and even prints out a little information about it.

typing the name of the function that you want, and then hit the "tab" key. RStudio will then display a little window like the one shown in Figure 5.1. In this figure, I've typed the letters `ro` at the command line, and then hit tab. The window has two panels. On the left, there's a list of variables and functions that start with the letters that I've typed shown in black text, and some grey text that tells you where that variable/function is stored. Ignore the grey text for now: it won't make much sense to you until we've talked about packages in Section 6.2. In Figure 5.1 you can see that there's quite a few things that start with the letters `ro`: there's something called `rock`, something called `round`, something called `round.Date` and so on. The one we want is `round`, but if you're typing this yourself you'll notice that when you hit the tab key the window pops up with the top entry (i.e., `rock`) highlighted. You can use the up and down arrow keys to select the one that you want. Or, if none of the options look right to you, you can hit the escape key ("esc") or the left arrow key to make the window go away.

In our case, the thing we want is the `round` option, so we'll select that. When you do this, you'll see that the panel on the right changes. Previously, it had been telling us something about the `rock` data set (i.e., "Measurements on 48 rock samples...") that is distributed as part of R. But when we select `round`, it displays information about the `round()` function, exactly as it is shown in Figure 5.1. This display is really handy. The very first thing it says is `round(x, digits = 0)`: what this is telling you is that the `round()` function has two arguments. The first argument is called `x`, and it doesn't have a default value. The second argument is `digits`, and it has a default value of 0. In a lot of situations, that's all the information you need. But RStudio goes a bit further, and provides some additional information about the function underneath. Sometimes that additional information is very helpful, sometimes it's not: RStudio pulls that text from the R help documentation, and my experience is that the helpfulness of that documentation varies wildly. Anyway, if you've decided that `round()` is the function that you want to use, you can hit the right arrow or the enter key, and RStudio will finish typing the rest of the function name for you.

The RStudio autocomplete tool works slightly differently if you've already got the name of the function typed and you're now trying to type the arguments. For instance, suppose I've typed `round(` into the console, and *then* I hit tab. RStudio is smart enough to recognize that I already know the name of the function that I want, because I've already typed it! Instead, it figures that what I'm interested in is the *arguments* to that function. So that's what pops up in the little window. You can see this in Figure 5.2. Again, the window has two panels, and you can interact with this window in exactly the same way that you did with the window shown in Figure 5.1. On the left hand panel, you can see a list of the argument names. On the right hand side, it displays some information about what the selected argument does.

5.5.2 Browsing your command history

One thing that R does automatically is keep track of your "command history". That is, it remembers all the commands that you've previously typed. You can access this history in a few different ways. The simplest way is to use the up and down arrow keys. If you hit the up key, the R console will show you the most recent command that you've typed. Hit it again, and it will show you the command before that. If you want the text on the screen to go away, hit escape. Incidentally, that always works: if you've started typing a command and you want to clear it and start again, hit escape. Using the up

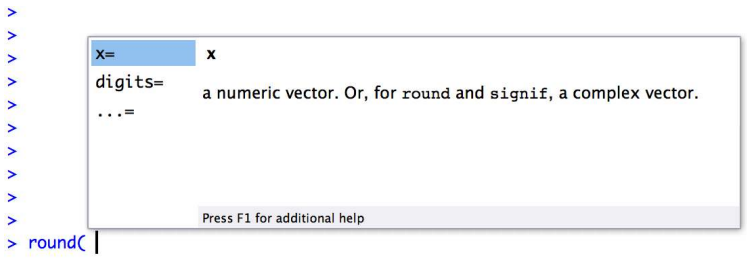


Figure 5.2: If you've typed the name of a function already along with the left parenthesis and then hit the "tab" key, RStudio brings up a different window to the one shown in Figure 5.1. This one lists all the arguments to the function on the left, and information about each argument on the right.

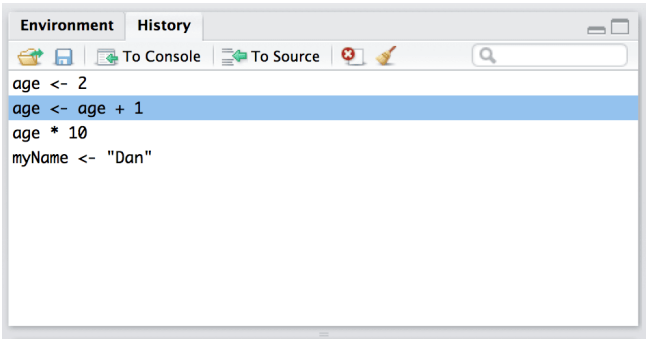


Figure 5.3: The history panel is located in the top right hand side of the Rstudio window. Click on the word "History" and it displays this panel.

and down keys can be really handy if you've typed a long command that had one typo in it. Rather than having to type it all again from scratch, you can use the up key to bring up the command and fix it.

The second way to get access to your command history is to look at the history panel in RStudio. On the upper right hand side of the RStudio window you'll see a tab labelled "History". Click on that, and you'll see a list of all your recent commands displayed in that panel: it should look something like Figure 5.3. If you double click on one of the commands, it will be copied to the R console. (You can achieve the same result by selecting the command you want with the mouse and then clicking the "To Console" button).

Another method is to start typing some text and then hit the Control key and the up arrow together (on Windows or Linux) or the Command key and the up arrow together (on a Mac). This will bring up a window showing all your recent commands that started with the same text as what you've currently typed. That can come in quite handy sometimes.

5.6 Storing many numbers as a vector

When I introduced variables in Section 5.3, I showed you how we can use variables to store a single number. In this section, we'll extend this idea and look at how to store multiple numbers within the one variable. In R the name for a variable that can store multiple values is a **vector**.

5.6.1 Creating a vector

Let's get bach to my silly "get rich by textbook writing" example. Suppose the textbook company sends me sales data on a monthly basis. Since my class start in late February, we might expect most of the sales to occur towards the start of the year. Let's suppose that I have 100 sales in February, 200 sales in March and 50 sales in April, and no other sales for the rest of the year. What I would like to do is have a variable – let's call it `sales.by.month` – that stores all this sales data. The first number stored should be `0` since I had no sales in January, the second should be `100`, and so on. The simplest

way to do this in R is to use the **combine** function, `c()`. To do so, all we have to do is type all the numbers you want to store in a comma separated list, like this:

```
> sales.by.month <- c(0, 100, 200, 50, 0, 0, 0, 0, 0, 0, 0, 0)
> sales.by.month
 [1]   0 100 200  50   0   0   0   0   0   0   0   0
```

To use the correct terminology here, we have a single variable here called `sales.by.month`: this variable is a vector that consists of 12 **elements**. Notice that I didn't specify any argument names here. The `c()` function is one of those cases where we don't use names. We just type all the numbers, and R just dumps them all in a single variable.

5.6.2 Showing the output

Now that we've learned how to put information into a vector, the next thing to understand is how to pull that information back out again. However, before I do so it's worth taking a slight detour. If you've been following along, typing all the commands into R yourself, it's possible that the output that you saw when we printed out the `sales.by.month` vector was slightly different to what I showed above. This would have happened if the window (or the Rstudio panel) that contains the R console is really, really narrow. If that were the case, you might have seen output that looks something like this:

```
> sales.by.month
 [1]   0 100 200  50   0   0   0   0
 [9]   0   0   0   0
```

Because there wasn't much room on the screen, R has printed out the results over two lines. But that's not the important thing to notice. The important point is that the first line has a `[1]` in front of it, whereas the second line starts with `[9]`. It's pretty clear what's happening here. For the first row, R has printed out the 1st element through to the 8th element, so it starts that row with a `[1]`. For the second row, R has printed out the 9th element of the vector through to the 12th one, and so it begins that row with a `[9]` so that you can tell where it's up to at a glance. It might seem a bit odd to you that R does this, but in some ways it's a kindness, especially when dealing with larger data sets!

5.6.3 Getting information out of vectors

To get back to the main story, let's consider the problem of how to get information out of a vector. At this point, you might have a sneaking suspicion that the answer has something to do with the `[1]` and `[9]` things that R has been printing out. And of course you are correct. Suppose I want to pull out the February sales data only. February is the second month of the year, so let's try this:

```
> sales.by.month[2]
[1] 100
```

Yep, that's the February sales all right. But there's a subtle detail to be aware of here: notice that R outputs `[1] 100`, *not* `[2] 100`. This is because R is being extremely literal. When we typed in `sales.by.month[2]`, we asked R to find exactly *one* thing, and that one thing happens to be the second element of our `sales.by.month` vector. So, when it outputs `[1] 100` what R is saying is that the first number *that we just asked for* is `100`. This behaviour makes more sense when you realise that we can use this trick to create new variables. For example, I could create a `february.sales` variable like this:

```
> february.sales <- sales.by.month[2]
> february.sales
[1] 100
```

Obviously, the new variable `february.sales` should only have one element and so when I print it out this new variable, the R output begins with a `[1]` because `100` is the value of the first (and only) element of `february.sales`. The fact that this also happens to be the value of the second element of `sales.by.month` is irrelevant. We'll pick this topic up again shortly (Section 5.9).

5.6.4 Altering the elements of a vector

Sometimes you'll want to change the values stored in a vector. Imagine my surprise when the publisher rings me up to tell me that the sales data for May are wrong. There were actually an additional 25 books sold in May, but there was an error or something so they hadn't told me about it. How can I fix my `sales.by.month` variable? One possibility would be to assign the whole vector again from the beginning, using `c()`. But that's a lot of typing. Also, it's a little wasteful: why should R have to redefine the sales figures for all 12 months, when only the 5th one is wrong? Fortunately, we can tell R to change only the 5th element, using this trick:

```
> sales.by.month[5] <- 25
> sales.by.month
 [1]   0 100 200  50  25   0   0   0   0   0   0   0
```

Another way to edit variables is to use the `fix()` function. I won't discuss them in detail right now, but you can check them out on your own.

5.6.5 Useful things to know about vectors

Before moving on, I want to mention a couple of other things about vectors. Firstly, you often find yourself wanting to know how many elements there are in a vector (usually because you've forgotten). You can use the `length()` function to do this. It's quite straightforward:

```
> length( x = sales.by.month )
[1] 12
```

Secondly, you often want to alter all of the elements of a vector at once. For instance, suppose I wanted to figure out how much money I made in each month. Since I'm earning an exciting $7 per book, what I want to do is multiply each element in the `sales.by.month` vector by `7`. R makes this pretty easy, as the following example shows:

```
> sales.by.month * 7
 [1]    0  700 1400  350    0    0    0    0    0    0    0    0
```

In other words, when you multiply a vector by a single number, all elements in the vector get multiplied. The same is true for addition, subtraction, division and taking powers. So that's neat. On the other hand, suppose I wanted to know how much money I was making per day, rather than per month. Since not every month has the same number of days, I need to do something slightly different. Firstly, I'll create two new vectors:

```
> days.per.month <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
> profit <- sales.by.month * 7
```

Obviously, the `profit` variable is the same one we created earlier, and the `days.per.month` variable is pretty straightforward. What I want to do is divide every element of `profit` by the *corresponding* element of `days.per.month`. Again, R makes this pretty easy:

```
> profit / days.per.month
 [1]  0.00000 25.00000 45.16129 11.66667  0.00000  0.00000  0.00000  0.00000  0.00000
[10]  0.00000  0.00000  0.00000
```

I still don't like all those zeros, but that's not what matters here. Notice that the second element of the output is 25, because R has divided the second element of `profit` (i.e. 700) by the second element of `days.per.month` (i.e. 28). Similarly, the third element of the output is equal to 1400 divided by 31, and so on. We'll talk more about calculations involving vectors later on, but that's enough detail for now.

5.7 Storing text data

A lot of the time your data will be numeric in nature, but not always. Sometimes your data really needs to be described using text, not using numbers. To address this, we need to consider the situation where our variables store text. To create a variable that stores the word "hello", we can type this:

```
> greeting <- "hello"
> greeting
[1] "hello"
```

When interpreting this, it's important to recognise that the quote marks here *aren't* part of the string itself. They're just something that we use to make sure that R knows to treat the characters that they enclose as a piece of text data, known as a **character string**. In other words, R treats `"hello"` as a string containing the word "hello"; but if I had typed `hello` instead, R would go looking for a variable by that name! You can also use `'hello'` to specify a character string.

Okay, so that's how we store the text. Next, it's important to recognise that when we do this, R stores the entire word `"hello"` as a *single* element: our `greeting` variable is not a vector of five different letters. Rather, it has only the one element, and that element corresponds to the entire character string `"hello"`. To illustrate this, if I actually ask R to find the first element of `greeting`, it prints the whole string:

```
> greeting[1]
[1] "hello"
```

Of course, there's no reason why I can't create a vector of character strings. For instance, if we were to continue with the example of my attempts to look at the monthly sales data for my book, one variable I might want would include the names of all 12 `months`. To do so, I could type in a command like this

```
> months <- c("January", "February", "March", "April", "May", "June",
+             "July", "August", "September", "October", "November", "December")
> months
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

This is a **character vector** containing 12 elements, each of which is the name of a month. So if I wanted R to tell me the name of the fourth month, all I would do is this:

```
> months[4]
[1] "April"
```

Working with text

Working with text data is somewhat more complicated than working with numeric data. For purposes of the current chapter we only need this bare bones sketch. The only other thing I want to do before moving on is show you an example of a function that can be applied to text data. So far, most of the functions that we have seen (i.e., `sqrt()`, `abs()` and `round()`) only make sense when applied to numeric data (e.g., you can't calculate the square root of "hello"), and we've seen one function that can be applied to pretty much any variable or vector (i.e., `length()`). So it might be nice to see an example of a function that can be applied to text.

The function I'm going to introduce you to is called `nchar()`, and what it does is count the number of individual characters that make up a string. Recall earlier that when we tried to calculate the `length()` of our `greeting` variable it returned a value of `1`: the `greeting` variable contains only the one string, which happens to be `"hello"`. But what if I want to know how many letters there are in the word? Sure, I could *count* them, but that's boring, and more to the point it's a terrible strategy if what I wanted to know was the number of letters in *War and Peace*. That's where the `nchar()` function is helpful:

```
> nchar( x = greeting )
[1] 5
```

That makes sense, since there are in fact 5 letters in the string `"hello"`. Better yet, you can apply `nchar()` to whole vectors. So, for instance, if I want R to tell me how many letters there are in the names of each of the 12 months, I can do this:

```
> nchar( x = months )
 [1] 7 8 5 5 3 4 4 6 9 7 8 8
```

So that's nice to know. The `nchar()` function can do a bit more than this, and there's a lot of other

functions that you can do to extract more information from text or do all sorts of fancy things. However, the goal here is not to teach any of that! The goal right now is just to see an example of a function that actually does work when applied to text.

5.8 Storing "true or false" data

Time to move onto a third kind of data. A key concept in that a lot of R relies on is the idea of a **logical value**. A logical value is an assertion about whether something is true or false. This is implemented in R in a pretty straightforward way. There are two logical values, namely `TRUE` and `FALSE`. Despite the simplicity, a logical values are very useful things. Let's see how they work.

5.8.1 Assessing mathematical truths

In George Orwell's classic book *1984*, one of the slogans used by the totalitarian Party was "two plus two equals five", the idea being that the political domination of human freedom becomes complete when it is possible to subvert even the most basic of truths. It's a terrifying thought, especially when the protagonist Winston Smith finally breaks down under torture and agrees to the proposition. R has rather firm opinions on the topic of what is and isn't true, at least as regards basic mathematics. If I ask it to calculate `2 + 2`, it always gives the same answer, and it's not 5:

```
> 2 + 2
[1] 4
```

Of course, so far R is just doing the calculations. I haven't asked it to explicitly assert that $2 + 2 = 4$ is a true statement. If I want R to make an explicit judgement, I can use a command like this:

```
> 2 + 2 == 4
[1] TRUE
```

What I've done here is use the **equality operator**, `==`, to force R to make a "true or false" judgment. Note that this is a very different operator to the assignment operator `=` that I talked about in Section 5.3. A common typo that people make when trying to write logical commands in R (or other languages, since the "`=` versus `==`" distinction is important in most programming languages) is to accidentally type `=` when you really mean `==`.

Okay, let's see what R thinks of the Party slogan:

```
> 2+2 == 5
[1] FALSE
```

What a relief! Or something like that. Anyway, it's worth having a look at what happens if I try to *force* R to believe that two plus two is five by making an assignment statement like `2 + 2 = 5` or `2 + 2 <- 5`. When I do this, here's what happens:

```
> 2 + 2 = 5
Error in 2 + 2 = 5 : target of assignment expands to non-language object
```

R doesn't like this very much. It recognises that `2 + 2` is *not* a variable (that's what the "non-language object" part is saying), and it won't let you try to "reassign" it. While R is pretty flexible, and actually does let you do some quite remarkable things to redefine parts of R itself, there are just some basic, primitive truths that it refuses to give up. It won't change the laws of addition, and it won't change the definition of the number `2`.

That's probably for the best.

5.8.2 Logical operations

So now we've seen logical operations at work, but so far we've only seen the simplest possible example. You probably won't be surprised to discover that we can combine logical operations with other operations and functions in a more complicated way, like this:

```
> 3*3 + 4*4 == 5*5
[1] TRUE
```

Table 5.2: Some logical operators. Technically I should call these "binary relational operators".

operation	operator	example input	answer
less than	`<`	`2 < 3`	`TRUE`
less than or equal to	`<=`	`2 <= 2`	`TRUE`
greater than	`>`	`2 > 3`	`FALSE`
greater than or equal to	`>=`	`2 >= 2`	`TRUE`
equal to	`==`	`2 == 3`	`FALSE`
not equal to	`!=`	`2 != 3`	`TRUE`
not	`!`	`!(1==1)`	`FALSE`
or	`|`	`(1==1) | (2==3)`	`TRUE`
and	`&`	`(1==1) & (2==3)`	`FALSE`

or this

```
> sqrt( 25 ) == 5
[1] TRUE
```

Not only that, but as Table 5.2 illustrates, there are several other logical operators that you can use, corresponding to some basic mathematical concepts. Hopefully these are all pretty self-explanatory: for example, the **less than** operator `<` checks to see if the number on the left is less than the number on the right. If it's less, then R returns an answer of `TRUE`:

```
> 99 < 100
[1] TRUE
```

but if the two numbers are equal, or if the one on the right is larger, then R returns an answer of `FALSE`, as the following two examples illustrate:

```
> 100 < 100
[1] FALSE
> 100 < 99
[1] FALSE
```

In contrast, the **less than or equal to** operator `<=` will do exactly what it says. It returns a value of `TRUE` if the number of the left hand side is less than or equal to the number on the right hand side. So if we repeat the previous two examples using `<=`, here's what we get:

```
> 100 <= 100
[1] TRUE
> 100 <= 99
[1] FALSE
```

And at this point I hope it's pretty obvious what the **greater than** operator `>` and the **greater than or equal to** operator `>=` do! Next on the list of logical operators is the **not equal to** operator `!=` which – as with all the others – does what it says it does. It returns a value of `TRUE` when things on either side are not identical to each other. Therefore, since $2 + 2$ isn't equal to 5, we get:

```
> 2 + 2 != 5
[1] TRUE
```

We're not quite done yet. There are three more logical operations that are worth knowing about, listed in Table 5.2. These are the **not** operator `!`, the **and** operator `&`, and the **or** operator `|`. Like the other logical operators, their behaviour is more or less exactly what you'd expect given their names. For instance, if I ask you to assess the claim that "either $2 + 2 = 4$ *or* $2 + 2 = 5$" you'd say that it's true. Since it's an "either-or" statement, all we need is for one of the two parts to be true. That's what the `|` operator does:

```
> (2+2 == 4) | (2+2 == 5)
[1] TRUE
```

On the other hand, if I ask you to assess the claim that "both $2 + 2 = 4$ *and* $2 + 2 = 5$" you'd say that it's false. Since this is an *and* statement we need both parts to be true. And that's what the `&`

operator does:

```
> (2+2 == 4) & (2+2 == 5)
[1] FALSE
```

Finally, there's the *not* operator, which is simple but annoying to describe in English. If I ask you to assess my claim that "it is not true that $2 + 2 = 5$" then you would say that my claim is true; because my claim is that "$2 + 2 = 5$ is false". And I'm right. If we write this as an R command we get this:

```
> ! (2+2 == 5)
[1] TRUE
```

In other words, since `2+2 == 5` is a `FALSE` statement, it must be the case that `!(2+2 == 5)` is a `TRUE` one. Essentially, what we've really done is claim that "not false" is the same thing as "true". Obviously, this isn't really quite right in real life. But R lives in a much more black or white world: for R everything is either true or false. No shades of gray are allowed. We can actually see this much more explicitly, like this:

```
> ! FALSE
[1] TRUE
```

Of course, in our $2 + 2 = 5$ example, we didn't really need to use "not" `!` and "equals to" `==` as two separate operators. We could have just used the "not equals to" operator `!=` like this:

```
> 2+2 != 5
[1] TRUE
```

But there are many situations where you really do need to use the `!` operator.

5.8.3 Storing and using logical data

Up to this point, I've introduced *numeric data* (in Sections 5.3 and 5.6) and *character data* (in Section 5.7). So you might not be surprised to discover that these `TRUE` and `FALSE` values that R has been producing are actually a third kind of data, called *logical data*. That is, when I asked R if `2 + 2 == 5` and it said `[1] FALSE` in reply, it was actually producing information that we can store in variables. For instance, I could create a variable called `is.the.Party.correct`, which would store R's opinion:

```
> is.the.Party.correct <- 2 + 2 == 5
> is.the.Party.correct
[1] FALSE
```

Alternatively, you can assign the value directly, by typing `TRUE` or `FALSE` in your command. Like this:

```
> is.the.Party.correct <- FALSE
> is.the.Party.correct
[1] FALSE
```

Better yet, because it's kind of tedious to type `TRUE` or `FALSE` over and over again, R provides you with a shortcut: you can use `T` and `F` instead (but it's case sensitive: `t` and `f` won't work). Warning! `TRUE` and `FALSE` are reserved keywords in R, so you can trust that they always mean what they say they do. Unfortunately, the shortcut versions `T` and `F` do not have this property. It's even possible to create variables that set up the reverse meanings, by typing commands like `T <- FALSE` and `F <- TRUE`. This is kind of insane, and something that is generally thought to be a design flaw in R. Anyway, the long and short of it is that it's safer to use `TRUE` and `FALSE`. So this works:

```
> is.the.Party.correct <- F
> is.the.Party.correct
[1] FALSE
```

but this doesn't:

```
> is.the.Party.correct <- f
Error: object 'f' not found
```

5.8.4 Vectors of logicals

The next thing to mention is that you can store vectors of logical values in exactly the same way that you can store vectors of numbers (Section 5.6) and vectors of text data (Section 5.7). Again, we can define them directly via the `c()` function, like this:

```
> x <- c(TRUE, TRUE, FALSE)
> x
[1]  TRUE  TRUE FALSE
```

or you can produce a vector of logicals by applying a logical operator to a vector. This might not make a lot of sense to you, so let's unpack it slowly. First, let's suppose we have a vector of numbers (i.e., a "non-logical vector"). For instance, we could use the `sales.by.month` vector that we were using in Section 5.6. Suppose I wanted R to tell me, for each month of the year, whether I actually sold a book in that month. I can do that by typing this:

```
> sales.by.month > 0
 [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

and again, I can store this in a vector if I want, as the example below illustrates:

```
> any.sales.this.month <- sales.by.month > 0
> any.sales.this.month
 [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

In other words, `any.sales.this.month` is a logical vector whose elements are `TRUE` only if the corresponding element of `sales.by.month` is greater than zero. For instance, since I sold zero books in January, the first element is `FALSE`.

5.8.5 Applying logical operation to text

In a moment (Section 5.9) I'll show you why these logical operations and logical vectors are so handy, but before I do so I want to very briefly point out that you can apply them to text as well as to logical data. It's just that we need to be a bit more careful in understanding how R interprets the different operations. In this section I'll talk about how the equal to operator `==` applies to text, since this is the most important one. Obviously, the not equal to operator `!=` gives the exact opposite answers to `==` so I'm implicitly talking about that one too, but I won't give specific commands showing the use of `!=`.

Okay, let's see how it works. In one sense, it's very simple. For instance, I can ask R if the word `"cat"` is the same as the word `"dog"`, like this:

```
> "cat" == "dog"
[1] FALSE
```

That's pretty obvious, and it's good to know that even R can figure that out. Similarly, R does recognise that a `"cat"` is a `"cat"`:

```
> "cat" == "cat"
[1] TRUE
```

Again, that's exactly what we'd expect. However, what you need to keep in mind is that R is not at all tolerant when it comes to grammar and spacing. If two strings differ in any way whatsoever, R will say that they're not equal to each other, as the following examples indicate:

```
> " cat" == "cat"
[1] FALSE
> "cat" == "CAT"
[1] FALSE
> "cat" == "c a t"
[1] FALSE
```

5.9 Indexing vectors

So far, whenever I've had to get information out of a vector, all I've done is typed something like `months[4]`; and when I do this R prints out the fourth element of the `months` vector. In this section,

I'll show you two additional tricks for getting information out of the vector.

5.9.1 Extracting multiple elements

One very useful thing we can do is pull out more than one element at a time. In the previous example, we only used a single number (i.e., `2`) to indicate which element we wanted. Alternatively, we can use a vector. So, suppose I wanted the data for February, March and April. What I could do is use the vector `c(2,3,4)` to indicate which elements I want R to pull out. That is, I'd type this:

```
> sales.by.month[ c(2,3,4) ]
[1] 100 200  50
```

Notice that the order matters here. If I asked for the data in the reverse order (i.e., April first, then March, then February) by using the vector `c(4,3,2)`, then R outputs the data in the reverse order:

```
 > sales.by.month[ c(4,3,2) ]
[1]  50 200 100
```

A second thing to be aware of is that R provides you with handy shortcuts for very common situations. For instance, suppose that I wanted to extract everything from the 2nd month through to the 8th month. One way to do this is to do the same thing I did above, and use the vector `c(2,3,4,5,6,7,8)` to indicate the elements that I want. That works just fine

```
> sales.by.month[ c(2,3,4,5,6,7,8) ]
[1] 100 200  50   0   0   0   0
```

but it's kind of a lot of typing. To help make this easier, R lets you use `2:8` as shorthand for `c(2,3,4,5,6,7,8)`, which makes things a lot simpler. First, let's just check that this is true:

```
> 2:8
[1] 2 3 4 5 6 7 8
```

Next, let's check that we can use the `2:8` shorthand as a way to pull out the 2nd through 8th elements of `sales.by.months`:

```
> sales.by.month[2:8]
[1] 100 200  50   0   0   0   0
```

So that's kind of neat.

5.9.2 Logical indexing

At this point, I can introduce an extremely useful tool called **logical indexing**. In the last section, I created a logical vector `any.sales.this.month`, whose elements are `TRUE` for any month in which I sold at least one book, and `FALSE` for all the others. However, that big long list of `TRUE`s and `FALSE`s is a little bit hard to read, so what I'd like to do is to have R select the names of the `months` for which I sold any books. Earlier on, I created a vector `months` that contains the names of each of the months. This is where logical indexing is handy. What I need to do is this:

```
> months[ sales.by.month > 0 ]
[1] "February" "March"    "April"    "May"
```

To understand what's happening here, it's helpful to notice that `sales.by.month > 0` is the same logical expression that we used to create the `any.sales.this.month` vector in the last section. In fact, I could have just done this:

```
> months[ any.sales.this.month ]
[1] "February" "March"    "April"    "May"
```

and gotten exactly the same result. In order to figure out which elements of `months` to include in the output, what R does is look to see if the corresponding element in `any.sales.this.month` is `TRUE`. Thus, since element 1 of `any.sales.this.month` is `FALSE`, R does not include `"January"` as part of the output; but since element 2 of `any.sales.this.month` is `TRUE`, R does include `"February"` in the output. Note that there's no reason why I can't use the same trick to find the actual sales numbers for those months. The command to do that would just be this:

```
> sales.by.month [ sales.by.month > 0 ]
[1] 100 200  50  25
```

In fact, we can do the same thing with text. Here's an example. Suppose that I later find out that the bookshop only had sufficient stocks for a few months of the year. They tell me that early in the year they had `"high"` stocks, which then dropped to `"low"` levels, and in fact for one month they were `"out"` of copies of the book for a while before they were able to replenish them. Thus, I might have a variable called `stock.levels` which looks like this:

```
> stock.levels
 [1] "high" "high" "low"  "out"  "out"  "high" "high" "high" "high" "high" "high"
[12] "high"
```

Thus, if I want to know the months for which the bookshop was out of my book, I could apply the logical indexing trick, but with the character vector `stock.levels`, like this:

```
> months[stock.levels == "out"]
[1] "April" "May"
```

Alternatively, if I want to know when the bookshop was either low on copies or out of copies, I could do this:

```
> months[stock.levels == "out" | stock.levels == "low"]
[1] "March" "April" "May"
```

or this

```
> months[stock.levels != "high" ]
[1] "March" "April" "May"
```

Either way, I get the answer I want.

At this point, I hope you can see why logical indexing is such a useful thing. It's a very basic, yet very powerful way to manipulate data. We do not talk about how to manipulate data in this short introduction.

5.10 Quitting

Assuming you're running R in the usual way (i.e., through RStudio or the default GUI on a Windows or Mac computer), then you can just shut down the application in the normal way. However, R also has a function, called `q()` that you can use to quit.

Regardless of what method you use to quit R, when you do so for the first time R will probably ask you if you want to save the "workspace image". We'll talk a lot more about loading and saving data in Section 6.5, but I figured we'd better quickly cover this now otherwise you're going to get annoyed when you close R at the end of the chapter. If you're using RStudio, you'll see a dialog box that asks you *Save workspace ...?* and if you're using a text based interface you'll see this:

```
> q()
Save workspace image? [y/n/c]:
```

The `y/n/c` part here is short for "yes / no / cancel". Type `y` if you want to save, `n` if you don't, and `c` if you've changed your mind and you don't want to quit after all.

What does this actually *mean*? What's going on is that R wants to know if you want to save all those variables that you've been creating, so that you can use them later. This sounds like a great idea, so it's really tempting to type `y` or click the "Save" button. To be honest though, I very rarely do this, and it kind of annoys me a little bit... what R is *really* asking is if you want it to store these variables in a "default" data file, which it will automatically reload for you next time you open R. And quite frankly, if I'd wanted to save the variables, then I'd have already saved them before trying to quit. Not only that, I'd have saved them to a location of *my* choice, so that I can find it again later.

5.11 Summary

This chapter covered the following topics:

- *Getting started.* We downloaded and installed R and Rstudio (Section 3).
- *Basic commands.* We talked a bit about the logic of how R works and in particular how to type commands into the R console (Section 5.1), and in doing so learned how to perform basic calculations using the arithmetic operators `+`, `-`, `*`, `/` and `^`. (Section 5.2)
- *Introduction to functions.* We saw several different functions, three that are used to perform numeric calculations (`sqrt()`, `abs()`, `round()`; Section 5.4), one that applies to text (`nchar()`; Section 5.7), and one that works on any variable (`length()`; Section 5.6.5). In doing so, we talked a bit about how argument names work, and learned about default values for arguments. (Section 5.4.1)
- *Introduction to variables.* We learned the basic idea behind variables, and how to assign values to variables using the assignment operator `<-` (Section 5.3). We also learned how to create vectors using the combine function `c()`. (Section 5.6)
- *Data types.* Learned the distinction between numeric, character and logical data; including the basics of how to enter and use each of them. (Sections 5.3 to 5.8)
- *Logical operations.* Learned how to use the logical operators `==`, `!=`, `<`, `>`, `<=`, `=>`, `!`, `&` and `|`. (Section 5.8). And learned how to use logical indexing. (Section 5.9)

Chapter 6

Additional R concepts

There are still quite a few things that we need to talk about now, otherwise you'll run into problems when trying to work with data and do statistics. The goal of this chapter is to build on the introductory content from the last chapter, **so that you get to the point that you can start learning, using, and discover R by your own**. The chapter comes in two parts. The first half is devoted to the "mechanics" of R:

- installing and loading packages,
- managing the workspace,
- navigating the file system, and
- loading and saving data.

The second half is about the different kinds of variables in R, i.e.:

- factors,
- data frames and
- formulas.

Moreover, we discuss how to get help in R and thorough platforms. The chapter should provide the basic foundations needed to tackle the content that may become important on your way to discover the possibilities of R.

6.1 Scripts

6.1.1 Using comments

The **comment** character, `#` has a simple meaning: it tells R to ignore everything else you've written on this line. Comments are very useful when writing scripts. While you don't need to use it, they make it easier to know what is going on in the code, for you and for others. For instance, if you read this:

```
> seeker <- 3.1415          # create the first variable
> lover <- 2.7183           # create the second variable
> keeper <- seeker * lover  # now multiply them to create a third one
> print()                           # print out the value of 'keeper'
[1] 8.539539
```

it's a lot easier to understand what I'm doing than if I just write this:

```
> seeker <- 3.1415
> lover <- 2.7183
> keeper <- seeker * lover
> print()
[1] 8.539539
```

Programs come in a few different forms: the kind of program that we're mostly interested in from the perspective of everyday data analysis using R is known as a **script**. The idea behind a script is that, instead of typing your commands into the R console one at a time, instead you write them all in a

text file. Then, once you've finished writing them and saved the text file, you can get R to execute all the commands in your file by using the source function. In a moment I'll show you exactly how this is done, but first I'd better explain why you should care.

6.1.2 Why use scripts?

To understand why scripts are so very useful, it may be helpful to consider the drawbacks to typing commands directly at the command prompt. The approach that we've been adopting so far, in which you type commands one at a time, and R sits there patiently in between commands, is referred to as the **interactive styl**e. Doing your data analysis this way is rather like having a conversation ... a very annoying conversation between you and your data set, in which you and the data aren't directly speaking to each other, and so you have to rely on R to pass messages back and forth.

This approach makes a lot of sense when you're just trying out a few ideas: maybe you're trying to figure out what analyses are sensible for your data, or maybe just you're trying to remember how the various R functions work, so you're just typing in a few commands until you get the one you want. In other words, the interactive style is very useful as a tool for exploring your data.

However, it has a number of drawbacks of which scripts help you get around all these difficulties:

- *It's hard to save your work effectively.* You can save the workspace, so that later on you can load any variables you created. You can save your plots as images. And you can even save the history or copy the contents of the R console to a file. Taken together, all these things let you create a reasonably decent record of what you did. But it does leave a lot to be desired. It seems like you ought to be able to save a single file that R could use (in conjunction with your raw data files) and reproduce everything (or at least, everything interesting) that you did during your data analysis.
- *It's annoying to have to go back to the beginning when you make a mistake.* Suppose you've just spent the last two hours typing in commands. Over the course of this time you've created lots of new variables and run lots of analyses. Then suddenly you realize that there was a nasty typo in the first command you typed, so all of your later numbers are wrong. Now you have to fix that first command, and then spend another hour or so combing through the R history to try and recreate what you did.
- *You can't leave notes for yourself.* Sure, you can scribble down some notes on a piece of paper, or even save a Word document that summarizes what you did. But what you really want to be able to do is write down an English translation of your R commands, preferably right "next to" the commands themselves. That way, you can look back at what you've done and actually remember what you were doing. In the simple examples we've engaged in so far, it hasn't been all that hard to remember what you were doing or why you were doing it, but only because everything we've done could be done using only a few commands, and you've never been asked to reproduce your analysis six months after you originally did it! When your data analysis starts involving hundreds of variables, and requires quite complicated commands to work, then you really, really need to leave yourself some notes to explain your analysis to, well, yourself.
- *It's nearly impossible to reuse your analyses later, or adapt them to similar problems.* Suppose that, sometime in January, you are handed a difficult data analysis problem. After working on it for ages, you figure out some really clever tricks that can be used to solve it. Then, in September, you get handed a really similar problem. You can sort of remember what you did, but not very well. You'd like to have a clean record of what you did last time, how you did it, and why you did it the way you did. Something like that would really help you solve this new problem.
- *It's hard to do anything except the basics.* There's a nasty side effect of these problems. Typos are inevitable. Even the best data analyst in the world makes a lot of mistakes. So the chance that you'll be able to string together dozens of correct R commands in a row are very small. So unless you have some way around this problem, you'll never really be able to do anything other than simple analyses.

- *It's difficult to share your work other people.* Because you don't have this nice clean record of what R commands were involved in your analysis, it's not easy to share your work with other people. Sure, you can send them all the data files you've saved, and your history and console logs, and even the little notes you wrote to yourself, but odds are pretty good that no-one else will really understand what's going on.

6.1.3 A first script

A script is just an ordinary text file, so you can write one in any text editor at all, but RStudio has a nice built in one so we'll use that. Go to the 'File' menu, select 'New File' and then choose 'R Script'. A new file will open in the top left panel of RStudio, into which we can type some commands. Let's type the following:

hello.R

```
1 x <- "hello world"
2 print(x)
```

The line at the top is the filename, and not part of the script itself. Below that, you can see the two R commands that make up the script itself. Next to each command I've included the line numbers. You don't actually type these into your script, but a lot of text editors (including the one built into RStudio) will show line numbers, since it's a very useful convention that allows you to say things like "line 1 of the script creates a new variable, and line 2 prints it out".

So how do we run the script? Assuming that the `hello.R` file has been saved to your working directory, then you can run the script using the following command:

```
> source( "hello.R" )
```

Then save the script using the menus (File > Save) as hello.R. If you like to save your scripts in a sub-folder called `scripts` of your working directory, then you need to run the script using the following command:

```
> source("./scripts/hello.R")
```

Just note that "." means the current folder. Most of the time you don't bother typing the `source` command manually. Instead you can achieve the same outcome by clicking on the "source" button in RStudio, shown in the top right of the screenshot above. If we now inspect the workspace using a command like `who` or `objects`, we discover that R has created the new variable `x` within the workspace, and not surprisingly `x` is a character string containing the text `"hello world"`. And just like that, you've written your first program R. It really is that simple.

Exercise 6.1 — Seconds A Year (Solution → p. 125)

Write a script that calculates the number of seconds in a year (assuming 365 days), and prints the result to the console After saving your script as `myscript.R` or whatever, close RStudio. It will probably ask if you want to 'save the workspace' as '.Rdata' or something similar. You can ignore that message and select 'no' (we'll talk about workspaces shortly). Reopen RStudio and run your script.

6.2 Installing and loading packages

A **package** is basically just a big collection of functions, data sets and other R objects that are all grouped together under a common name. Some packages are already installed when you put R on your computer, but the vast majority of them of R packages are out there on the internet, waiting for you to download, install and use them. R packages are collections of functions and data sets developed by the community. They increase the power of R by improving existing base R functionalities, or by adding new ones. For example, if you are usually working with data frames, probably you will have heard about `dplyr` or `data.table`, two of the most popular R packages. More than 10,000 packages are available at the official repository (CRAN) and many more are publicly available through the internet.

In this section, I'll describe how to work with packages using the RStudio tools. Along the way, you'll see that whenever you get RStudio to do something (e.g., install a package), you'll actually see the R commands that get created.

Images sourced from https://www.wikihow.com/Change-a-Light-Bulb

Figure 6.1: Installing packages

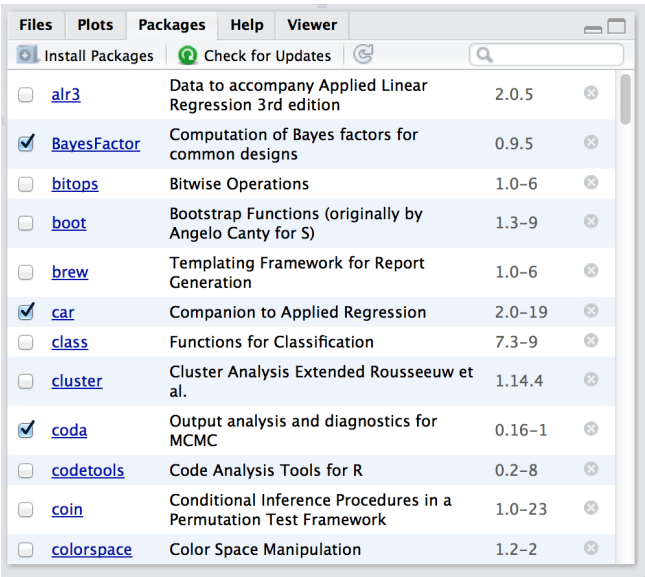


Figure 6.2: The packages panel.

However, before we get started, there's a critical distinction that you need to understand, which is the difference between having a package **installed** on your computer, and having a package **loaded** in R. When you install R on your computer only a small number of packages come bundled with the basic R installation. The installed packages are "on your computer somewhere". The critical thing to remember is that just because something is on your computer doesn't mean R can use it. In order for R to be able to *use* one of your installed packages, that package must also be "loaded". Generally, when you open up R, only a few of these packages (about 7 or 8) are actually loaded. Basically what it boils down to is this:

1. A package must be installed before it can be loaded.
2. A package must be loaded before it can be used.

We only need to install a package once on our computer. However, to use the package, we need to load the library every time we start a new R environment. You can think of this as installing a bulb versus turning on the light.

The two step process might seem a little odd at first, but the designers of R had very good reasons to do it this way. That is, there are more than 10.000 packages, and probably about 8000 authors of packages, and no-one really knows what all of them do. Keeping the installation separate from the loading minimizes the chances that two packages will interact with each other in a nasty way. Moreover having installed all available packages would probably blow your hard disk.

6.2.1 The package panel in RStudio

In the lower right hand panel in RStudio, you'll see a tab labelled "Packages". Click on the tab, and you'll see a list of packages that looks something like Figure 6.2. Every row in the panel corresponds to a different package, and every column is a useful piece of information about that package. (If you're using the command line, you can get the same information by typing `library()` at the command line.) Going from left to right, here's what each column is telling you:

- The check box on the far left column indicates whether or not the package is loaded.
- The one word of text immediately to the right of the check box is the name of the package.
- The short passage of text next to the name is a brief description of the package.
- The number next to the description tells you what version of the package you have installed.
- The little x-mark next to the version number is a button that you can push to uninstall the package from your computer (you almost never need this).

6.2.2 Loading a package

That seems straightforward enough, so let's try loading and unloading packades. For this example, I'll use the `foreign` package. The `foreign` package is a collection of tools that are very handy when R needs to interact with files that are produced by other software packages (e.g., SPSS). It comes bundled with R, so it's one of the ones that you have installed already, but it won't be one of the ones loaded. Inside the `foreign` package is a function called `read.spss()`. It's a handy little function that you can use to import an SPSS data file into R, so let's pretend we want to use it. Currently, the `foreign` package isn't loaded, so if I ask R to tell me if it knows about a function called `read.spss()` it tells me that there's no such thing...

```
> exists( "read.spss" )
[1] FALSE
```

Now let's load the package. In Rstudio, the process is dead simple: go to the package tab, find the entry for the `foreign` package, and check the box on the left hand side. The moment that you do this, you'll see a command like this appear in the R console:

```
> library("foreign")
```

The version you'll see in the console also specifies the `lib.loc` argument, which tells R where the package is stored on your computer, but mostly you don't need to worry about that yourself.

Throughout these notes, you'll sometimes see me typing in `library` commands. You don't actually have to type them in yourself: you can use the RStudio package panel to do all your package loading for you. The only reason I include the `library` commands sometimes is as a reminder to you to make sure that you have the relevant package loaded. Oh, and I suppose we should check to see if our attempt to load the package actually worked. Let's see if R now knows about the existence of the `read.spss` function...

```
> exists( "read.spss" )
[1] TRUE
```

Yep. All good.

6.2.3 Unloading a package

Sometimes, especially after a long session of working with R, you find yourself wanting to get rid of some of those packages that you've loaded. The RStudio package panel makes this exactly as easy as loading the package in the first place. Find the entry corresponding to the package you want to unload, and uncheck the box. When you do that for the `foreign` package, you'll see this command appear on screen:

```
> detach("package:foreign", unload=TRUE)
```

And the package is unloaded. We can verify this by seeing if the `read.spss()` function still `exists()`:

```
> exists( "read.spss" )
[1] FALSE
```

Nope. Definitely gone.

6.2.4 Conflicts

Something else you should be aware of. Sometimes you'll attempt to load a package, and R will print out a message on screen telling you that something or other has been "masked". This will be confusing to you if I don't explain it now, and it actually ties very closely to the whole reason why R forces you to load packages separately from installing them. Here's an example. Two of the package that you often see in several examples

and code snippets lot in this book are called `car` and `psych`. The `car` package is short for "Companion to Applied Regression" (which is a really great book, see Fox and Weisberg (2018)). The `car` package was written by a guy called John Fox, who has written a lot of great statistical tools for social science applications. The `psych` package was written by William Revelle, and it has a lot of functions that are very useful for psychologists in particular, especially in regards to psychometric techniques. For the most part, `car` and `psych` are quite unrelated to each other. They do different things, so not surprisingly almost all of the function names are different. But... there's one exception to that. The `car` package and the `psych` package *both* contain a function called `logit()`.[1] This creates a naming conflict. If I load both packages into R, an ambiguity is created. If the user types in `logit(100)`, should R use the `logit()` function in the `car` package, or the one in the `psych` package? The answer is: R uses whichever package you loaded most recently, and it tells you this very explicitly. Here's what happens when I load the `car` package, and then afterwards load the `psych` package:

```
> library(car)
> library(psych)
Attaching package: 'psych'

The following object is masked from 'package:car':

    logit
```

The output here is telling you that the `logit` object (i.e., function) in the `car` package is no longer accessible to you. It's been hidden (or "masked") from you by the one in the `psych` package.

Please notice: You can get R to use the one from the `car` package by using `car::logit()` as your command rather than `logit()`, since the `car::` part tells R explicitly which package to use.

6.2.5 Downloading new packages

How can we download and install packages? There is a big repository of packages called the "Comprehensive R Archive Network" (CRAN), and the easiest way of getting and installing a new package is from one of the many CRAN mirror sites (`https://cran.r-project.org/`). Conveniently for us, R provides a function called `install.packages()` that you can use to do this. Even *more* conveniently, the Rstudio team runs its own CRAN mirror and RStudio has a clean interface that lets you install packages without having to learn how to use the `install.packages()` command.

Using the RStudio tools is, again, dead simple. In the top left hand corner of the packages panel (Figure 6.2) you'll see a button called "Install Packages". If you click on that, it will bring up a window like the one shown in Figure 6.3a. There are a few different buttons and boxes you can play with. Ignore most of them. Just go to the line that says "Packages" and start typing the name of the package that you want. As you type, you'll see a dropdown menu appear (Figure 6.3b), listing names of packages that start with the letters that you've typed so far. You can select from this list, or just keep typing. Either way, once you've got the package name that you want, click on the install button at the bottom of the window. When you do, you'll see the following command appear in the R console:

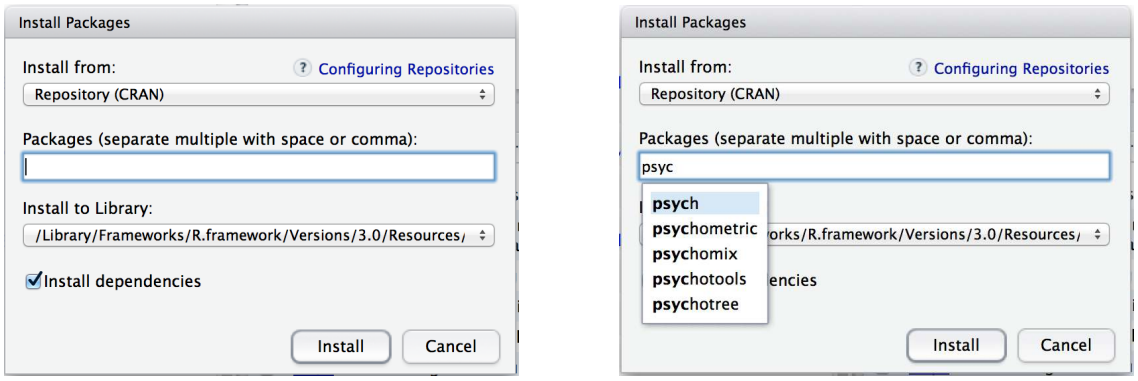
```
> install.packages("psych")
```

This is the R command that does all the work. R then goes off to the internet, has a conversation with CRAN, downloads some stuff, and installs it on your computer. You probably don't care about all the details of R's little adventure on the web, but the `install.packages()` function is rather chatty, so it reports a bunch of gibberish that you really aren't all that interested in:

```
Installing package into '/home/sthu/R/x86_64-pc-linux-gnu-library/4.0'
(as 'lib' is unspecified)
trying URL 'https://cloud.r-project.org/src/contrib/psych_2.0.12.tar.gz'
Content type 'application/x-gzip' length 2696314 bytes (2.6 MB)
==================================================
downloaded 2.6 MB

* installing *source* package 'psych' ...
** package 'psych' successfully unpacked and MD5 sums checked
** using staged installation
```

[1] The logit function a simple mathematical function that happens not to have been included in the basic R distribution.



(a) (b)

Figure 6.3: The package installation dialog box in Rstudio (panel a). When you start typing, you'll see a dropdown menu suggest a list of possible packages that you might want to install (panel b)

```
** R
** data
*** moving datasets to lazyload DB
** inst
** byte-compile and prepare package for lazy loading
```

Despite the long and tedious response, all thar really means is "I've installed the psych package". I find it best to humour the talkative little automaton. I don't actually read any of this garbage, I just politely say "thanks" and go back to whatever I was doing.

6.2.6 Installing from Github

Most of the time the packages that you'll want to install have been made available on CRAN, so the `install.packages` function is what you'd want to use. However, sometimes people write packages that they don't bother to submit to CRAN, and sometimes you might want to try out a package that is currently under development. In these situations, people who write packages will often make them available on sites like `GitHub.com`, `Bitbucket.org` or `about.gitlab.com`. It's perfectly possible to install packages directly from those sites, using the `devtools` package.

The first thing you'll need to do is install devtools, which is easy because that package is available on CRAN.

```
> install.packages("devtools")
```

Once you have `devtools` on installed, you can use the `install_github` command to install a package directly from a GitHub repository using the following function:

```
> install_github("DeveloperName/PackageName")
```

Further information on this can be found here: `https://cran.r-project.org/web/packages/githubinstall/vignettes/githubinstall.html`

6.2.7 Updating R and R packages

Every now and then the authors of packages release updated versions to add new functionality or fix bugs. It's generally a good idea to update your packages periodically. There's an `update.packages()` function that you can use to do this, but it's probably easier to stick with the RStudio tool. In the packages panel, click on the "Update Packages" button. This will bring up a window that shows you packages to be updated. You can to tell R which updates you want to install by checking the boxes on the left. If you're feeling lazy and just want to update everything, click the "Select All" button, and then click the "Install Updates" button. R then prints out a *lot* of garbage on the screen, individually downloading and installing all the new packages. This might take a while to complete depending on how good your internet connection is.

About every six months or so, a new version of R is released. You can't update R from within RStudio (not to my knowledge, at least): to get the new version you can go to the CRAN website and download the most recent version of R, and install it in the same way you did when you originally installed R on your computer. This

used to be a slightly frustrating event, because whenever you downloaded the new version of R, you would lose all the packages that you'd downloaded and installed, and would have to repeat the process of re-installing them. This was pretty annoying, and there were some neat tricks you could use to get around this. However, newer versions of R don't have this problem so I no longer bother explaining the workarounds for that issue.

6.3 Managing the workspace

Let's suppose that you're reading through this book, and what you're doing is sitting down with it once a week and working through a whole chapter in each sitting. Not only that, you've been following my advice and typing in all these commands into R. So far during this chapter, you'd have typed quite a few commands, although the only ones that actually involved creating variables were the ones you typed during Section 6.1.1. As a result, you currently have three variables; `seeker`, `lover`, and `keeper`. These three variables are the contents of your **workspace**, also referred to as the **global environment**. The workspace is a key concept in R, so in this section we'll talk a lot about what it is and how to manage its contents.

6.3.1 Listing the contents of the workspace

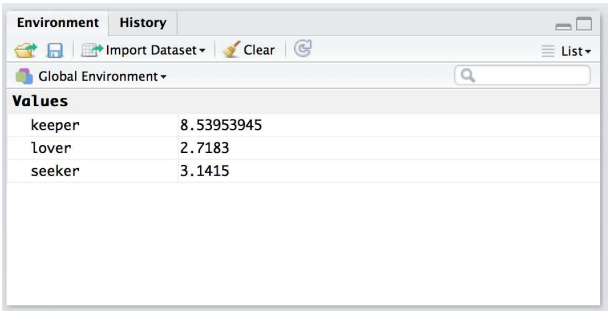


Figure 6.4: The Rstudio "Environment" panel shows you the contents of the workspace. The view shown above is the "list" view. To switch to the grid view, click on the menu item on the top right that currently reads "list". Select "grid" from the dropdown menu, and then it will switch to a view like the one shown in Figure 6.5.

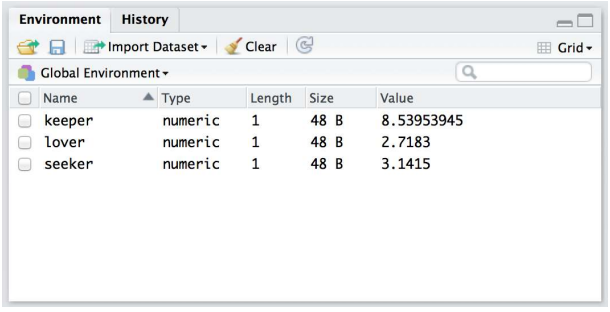


Figure 6.5: The Rstudio "Environment" panel shows you the contents of the workspace. Compare this "grid" view to the "list" view in Figure 6.4

The first thing that you need to know how to do is examine the contents of the workspace. If you're using Rstudio, you will probably find that the easiest way to do this is to use the "Environment" panel in the top right hand corner. Click on that, and you'll see a list that looks very much like the one shown in Figures 6.4 and 6.5. If you're using the commmand line, then the `objects()` function may come in handy:

```
> objects()
[1] "keeper" "lover"  "seeker"
```

Of course, in the true R tradition, the `objects()` function has a lot of fancy capabilities that I'm glossing over in this example. Moreover there are also several other functions that you can use, including `ls()` which is pretty much identical to `objects()`, and `ls.str()` which you can use to get a fairly detailed description of all the variables in the workspace.

The lsr package, for example, includes a function that is informative that you can use for this purpose, called who(). In comparison to the ls.str() function it is more concise. To have access to the who() function, we need to install the lsr package and load it:

```
> install.packages("lsr")
> library("lsr")
```

and now we can use the who() function:

```
> ls()
   -- Name --    -- Class --    -- Size --
   keeper        numeric        1
   lover         numeric        1
   seeker        numeric        1
```

As you can see, the who() function lists all the variables and provides some basic information about what kind of variable each one is and how many elements it contains. Personally, I find this output much easier more useful than the very compact output of the objects() function, but less overwhelming than the extremely verbose ls.str() function.

6.3.2 Removing variables from the workspace

Looking over that list of variables, it occurs to me that I really don't need them any more. I created them originally just to make a point, but they don't serve any useful purpose anymore, and now I want to get rid of them. I'll show you how to do this, but first I want to warn you – there's no "undo" option for variable removal. Once a variable is removed, it's gone forever unless you save it to disk. I'll show you how to do *that* in Section 6.5, but quite clearly we have no need for these variables at all, so we can safely get rid of them.

In Rstudio, the easiest way to remove variables is to use the environment panel. Assuming that you're in grid view (i.e., Figure 6.5), check the boxes next to the variables that you want to delete, then click on the "Clear" button at the top of the panel. When you do this, Rstudio will show a dialog box asking you to confirm that you really do want to delete the variables. It's always worth checking that you really do, because as Rstudio is at pains to point out, you can't undo this. Once a variable is deleted, it's gone.[2] In any case, if you click "yes", that variable will disappear from the workspace: it will no longer appear in the environment panel, and it won't show up when you use the who() command.

Suppose you don't access to Rstudio, and you still want to remove variables. This is where the **remove** function rm() comes in handy. The simplest way to use rm() is just to type in a (comma separated) list of all the variables you want to remove. Let's say I want to get rid of seeker and lover, but I would like to keep keeper. To do this, all I have to do is type:

```
> rm( seeker, lover )
```

There's no visible output, but if I now inspect the workspace

```
> ls()
[1] "keeper"
```

I see that there's only the keeper variable left.

6.4 Navigating the file system

In this section I talk a little about how R interacts with the file system on your computer. It's not a terribly interesting topic, but it's useful. As background to this discussion, I'll talk a bit about how file system locations work in Section 6.4.1. Once upon a time *everyone* who used computers could safely be assumed to understand how the file system worked, because it was impossible to successfully use a computer if you didn't! However, modern operating systems are much more "user friendly", and as a consequence of this they go to great lengths to hide the file system from users. So these days it's not at all uncommon for people to have used computers most of their life and not be familiar with the way that computers organize files. If you already know this stuff, skip straight to Section 6.4.2. Otherwise, read on. I'll try to give a brief introduction that will be useful for those of you who have never been forced to learn how to navigate around a computer using a DOS or UNIX shell.

[2] Mind you, all that means is that it's been removed from the workspace. If you've got the data saved to file somewhere, then that *file* is perfectly safe.

6.4.1 The file system itself

In this section I describe the basic idea behind file locations and file paths. Regardless of whether you're using Window, Mac OS or Linux, every file on the computer is assigned a (fairly) human readable address, and every address has the same basic structure: it describes a *path* that starts from a *root* location , through as series of *folders* (or if you're an old-school computer user, *directories*), and finally ends up at the file.

On a Windows computer the root is the physical drive on which the file is stored, and for many home computers the name of the hard drive that stores all your files is `C:` and therefore most file names on Windows begin with `C:`. After that comes the folders, and on Windows the folder names are separated by a backslash symbol "\". So, the complete path to this book on my Windows computer might be something like this:

```
C:\Users\huber\Rbook\rcourse-book.pdf
```

and what that *means* is that the book is called `rcourse-book.pdf`, and it's in a folder called `Rbook` which itself is in a folder called `huber` which itself is ... well, you get the idea. On Linux, Unix and Mac OS systems, the addresses look a little different, but they're more or less identical in spirit. Instead of using the backslash, folders are separated using a forward slash, and unlike Windows, they don't treat the physical drive as being the root of the file system. So, the path to this book on my Mac might be something like this: -

```
/Users/huber/Rbook/rcourse-book.pdf
```

So that's what we mean by the "path" to a file. The next concept to grasp is the idea of a **working directory** and how to change it. For those of you who have used command line interfaces previously, this should be obvious already. But if not, here's what I mean. The working directory is just "whatever folder I'm currently looking at". Suppose that I'm currently looking for files in Explorer (if you're using Windows) or using Finder (on a Mac). The folder I currently have open is my user directory (i.e., `C:\Users\huber` or `/Users/huber`). That's my current working directory.

The fact that we can imagine that the program is "in" a particular directory means that we can talk about moving *from* our current location *to* a new one. What that means is that we might want to specify a new location in relation to our current location. To do so, we need to introduce two new conventions. Regardless of what operating system you're using, we use `.` to refer to the current working directory, and `..` to refer to the directory above it. This allows us to specify a path to a new location in relation to our current location, as the following examples illustrate. Let's assume that I'm using my Windows computer, and my working directory is `C:\Users\huber\Rbook`). The table below shows several addresses in relation to my current one:

absolute path (i.e., from root)	relative path (i.e. from `C:\Users\huber\Rbook`)
`C:\Users\huber`	`..`
`C:\Users`	`..\..`
`C:\Users\huber\Rbook\source`	`.\source`
`C:\Users\huber\nerdstuff`	`..\nerdstuff`

There's one last thing I want to call attention to: the `~` directory. I normally wouldn't bother, but R makes reference to this concept sometimes. It's quite common on computers that have multiple users to define `~` to be the user's home directory. On my Mac, for instance, the home directory `~` for the "huber" user is `\Users\huber\`. And so, not surprisingly, it is possible to define other directories in terms of their relationship to the home directory. For example, an alternative way to describe the location of the `rcourse-book.pdf` file on my Mac would be

```
~\Rbook\rcourse-book.pdf
```

That's about all you really need to know about file paths. And since this section already feels too long, it's time to look at how to navigate the file system in R .

6.4.2 Navigating the file system using the R console

In this section I'll talk about how to navigate this file system from within R itself. It's not particularly user friendly, and so you'll probably be happy to know that RStudio provides you with an easier method, and I will describe it in Section 6.4.4. So in practice, you won't *really* need to use the commands that I babble on about in this section, but I do think it helps to see them in operation at least once before forgetting about them forever.

Okay, let's get started. When you want to load or save a file in R it's important to know what the working directory is. You can find out by using the `getwd()` command. For the moment, let's assume that I'm using Mac OS or Linux, since there's some subtleties to Windows. Here's what happens:

```
> getwd()
[1] "/Users/huber"
```

We can change the working directory quite easily using `setwd()`. The `setwd()` function has only the one argument, `dir`, is a character string specifying a path to a directory, or a path relative to the working directory. Since I'm currently located at `/Users/huber`, the following two are equivalent:

```
> setwd("/Users/huber/Rbook/data")
> setwd("./Rbook/data")
```

Now that we're here, we can type `list.files()` command to get a listing of all the files in that directory. Since this is the directory in which I store all of the data files that we'll use in this book, here's what we get as the result:

```
> list.files()
 [1] "afl24.Rdata"              "aflsmall.Rdata"          "aflsmall2.Rdata"
 [4] "agpp.Rdata"               "all.zip"                 "annoying.Rdata"
 [7] "anscombesquartet.Rdata"   "awesome.Rdata"           "awesome2.Rdata"
[10] "booksales.csv"            "booksales.Rdata"         "booksales2.csv"
[13] "cakes.Rdata"              "cards.Rdata"             "chapek9.Rdata"
[16] "chico.Rdata"              "clinicaltrial_old.Rdata" "clinicaltrial.Rdata"
[19] "coffee.Rdata"             "drugs.wmc.rt.Rdata"      "dwr_all.Rdata"
[22] "effort.Rdata"             "happy.Rdata"             "harpo.Rdata"
[25] "harpo2.Rdata"             "likert.Rdata"            "nightgarden.Rdata"
[28] "nightgarden2.Rdata"       "parenthood.Rdata"        "parenthood2.Rdata"
[31] "randomness.Rdata"         "repeated.Rdata"          "rtfm.Rdata"
[34] "salem.Rdata"              "zeppo.Rdata"
```

Not terribly exciting, I'll admit, but it's useful to know about. In any case, there's only one more thing I want to make a note of, which is that R also makes use of the home directory. You can find out what it is by using the `path.expand()` function, like this:

```
> path.expand("~")
[1] "/Users/huber"
```

You can change the user directory if you want, but we're not going to make use of it very much so there's no reason to. The only reason I'm even bothering to mention it at all is that when you use RStudio to open a file, you'll see output on screen that defines the path to the file relative to the `~` directory. I'd prefer you not to be confused when you see it.

One additional thing worth calling your attention to is the `file.choose()` function. Suppose you want to load a file and you don't quite remember where it is, but would like to browse for it. Typing `file.choose()` at the command line will open a window in which you can browse to find the file; when you click on the file you want, R will print out the full path to that file. This is kind of handy.

6.4.3 Why do the Windows paths use the wrong slash?

Let's suppose I'm using a computer with Windows. As before, I can find out what my current working directory is like this:

```
> getwd()
[1] "C:/Users/huber/
```

This seems about right, but you might be wondering why R is displaying a Windows path using the wrong type of slash. The answer is slightly complicated, and has to do with the fact that R treats the `\` character as "special". If you're deeply wedded to the idea of specifying a path using the Windows style slashes, then what you need to do is to type `\\` whenever you mean `\`. In other words, if you want to specify the working directory on a Windows computer, you need to use one of the following commands:

```
> setwd( "C:/Users/huber" )
> setwd( "C:\\Users\\huber" )
```

It's kind of annoying to have to do it this way, but it's a necessary evil. Fortunately, as we'll see in the next section, RStudio provides a much simpler way of changing directories...

6.4.4 Navigating the file system using the Studio file panel

Although I think it's important to understand how all this command line stuff works, in many (maybe even most) situations there's an easier way. For our purposes, the easiest way to navigate the file system is to make use of Rstudio's built in tools. The "file" panel – the lower right hand area in Figure 6.6 – is actually a pretty decent file browser. Not only can you just point and click on the names to move around the file system, you can also use it to set the working directory, and even load files.

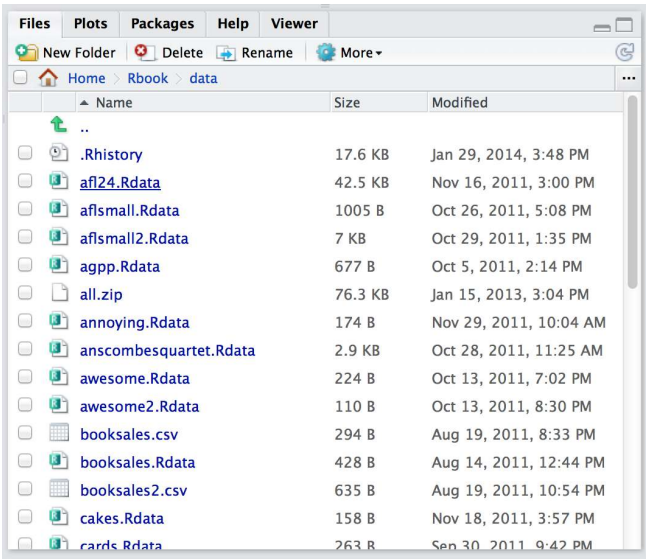


Figure 6.6: The "file panel" is the area shown in the lower right hand corner. It provides a very easy way to browse and navigate your computer using R.

Here's what you need to do to change the working directory using the file panel. Let's say I'm looking at the actual screen shown in Figure 6.6. At the top of the file panel you see some text that says "Home > Rbook > data". What that means is that it's *displaying* the files that are stored in the

`/Users/huber/Rbook/data`

directory on my computer. It does *not* mean that this is the R working directory. If you want to change the R working directory, using the file panel, you need to click on the button that reads "More". This will bring up a little menu, and one of the options will be "Set as Working Directory". If you select that option, then R really will change the working directory. You can tell that it has done so because this command appears in the console:

```
> setwd("~/Rbook/data")
```

In other words, Studio sends a command to the R console, exactly as if you'd typed it yourself. The file panel can be used to do other things too. If you want to move "up" to the parent folder (e.g., from `/Users/huber/Rbook/data` to `/Users/huber/Rbook`) click on the ".." link in the file panel. To move to a subfolder, click on the name of the folder that you want to open. You can open some types of file by clicking on them. You can delete files from your computer using the "delete" button, rename them with the "rename" button, and so on.

As you can tell, the file panel is a very handy little tool for navigating the file system. But it can do more than just navigate. As we'll see later, it can be used to open files. And if you look at the buttons and menu options that it presents, you can even use it to rename, delete, copy or move files, and create new folders. However, since most of that functionality isn't critical to the basic goals of this book, I'll let you discover those on your own.

6.5 Loading and saving data

There are several different types of files that are likely to be relevant to us when doing data analysis. There are three in particular that are especially important from the perspective of this book:

- *Workspace files* are those with a `.Rdata` file extension. This is the standard kind of file that R uses to store data and variables. They're called "workspace files" because you can use them to save your whole workspace.

- *Comma separated value (CSV) files* are those with a `.csv` file extension. These are just regular old text files, and they can be opened with almost any software. It's quite typical for people to store data in CSV files, precisely because they're so simple.
- *Script files* are those with a `.R` file extension. These aren't data files at all; rather, they're used to save a collection of commands that you want R to execute later. They're just text files.

There are also several other types of file that R makes use of,[3] but they're not really all that central to our interests. There are also several other kinds of data file that you might want to import into R. For instance, you might want to open Microsoft Excel spreadsheets (`.xls` files), or data files that have been saved in the native file formats for other statistics software, such as SPSS, SAS, Minitab, Stata or Systat. Finally, you might have to handle databases. R tries hard to play nicely with other software, so it has tools that let you open and work with any of these and many others. For now, I want to focus primarily on the two kinds of data file that you're most likely to need: `.Rdata` files and `.csv` files. In this section I'll talk about how to load a workspace file, how to import data from a CSV file, and how to save your workspace to a workspace file.

6.5.1 Loading workspace files using R

When I used the `list.files()` command to list the contents of the `/Users/huber/Rbook/data` directory (in Section 6.4.2), the output referred to a file called `booksales.Rdata`. Let's say I want to load the data from this file into my workspace. The way I do this is with the `load()` function. There are two arguments to this function, but the only one we're interested in is `file="..."`. This should be a character string that specifies a path to the file that needs to be loaded. You can use an absolute path or a relative path to do so. Using the absolute file path, the command would look like this:

```
> load( file = "/Users/huber/Rbook/data/booksales.Rdata" )
```

but this is pretty lengthy. Given that the working directory (remember, we changed the directory at the end of Section 6.4.4) is `/Users/huber/Rbook/data`, I could use a relative file path, like so:

```
> load( file = "../data/booksales.Rdata" )
```

However, my preference is usually to change the working directory first, and *then* load the file. What that would look like is this:

```
> setwd( "../data" )         # move to the data directory
> load( "booksales.Rdata" )  # load the data
```

If I were then to type `ls()` or `ls.str()`, I'd see that there are several new variables in my workspace now. Throughout this book, whenever you see me loading a file, I will assume that the file is actually stored in the working directory, or that you've changed the working directory so that R is pointing at the directory that contains the file. Obviously, *you* don't need type that command yourself: you can use the RStudio file panel to do the work.

6.5.2 Loading workspace files using RStudio

Okay, so how do we open an .Rdata file using the RStudio file panel? It's terribly simple. First, use the file panel to find the folder that contains the file you want to load. If you look at Figure 6.6, you can see that there are several .Rdata files listed. Let's say I want to load the `booksales.Rdata` file. All I have to do is click on the file name. RStudio brings up a little dialog box asking me to confirm that I do want to load this file. I click yes. The following command then turns up in the console,

```
> load("~/Rbook/data/booksales.Rdata")
```

and the new variables will appear in the workspace (you'll see them in the Environment panel in RStudio, or if you type `ls()`). So easy it barely warrants having its own section.

6.5.3 Importing data from CSV files using R

One quite commonly used data format is the humble "comma separated value" file, also called a CSV file, and usually bearing the file extension `.csv`. CSV files are just plain old-fashioned text files, and what they store is basically just a table of data. This is illustrated in Figure 6.7, which shows a file called `booksales.csv` that I've

[3] Notably those with `.rda`, `.Rd`, `.Rhistory`, `.rdb` and `.rdx` extensions

created. As you can see, each row corresponds to a variable, and each row represents the book sales data for one month. The first row doesn't contain actual data though: it has the names of the variables.

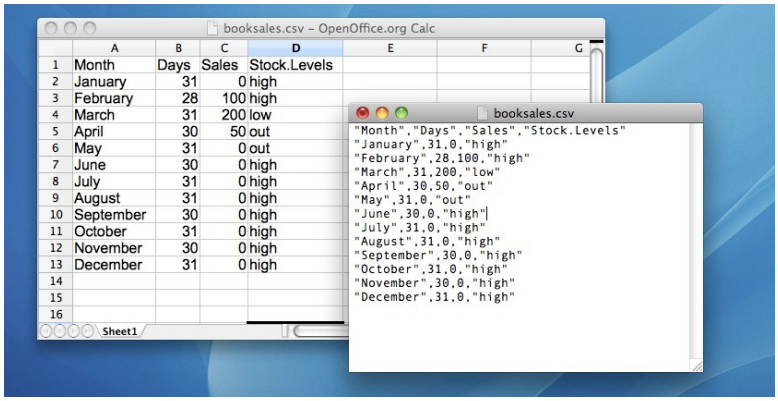


Figure 6.7: The `booksales.csv` data file. On the left, I've opened the file in using a spreadsheet program (OpenOffice), which shows that the file is basically a table. On the right, the same file is open in a standard text editor (the TextEdit program on a Mac), which shows how the file is formatted. The entries in the table are wrapped in quote marks and separated by commas.

If RStudio were not available to you, the easiest way to open this file would be to use the `read.csv()` function. This function is pretty flexible and can be discussed in greater depth, but for now there's only two arguments to the function that I'll mention:

- `file`. This should be a character string that specifies a path to the file that needs to be loaded. You can use an absolute path or a relative path to do so.
- `header`. This is a logical value indicating whether or not the first row of the file contains variable names. The default value is `TRUE`.

Therefore, to import the CSV file, the command I need is:

```
> books <- read.csv( file = "booksales.csv" )
```

There are two very important points to notice here. Firstly, notice that I *didn't* try to use the `load()` function, because that function is only meant to be used for `.Rdata` files. If you try to use `load()` on other types of data, you get an error. Secondly, notice that when I imported the CSV file I assigned the result to a variable, which I imaginatively called `books`. Let's have a look at what we've got:

```
> print( books )
       Month Days Sales Stock.Levels
1    January   31     0         high
2   February   28   100         high
3      March   31   200          low
4      April   30    50          out
5        May   31     0          out
6       June   30     0         high
7       July   31     0         high
8     August   31     0         high
9  September   30     0         high
10   October   31     0         high
11  November   30     0         high
12  December   31     0         high
```

It worked, but the format of this output is a bit unfamiliar. We haven't seen anything like this before. What you're looking at is a **data frame**, which is a very important kind of variable in R, and one I'll discuss in Section 6.8.

6.5.4 Importing data from CSV files using Studio

You can also import data using Studio . In the environment panel in Studio you should see a button called "Import Dataset". Click on that, and it will give you a couple of options: select the "From Text File..." option, and it will open up a very familiar dialog box asking you to select a file: if you're on a Mac, it'll look like the usual Finder window that you use to choose a file; on Windows it looks like an Explorer window. I'm assuming that you're familiar with your own computer, so you should have no problem finding the CSV file that you want to import! Find the one you want, then click on the "Open" button. When you do this, you'll see a window that looks like the one in Figure 6.8.

The import data set window is relatively straightforward to understand. In the top left corner, you need to type the name of the variable you R to create. By default, that will be the same as the file name: our file is called `booksales.csv`, so Rstudio suggests the name `booksales`. If you're happy with that, leave it alone. If not, type something else. Immediately below this are a few things that you can tweak to make sure that the data gets imported correctly:

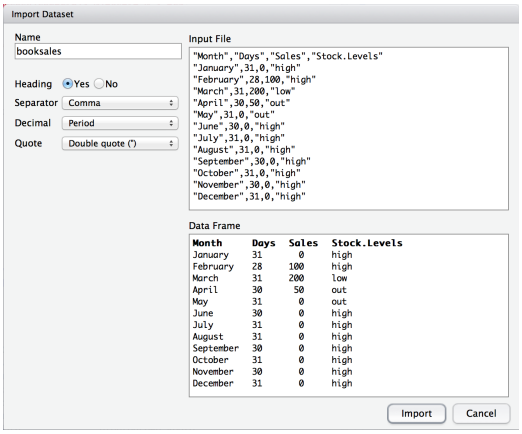


Figure 6.8: The Studio window for importing a CSV file into R .

- Heading. Does the first row of the file contain raw data, or does it contain headings for each variable? The `booksales.csv` file has a header at the top, so I selected "yes".
- Separator. What character is used to separate different entries? In most CSV files this will be a comma (it is "comma separated" after all). But you can change this if your file is different.
- Decimal. What character is used to specify the decimal point? In English speaking countries, this is almost always a period (i.e., `.`). That's not universally true: many European countries use a comma. So you can change that if you need to.
- Quote. What character is used to denote a block of text? That's usually going to be a double quote mark. It is for the `booksales.csv` file, so that's what I selected.

The nice thing about the Rstudio window is that it shows you the raw data file at the top of the window, and it shows you a preview of the data at the bottom. If the data at the bottom doesn't look right, try changing some of the settings on the left hand side. Once you're happy, click "Import". When you do, two commands appear in the R console:

```
> booksales <- read.csv("~/Rbook/data/booksales.csv")
> View(booksales)
```

The first of these commands is the one that loads the data. The second one will display a pretty table showing the data in Rstudio.

6.5.5 Saving a workspace file using R

Not surprisingly, saving data is very similar to loading data. Although Studio provides a simple way to save files (see below), it's worth understanding the actual commands involved. There are two commands you can use to do this, `save()` and `save.image()`. If you're happy to save *all* of the variables in your workspace into the data file, then you should use `save.image()`. And if you're happy for R to save the file into the current working directory, all you have to do is this:

```
> save.image( file = "myfile.Rdata" )
```

Since `file` is the first argument, you can shorten this to `save.image("myfile.Rdata")`; and if you want to save to a different directory, then (as always) you need to be more explicit about specifying the path to the file, just as we discussed in Section 6.4. Suppose, however, I have several variables in my workspace, and I only want to save some of them. For instance, I might have this as my workspace:

```
-- Name --    -- Class --    -- Size --
data          data.frame     3 x 2
handy         character      1
junk          numeric        1
```

I want to save `data` and `handy`, but not `junk`. But I don't want to delete `junk` right now, because I want to use it for something else later on. This is where the `save()` function is useful, since it lets me indicate exactly which variables I want to save. Here is one way I can use the `save` function to solve my problem:

```
> save(data, handy, file = "myfile.Rdata")
```

Importantly, you *must* specify the name of the `file` argument. The reason is that if you don't do so, R will think that `"myfile.Rdata"` is actually a *variable* that you want to save, and you'll get an error message. Finally, I should mention a second way to specify which variables the `save()` function should save, which is to use the `list` argument. You do so like this:

```
> save.me <- c("data", "handy")   # the variables to be saved
> save( file = "booksales2.Rdata", list = save.me )   # the command to save them
```

6.5.6 Saving a workspace file using RStudio

Rstudio allows you to save the workspace pretty easily. In the environment panel you can see the "save" button. There's no text, but it's the same icon that gets used on every computer everywhere: it's the one that looks like a floppy disk. You know, those things that haven't been used in about 20 years. Alternatively, go to the "Session" menu and click on the "Save Workspace As..." option. This will bring up the standard "save" dialog box for your operating system. Type in the name of the file that you want to save it to, and all the variables in your workspace will be saved to disk. You'll see an R command like this one

```
> save.image("~/Desktop/Untitled.RData")
```

Pretty straightforward, really.

> A word of warning: what you don't want to do is use the "File" menu. If you look in the "File" menu you will see "Save" and "Save As..." options, but they don't save the workspace. Those options are used for dealing with *scripts*, and so they'll produce `.R` files.

6.6 Useful things to know about variables

In Chapter 5 I talked a lot about variables, how they're assigned and some of the things you can do with them, but there's a lot of additional complexities. That's not a surprise of course. However, some of those issues are worth drawing your attention to now. So that's the goal of this section; to cover a few extra topics. As a consequence, this section is basically a bunch of things that I want to briefly mention, but don't really fit in anywhere else. In short, I'll talk about several different issues in this section, which are only loosely connected to one another.

6.6.1 Special values

The first thing I want to mention are some of the "special" values that you might see R produce. Most likely you'll see them in situations where you were expecting a number, but there are quite a few other ways you can encounter them. These values are `Inf`, `NaN`, `NA` and `NULL`. These values can crop up in various different places, and so it's important to understand what they mean.

- *Infinity* (`Inf`). The easiest of the special values to explain is `Inf`, since it corresponds to a value that is infinitely large. You can also have `-Inf`. The easiest way to get `Inf` is to divide a positive number by 0:

```
> 1 / 0
[1] Inf
```

In most real world data analysis situations, if you're ending up with infinite numbers in your data, then something has gone awry. Hopefully you'll never have to see them.

- *Not a Number* (`NaN`). The special value of `NaN` is short for "not a number", and it's basically a reserved keyword that means "there isn't a mathematically defined number for this". If you can remember your high school maths, remember that it is conventional to say that 0/0 doesn't have a proper answer: mathematicians would say that 0/0 is *undefined*. R says that it's not a number:

  ```
  > 0 / 0
  [1] NaN
  ```

  Nevertheless, it's still treated as a "numeric" value. To oversimplify, `NaN` corresponds to cases where you asked a proper numerical question that genuinely has *no meaningful answer*.

- *Not available* (`NA`). `NA` indicates that the value that is "supposed" to be stored here is missing. To understand what this means, it helps to recognise that the `NA` value is something that you're most likely to see when analysing data from real world experiments. Sometimes you get equipment failures, or you lose some of the data, or whatever. The point is that some of the information that you were "expecting" to get from your study is just plain missing. Note the difference between `NA` and `NaN`. For `NaN`, we really do know what's supposed to be stored; it's just that it happens to correspond to something like 0/0 that doesn't make any sense at all. In contrast, `NA` indicates that we actually don't know what was supposed to be there. The information is *missing*.

- *No value* (`NULL`). The `NULL` value takes this "absence" concept even further. It basically asserts that the variable genuinely has no value whatsoever. This is quite different to both `NaN` and `NA`. For `NaN` we actually know what the value is, because it's something insane like 0/0. For `NA`, we believe that there is supposed to be a value "out there", but a dog ate our homework and so we don't quite know what it is. But for `NULL` we strongly believe that there is *no value at all*.

6.6.2 Assigning names to vector elements

One thing that is sometimes a little unsatisfying about the way that R prints out a vector is that the elements come out unlabelled. Here's what I mean. Suppose I've got data reporting the quarterly profits for some company. If I just create a no-frills vector, I have to rely on memory to know which element corresponds to which event. That is:

```
> profit <- c( 3.1, 0.1, -1.4, 1.1 )
> profit
[1]  3.1  0.1 -1.4  1.1
```

You can probably guess that the first element corresponds to the first quarter, the second element to the second quarter, and so on, but that's only because I've told you the back story and because this happens to be a very simple example. In general, it can be quite difficult. This is where it can be helpful to assign `names` to each of the elements. Here's how you do it:

```
> names(profit) <- c("Q1","Q2","Q3","Q4")
> profit
  Q1   Q2   Q3   Q4
 3.1  0.1 -1.4  1.1
```

This is a slightly odd looking command, admittedly, but it's not too difficult to follow. All we're doing is assigning a vector of labels (character strings) to `names(profit)`. You can always delete the names again by using the command `names(profit) <- NULL`. It's also worth noting that you don't have to do this as a two stage process. You can get the same result with this command:

```
> profit <- c( "Q1" = 3.1, "Q2" = 0.1, "Q3" = -1.4, "Q4" = 1.1 )
> profit
  Q1   Q2   Q3   Q4
 3.1  0.1 -1.4  1.1
```

The important things to notice are that (a) this does make things much easier to read, but (b) the names at the top aren't the "real" data. The *value* of `profit[1]` is still `3.1`; all I've done is added a *name* to `profit[1]` as well. Nevertheless, names aren't purely cosmetic, since R allows you to pull out particular elements of the vector by referring to their names:

```
> profit["Q1"]
 Q1
3.1
```

And if I ever need to pull out the names themselves, then I just type `names(profit)`.

6.6.3 Variable classes

As we've seen, R allows you to store different kinds of data. In particular, the variables we've defined so far have either been character data (text), numeric data, or logical data.[4] It's important that we remember what kind of information each variable stores (and even more important that R remembers) since different kinds of variables allow you to do different things to them. For instance, if your variables have numerical information in them, then it's okay to multiply them together:

```
> x <- 5   # x is numeric
> y <- 4   # y is numeric
> x * y
[1] 20
```

But if they contain character data, multiplication makes no sense whatsoever, and R will complain if you try to do it:

```
> x <- "apples"   # x is character
> y <- "oranges"  # y is character
> x * y
Error in x * y : non-numeric argument to binary operator
```

Even R is smart enough to know you can't multiply `"apples"` by `"oranges"`. It knows this because the quote marks are indicators that the variable is supposed to be treated as text, not as a number.

This is quite useful, but notice that it means that R makes a big distinction between `5` and `"5"`. Without quote marks, R treats `5` as the number five, and will allow you to do calculations with it. With the quote marks, R treats `"5"` as the textual character five, and doesn't recognise it as a number any more than it recognises `"p"` or `"five"` as numbers. As a consequence, there's a big difference between typing `x <- 5` and typing `x <- "5"`. In the former, we're storing the number `5`; in the latter, we're storing the character `"5"`. Thus, if we try to do multiplication with the character versions, R gets stroppy:

```
> x <- "5"   # x is character
> y <- "4"   # y is character
> x * y
Error in x * y : non-numeric argument to binary operator
```

Okay, let's suppose that I've forgotten what kind of data I stored in the variable `x` (which happens depressingly often). R provides a function that will let us find out. Or, more precisely, it provides *three* functions: `class()`, `mode()` and `typeof()`. Why the heck does it provide three functions, you might be wondering? Basically, because R actually keeps track of three different kinds of information about a variable:

1. The **class** of a variable is a "high level" classification, and it captures psychologically (or statistically) meaningful distinctions. For instance `"2011-09-12"` and `"my birthday"` are both text strings, but there's an important difference between the two: one of them is a date. So it would be nice if we could get R to recognise that `"2011-09-12"` is a date, and allow us to do things like add or subtract from it. The class of a variable is what R uses to keep track of things like that. Because the class of a variable is critical for determining what R can or can't do with it, the `class()` function is very handy.
2. The **mode** of a variable refers to the format of the information that the variable stores. It tells you whether R has stored text data or numeric data, for instance, which is kind of useful, but it only makes these "simple" distinctions. It can be useful to know about, but it's not the main thing we care about.

[4] Or functions. But let's ignore functions for the moment.

3. The **type** of a variable is about the storage mode of any object.

In practice, it's the `class()` of the variable that we care most about. The following examples illustrate the use of the `class()` function:

```
> x <- "hello world"     # x is text
> class(x)
[1] "character"

> x <- TRUE     # x is logical
> class(x)
[1] "logical"

> x <- 100     # x is a number
> class(x)
[1] "numeric"
```

6.7 Factors

6.7.1 Introducing factors

Suppose, I was doing a study in which people could belong to one of three different treatment conditions. Each group of people were asked to complete the same task, but each group received different instructions. Not surprisingly, I might want to have a variable that keeps track of what group people were in. So I could type in something like this

```
> group <- c(1,1,1,2,2,2,3,3,3)
```

so that `group[i]` contains the group membership of the `i`-th person in my study. Clearly, this is numeric data, but equally obviously this is a nominal scale variable. There's no sense in which "group 1" plus "group 2" equals "group 3", but nevertheless if I try to do that, R won't stop me because it doesn't know any better:

```
> group + 2
 [1] 3 3 3 4 4 4 5 5 5
```

Apparently R seems to think that it's allowed to invent "group 4" and "group 5", even though they didn't actually exist. Unfortunately, R is too stupid to know any better: it thinks that `3` is an ordinary number in this context, so it sees no problem in calculating `3 + 2`. But since *we're* not that stupid, we'd like to stop R from doing this. We can do so by instructing R to treat `group` as a factor. This is easy to do using the `as.factor()` function.

```
> group <- as.factor(group)
> group
 [1] 1 1 1 2 2 2 3 3 3
 Levels: 1 2 3
```

It looks more or less the same as before (though it's not immediately obvious what all that `Levels` rubbish is about), but if we ask R to tell us what the class of the `group` variable is now, it's clear that it has done what we asked:

```
> class(group)
[1] "factor"
```

Neat. Better yet, now that I've converted `group` to a factor, look what happens when I try to add 2 to it:

```
> group + 2
[1] NA NA NA NA NA NA NA NA NA
Warning message:
In Ops.factor(group, 2) : + not meaningful for factors
```

This time even R is smart enough to know that I'm being an idiot, so it tells me off and then produces a vector of missing values. Figures 6.9 to 6.11 visualize the way how calculation with vectors works in R.

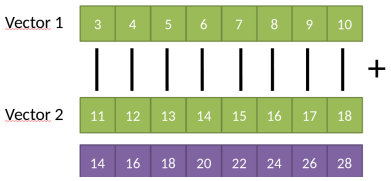


Figure 6.9: How to calculate with vectors (1)

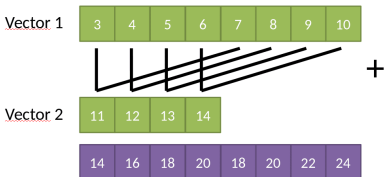


Figure 6.10: How to calculate with vectors (2)

6.7.2 Labelling the factor levels

I have a confession to make. My memory is not infinite in capacity; and it seems to be getting worse as I get older. So it kind of annoys me when I get data sets where there's a nominal scale variable called `gender`, with two levels corresponding to males and females. But when I go to print out the variable I get something like this:

```
> gender
[1] 1 1 1 1 1 2 2 2 2
Levels: 1 2
```

That's not helpful at all. Which number corresponds to the males and which one corresponds to the females? Wouldn't it be nice if R could actually keep track of this? It's way too hard to remember which number corresponds to which gender. And besides, the problem that this causes is much more serious than a single sad nerd... because R has no way of knowing that the `1`s in the `group` variable are a very different kind of thing to the `1`s in the `gender` variable. So if I try to ask which elements of the `group` variable are equal to the corresponding elements in `gender`, R thinks this is totally kosher, and gives me this:

```
> group == gender
[1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
```

Well, that's ... especially stupid.[5] The problem here is that R is very literal minded. Even though you've declared both `group` and `gender` to be factors, it still assumes that a `1` is a `1` no matter which variable it appears in.

To fix both of these problems (my memory problem, and R 's infuriating literal interpretations), what we need to do is assign meaningful labels to the different *levels* of each factor. We can do that like this:

```
> levels(group) <- c("group 1", "group 2", "group 3")
> print(group)
[1] group 1 group 1 group 1 group 2 group 2 group 2 group 3 group 3 group 3
```

[5] Some users might wonder why R even allows the == operator for factors. The reason is that sometimes you really do have different factors that have the same levels. For instance, if I was analysing data associated with football games, I might have a factor called `home.team`, and another factor called `winning.team`. In that situation I really should be able to ask if `home.team == winning.team`.

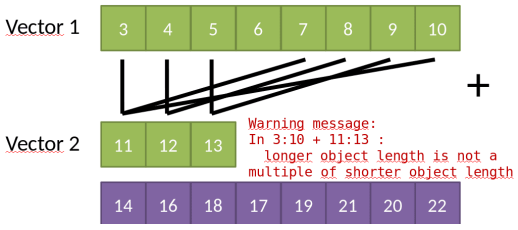


Figure 6.11: How to calculate with vectors (3)

```
Levels: group 1 group 2 group 3
>
> levels(gender) <- c("male", "female")
> print(gender)
[1] male   male   male   male   male   female female female female
Levels: male female
```

That's much easier on the eye, and better yet, R is smart enough to know that `"female"` is not equal to `"group 2"`, so now when I try to ask which group memberships are "equal to" the gender of the corresponding person,

```
> group == gender
Error in Ops.factor(group, gender) : level sets of factors are different
```

R correctly tells me that I'm an idiot.

6.8 Data frames

It's now time to go back and deal with the somewhat confusing thing that happened in Section 6.5.3 when we tried to open up a CSV file. Apparently we succeeded in loading the data, but it came to us in a very odd looking format. At the time, I told you that this was a **data frame**. Now I'd better explain what that means.

6.8.1 Introducing data frames

In order to understand why R has created this funny thing called a data frame, it helps to try to see what problem it solves. So let's go back to the little scenario that I used when introducing factors in Section 6.7. In that section I recorded the `group` and `gender` for all 9 participants in my study. Let's also suppose I recorded their ages and their `score` on a test:

```
> age <- c(17, 19, 21, 37, 18, 19, 47, 18, 19)
> score <- c(12, 10, 11, 15, 16, 14, 25, 21, 29)
```

Assuming no other variables are in the workspace, if I type `who()` I get this:[6]

```
> who()
   -- Name --    -- Class --    -- Size --
   age           numeric        9
   gender        factor         9
   group         factor         9
   score         numeric        9
```

So there are four variables in the workspace, `age`, `gender`, `group` and `score`. And it just so happens that all four of them are the same size (i.e., they're all vectors with 9 elements). And, it just so happens that `age[1]` corresponds to the age of the first person, and `gender[1]` is the gender of that very same person, etc. In other words, you and I both know that all four of these variables correspond to the *same* data set, and all four of them are organised in exactly the same way.

However, R *doesn't* know this! As far as it's concerned, there's no reason why the `age` variable has to be the same length as the `gender` variable; and there's no particular reason to think that `age[1]` has any special relationship to `gender[1]` any more than it has a special relationship to `gender[4]`. In other words, when we store everything in separate variables like this, R doesn't know anything about the relationships between things. It doesn't even really know that these variables actually refer to a proper data set. The data frame fixes this: if we store our variables inside a data frame, we're telling R to treat these variables as a single, fairly coherent data set.

To see how they do this, let's create one. So how do we create a data frame? One way we've already seen: if we import our data from a CSV file, R will store it as a data frame. A second way is to create it directly from some existing variables using the `data.frame()` function. All you have to do is type a list of variables that you want to include in the data frame. The output of a `data.frame()` command is, well, a data frame. So, if I want to store all four variables from my experiment in a data frame called `expt` I can do so like this:

[6] Please make sure that you've installed and loaded the `lsr` package.

```
> expt <- data.frame ( age, gender, group, score )
> expt
  age gender   group score
1  17   male group 1    12
2  19   male group 1    10
3  21   male group 1    11
4  37   male group 2    15
5  18   male group 2    16
6  19 female group 2    14
7  47 female group 3    25
8  18 female group 3    21
9  19 female group 3    29
```

Note that `expt` is a completely self-contained variable. Once you've created it, it no longer depends on the original variables from which it was constructed. That is, if we make changes to the original `age` variable, it will *not* lead to any changes to the age data stored in `expt`.

6.8.2 Pulling out the contents of the data frame using `$`

At this point, our workspace contains only the one variable, a data frame called `expt`. But as we can see when we told R to print the variable out, this data frame contains 4 variables, each of which has 9 observations. So how do we get this information out again? After all, there's no point in storing information if you don't use it, and there's no way to use information if you can't access it. So let's talk a bit about how to pull information out of a data frame.

The first thing we might want to do is pull out one of our stored variables, let's say `score`. One thing you might try to do is ignore the fact that `score` is locked up inside the `expt` data frame. For instance, you might try to print it out like this:

```
> score
Error: object 'score' not found
```

This doesn't work, because R doesn't go "peeking" inside the data frame unless you explicitly tell it to do so. There's actually a very good reason for this, which I'll explain in a moment, but for now let's just assume R knows what it's doing. How do we tell R to look inside the data frame? As is always the case with R there are several ways. The simplest way is to use the $ operator to extract the variable you're interested in, like this:

```
> expt$score
[1] 12 10 11 15 16 14 25 21 29
```

6.8.3 Getting information about a data frame

One problem that sometimes comes up in practice is that you forget what you called all your variables. Normally you might try to type `objects()` or `who()`, but neither of those commands will tell you what the names are for those variables inside a data frame! One way is to ask R to tell you what the *names* of all the variables stored in the data frame are, which you can do using the `names()` function:

```
> names(expt)
[1] "age"    "gender" "group"  "score"
```

An alternative method is to use the `who()` function, as long as you tell it to look at the variables inside data frames. If you set `expand = TRUE` then it will not only list the variables in the workspace, but it will "expand" any data frames that you've got in the workspace, so that you can see what they look like. That is:

```
> who(expand = TRUE)
   -- Name --    -- Class --    -- Size --
   expt          data.frame     9 x 4
    $age         numeric        9
    $gender      factor         9
    $group       factor         9
    $score       numeric        9
```

or, since `expand` is the first argument in the `who()` function you can just type `who(TRUE)`. I'll do that a lot in this book.

There's a lot more that can be said about data frames: they're fairly complicated beasts, and the longer you use R the more important it is to make sure you really understand them.

6.9 Lists

The next kind of data I want to mention are **lists**. Lists are an extremely fundamental data structure in R, and as you start making the transition from a novice to a savvy R user you will use lists all the time. I don't use lists very often in this book – not directly – but most of the advanced data structures in R are built from lists (e.g., data frames are actually a specific type of list). Because lists are so important to how R stores things, it's useful to have a basic understanding of them. Okay, so what is a list, exactly? Like data frames, lists are just "collections of variables." However, unlike data frames – which are basically supposed to look like a nice "rectangular" table of data – there are no constraints on what kinds of variables we include, and no requirement that the variables have any particular relationship to one another. In order to understand what this actually *means*, the best thing to do is create a list, which we can do using the `list()` function. If I type this as my command:

```
> huber <- list( age = 38,
+              nerd = TRUE,
+              parents = c("Edith","Michael")
+ )
```

R creates a new list variable called `huber`, which is a bundle of three different variables: `age`, `nerd` and `parents`. Notice, that the `parents` variable is longer than the others. This is perfectly acceptable for a list, but it wouldn't be for a data frame. If we now print out the variable, you can see the way that R stores the list:

```
> print( huber )
$age
[1] 38

$nerd
[1] TRUE

$parents
[1] "Edith" "Michael"
```

As you might have guessed from those $ symbols everywhere, the variables are stored in exactly the same way that they are for a data frame (again, this is not surprising: data frames *are* a type of list). So you will (I hope) be entirely unsurprised and probably quite bored when I tell you that you can extract the variables from the list using the $ operator, like so:

```
> huber$nerd
[1] TRUE
```

If you need to add new entries to the list, the easiest way to do so is to again use $, as the following example illustrates. If I type a command like this

```
> huber$children <- c("Rosa", "Ignaz", "Zita", "Alois")
```

then R creates a new entry to the end of the list called `children`, and assigns it four values. If I were now to `print()` this list out, you'd see a new entry at the bottom of the printout. Finally, it's actually possible for lists to contain other lists, so it's quite possible that I would end up using a command like `huber$children$age` to find out how old my children are.

6.10 Formulas

The last kind of variable that I want to introduce before finally being able to start talking about statistics is the **formula**. Formulas were originally introduced into R as a convenient way to specify a particular type of statistical model but they're such handy things that they've spread. Formulas are now used in a lot of different contexts, so it makes sense to introduce them early.

Stated simply, a formula object is a variable, but it's a special type of variable that specifies a relationship between other variables. A formula is specified using the "tilde operator" `~`. A very simple example of a formula is shown below:[7]

```
> formula1 <- out ~ pred
> formula1
out ~ pred
```

The *precise* meaning of this formula depends on exactly what you want to do with it, but in broad terms it means "the `out` (outcome) variable, analysed in terms of the `pred` (predictor) variable". That said, although the simplest and most common form of a formula uses the "one variable on the left, one variable on the right" format, there are others. For instance, the following examples are all reasonably common

```
> formula2 <-  out ~ pred1 + pred2   # more than one variable on the right
> formula3 <-  out ~ pred1 * pred2   # different relationship between predictors
> formula4 <-  ~ var1 + var2         # a 'one-sided' formula
```

and there are many more variants besides. Formulas are pretty flexible things, and so different functions will make use of different formats, depending on what the function is intended to do.

6.11 User-defined functions

One of the great strengths of R is the user's ability to add functions. Sometimes there is a small task (or series of tasks) you need done and you find yourself having to repeat it multiple times. In these types of situations it can be helpful to create your own custom function. The structure of a function is given below:

```
name_of_function <- function(argument1, argument2) {
    statements or code that does something
    return(something)
}
```

First you give your function a name. Then you assign value to it, where the value is the function.

When defining the function you will want to provide the list of arguments required (inputs and/or options to modify behavior of the function), and wrapped between curly brackets place the tasks that are being executed on/using those arguments. The argument(s) can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way.

Finally, you can "return" the value of the object from the function, meaning pass the value of it into the global environment. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function.

NOTE: You can also have a function that doesn't require any arguments, nor will it return anything.

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```
square_it <- function(x) {
   square <- x * x
   return(square)
}
```

Now, we can use the function as we would any other function. We type out the name of the function, and inside the parentheses we provide a numeric value `x`:

```
square_it(5)

## [1] 25
```

Pretty simple, right?

In this case, we only had one line of code that was run, but in theory you could have many lines of code to get obtain the final results that you want to "return" 'to the user. We have only scratched the surface here when it comes to creating functions! If you are interested please find more detailed information on this R-bloggers site[8].

[7] Note that, when I write out the formula, R doesn't check to see if the `out` and `pred` variables actually exist: it's only later on when you try to use the formula for something that this happens.

[8] https://www.r-bloggers.com/2014/06/how-to-write-and-debug-an-r-function/

6.12 Summary

This chapter continued where Chapter 5 left off. The focus was still primarily on introducing basic R concepts, but this time at least you can see how those concepts are related to data analysis:

- *Installing, loading and updating packages.* Knowing how to extend the functionality of R by installing and using packages is critical to becoming an effective R user (Section 6.2)
- *Getting around.* Section 6.3 talked about how to manage your workspace and how to keep it tidy. Similarly, Section 6.4 talked about how to get R to interact with the rest of the file system.
- *Loading and saving data.* Finally, we encountered actual data files. Loading and saving data is obviously a crucial skill, one we discussed in Section 6.5.
- *Useful things to know about variables.* In particular, we talked about special values, element names and classes (Section 6.6).
- *More complex types of variables.* R has a number of important variable types that will be useful when analysing real data. I talked about factors in Section 6.7, data frames in Section 6.8, lists in Section 6.9 and formulas in Section 6.10.
- *Generic functions.* How is it that some function seem to be able to do lots of different things? Section 6.11 tells you how.
- *Getting help.* Assuming that you're not looking for counselling, Section **??** covers several possibilities. If you are looking for counselling, well, this book really can't help you there. Sorry.

Taken together, Chapters 5 and 6 provide enough of a background that you can finally get started doing some data work! Yes, there's a lot more R concepts that you ought to know, but I think that we've talked quite enough about programming for the moment. It's time for you to get your hands on R! Don't be afraid, you can do it!

Chapter 7

The tidyverse universe

7.1 Introduction

Required reading:

- Wickham and Grolemund (2018, ch. 2). See: `https://r4ds.had.co.nz/workflow-basics.html`‘

Required exercise:

- My swirl learning module ‘tidyverse’. See 141.

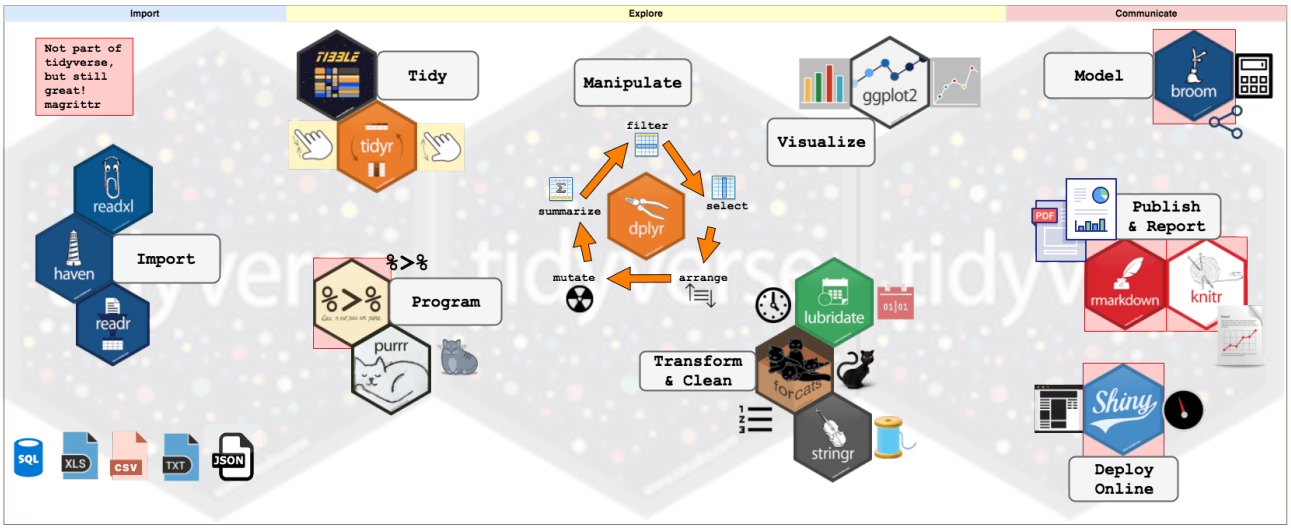


Figure 7.1: The tidyverse universe

If you want to work with R, I guess you should get known to the `tidyverse` package. The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. The core packages are ggplot2, dplyr, tidyr, readr, purrr, tibble, stringr, and forcats, which provide functionality to model, transform, and visualize data. An additional 12 packages assist the core. As of November 2018, the tidyverse package and some of its individual packages comprise 5 out of the top 10 most downloaded R packages. The tidyverse is the subject of multiple books and papers.

Install the complete tidyverse with:

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

See how the tidyverse makes data science faster, easier and more fun with the book *R for Data Science* from Wickham and Grolemund (2018). Read it online, buy the book or try another resource from the community, see e.g.: `www.tidyverse.org`. On YouTube you find tons of material that exemplifies how to use R.

7.2 The pipe operator

Required exercise:

- Exercise 'The pipe operator and others' on section 12.3. Also see https://htmlpreview.github.io/?https://github.com/hubchev/courses/blob/main/rmd/pipe-operator.html.

7.3 Data manipulation with `dplyr`

Required reading:

- Wickham and Grolemund (2018, ch. 3). See: https://r4ds.had.co.nz/transform.html

Wickham and Grolemund (2018, ch. 3) is about the five key dplyr functions who allow you to solve the vast majority of your data-manipulation challenges:

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarise()`).

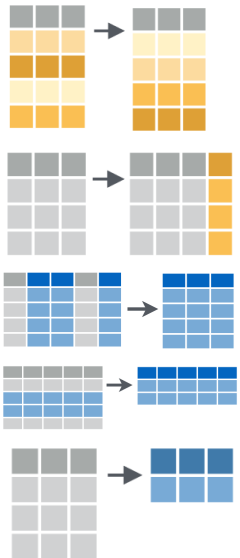
These functions can all be used in conjunction with `group_by()` , which changes the scope of each function from operating on the entire dataset to operating on it group-by-group.

All functions work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame.
3. The result is a new data frame.

Exercise 7.1 — dplyr cheatsheets (Solution → p. ??)

Here https://www.rstudio.com/resources/cheatsheets/ you find some cheatsheets on the `dplyr` package. Assign the following graphical sketches to one of the four functions mentioned above that are part of the `dplyr` package.



Exercise 7.2 — Subsetting (Solution → p. ??)

1. Load the following packages: `tidyverse`, `dplyr`, and `tibble`.
2. Check to see if you have the `mtcars` dataset by entering the command `mtcars`.
3. Save the `mtcars` dataset in an object named `cars`.
4. What class is `cars`?
5. How many observations (rows) and variables (columns) are in the mtcars dataset?
6. Rename mpg in cars to `MPG`. Use `rename()`.
7. Convert the column names of cars to all upper case. Use `rename_all`, and the `toupper` command.
8. Convert the rownames of `cars` to a column called `car` using `rownames_to_column`.
9. Subset the columns from `cars` that end in "`p`" and call it `pvars` using `ends_with()`.
10. Create a subset cars that only contains the columns: `wt`, `qsec`, and `hp` and assign this object to `carsSub`. (Use `select()`.)
11. What are the dimensions of `carsSub`? (Use `dim()`.)
12. Convert the column names of `carsSub` to all upper case. Use `rename_all()`, and `toupper()` (or `colnames()`).
13. Subset the rows of cars that get more than 20 miles per gallon (`mpg`) of fuel efficiency. How many are there? (Use `filter()`.)
14. Subset the rows that get less than 16 miles per gallon (`mpg`) of fuel efficiency and have more than 100 horsepower (`hp`). How many are there? (Use `filter()` and the pipe operator.)
15. Create a subset of the cars data that only contains the columns: `wt`, `qsec`, and `hp` for cars with 8 cylinders (`cyl`) and reassign this object to `carsSub`. What are the dimensions of this dataset? Don't use the pipe operator.
16. Create a subset of the cars data that only contains the columns: `wt`, `qsec`, and `hp` for cars with 8 cylinders (`cyl`) and reassign this object to `carsSub2`. Use the pipe operator.
17. Re-order the rows of `carsSub` by weight (`wt`) in increasing order. (Use `arrange()`.)
18. Create a new variable in carsSub called wt2, which is equal to wt^2, using `mutate()` and piping `%>%`.

Please find solutions here: `https://raw.githubusercontent.com/hubchev/courses/main/scr/exe_subset.R`

Chapter 8

Using graphs and visualizing data

Required readings:

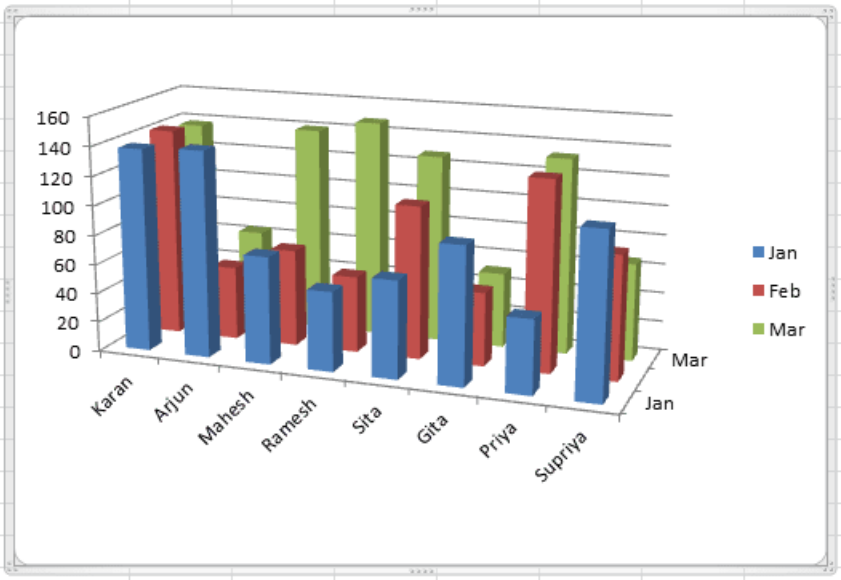
- *Using Graphs and Visualising Data* by Oliver Kirchkamp (2018): https://www.kirchkamp.de/oekonometrie/pdf/gra-p.pdf
- Wickham and Grolemund (2018, ch. 1): *Data Visualization with ggplot2*, See: https://r4ds.had.co.nz/data-visualisation.html

Required exercises:

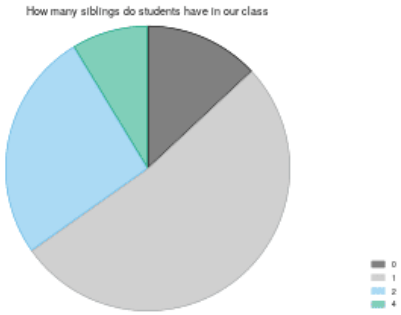
- The *Convergence* exercise found in section 12.5 and here: https://github.com/hubchev/courses/blob/main/scr/convergence.R
- The *Regression analysis presentation* that can be found in the Appendix of these notes on pages 125f and here: https://htmlpreview.github.io/?https://github.com/hubchev/courses/blob/main/rmd/regress_lecture.html

Exercise 8.1 — What makes a graph ugly? (Solution → p. ??)

Discuss what are the features of a good and bad graphical visualization of data. What do you think about the following graph?



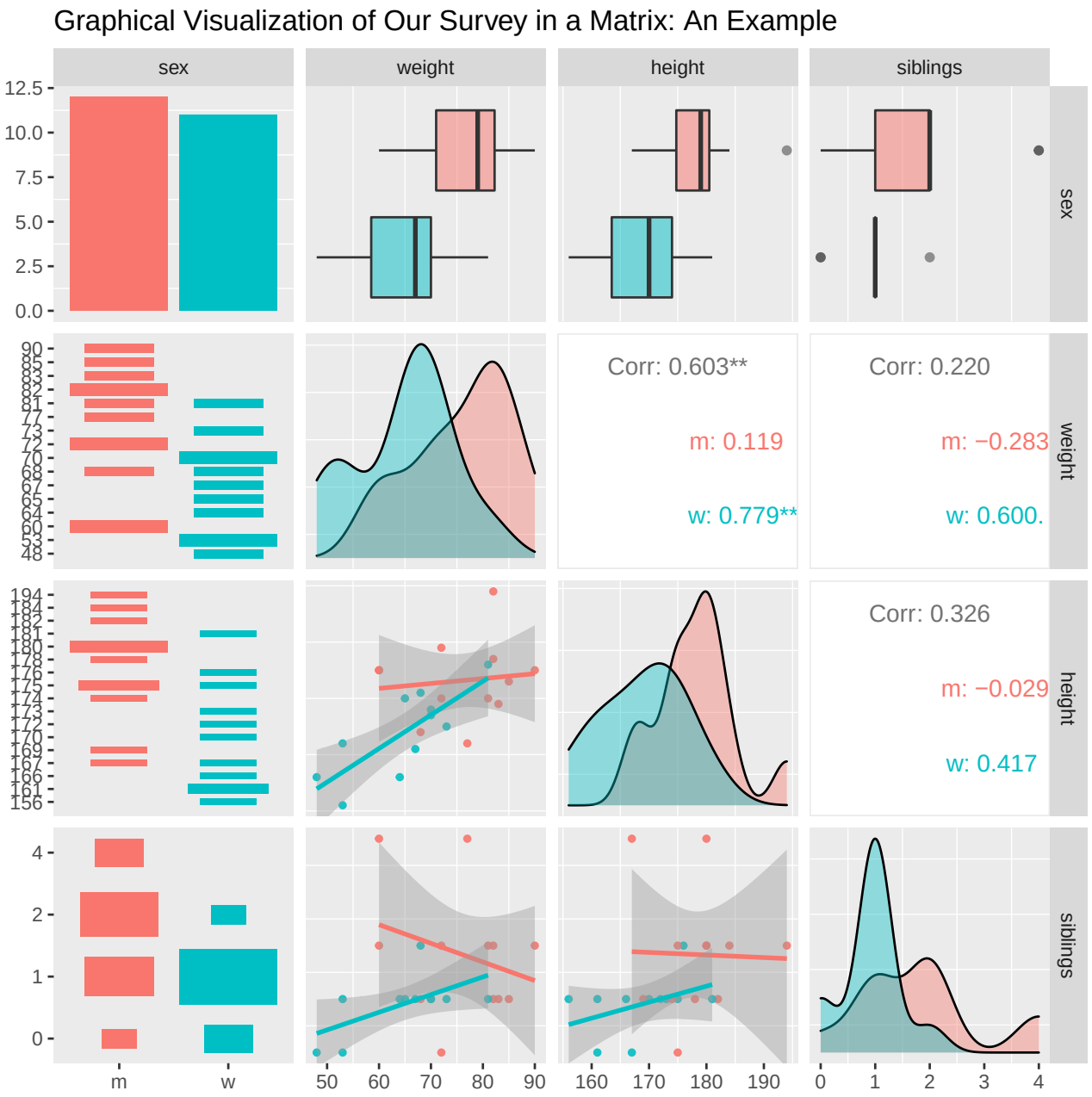
A note on pie chart A pie chart shows a circle which is divided into sectors that each represent a proportion of the whole.



Pie charts may look simple, but they're tricky to get right. The brain's not very good at comparing the size of angles and because there's no scale in pie plots, reading accurate values is difficult. Thus, pie charts are often poor at communicating data. They take up more space and are harder to read than alternatives. Unfortunately, people like to look at pies.

Matrix of Plots

```
read_csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/classdata.csv") %>%
        select(sex,weight,height,siblings) %>%
        ggpairs(.,
        title = "Graphical Visualization of Our Survey in a Matrix: An Example",
        mapping = ggplot2::aes(colour=sex),
        lower = list(continuous = wrap("smooth", alpha = 0.9, size=1.1),
        discrete = "blank", combo="count"),
        diag = list(discrete="barDiag",
        continuous = wrap("densityDiag", alpha=0.4 )),
        upper = list(combo = wrap("box_no_facet", alpha=0.5),
        continuous = wrap("cor", size=4, align_percent=0.8))) +
        theme(panel.grid.major = element_blank())    # remove gridlines
```



- Can you interpret each plot in the matrix above?
- Can you name the type of graph for each plot above?

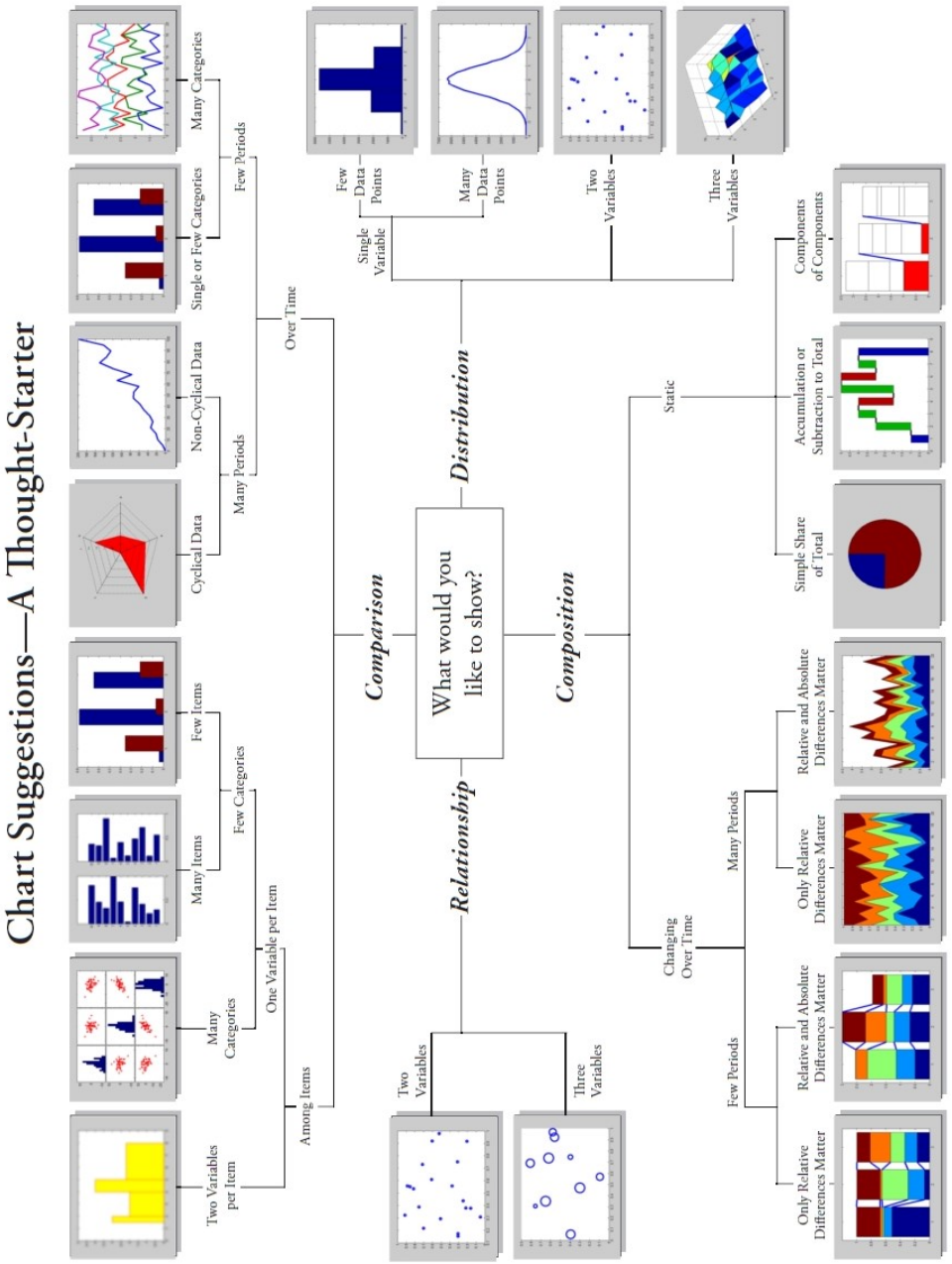
Do's and don'ts of data visualization

Graphical representation is another way of analyzing numerical data. Data visualization can be a great way to get insights. It can help to communicate a large amount of information simply and intuitively. However, you should avoid a few all-too-common mistakes.

- Graphs can be misleading and that may hold even true for *good* graphs sometimes in a way!
- An overview of different types of graphs: https://visme.co/blog/types-of-graphs/
- Nice animated graphs and the corresponding R code: https://www.r-graph-gallery.com/animation.html
- Good graphs are easy to understand and eye catching.
- Minimize colors and other attention-grabbing elements that are not directly related to the data of interest.
- Show the full scale of the graph, then zoom to show the data of interest, if necessary. In other words, don't truncate the an axis or change the scaling within an axis just to make you your story more dramatic.

- Label and describe your chart sufficiently so that everybody can fully understand the content of the shown data set and statistics, respectively.
- For more tips, see: https://guides.library.duke.edu/datavis/topten

Color blindness Worldwide, there are approximately 300 million people with color blindness. Most colour blind people are able to see things as clearly as other people but they are unable to fully *see* red, green or blue light. Thus, better rely on color schemes that are easy to *see* for colorblind people.



Chapter 9

Regression Analysis

Required readings:

- My presentation about regression analysis available at my github account: https://htmlpreview.github.io/?https://github.com/hubchev/courses/blob/main/rmd/regress_lecture.html and attached in the Appendix of these notes, see pages 125f.

9.1 Simple linear regression

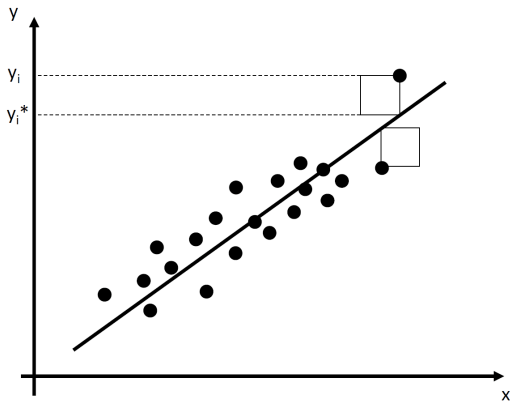
- Linear regression analysis is one of the most commonly used predictive modeling techniques. The aim of linear regression is to find a mathematical equation for a continuous response variable y as a function of one (simple linear model) or more variables (multiple linear regression), x. So that you can use this regression model to predict y when only the x is known.
- Of course, for a meaningful prediction you need more than two observations, i, that must be available for the variables of interest.
- The linear regression model is

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

  where
  - the index i runs over the observations, $i = 1, \ldots, n$
  - y_i is the **dependent variable**, the regressand, or simply the left-hand variable
  - x_i is the **independent variable**, the regressor, or simply the right-hand variable
  - β_0 is the **intercept** of the population regression line or the constant
  - β_1 is the slope of the population regression line
  - ϵ_i is the **error term** or the residual.

Estimating the coefficients of the linear regression model

- In practice, the intercept and slope of the regression are unknown. Therefore, we must employ data to estimate both unknown parameters.
- The method we use to *fit a line* is called the ordinary least squared (OLS) method. The idea is to minimize the sum of the squared differences of all y_i and y_i^* as sketched in the plot below.



- In more formal words, we try minimize the squared residuals by choosing the estimated coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$

$$\min_{\hat{\beta}_0,\hat{\beta}_1} \sum_{i=1} \epsilon_i^2 = \sum_{i=1} \left[y_i - \underbrace{(\hat{\beta}_0 + \hat{\beta}_1 x_i)}_{\text{predicted values} \equiv y_i^*} \right]^2$$
$$\Leftrightarrow = \sum_{i=1} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

- Minimizing the function requires to calculate the first order conditions with respect to alpha and beta and set them zero:

$$\frac{\partial \sum_{i=1} \epsilon_i^2}{\partial \beta_0} = 2 \sum_{i=1} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) = 0$$
$$\frac{\partial \sum_{i=1} \epsilon_i^2}{\partial \beta_1} = 2 \sum_{i=1} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) x_i = 0$$

- This is just a linear system of two equations with two unknowns β_0 and β_1, which we can mathematically solve for β_0:

$$\sum_{i=1} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i) = 0$$
$$\Leftrightarrow \hat{\beta}_0 = \frac{1}{n} \sum_{i=1} (y_i - \hat{\beta}_1 x_i)$$
$$\Leftrightarrow \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

- and for β_1:

$$
\begin{aligned}
&\sum_{i=1}(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)x_i = 0 \\
\Leftrightarrow &\sum_{i=1} y_i x_i - \underbrace{\hat{\beta}_0}_{\bar{y}-\hat{\beta}_1\bar{x}} x_i - \hat{\beta}_1 x_i^2 = 0 \\
\Leftrightarrow &\sum_{i=1} y_i x_i - (\bar{y} - \hat{\beta}_1\bar{x})x_i - \hat{\beta}_1 x_i^2 = 0 \\
\Leftrightarrow &\sum_{i=1} y_i x_i - \bar{y}x_i - \hat{\beta}_1\bar{x}x_i - \hat{\beta}_1 x_i^2 = 0 \\
\Leftrightarrow &\sum_{i=1}(y_i - \bar{y} - \hat{\beta}_1\bar{x} - \hat{\beta}_1 x_i)x_i = 0 \\
\Leftrightarrow &\sum_{i=1}(y_i - \bar{y})x_i - \hat{\beta}_1(\bar{x} - x_i)x_i = 0 \\
\Leftrightarrow &\sum_{i=1}(y_i - \bar{y})x_i = \hat{\beta}_1 \sum_{i=1}(\bar{x} - x_i)x_i \\
\Leftrightarrow &\hat{\beta}_1 = \frac{\sum_{i=1}(y_i - \bar{y})x_i}{\sum_{i=1}(\bar{x} - x_i)x_i} \\
\Leftrightarrow &\hat{\beta}_1 = \frac{\sum_{i=1}(y_i - \bar{y})(x_i - \bar{x})}{\sum_{i=1}(\bar{x} - x_i)^2} \\
\Leftrightarrow &\hat{\beta}_1 = \frac{\sigma_{x,y}}{\sigma_x^2}
\end{aligned}
$$

- The estimated regression coefficient $\hat{\beta}_1$ equals the covariance between y and x divided by the variance of x.
- The formulas presented above may not be very intuitive at first glance. On https://www.econometrics-with-r.org/4-2-estimating-the-coefficients-of-the-linear-regression-model.html however, an interactive application can be found. This application aims to help you understand the mechanics of OLS. You can add observations by clicking into the coordinate system where the data are represented by points. Once two or more observations are available, the application computes a regression line using OLS and some statistics which are displayed in the right panel. The results are updated as you add further observations to the left panel. A double-click resets the application, i.e., all data are removed.

The least squares assumptions

OLS performs well under a quite broad variety of different circumstances. However, there are some assumptions which need to be satisfied in order to ensure that the estimates are normally distributed in large samples.

The Least Squares Assumptions should fullfil the following assumptions:

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i, \ i = 1, \dots, n$$

1. The error term ϵ_i has conditional mean zero given X_i : $E(u_i|X_i) = 0$.
2. $(Xi, Yi), i = 1, \dots, n$ are independent and identically distributed (i.i.d.) draws from their joint distribution.
3. Large outliers are unlikely: X_i and Y_i have nonzero finite fourth moments. That means, assumption 3 requires that X and Y have a finite kurtosis.

Measures of fit

After fitting a linear regression model, a natural question is how well the model describes the data. Visually, this amounts to assessing whether the observations are tightly clustered around the regression line. Both the coefficient of determination and the standard error of the regression measure how well the OLS Regression line fits the data.

R^2 the coefficient of determination, is the fraction of the sample variance of Y_i that is explained by X_i. Mathematically, the R^2 can be written as the ratio of the explained sum of squares to the total sum of squares. The explained sum of squares (ESS) is the sum of squared deviations of the predicted values $\hat{Y}_i$, from the average of the Y_i. The total sum of squares (TSS) is the sum of squared deviations of the Y_i from their average. Thus

we have

$$ESS = \sum_{i=1}^{n} \left(\hat{Y}_i - \overline{Y}\right)^2, \tag{9.1}$$

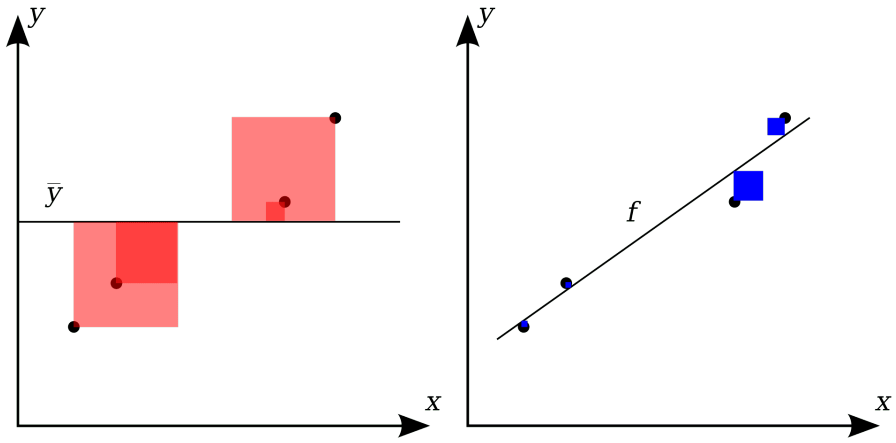
$$TSS = \sum_{i=1}^{n} \left(Y_i - \overline{Y}\right)^2, \tag{9.2}$$

$$R^2 = \frac{ESS}{TSS}. \tag{9.3}$$

Since $TSS = ESS + SSR$ we can also write

$$R^2 = 1 - \frac{SSR}{TSS}$$

with $SSR = \sum_{i=1}^{n} \epsilon^2$



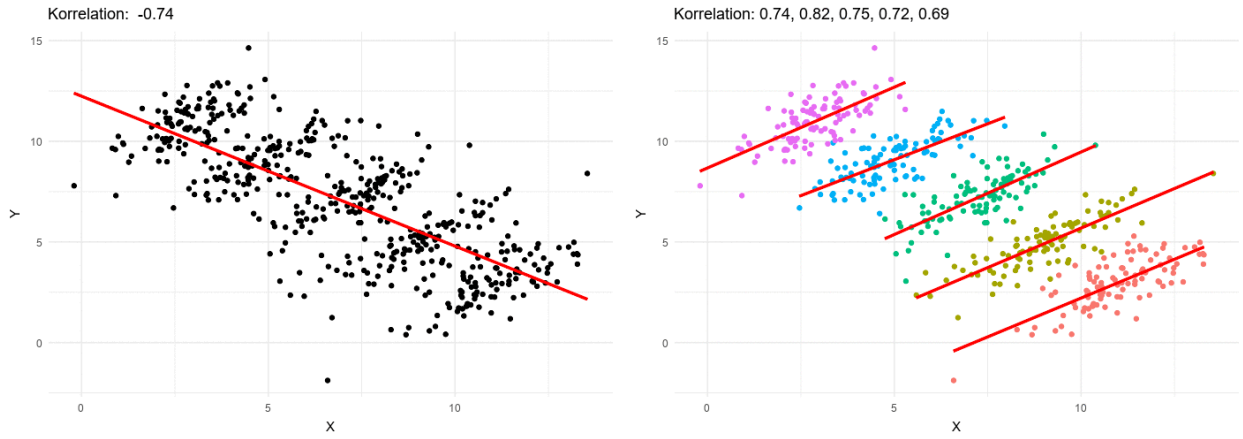
R^2 lies between 0 and 1. It is easy to see that a perfect fit, i.e., no errors made when fitting the regression line, implies $R2 = 1$ since then we have $SSR = 0$. On the contrary, if our estimated regression line does not explain any variation in the Y_i, we have $ESS = 0$ and consequently $R^2 = 0$.

9.2 Multiple linear regression

In what follows I introduce linear regression models that use more than just one explanatory variable and discuss important key concepts in multiple regression. As we broaden our scope beyond the relationship of only two variables (the dependent variable and a single regressor) some potential new issues arise such as *multicollinearity* and *omitted variable bias* (OVB). In particular, this section deals its implication for causal interpretation of OLS-estimated coefficients.

Simpsons paradox

Simpson's paradox is an effect that occurs when the marginal association between two categorical variables is qualitatively different from the partial association between the same two variables **after controlling for one or more other variables**. Simpson's paradox is important for three critical reasons. First, people often expect statistical relationships to be immutable. They often are not. The relationship between two variables might increase, decrease, or even change direction depending on the set of variables being controlled. Second, Simpson's paradox is not simply an obscure phenomenon of interest only to a small group of statisticians. Simpson's paradox is actually one of a large class of association paradoxes. Third, Simpson's paradox reminds researchers that causal inferences, particularly in nonexperimental studies, can be hazardous. Uncontrolled and even unobserved variables that would eliminate or reverse the association observed between two variables might exist.



Example: Understanding Simpson's paradox is easiest in the context of a simple example. Suppose that a university is concerned about sex bias during the admission process to graduate school. To study this, applicants to the university's graduate programs are classified based on sex and admissions outcome. These data would seem to be consistent with the existence of a sex bias because men (40 percent were admitted) were more likely to be admitted to graduate school than women (25 percent were admitted).

To identify the source of the difference in admission rates for men and women, the university subdivides applicants based on whether they applied to a department in the natural sciences or to one in the social sciences and then conducts the analysis again. Surprisingly, the university finds that the direction of the relationship between sex and outcome has reversed. In natural science departments, women (80 percent were admitted) were more likely to be admitted to graduate school than men (46 percent were admitted); similarly, in social science departments, women (20 percent were admitted) were more likely to be admitted to graduate school than men (4 percent were admitted).

Although the reversal in association that is observed in Simpson's paradox might seem bewildering, it is actually straightforward. In this example, it occurred because both sex and admissions were related to a third variable, namely, the department. First, women were more likely to apply to social science departments, whereas men were more likely to apply to natural science departments. Second, the acceptance rate in social science departments was much less than that in natural science departments. Because women were more likely than men to apply to programs with low acceptance rates, when department was ignored (i.e., when the data were aggregated over the entire university), it seemed that women were less likely than men to be admitted to graduate school, whereas the reverse was actually true. Although hypothetical examples such as this one are simple to construct, numerous real-life examples can be found easily in the social science and statistics literature.

Regression model and estimation

The multiple regression model is

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{3i} + \cdots + \beta_k X_{ki} + u_i, \ \ i = 1, \ldots, n.$$

How can we estimate the coefficients of the multiple regression model? As in the simple model, we seek to minimize the sum of squared mistakes by choosing estimated coefficients $\beta_0, \beta_1, \ldots, \beta_k$ such that

$$\sum_{i=1}^{n} (Y_i - b_0 - b_1 X_{1i} - b_2 X_{2i} - \cdots - b_k X_{ki})^2$$

This demands matrix notation. This goes beyond the scope of this introduction. I refer to the open book *Applied Statistics with R*, see: https://daviddalpiaz.github.io/appliedstats/multiple-linear-regression.html. Here the derivation of the OLS estimator is explained in further detail.

Confounding variables

A confounding variable is a characteristic which was not included or controlled for in the study, but can influence the results. That is, the real effects due to the treatment are confounded, or clouded, due to this variable.

For example, if you select a group of people who take vitamin C daily, and a group who don't, and follow them all for a year's time counting how many colds they get, you might notice the group taking vitamin C had fewer colds than the group who didn't take vitamin C. However, you cannot conclude that vitamin C reduces colds. Because this was not a true experiment but rather an observational study, there are many confounding

variables at work. One possible confounding variable is the person's level of health consciousness; people who take vitamins daily may also wash their hands more often, thereby heading off germs.

How do researchers handle confounding variables? **Control** is what it's all about. Here you could pair up people who have the same level of health-consciousness and randomly assign one person in each pair to taking vitamin C each day (the other person gets a fake pill). Any difference in number of colds found between the groups is more likely due to the vitamin C, compared to the original observational study. Good experiments control for potential confounding variables.

Suppose a researcher claims that eating seaweed helps you live longer; you read interviews with the subjects and dis- cover that they were all over 100, ate very healthy foods, slept an average of 8 hours a day, drank a lot of water, and exer- cised. Can we say the long life was caused by the seaweed? You can't tell, because so many other variables exist that could also promote long life (the diet, the sleeping, the water, the exercise); these are all confounding variables. A common error in research studies is to fail to control for confounding variables, leaving the results open to scrutiny. The best way to head off confounding variables is to do a well- designed experiment in a controlled setting. Observational studies are great for surveys and polls, but not for showing cause-and-effect relationships, because they don't control for confounding variables. A well-designed experiment provides much stronger evidence.

Exercise 9.1 — Look at the Output (Solution → p. ??)

price	Coef.	Std. Err.	t	P>|t|
weight	3.464706	.630749	5.49	0.000
mpg	21.8536	74.22114		
foreign	3673.06	683.9783	5.37	
_cons	-5853.696	3376.987		0.087

Above you see an excerpt of a regression output taken from a statistical program. Some t-values and p-values are missing.

a) Calculate the t-value of the coefficient `mpg`. Is the coefficient at a level of $\alpha = 0.05$ statistically significant?

b) Is the coefficient `foreign` at a level of $\alpha = 0.05$ statistically significant?

c) Is the constant (i.e., `_cons`) at a level of $\alpha = 0.05$ statistically significant?

Exercise 9.2 — Look at Stata Output (Solution → p. ??)

Below you find two regression outputs from Stata. Try to interpret the p-values and the confidence intervals. How are the t-values calculated. Can you use the *magic number* 1.96 to say if a corresponding estimated coefficient is statistically significant, or not? Which estimated model is *better*?

```
. reg weight height sex_n
```

Source	SS	df	MS	Number of obs	=	23
				F(2, 20)	=	7.02
Model	1122.42818	2	561.214089	Prob > F	=	0.0049
Residual	1599.05008	20	79.9525041	R-squared	=	0.4124
				Adj R-squared	=	0.3537
Total	2721.47826	22	123.703557	Root MSE	=	8.9416

weight	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
height	.5923091	.2671132	2.22	0.038	.0351207	1.149498
sex_n	-5.78938	4.477272	-1.29	0.211	-15.12881	3.550045
_cons	-23.74037	50.40112	-0.47	0.643	-128.8753	81.39453

```
. reg weight height sex_n I_sex_height
```

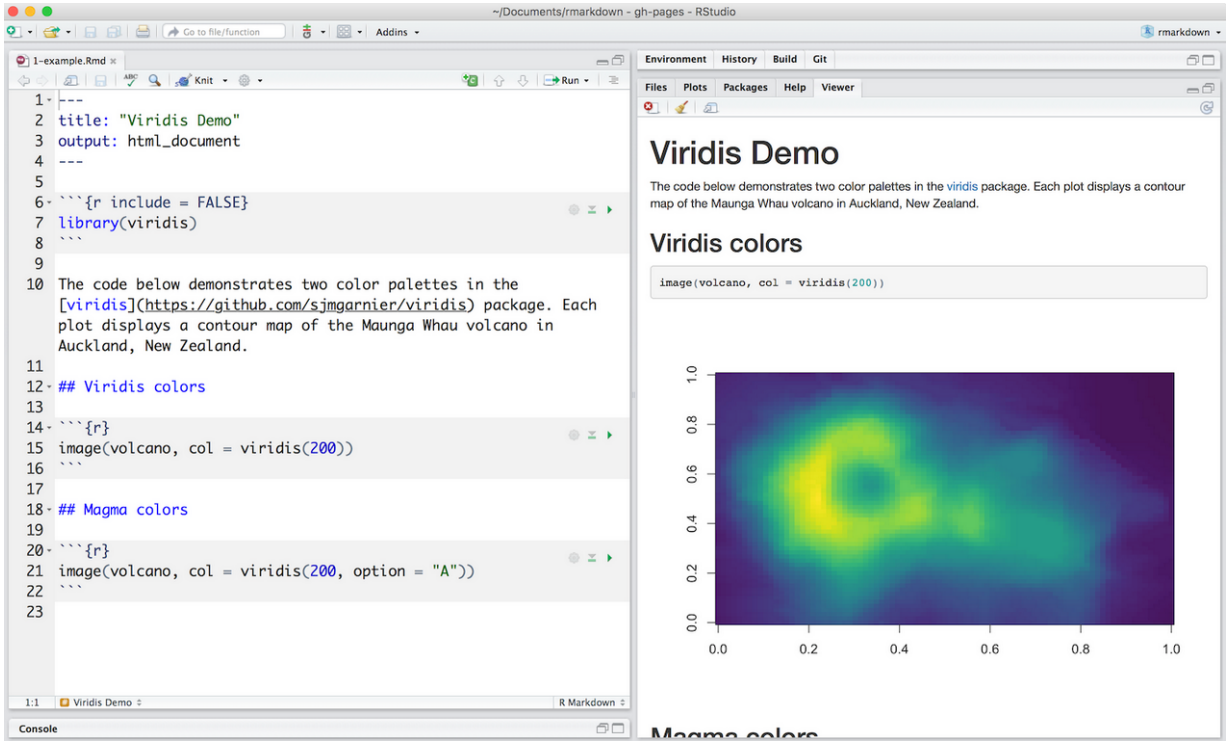
Source	SS	df	MS	Number of obs	=	23
				F(3, 19)	=	6.02
Model	1326.55663	3	442.185544	Prob > F	=	0.0046
Residual	1394.92163	19	73.4169278	R-squared	=	0.4874
				Adj R-squared	=	0.4065
Total	2721.47826	22	123.703557	Root MSE	=	8.5684

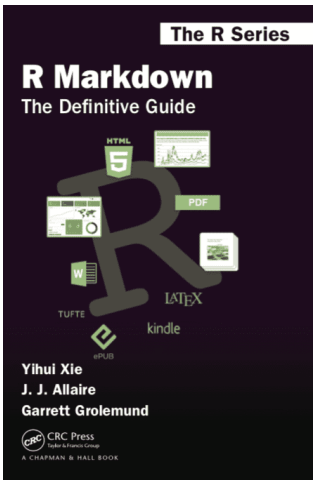
weight	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
height	-.6916747	.8114546	-0.85	0.405	-2.390069	1.006719
sex_n	-153.9612	88.96469	-1.73	0.100	-340.1665	32.244
I_sex_height	.8536423	.5119438	1.67	0.112	-.2178683	1.925153
_cons	201.104	143.2315	1.40	0.176	-98.6829	500.8909

Chapter 10

Markdown

Markdown provides an authoring framework for data science. You can use a single Markdown file to both save and execute code and generate high quality reports that can be shared with an audience. Please have a look at the official tutorial of Studio : https://rmarkdown.rstudio.com/lesson-1.html





The book *R Markdown: The Definitive Guide* by Xie et al. (2018) offers a comprehensive introduction. The online version of the book is regularly updated and free of costs: https://bookdown.org/yihui/rmarkdown/

Chapter 11

Git and GitHub

11.1 Git: Version control system

- Here you find always up to data reference manuals, books, videos and links that help you to learn, understand, and apply git: https://git-scm.com/doc
- Chacon and Straub (2022) is a detailed book (freely available) that answers all your questions about Git (and github): https://git-scm.com/book/en/v2
- Here you find a cheatsheet for git: https://training.github.com/

11.2 GitHub: platform to share and collaborate

GitHub.com is an online platform "where the world builds software". It hosts files for software development and offers the distributed version control functionality of Git, plus its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, continuous integration, and wikis for every project.

It is commonly used to host open-source projects. As of May 2022, GitHub reports having over 83 million developers, more than 4 million organizations, and more than 200 million repositories. It is currently the largest source code host and I guess that will not change in the future.

- A quick introduction can be found here: https://docs.github.com/en/get-started/quickstart/hello-world
- For a quick start I highly recommend to make your first contribution following this tutorial: https://github.com/firstcontributions/first-contributions

Chapter 12

Exercises

12.1 Run a script

Download this R-Script[1] and try to run and use it in RStudio. This script shows some important features of R and in particular, it installs the most important packages for you. In order to avoid problems in some examples later on in this course, please run this script.

```
## ----------------------------
## Script name: SH-R-01
## Purpose of script: First script students should run
## ----------------------------
## Author: Dr. Stephan Huber
## Date Created: 2020-09-30
## Copyright (c) Stephan Huber, 2020
## Email: Stephan.Huber(at)hs-fresenius.de
## ----------------------------


# Let us start with using R within RStudio
# with a # you can write a comment in your script

# with "Ctrl+Enter" you can execute a line of command, try it out:
print("Hello, world!")

# we use the book: "R for Dummies" in the course
# when reading the book, always try to execute the code snippets by your own
# For example, p. 3:
# Simulate 1 million throws of two six-sided dices:
set.seed(42)
throws <- 1e6
dice <- replicate(2,
sample(1:6, throws, replace = TRUE)
)
table(rowSums(dice))

# with a "?" you get some help/information about the functions and operators used
?set.seed
?assignOps
?sample
?hist
?table

# there are other ways to get help and learn within R
```

[1] https://raw.githubusercontent.com/hubchev/R-Intro/master/scripts/SH-R-01.R **Hint:** make a right-click and do *save as*.

```
help.start()
example(hist)
apropos("hist")
vignette()
vignette("ggwordcloud")

# Performing multiple calculations with vectors
24 + 7 + 11
1 + 2 + 3 + 4 + 5
14 / 2
# but
14 : 2 # this operator is called sequence
3*2
3      *       2
# but > 3 x 2 does not work!
x <- 1:5  # see assignment operator ?assignOps
x
x + 2
x + 6:10
x
# alternatively:
x2 <- x + 6:10
x3 <- 6:10
x4 <- x + x3

# To construct a vector, type into the console:
c(1, 2, 3, 4, 5)
?c()
b <- c(1, 2, 3, 4, 5)
b

# Packages are sort of R's brain so let's
# SEE WHAT PACKAGES ARE AROUND
# https://www.r-pkg.org/
# https://cran.r-project.org/web/views/
# For example tidyvers and ggplot2 are powerfull packages to create data-visualizations, se
# https://www.r-graph-gallery.com/

# LOAD PACKAGES
# There are a lot of packages pre-installed but not loaded, see here:
installed.packages()

# A package can be installed using install.packages("<package name>").
install.packages("dplyr")
# A package can be removed using remove.packages("<package name>").
remove.packages("dplyr")

# I recommend "pacman" for managing add-on packages. It will
# install packages, if needed, and then load the packages.
install.packages("pacman")

# Then load the package by using either of the following:
require(pacman)  # Gives a confirmation message.
library(pacman)  # No message.

# Or, by using "pacman::p_load" you can use the p_load
# function from pacman without actually loading pacman.
# These are packages I load every time.
pacman::p_load(pacman, dplyr, GGally, ggplot2, ggthemes,
ggvis, httr, lubridate, plotly, rio, rmarkdown, shiny,
stringr, tidyr)
```

```
# the package "datasets" contains some datasets
library(datasets)  # Load/unload base packages manually
?datasets
library(help = "datasets")

# CLEAN UP ##########################################################

# Clear packages
p_unload(dplyr, tidyr, stringr) # Clear specific packages
p_unload(all)  # Easier: clears all add-ons

detach("package:datasets", unload = TRUE)  # For base

# Clear console
cat("\014")  # shortut for doing that is ctrl+L
```

12.2 How to load R built-in R data set mtcars and explore the data

Go to https://t1p.de/R-mtcars andsb-d write a script that contains the content of the page. Try to understand what is going on. The pdf version of the file can be found on the following pages.

How to load R Built-in R Data set Mtcars and explore the data

R comes with several built-in data sets, which are generally used as demo data for playing with R functions.

In this vingette,we will describe how to load and use R built-in data sets focusing on the Mtcar dataset.

We will be exploring the basic functions of the dataset using a few basic exploration R functions.

List of pre-loaded data

Once you start your R program, there are example data sets available within R along with loaded packages.

If you just want to play with some test data to see how they load and what basic functions you can run, the default installation of R comes with several data sets.

The benefits of starting off using the pre-loaded data is that it gives you a chance to try analysis and plotting commands and there are a lot of online tutorials that use these sample sets.

To see the list of pre-loaded data, type the function data() into the R console and you will get a listing of pre-loaded data sets:

```
data()
```

Loading a built-in R data

Load and print mtcars data as follow:

Loading

```
data(mtcars)
```

Print the first 6 rows

This functions will allow you to view the first 6 rows of the data set.

```
head(mtcars, 6)
```

```
##                   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
```

If you want learn more about mtcars data sets, type this:

```
?mtcars
```

```
## starting httpd help server ... done
```

Most used R built-in data sets

mtcars: Motor Trend Car Road Tests

The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models)

Format

A data frame with 32 observations on 11(numeric) variables.

View the content of mtcars data set:

1. Loading

```
data("mtcars")
```

2. Print

```
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
```

It contains 32 observations and 11 variables: # Number of rows (observations)

```
nrow(mtcars)
```

```
## [1] 32
```

[1] 32 # Number of columns (variables)

```
ncol(mtcars)
```

```
## [1] 11
```

If you want to learn more about mtcars, type this:

```
?mtcars
```

select head of mtcars data set

```
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
```

select end of mtcars data set

```
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
```

summaries the data set

```
summary(mtcars)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

quantiles of dataset

```
quantile(mtcars$wt)
```

```
##      0%     25%     50%     75%    100% 
## 1.51300 2.58125 3.32500 3.61000 5.42400
```

select quantiles by percent

To calculate the quantiles by percent:

```
quantile(mtcars$wt, c(.2, .4, .8))
```

```
##   20%   40%   80% 
## 2.349 3.158 3.770
```

variance of weight

To calculate the variance of weight:

```
var(mtcars$wt)
```

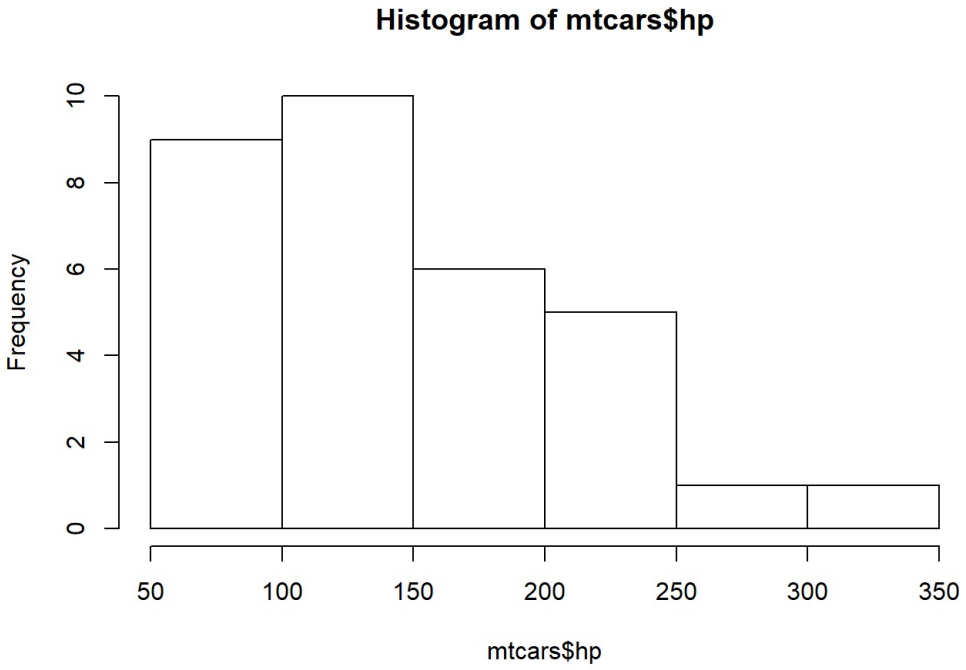
```
## [1] 0.957379
```

Historgram

Histograms are a classic way to assess the shape of the distribution of a single variable.

To get the histogram of hp, the code below will produce a histogram:

```
hist(mtcars$hp)
```



Summary

We explored some of the common functions in R that I like to use to explore a data frame before I conduct any statistical analysis. We used the built-in data set mtcars to illustrate these functions.

12.3 The pipe operator and others

The Pipe Operator

Stephan Huber

14 12 2021

What is the pipe operator, %>% in R?

The pipe operator, %>%, comes from the magrittr package which is also a part of the tidyverse package. The pipe is to help you write code in a way that is easier to read and understand. As R is a functional language, code often contains a lot of parenthesis, (and). Nesting those parentheses together is complex and you easily get lost. This makes your R code hard to read and understand. Here's where %>% comes in to the rescue! Consider the following chunk of code to explain the usage of the pipe:

```
# create some data `x`
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
x
```

```
## [1] 0.109 0.359 0.630 0.996 0.515 0.142 0.017 0.829 0.907
```

```
# take the logarithm of `x`,
x2 <- log(x)
x2
```

```
## [1] -2.216407397 -1.024432890 -0.462035460 -0.004008021 -0.663588378
## [6] -1.951928221 -4.074541935 -0.187535124 -0.097612829
```

```
# compute the lagged and iterated differences (see `diff()`)
x3 <- diff(x2)
x3
```

```
## [1]  1.19197451  0.56239743  0.45802744 -0.65958036 -1.28833984 -2.12261371
## [7]  3.88700681  0.08992229
```

```
# compute the exponential function
x4 <- exp(x3)
x4
```

```
## [1]  3.2935780  1.7548747  1.5809524  0.5170683  0.2757282  0.1197183 48.7647059
## [8]  1.0940893
```

```
#  Make yourself familiar with the functions round() and round the result (1 digit)
x5 <- round(x4, 1)
x5
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

That is rather long and we actually don't need objects x2, x3, and x4. Well, then let us write that in a nested function:

```
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)

round(exp(diff(log(x))), 1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

This is short but you easily loose overview. The solution is the *pipe*:

```
# load one of these packages: `magrittr` or `tidyverse`
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──
```

```
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.2     ✓ dplyr   1.0.7
## ✓ tidyr   1.1.3     ✓ stringr 1.4.0
## ✓ readr   2.1.1     ✓ forcats 0.5.1
```

```
## ── Conflicts ─────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
# Perform the same computations on `x` as above
x %>% log() %>%
    diff() %>%
    exp() %>%
    round(1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

You can read the %>% with “and then” because it takes the results of some function “and then” does something with that in the next. Another example can be found in this short clip: Using the pipe operator in R (https://youtu.be/PX5NuteZ3Vg)

Read out loud the following code:

```
library("datasets")
iris %>%
  subset(Sepal.Length > 5) %>%
  aggregate(. ~ Species, ., mean)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa     5.313636    3.713636     1.509091   0.2772727
## 2 versicolor     5.997872    2.804255     4.317021   1.3468085
## 3  virginica     6.622449    2.983673     5.573469   2.0326531
```

A solution may be the following: “you take the Iris data *and then* you subset the data *and then* you aggregate the data and show the mean”.

What sort of *extract operators* exist in R and how can they be used?

The extract operator (https://stat.ethz.ch/R-manual/R-devel/library/base/html/Extract.html) is used to retrieve data from objects in R. The operator may take four forms, including `[`, `[[`, `$`, and `@`. The fourth form, `@`, is called the slot operator (https://stat.ethz.ch/R-manual/R-devel/library/methods/html/slot.html), and is a more advanced topic so we won’t discuss it here.

The first form, `[`, can be used to extract content from vector, lists, or data frames. Since vectors are one dimensional, i.e., they contain between 1 and N elements, we apply the extract operator to the vector as a single number or a list of numbers as follows.

```
x[ selection criteria here ]
```

The following code defines a vector and then extracts the last 3 elements from it using two techniques. The first technique directly references elements 13 through 15. The second approach uses the length of the vector to calculate the indexes of last three elements.

```
    x <- 16:30 # define a vector
    x[13:15] # extract last 3 elements
```

```
## [1] 28 29 30
```

```
    x[(length(x)-2):length(x)] # extract last 3 elements
```

```
## [1] 28 29 30
```

When used with a list, `[` extracts one or more elements from the list. When used with a data frame, the extract operator can select rows, columns, or both rows and columns. Therefore, the extract opertor takes the following form: rows then a comma, then columns.

```
x[select criteria for rows , select criteria for columns]
```

The second and third forms of the extract operator, `[[` and `$` extract a single item from an object. It is used to refer to an element in a list or a column in a data frame. The easiest way to see how the various features of the extract operator work is to get through some examples.The following snippetsuse the `mtcars` data set from the `datasets` package.

```
    library(datasets)
    data(mtcars)

    # Here, we set up a column name in a variable to illustrate use
    # of various forms of the extract operator with a column name stored in
    # another R object
    theCol <- "cyl"

    # approach 1: use [[ form of extract operator to extract a column
    #             from the data frame as a vector
    #             this works because a data frame is also a list
    mtcars[[theCol]]
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
    # approach 2: use variable name in column dimension of data frame
    mtcars[,theCol]
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
    # approach 3: use the $ form of extract operator. Note that since this
    #             form accesses named elements from the list, you can't use
    #             variable substitution (e.g. theCol) with this version of
    #             extract
    mtcars$cyl
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
    # this version fails because the `$` version of extract does not
    # work with variable substitution (i.e. a computed index)
    mtcars$theCol
```

```
## NULL
```

x$y is actually just a short form for x[[“y”]].

The difference of `[ ]` and `[[ ]]` is that `[[ ]]` is used to access a component in a list or matrix whereas `[ ]` is used to access a single element in a matrix or array.

```
object <- list(a = 5, b = 6)

object ['a']
```

```
## $a
## [1] 5
```

```
object [['a']]
```

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

What is the *Console* in RStudio good for?

You can also execute R code straight in the console. This is a good way to experiment with R code, as your submission is not checked for correctness.

Creating sequences

We just learned about the `c()` operator, which forms a vector from its arguments. If we’re trying to build a vector containing a sequence of numbers, there are several useful functions at our disposal. These are the colon operator `:` and the sequence function `seq()`.

`:` Colon operator:

```
1:10 # Numbers 1 to 10
```

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

```
127:132 # Numbers 127 to 132
```

```
## [1] 127 128 129 130 131 132
```

`seq` function: `seq(from, to, by)`

```
seq(1,10,1) # Numbers 1 to 10
```

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

```
seq(1,10,2) # Odd numbers from 1 to 10
```

```
## [1] 1 3 5 7 9
```

```
seq(2,10,2) # Even numbers from 2 to 10
```

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

(a) Use `:` to output the sequence of numbers from 3 to 12

```
3:12
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12
```

(b) Use `seq()` to output the sequence of numbers from 3 to 30 in increments of 3

```
seq(3, 30, 3)
```

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

(c) Save the sequence from (a) as a variable `x`, and the sequence from (b) as a variable `y`. Output their product `x*y`

```
x <- 3:12
y <- seq(3, 30, 3)
x * y
```

```
##  [1]   9  24  45  72 105 144 189 240 297 360
```

Solve the following exercises in your own R-Script.

1. RStudio offers a lot of helpful so-called Cheat Sheets (see: https://rstudio.com/resources/cheatsheets/ (https://rstudio.com/resources/cheatsheets/)) Download and make yourself familiar with the following Base R Cheeat Sheet (https://rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf)
2. Open RStudio, create a R-script, set your working directory, load the data package, calculate *3 + 4* in R and add a comment to it.
3. Further, calculate

- divide 697 by 41
- take the square root of 12
- take 3 to the power of 12

2. Create a vector called vec1 that contain the numbers 2 5 8 12 16.
3. Use `x:y` notation to make (select) a second vector called vec2 containing the numbers 5 to 9.
4. Subtract vec2 from vec1 and look at the result.
5. Use seq() to make a vector of 100 values starting at 2 and increasing by 3 each time. Name the new vector nseries. (Hint: The example of the *creating sequences* section may be helpful.)
6.

- Extract the values at positions 5,10,15 and 20 in the vector of values you just created.
- Extract the values at positions 10 to 30
- Hint: Both of these actions require making a selection in the vector using the [ ] notation. Inside the square brackets you put a vector of index positions, so the problem here is to create the vector of index positions.

Solution:

```
697 / 41
```

```
## [1] 17
```

```
sqrt(12)
```

```
## [1] 3.464102
```

```
3 ^ 12
```

```
## [1] 531441
```

```
vec1 <- c(2,5,8,12,16)
vec2 <- 5:9
vec1 - vec2
```

```
## [1] -3 -1  1  4  7
```

```
number.series <- seq(from=2,by=3,length.out=100)
number.series
```

```
##   [1]   2   5   8  11  14  17  20  23  26  29  32  35  38  41  44  47  50  53
##  [19]  56  59  62  65  68  71  74  77  80  83  86  89  92  95  98 101 104 107
##  [37] 110 113 116 119 122 125 128 131 134 137 140 143 146 149 152 155 158 161
##  [55] 164 167 170 173 176 179 182 185 188 191 194 197 200 203 206 209 212 215
##  [73] 218 221 224 227 230 233 236 239 242 245 248 251 254 257 260 263 266 269
##  [91] 272 275 278 281 284 287 290 293 296 299
```

```
number.series[c(5,10,15,20)]
```

```
## [1] 14 29 44 59
```

```
number.series[seq(from=5,to=20,by=5)]
```

```
## [1] 14 29 44 59
```

```
number.series[10:30]
```

```
##  [1] 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89
```

Can you plot data?

Please explain your classmates how to create the following plots with R:

- Pie Charts (https://www.tutorialspoint.com/r/r_pie_charts.htm)
- Bar Charts (https://www.tutorialspoint.com/r/r_bar_charts.htm)
- Boxplots (https://www.tutorialspoint.com/r/r_boxplots.htm)
- Histograms (https://www.tutorialspoint.com/r/r_histograms.htm)
- Line Graphs (https://www.tutorialspoint.com/r/r_line_graphs.htm)
- Scatterplots (https://www.tutorialspoint.com/r/r_scatterplots.htm)

Can you create data frames and merge them?

Please try to understand the following lines of code:

```
data1 <- data.frame(id = 1:6,                                 # Create first example data frame
                    x1 = c(5, 1, 4, 9, 1, 2),
                    x2 = c("A", "Y", "G", "F", "G", "Y"))
data2 <- data.frame(id = 4:9,                                 # Create second example data frame
                    y1 = c(3, 3, 4, 1, 2, 9),
                    y2 = c("a", "x", "a", "x", "a", "x"))

merge(data1, data2, by = "id")                                # Merge data frames by columns names
```

```
##   id x1 x2 y1 y2
## 1  4  9  F  3  a
## 2  5  1  G  3  x
## 3  6  2  Y  4  a
```

```
merge(data1, data2, by = "id", all.x = TRUE)                  # Keep all rows of x-data
```

```
##   id x1 x2 y1   y2
## 1  1  5  A NA <NA>
## 2  2  1  Y NA <NA>
## 3  3  4  G NA <NA>
## 4  4  9  F  3    a
## 5  5  1  G  3    x
## 6  6  2  Y  4    a
```

```
merge(data1, data2, by = "id", all.y = TRUE)                  # Keep all rows of y-data
```

```
##   id x1   x2 y1 y2
## 1  4  9    F  3  a
## 2  5  1    G  3  x
## 3  6  2    Y  4  a
## 4  7 NA <NA>  1  x
## 5  8 NA <NA>  2  a
## 6  9 NA <NA>  9  x
```

```
merge(data1, data2, by = "id", all.x = TRUE, all.y = TRUE)    # Keep all rows of both data frames
```

```
##   id x1   x2 y1   y2
## 1  1  5    A NA <NA>
## 2  2  1    Y NA <NA>
## 3  3  4    G NA <NA>
## 4  4  9    F  3    a
## 5  5  1    G  3    x
## 6  6  2    Y  4    a
## 7  7 NA <NA>  1    x
## 8  8 NA <NA>  2    a
## 9  9 NA <NA>  9    x
```

```
data3 <- data.frame(id = 5:6,                                 # Create third example data frame
                    z1 = c(3, 2),
                    z2 = c("K", "b"))

data12 <- merge(data1, data2, by = "id")                      # Merge data 1 & 2 and store
merge(data12, data3, by = "id")                               # Merge multiple data frames
```

```
##   id x1 x2 y1 y2 z1 z2
## 1  5  1  G  3  x  3  K
## 2  6  2  Y  4  a  2  b
```

Can you calculate growth rates and build up data frames?

Do the following:

- Load the `sunspot.year` data which is part of R's datasets package `datasets`,
- generate a vector that contains the years 1700 to 1988,
- combine the two vectors into a data frame using `data.frame()`,
- calculate the growth rate of yearly sunspot using `growth.rate()` which is part of the `tis` package,
- add the growth variable to the data frame.

```
library("datasets")
data("sunspot.year")
year <- 1700:1988
sunspot.frame <- data.frame(year, sunspot.year)

install.packages("tis")
```

```
## Installing package into '/home/sthu/R/x86_64-pc-linux-gnu-library/4.1'
## (as 'lib' is unspecified)
```

```
library("tis")
```

```
##
## Attaching package: 'tis'
```

```
## The following object is masked from 'package:dplyr':
##
##     between
```

```
growth <- growth.rate(sunspot.frame$sunspot.year, lag=1, simple = T)
year <- 1701:1988
sunspot.frame2 <- data.frame(year, growth)
merge(sunspot.frame, sunspot.frame2, by = "year", all.x = TRUE, all.y = TRUE)
```

Can you load R Built-in R Data set?

Study this webpage R Built-in R Data set (https://rstudio-pubs-static.s3.amazonaws.com/481654_883a4b47c9b244d4859dd1db235f0165.html) and show how can you load R's *Motor Trend Car Road Tests* dataset?

Can you inspect data sets?

Here is a non-exhaustive list of functions to get a sense of the content/structure of data.

- All data structures - content display:
  - **`str()`:** compact display of data contents (env.)
  - **`class()`:** data type (e.g. character, numeric, etc.) of vectors and data structure of dataframes, matrices, and lists.
  - **`summary()`:** detailed display, including descriptive statistics, frequencies
  - **`head()`:** will print the beginning entries for the variable
  - **`tail()`:** will print the end entries for the variable
- Vector and factor variables:
  - **`length()`:** returns the number of elements in the vector or factor
- Dataframe and matrix variables:
  - **`dim()`:** returns dimensions of the dataset
  - **`nrow()`:** returns the number of rows in the dataset
  - **`ncol()`:** returns the number of columns in the dataset
  - **`rownames()`:** returns the row names in the dataset
  - **`colnames()`:** returns the column names in the dataset

Load the mtcars data set and play around with the functions above.

Do you know the cars data?

The `cars` data comes with the default installation of R. To see the first few columns of the data, just type `head(cars)`.

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

We'll do a bad thing here and use the `attach()` command, which will allow us to access the `speed` and `dist` columns of `cars` as though they were vectors in our workspace. The `attach()` function has the side effect of altering the search path and this can easily lead to the wrong object of a particular name being found. People do often forget to detach databases. Thus, it is better to use `$`.

```
attach(cars) # Using this command is poor style.  We will avoid it in the future.
speed
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15 15
## [26] 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24 24 25
```

```
dist
```

```
##  [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34  34  46
## [20]  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76  84  36  46  68
## [39]  32  48  52  56  64  66  54  70  92  93 120  85
```

(a) Calculate the average and standard deviation of speed and distance.

```
mean(speed)
```

```
## [1] 15.4
```

```
sd(speed)
```

```
## [1] 5.287644
```

```
mean(dist)
```

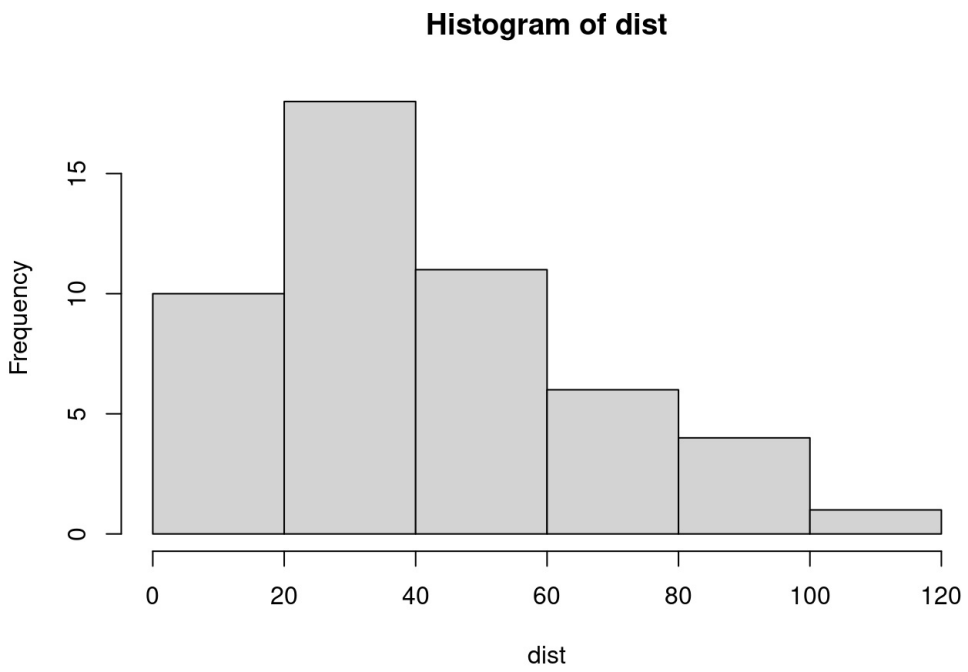
```
## [1] 42.98
```

```
sd(dist)
```

```
## [1] 25.76938
```

(b) Make a a histogram of stopping distance using the `hist` function.

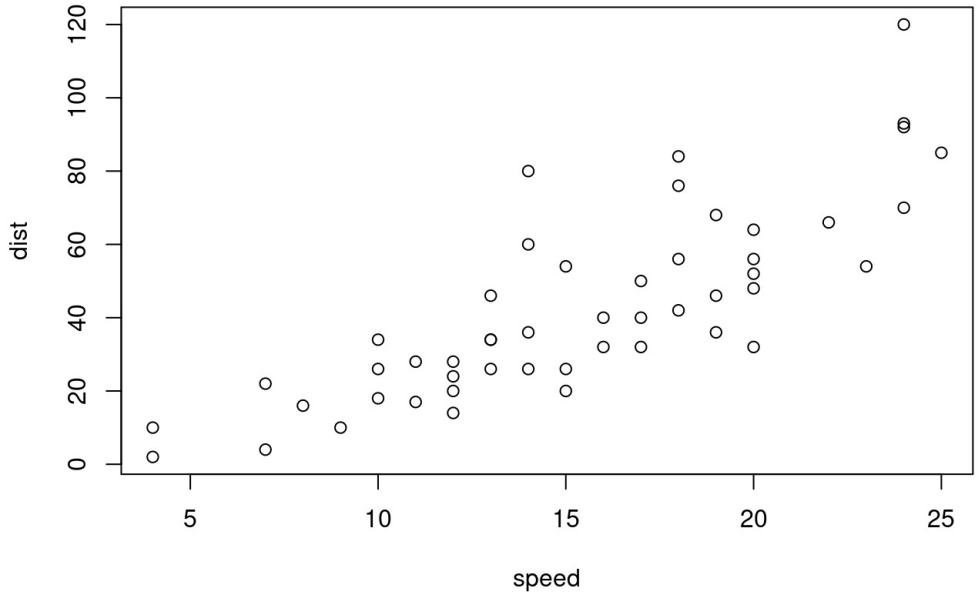
```
hist(dist) # Histogram of stopping distance
```



The `plot(x,y,...)` function plots a vector `y` against a vector `x`. You can type `?plot` into the Console to learn more about the basic plot function.

(c) Use the `plot(x,y)` function to create a scatterplot of dist against speed.

```
plot(speed, dist)
```



Describe the meaning of the # sign and the <- operator.

- The `#` sign is used in R-scripts to add comments. That is usefull that you and others can understand what the R code is about. Comments are not run as R code, so they will not influence your result.
- `<-` is an assignment operator that assigns a value to a name/variable. A variable allows you to store a value or an object in R and you can create that with the assignment operator. You can then later use it to access the value or the object that is stored within this variable.

Explain briefly ways to get help via the Console.

- ?: Search R documentation for a specific term.
- ?? Search R help files for a word or phrase.
- RSiteSearch(): Search search.r-project.org
- findFn(): Search search.r-project.org for functions (Hint: requires the “sos” library loaded!)
- help.start(): Access to html manuals and documentations implemented in R
- apropos(): Returns a character vector giving the names of objects in the search list matching (as a regular expression) what.
- find(): Returns where objects of a given name can be found.
- vignette(): View a specified package vignette, i.e., supporting material such as introductions.

Explain what the webpage RSeek.org can do for you.

It makes it easy to search for information about R while filtering out sites hat match “R” but don’t contain the information you’re looking for. It’s just like a Google search, but restricts the search just to those sites known to contain information about R.

Exercise

The `ggplot2` package is part of the `tidyverse` suite of integrated packages (https://www.tidyverse.org/packages/) which was designed to work together to make common data science operations more user-friendly. Maybe we use the `tidyverse` suite in later lessons, so let’s install it.

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

Exercise

Search in the rdocumention.org (https://www.rdocumentation.org/) for ggplot2.

User-defined Functions

One of the great strengths of R is the user’s ability to add functions. Sometimes there is a small task (or series of tasks) you need done and you find yourself having to repeat it multiple times. In these types of situations it can be helpful to create your own custom function. The **structure of a function is given below**:

```
name_of_function <- function(argument1, argument2) {
    statements or code that does something
    return(something)
}
```

- First you give your function a name.
- Then you assign value to it, where the value is the function.

When **defining the function** you will want to provide the **list of arguments required** (inputs and/or options to modify behaviour of the function), and wrapped between curly brackets place the **tasks that are being executed on/using those arguments**. The argument(s) can be any type of object (like a scalar, a matrix, a dataframe, a vector, a logical, etc), and it's not necessary to define what it is in any way.

Finally, you can **"return" the value of the object from the function**, meaning pass the value of it into the global environment. The important idea behind functions is that objects that are created within the function are local to the environment of the function – they don't exist outside of the function.

> **NOTE:** You can also have a function that doesn't require any arguments, nor will it return anything.

Let's try creating a simple example function. This function will take in a numeric value as input, and return the squared value.

```
square_it <- function(x) {
    square <- x * x
    return(square)
}
```

Now, we can use the function as we would any other function. We type out the name of the function, and inside the parentheses we provide a numeric value `x`:

```
square_it(5)
```

```
## [1] 25
```

Pretty simple, right?

R Markdown

R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both

- save and execute code and
- generate high quality reports that can be shared with an audience.

Please have a look at the following tutorials:

- R Markdown from RStudio (https://rmarkdown.rstudio.com/lesson-1.html)
- R Markdown Cheat Sheet (https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf)

12.4 Data transformation

Please download and open the following R-script and try to answer the questions therein:
https://raw.githubusercontent.com/hubchev/courses/main/scr/data_transformation.R

```
# ============================================================================
# 0.
# Set your working directory and clear the environment.

# ============================================================================
# 1.
# Download and read the most recent version of the lecture notes
# (rcourse_book.pdf) from ILIAS.

# ============================================================================
# 2.
# Read: Wickham and Grolemund (2018, ch. "Workflow: basics").
# See: https://r4ds.had.co.nz/workflow-basics.html

# ============================================================================
# 3.
# Answer the following questions:

# ===
# a)
# Run the two following lines of code
# (a.k.a. send it from the script to the console)
# Why does the second line of code not work? Look carefully!
# This may seem like an exercise in pointlessness, but training your brain
# to notice even the tiniest difference will pay off when programming.)

my_variable <- 10
my_varıable

# ===
# b)
# Tweak each of the following R commands so that they run correctly:

library(tidyverse)

ggplot(dota = mpg) +
geom_point(mapping = aes(x = displ, y = hwy))

fliter(mpg, cyl = 8)
filter(diamond, carat > 3)

# Solutions:
ggplot(data = mpg) +
geom_point(mapping = aes(x = displ, y = hwy))

ggplot(data = mpg) +
geom_point() +
aes(x = displ, y = hwy)

filter(diamonds, carat > 3)
filter(mpg, cyl == 8)
# ===
# c)
# Press Alt + Shift + K. What happens? How can you get to the same place using
# the menus?
```

```
# ============================================================================
# 4.
# Read Wickham and Grolemund (2018, ch. "Data transformation")
# See: https://r4ds.had.co.nz/transform.html
#
# Answer the following questions (Also see lecture notes excercise "Subsetting")
#
# --- Solutions to these questions can be found here:
# https://raw.githubusercontent.com/hubchev/courses/main/scr/exe_subset.R
# !!! For your own benefit: try to come up with the solution yourself.
# ===
# Load the following packages: tidyverse, dplyr, and tibble.
# ===
# Check to see if you have the mtcars dataset by entering the command mtcars.
# ===
# Save the mtcars dataset in an object named cars.
# ===
# What class is cars?
# ===
# How many observations (rows) and variables (columns) are in the
# mtcars dataset?
# ===
# Rename mpg in cars to MPG. Use rename().
# ===
# Convert the column names of cars to all upper case. Use rename\_all,
# and the toupper command.
# ===
# Convert the rownames of cars to a column called car using
# rownames\_to\_column.
# ===
# Subset the columns from cars that end in "p" and call it pvars
# using ends\_with().
# ===
# Create a subset cars that only contains the columns: wt, qsec, and hp and
# assign this object to carsSub.  (Use select().)
# ===
# What are the dimensions of carsSub? (Use dim().)
# ===# Convert the column names of carsSub to all upper case. Use rename\_all(),
# and toupper() (or colnames()).
# ===
# Subset the rows of cars that get more than 20 miles per gallon (mpg)
# of fuel efficiency. How many are there? (Use filter().)
# ===
# Subset the rows that get less than 16 miles per gallon (mpg) of fuel
# efficiency and have more than 100 horsepower (hp).
# How many are there? (Use filter() and the pipe operator.)
# ===
# Create a subset of the cars data that only contains the columns: wt, qsec, and
# hp for cars with 8 cylinders (cyl) and reassign this object to carsSub.
# What are the dimensions of this dataset? Don't use the pipe operator.
# ===
# Create a subset of the cars data that only contains the columns:
# wt, qsec, and hp for cars with 8 cylinders (cyl) and reassign
# this object to carsSub2. Use the pipe operator.
# ===
# Re-order the rows of carsSub by weight (wt) in increasing order.
# (Use arrange().)
# ===
# Create a new variable in carsSub called wt2, which is equal to $wt^2$,
# using mutate() and piping \%>\%.
```

```
# ============================================================================
# 5.
# Read Wickham and Grolemund (2018, ch. "Data transformation")
# See: https://r4ds.had.co.nz/transform.html
# Answer the question (1) of exercise 5.2.4 of Wickham and Grolemund (2018)
# See:
# https://r4ds.had.co.nz/transform.html#exercises-8
#
# Solutions can be found here:
# https://jrnold.github.io/r4ds-exercise-solutions/transform.html
# !!! For your own benefit: try to come up with the solution yourself.
# ===
# Find all flights that
# ===
# Had an arrival delay of two or more hours
# ===
# Flew to Houston (IAH or HOU)
# ===
# Were operated by United, American, or Delta
# ===
# Departed in summer (July, August, and September)
# ===
# Arrived more than two hours late, but didn't leave late
# ===
# Were delayed by at least an hour, but made up over 30 minutes in flight
# ===
# Departed between midnight and 6 am (inclusive)
```

12.5 Convergence

The dataset convergence.dta, see https://github.com/hubchev/courses/blob/main/dta/convergence.dta, contains the per capita GDP of 1960 (gdppc60) and the average growth rate of GDP per capita between 1960 and 1995 (growth) for different countries (country), as well as 3 dummy variables indicating the belonging of a country to the region Asia (asia), Western Europe (weurope) or Africa (africa).

1) Some countries are not assigned to a certain country group. Name the countries which are assign to be part of Western Europe, Africa or Asia. If you find countries that are members of the EU, assign them a '1' in the variable weurope.

2) Create a table that shows the average GDP per capita for all available points in time. Group by Western European, Asian, African, and the remaining countries.

3) Create the growth rate of GDP per capita from 1960 to 1995 and call it gdpgrowth. (Note: The log value X minus the log value X of the previous period is approximately equal to the growth rate).

4) Calculate the unconditional convergence of all countries by constructing a graph in which a scatterplot shows the GDP per capita growth rate between 1960 and 1995 (gdpgrowth) on the y-axis and the 1960 GDP per capita (gdppc60) on the x-axis. Add to the same graph the estimated linear relationship. You do not need to label the graph further, just two things: title the graph 'world' and label the individual observations with the country names.

5) Create three graphs describing the same relationship for the sample of Western European, African and Asian countries. Title the graph accordingly with 'weurope', 'africa' and 'asia'.

6) Combine the 4 graphs into one image. Discuss how an upward or downward sloping regression line can be interpreted.

7) Estimate the relationships illustrated in the 4 graphs using the least squares method. Present the 4 estimation results in a table, indicating the significance level with stars. In addition, the Akaike information criterion, and the number of observations should be displayed in the table. Interpret the four estimation results regarding their significance.

8) Put the data set into the so-called *long format* and calculate the GDP per capita growth rates for the available time points in the countries.

Please find solutions here: https://raw.githubusercontent.com/hubchev/courses/main/scr/convergence.R

12.6 Calories and weight

Please find solutions here: https://raw.githubusercontent.com/hubchev/courses/main/scr/exe_calories.R

This exercise contains 17 questions for the total of 80 points.

Please answer all (!) questions in a R script. Text should be written as a comment using the '#' to *comment out* the text.[1] Make sure that the script runs without errors.

1. (0 points) Write down your name, your matriculation number, and the date.
2. (5 points) Set your working directory.
3. (5 points) Clear your global environment.
4. (5 points) Load the following package(s): `tidyverse`

   The following table stems from a survey carried out at the Campus of the German Sport University of Cologne at Opening Day (first day of the new semester) between 8:00am and 8:20am. The questions were asked in an open face-to-face communication. The survey consists of 6 individuals with the following information:

id	1	2	3	4	5	6
sex	f	f	f	m	m	m
age	21	19	23	18	20	61
weight	48	55	50	71	77	85
calories	1700	1800	2300	2000	2800	2500
sport	60	120	180	60	240	30

   Data description:

   id: Variable with an anonymized identifier for each participant.

   sex: Gender, i.e., the participants replied to be either male (m) or female (f).

   age The age in years of the participants at the time of the survey.

   weight Number of kg the participants pretended to weight.

   calories Estimate of the participants on their average daily consumption of calories.

   sport Estimate of the participants on their average daily time that they spend on doing sports (measured in minutes).

5. (5 points) Which type of data do we have here? (Panel data, repeated cross-sectional data, cross-sectional data, time Series data)
6. (10 points) Store each of the five variables in a vector and put all five variables into a dataframe with the title `df`. If you fail here, read in the data using this line of code:

```
df <- read_csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/df-calories.csv")
```

7. (5 points) Show for all numerical variables the summary statistics including the mean, median, minimum, and the maximum.

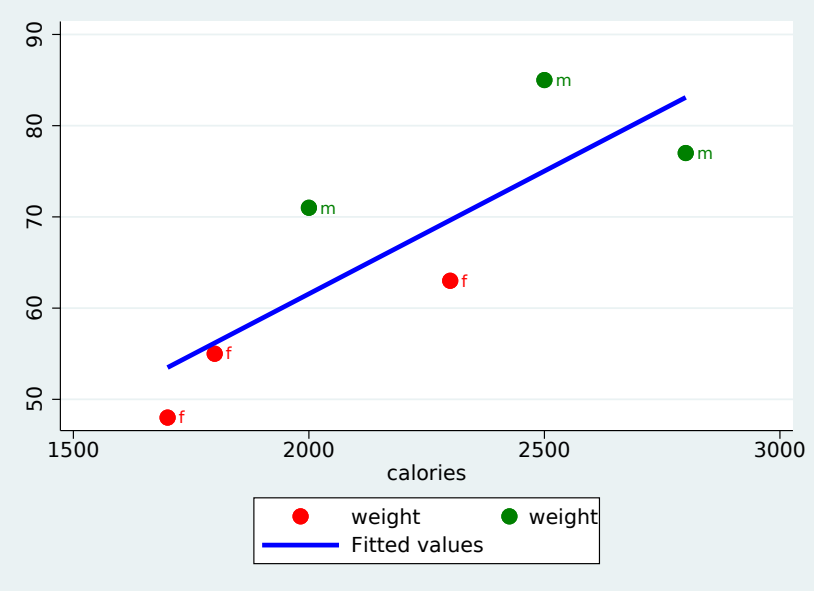
[1] The easiest way to create a multi-line comment in RStudio is to highlight the text and press Ctrl+Shift+C. For macOS, use Command+Shift+C.

Please go on to the next page...

8. (5 points) Show for all numerical variables the summary statistics including the mean and the standard deviation, **separated by male and female**. Use therefore the pipe operator.

9. (5 points) Suppose you want to analyze the general impact of average calories consumption per day on the weight. Discuss if the sample design is appropriate to draw conclusions on the population. What may cause some bias in the data? Discuss possibilities to improve the sampling and the survey, respectively.

10. (5 points) The following plot visualizes the two variables weight and calories. Discuss what can be improved in the graphical visualization.



11. (5 points) Make a scatterplot matrix containing all numerical variables.

12. (5 points) Calculate the Pearson Correlation Coefficient of the two variables

    a) `calories` and `sport`, and

    b) `weight` and `calories`.

13. (5 points) Make a scatterplot with `weight` in the y-axis and `calories` on the x-axis. Additionally, the plot should contain a linear fit and the points should be labeled with the `sex` just like in the figure shown above.

14. Estimate the following regression specification using the OLS method:

$$weight_i = \beta_0 + \beta_1 calories_i + \epsilon_i.$$

    Show a summary of the estimates that look like this:

```
Call:
lm(formula = weight ~ calories, data = df)

Residuals:
1       2       3       4       5       6
-5.490 -1.182 -6.640  9.435 -6.099  9.976
```

```
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept)  7.730275  20.197867   0.383   0.7214
calories     0.026917   0.009107   2.956   0.0417 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.68 on 4 degrees of freedom
Multiple R-squared:  0.6859,Adjusted R-squared:  0.6074
F-statistic: 8.735 on 1 and 4 DF,  p-value: 0.04174
```

15. (5 points) Interpret the results. In particular, interpret how many kg the estimated weight increases—on average and ceteris paribus—if calories increase by 100 calories. Additionally, discuss the statistical properties of the estimated coefficient $\hat{\beta}_1$ and the meaning of the *Adjusted R-squared*.

16. (5 points) OLS estimates can suffer from omitted variable bias. State the two conditions that need to be fulfilled for omitted bias to occur.

17. (5 points) Discuss potential confounding variables that may cause omitted variable bias. Given the dataset above how can some of the confounding variables be *controlled for*?

End of exam

12.7 Forest and GDP

Exercise: Forest and GDP

Stephan.Huber@hs-fresenius.de

Data Science and Data Analytics (short mock exam)

Please answer all (!) questions in an R script. Normal text should be written as comments, using the '#' to comment out text. Make sure the script runs without errors before submitting it. Each task (starting with 1) is worth XXX points. You have a total of XXX minutes of editing time. Please do not forget to number your answers.

When you are done with your work, save the R script, export the script to pdf format and upload the pdf file.

Suppose you aim to empirically examine the impact of economic activity on the environment, i.e., forest area (% of land area). The data set that we use in the following is 'forest.Rdata'. I downloaded from the Worldbank. In 'data_forest.R' I document the data preperation. See:

`https://raw.githubusercontent.com/hubchev/courses/main/scr/data_forest.R`

Data description

gdp GDP (constant 2015 US$)

GDP at purchaser's prices is the sum of gross value added by all resident producers in the

economy plus any product taxes and minus any subsidies not included in the value of the products. It is calculated without making deductions for depreciation of fabricated assets or for depletion and degradation of natural resources. Data are in constant 2015 prices, expressed in U.S. dollars. Dollar figures for GDP are converted from domestic currencies using 2015 official exchange rates. For a few countries where the official exchange rate does not reflect the rate effectively applied to actual foreign exchange transactions, an alternative conversion factor is used. (NY.GDP.MKTP.KD)

gdp_growth GDP growth (annual %)

Annual percentage growth rate of GDP at market prices based on constant local currency. Aggregates are based on constant 2015 prices, expressed in U.S. dollars. GDP is the sum of gross value added by all resident producers in the economy plus any product taxes and minus any subsidies not included in the value of the products. It is calculated without making deductions for depreciation of fabricated assets or for depletion and degradation of natural resources. (NY.GDP.MKTP.KD.ZG)

unemployment Unemployment, total (% of total labor force) (modeled ILO estimate)

Unemployment refers to the share of the labor force that is without work but available for and seeking employment. (SL.UEM.TOTL.ZS) See: https://data.worldbank.org/indicator/SL.UEM.TOTL.ZS

income World Bank Country and Lending Groups

For the current 2022 fiscal year, low-income economies are defined as those with a GNI per capita, calculated using the World Bank Atlas method, of $1,045 or less in 2020; lower middle-income economies are those with a GNI per capita between $1,046 and $4,095; upper middle-income economies are those with a GNI per capita between $4,096 and $12,695; high-income economies are those with a GNI per capita of $12,696 or more.

forest Forest area (% of land area)

Forest area is land under natural or planted stands of trees of at least 5 meters in situ, whether productive or not, and excludes tree stands in agricultural production systems (for example, in fruit plantations and agroforestry systems) and trees in urban parks and gardens. (AG.LND.FRST.ZS)

pop Population, total - Spain

Total population is based on the de facto definition of population, which counts all residents regardless of legal status or citizenship. The values shown are midyear estimates. (SP.POP.TOTL)

unemployment_dif

Yearly change in unemployment: unemployment(t)-unemployment(t-1)

gdppc GDP per capita (gdppc = gdp/pop)

---

(1) Set your working directory.

(2) Clear your global environment.

(3) Install and load the following packages: 'tidyverse', 'sjPlot', and 'ggpubr'

(4) Download and load the data, respectively, with the following code:

```
load(url("https://github.com/hubchev/courses/raw/main/dta/forest.Rdata"))
```

If that is not working, you can also download the data from ILIAS, save it in your working directory and load it from there with:

```
load("forest.Rdata")
```

(5) Show for all numerical variables the summary statistics including the mean, median, minimum, and the maximum.

(6) Rename the variable 'country.x' to 'country' in the dataset 'df'.

(7) Create a variable that indicates the gdp in million US $ ('gdp' divided by 1,000,000). Name the variable 'gdp_mio'.

(8) Create a table showing the mean values of the variables 'gdp_mio', and 'forest' over time separately by region. Use the pipe operator. (Tip: See below for how your result should look like.)

```
## # A tibble: 7 x 3
##   region                      m_gdp_mio m_forest
##   <chr>                           <dbl>    <dbl>
## 1 East Asia & Pacific           623388.    48.2
## 2 Europe & Central Asia         382810.    29.4
## 3 Latin America & Caribbean     135439.    46.6
## 4 Middle East & North Africa    117698.     3.04
## 5 North America                9117487.    35.8
## 6 South Asia                    235150.    25.4
## 7 Sub-Saharan Africa             25952.    32.3
```

(9) Investigate the relationship of economic activity measured by the GDP and the GDP per capita with a country's forest area. Therefore, graphically visualize the relationsship and consider things like correlation analysis and regression analysis.

Bibliography

Békés, G. and Kézdi, G. (2021). *Data Analysis for Business, Economics, and Policy*. Cambridge University Press. [Cited on page 5.]

Blumberg, R. and Atre, S. (2003). The problem with unstructured data. *DM Review*, 13(42-49):62. [Cited on pages 15 and 16.]

Chacon, S. and Straub, B. (2022). *Pro Git*. Apress, 2 edition. freely available online: https://git-scm.com/book/en/v2. [Cited on page 92.]

Chang, W. (2018). *R Graphics Cookbook: Practical Recipes for Visualizing Data*. O'Reilly Media. [Cited on page 27.]

Crawley, M. J. (2013). *The R Book*. John Wiley & Sons, 2nd edition. [Cited on page 26.]

Cunningham, S. (2021). *Causal Inference: The Mixtape*. Yale University Press. [Cited on page 5.]

Davenport, T. H. and Patil, D. (2012). Data scientist. *Harvard Business Review*, 90(5):70–76. [Cited on page 20.]

Demetis, D. and Lee, A. S. (2018). When humans using the IT artifact becomes IT using the human artifact. *Journal of the Association for Information Systems*, 19(10):5. [Cited on page 21.]

Fox, J. and Weisberg, S. (2018). *An R Companion to Applied Regression*. SAGE, 3 edition. [Cited on page 57.]

Gardner, M. (1962). How to build a game-learning machine and then teach it to play and to win. *Scientific American*, 206(3):138–144. [Cited on page 8.]

Harford, T. (2020). *How to Make the World Add Up: Ten Rules for Thinking Differently about Numbers*. The Bridge Street Press. [Cited on page 5.]

Huntington-Klein, N. (2021). *The Effect: An Introduction to Research Design and Causality*. Chapman and Hall/CRC. [Cited on page 5.]

Irizarry, R. A. (2020). *Introduction to Data Science. Data Analysis and Prediction Algorithms with R*. https://rafalab.github.io/dsbook/. [Cited on page 26.]

Kelleher, J. D. and Tierney, B. (2018). *Data Science*. MIT Press. [Cited on pages 5 and 6.]

Kirchkamp, O. (2018). Using graphs and visualising data. Technical report. https://www.kirchkamp.de/oekonometrie/pdf/gra-p.pdf (retrieved on 2022/05/20). [Cited on page 80.]

Muenchen, B. (2019). Data science jobs report 2019: Python way up, tensorflow growing rapidly, r use double sas. News 19:n23, KDNuggets. [Cited on page 8.]

Muschelli, J. and Jaffe, A. (2022). Introduction to r for public health researchers. This work is licensed under a creative commons attribution-noncommercial-sharealike 4.0 international license., GitHub. https://github.com/muschellij2/intro_to_r. [Cited on page 1.]

Navarro, D. (2020). *Learning Statistics with R*. Version 0.6 edition. https://learningstatisticswithr.com/. [Cited on page 1.]

Provost, F. and Fawcett, T. (2013). *Data Science for Business: What you need to know about data mining and data-analytic thinking*. O'Reilly Media. [Cited on pages 10 and 11.]

Teetor, P. (2011). *R Cookbook: Proven Recipes for Data Analysis, Statistics, and Graphics*. O'Reilly Media. [Cited on page 27.]

Timbers, T., Campbell, T., and Lee, M. (2022). *Data Science: A First Introduction*. CRC Press. [Cited on page 26.]

Venables, W. N., Smith, D. M., and R Core Team (2022). *An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics*. This manual is for r, version 4.1.3 (2022-03-10) edition. http://cran.r-project.org/doc/manuals/R-intro.pdf (retrieved on 2022/04/06). [Cited on page 26.]

Wickham, H. and Grolemund, G. (2018). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly. [Cited on pages 26, 77, 78, and 80.]

Xie, Y., Allaire, J. J., and Grolemund, G. (2018). *R Markdown: The Definitive Guide*. Chapman and Hall/CRC. [Cited on page 91.]

Zheng, Y. (2011). T-drive trajectory data sample. Technical report, microsoft.com. [Cited on page 13.]

Appendix A

Appendix

A.1 Solution to exercises

Seconds A Year Solution to Exercise 6.1

```
seconds <- 60
minutes <- 60
hours <- 24
days <- 365

answer <- seconds * minutes * hours * days
print(answer)
```

A.2 Regression analysis presentation

Regression Analysis

Stephan.Huber@hs-fresenius.de (mailto:Stephan.Huber@hs-fresenius.de)
Hochschule Fresenius: Data Science

Compiled at 17 May, 2022

Word count: 1614

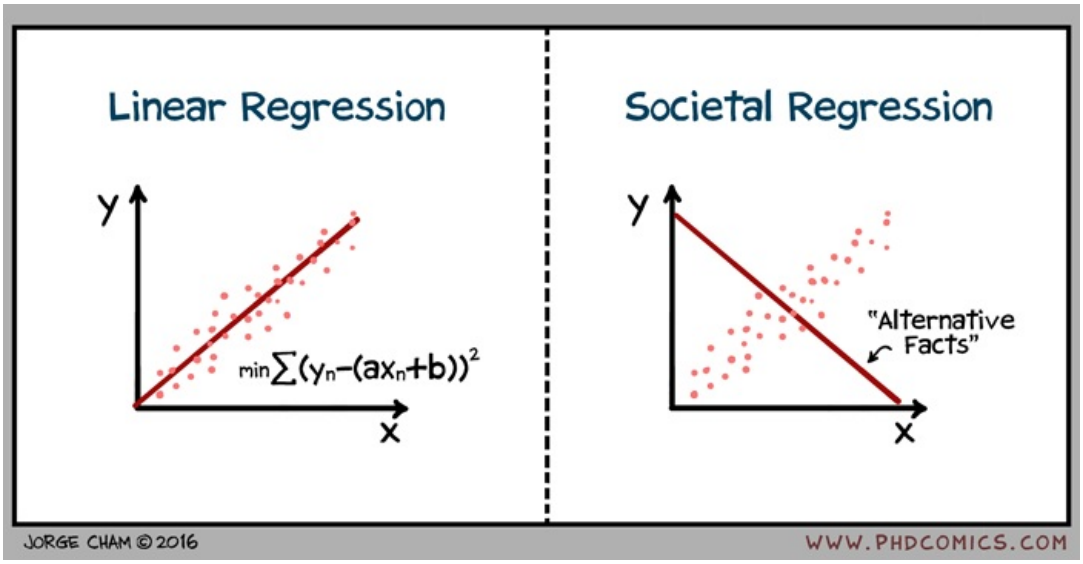
1 Literature

Regression analysis is covered by almost all econometric and statistical textbooks. Some are more formal some are more illustrative and intuitive. Here is my selection with a focus on R:

- Book: Applied Statistics with R (https://daviddalpiaz.github.io/appliedstats/)
- Book: Introduction to Econometrics with R (https://www.econometrics-with-r.org/)
- A more formal approach from my Alma Mater: slides (https://www.uni-regensburg.de/wirtschaftswissenschaften/vwl-tschernig/medien/zeitreihenoekonometrie/eoe_eng_2020_08_chapter_all.pdf) and handout (https://www.uni-regensburg.de/wirtschaftswissenschaften/vwl-tschernig/medien/methoden-der-oekonometrie/methoden_oekonometrie_handout_2018_11_29.pdf)
- This presentation is available on my github account (https://github.com/hubchev/courses)

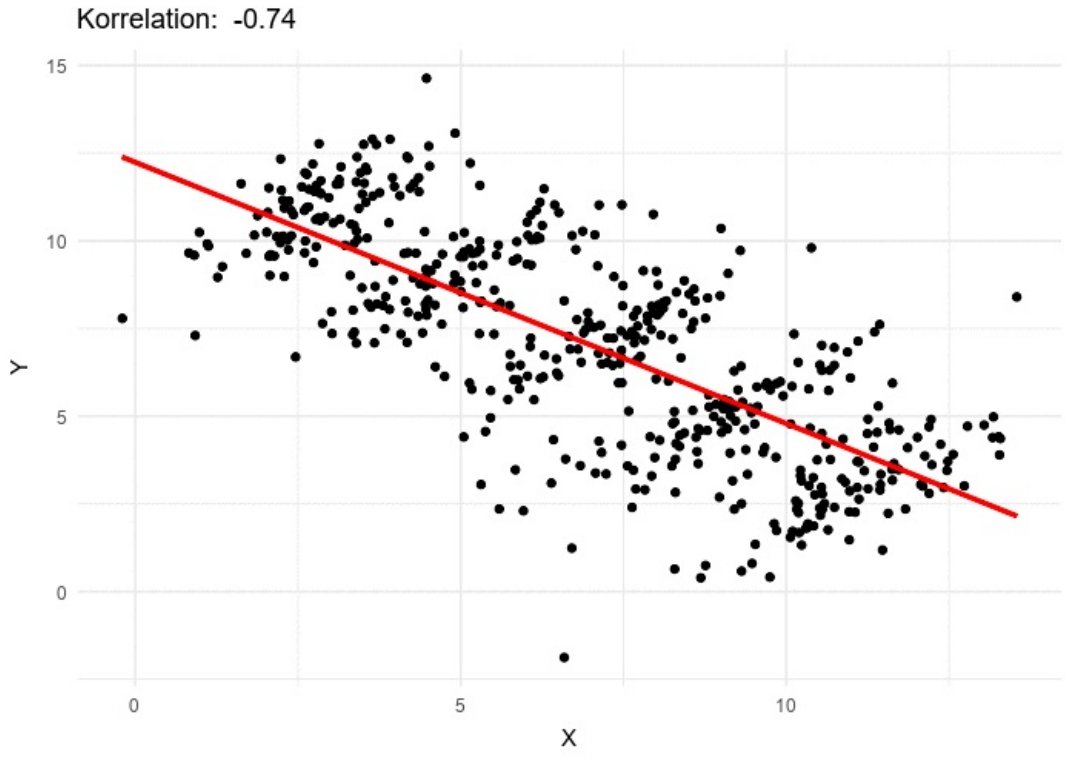
2 Why regression analysis?

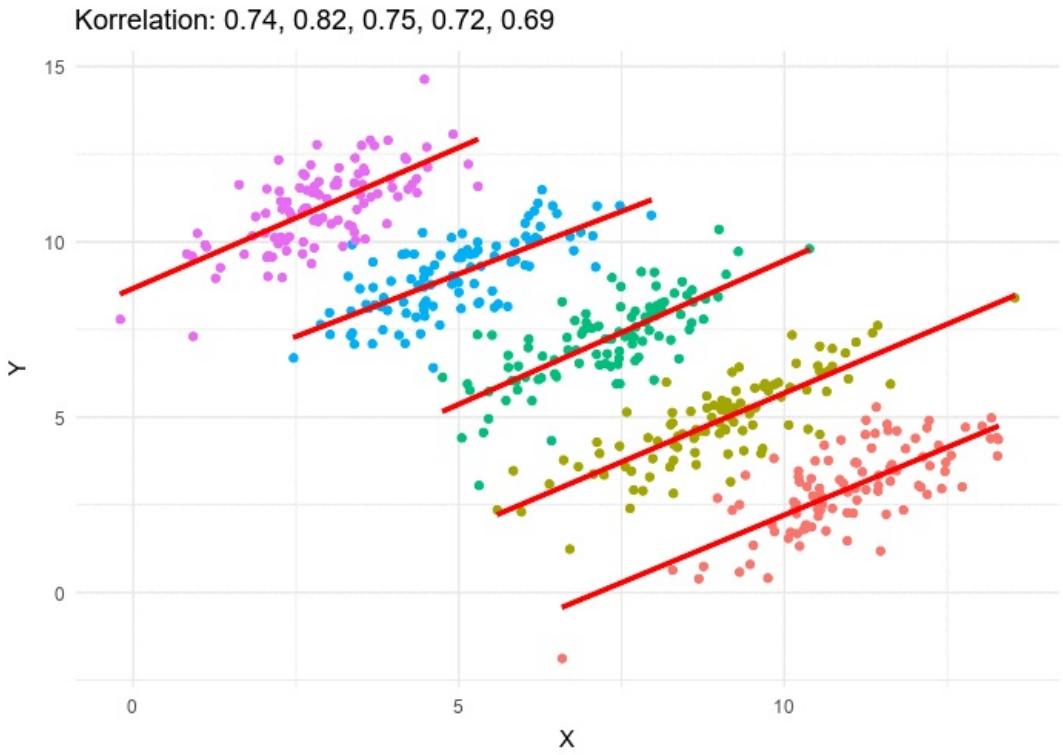
2.1 It is a medicine agains alternative facts



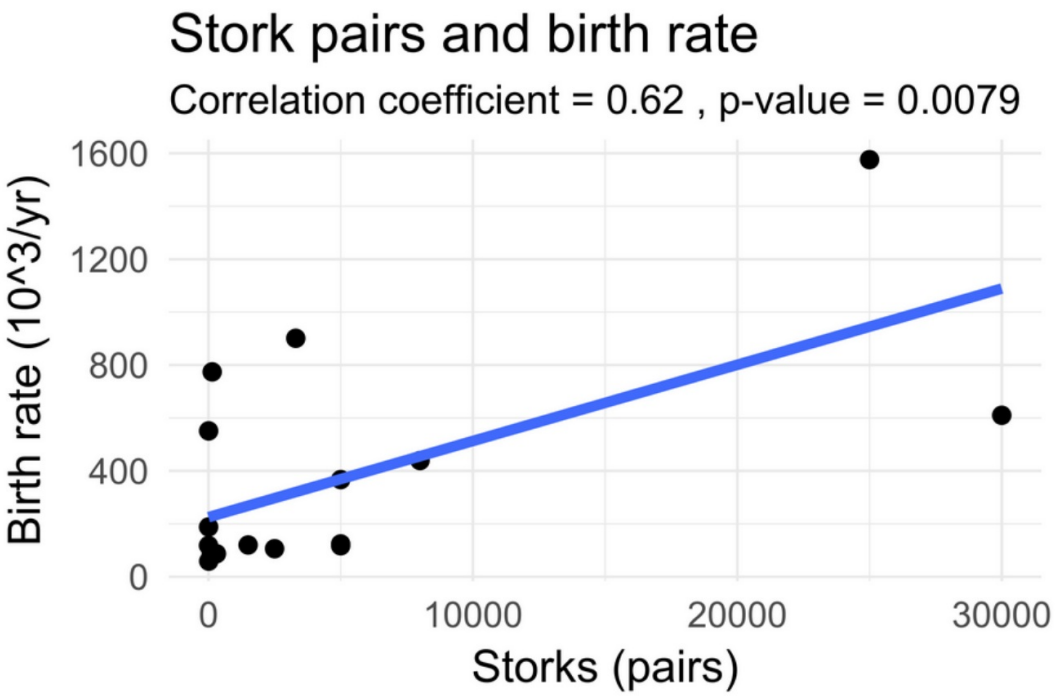
- Regressions allow us to **draw insights** from data,
- to analyze and **interpret** the strength of relationships and
- to reduce the likeliness of **causal fallacy**.

2.2 Simpsons paradox

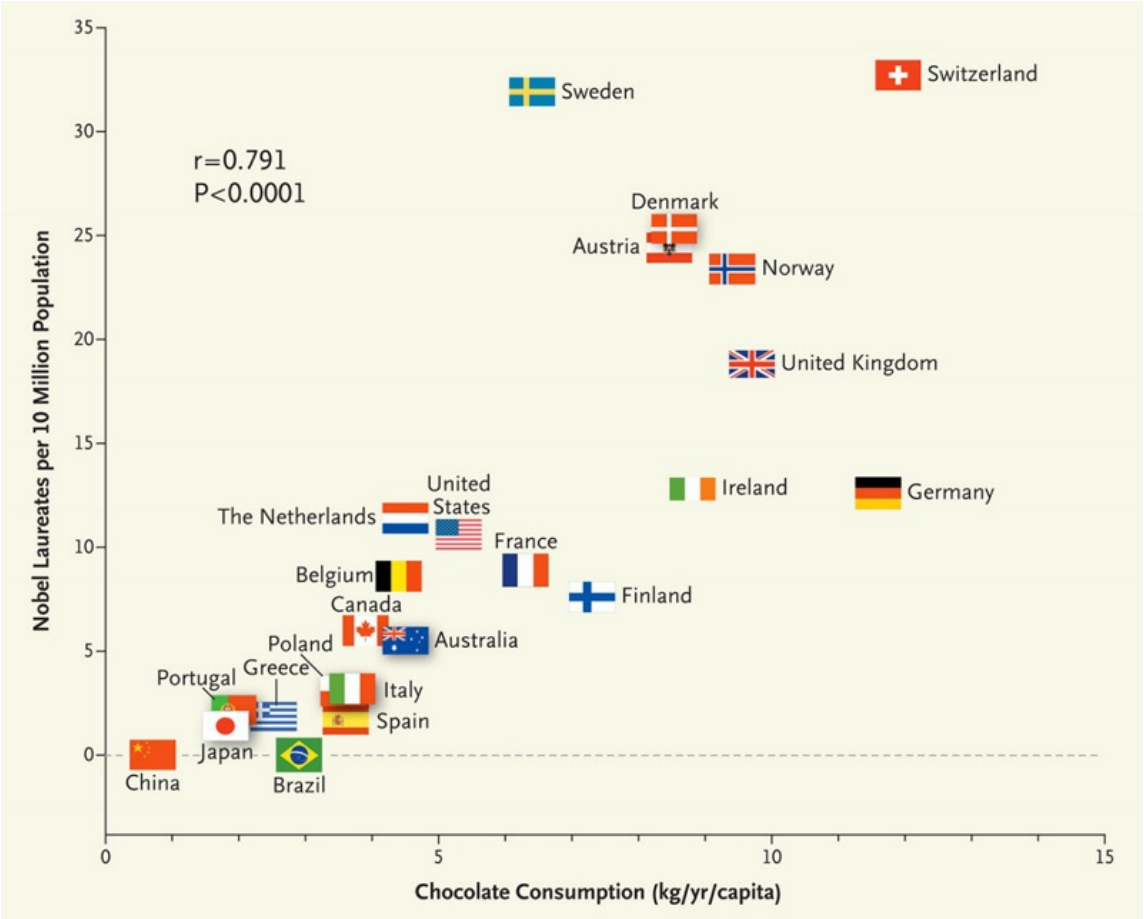




2.3 Correlation does not imply causation



Matthews, R. (2000), Storks Deliver Babies (p= 0.008). Teaching Statistics, 22: 36–38.

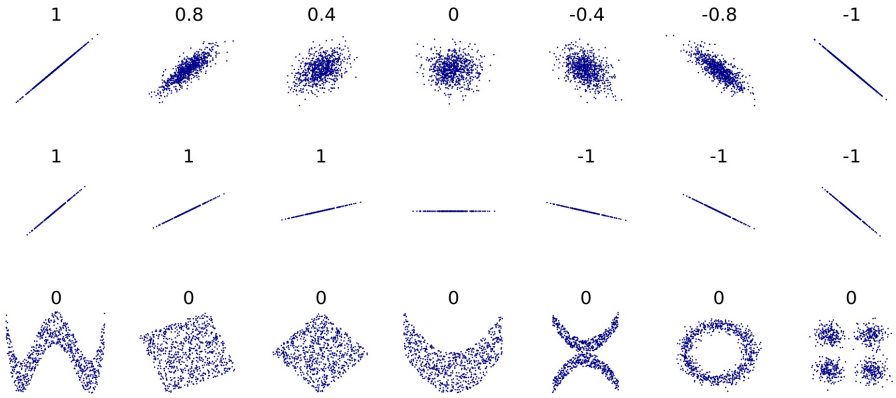


http://www.nejm.org/doi/full/10.1056/NEJMon1211064 (http://www.nejm.org/doi/full/10.1056/NEJMon1211064)

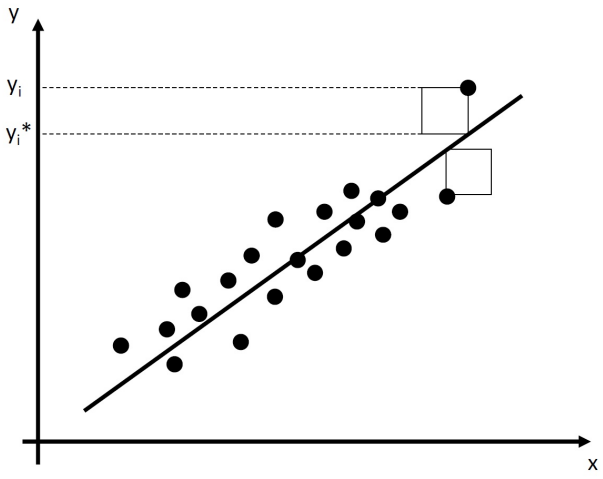
Sometimes called spurious correlation*

2.3.1 Correlation is often useless

- Correlation reflects the of a linear relationship (top row), it tells us nothing about the slope (strength) of that relationship (middle). Thus, we cannot interpret them economically.
- Moreover, many aspects of nonlinear relationships (bottom) remain unexplained. The figure in the center has a slope of 0 but in that case the correlation coefficient is undefined because the variance of \(Y\) is zero.



- **Linear regression** is a predictive modeling techniques that aims to find a mathematical equation for a variable \(y\) as a function of one (simple linear model) or more variables (multiple linear regression), \(x\).
- The method to is called the **ordinary least squared (OLS) method** as it minimizes the sum of the squared differences of all \(y_i\) and \(y_i^*\) as sketched below.



The simple linear regression model is \[y_i = \beta_{0} + \beta_{1} x_i + \epsilon_i \] where

- the index \(i\) runs over the observations, \(i=1,\dots,n\)
- \(y_i\) is the , the regressand
- \(x_i\) is the , the regressor
- \(\beta_0\) is the or constant
- \(\beta_1\) is the slope of regression line
- \(\epsilon_i\) is the or the residual.

3 OLS estimation method

- minimize the squared residuals by choosing the estimated coefficients \(\hat{\beta_{0}}\) and \(\hat{\beta_{1}}\)

\[\min_{\hat{\beta_{0}}, \hat{\beta_{1}}}\sum_{i=1} \epsilon_i^2 = \sum_{i=1} (y_i - \hat{\beta_{0}} - \hat{\beta_{1}} x_i)^2\]

- Minimizing the function requires to calculate the first order conditions with respect to \(\hat{\beta_{0}}\) and \(\hat{\beta_{1}}\) and set them zero (see exercises)
- The estimators are: \[\hat{\beta_{0}} =\bar{y}-\hat{\beta_{1}}\bar{x}\]

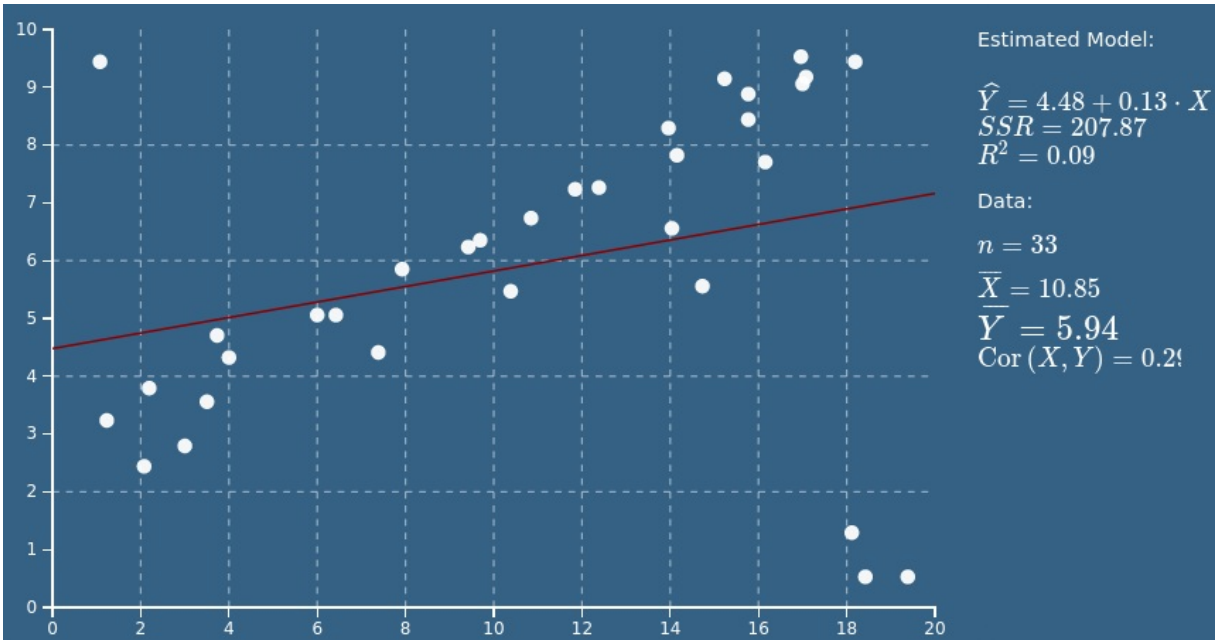
\[\hat{\beta_{1}} =\frac{\sum_{i=1}(y_i -\bar{y})(x_i-\bar{x})}{\sum_{i=1} (\bar{x} - x_i)^2 }\]

In the multiple regression model the OLS is derived similarly but we skip the derivation. The \(\hat{\beta}\) coefficient vector can be expressed as follows: \[\hat{\beta}=\left(\mathrm{X} ^{\mathsf {T}}\mathrm {X} \right)^{-1}\mathrm {X} ^{\mathsf {T}}\mathbf {y}\]

Here you find the derivation of the OLS estimator for the multiple regression model: ch. 9.1 Applied Statistics with R (https://daviddalpiaz.github.io/appliedstats/multiple-linear-regression.html)

4 Caveats of OLS (outliers are bad)

On this website (https://www.econometrics-with-r.org/4-2-estimating-the-coefficients-of-the-linear-regression-model.html) you find an interactive application. Play around with it and discuss possible caveats of the OLS method.



5 Example

In the statistic course of WS 2020, I asked 23 students about their weight, height, sex, and number of siblings:

```
library("haven")
classdata <- read.csv("https://raw.githubusercontent.com/hubchev/courses/main/dta/classdata.csv")

head(classdata)
```

```
##   id sex weight height siblings row
## 1  1   w     53    156        1   g
## 2  2   w     73    170        1   g
## 3  3   m     68    169        1   g
## 4  4   w     67    166        1   g
## 5  5   w     65    175        1   g
## 6  6   w     48    161        0   g
```

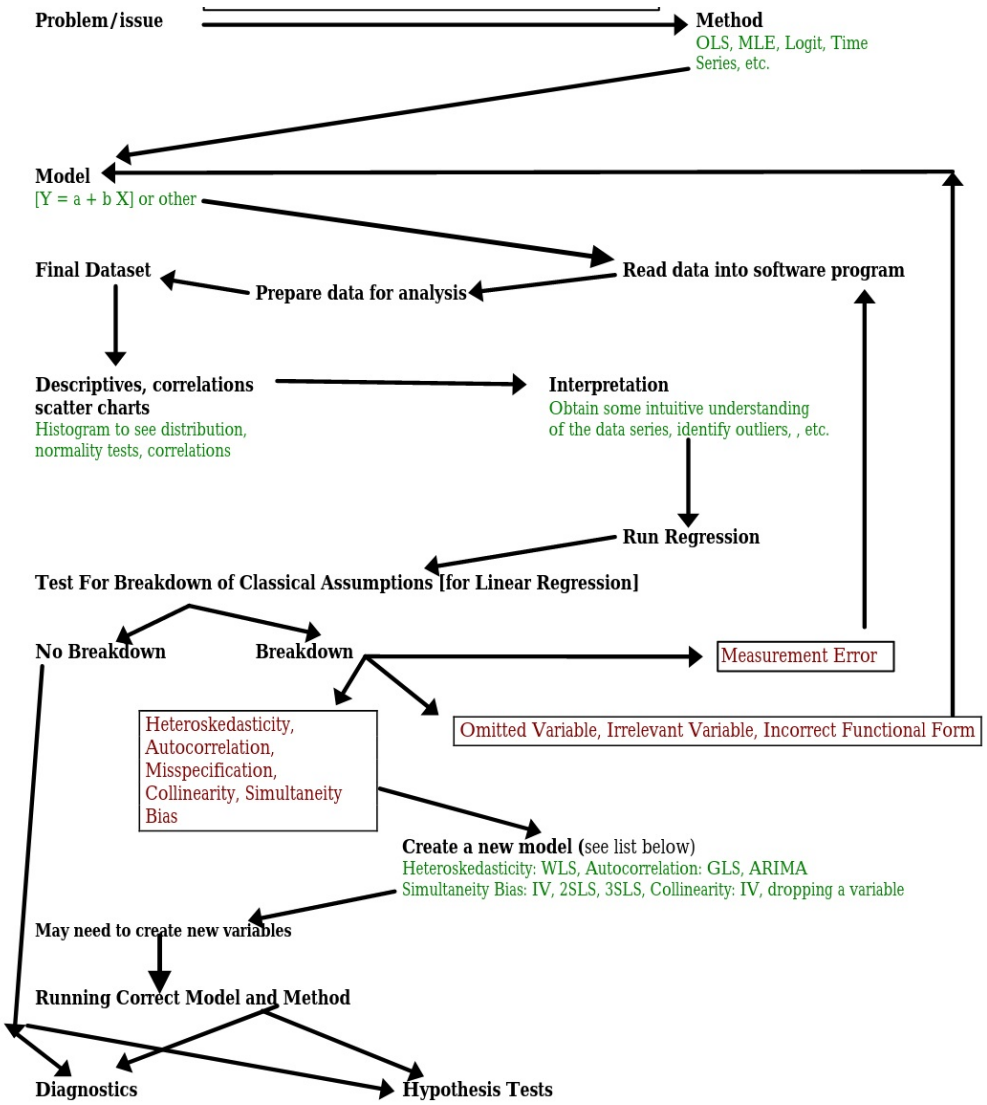
```
summary(classdata)
```

```
##        id           sex                weight          height
##  Min.   : 1.0   Length:23          Min.   :48.00   Min.   :156.0
##  1st Qu.: 6.5   Class :character   1st Qu.:64.50   1st Qu.:168.0
##  Median :12.0   Mode  :character   Median :70.00   Median :175.0
##  Mean   :12.0                      Mean   :70.61   Mean   :173.7
##  3rd Qu.:17.5                      3rd Qu.:81.00   3rd Qu.:180.0
##  Max.   :23.0                      Max.   :90.00   Max.   :194.0
##     siblings         row
##  Min.   :0.000   Length:23
##  1st Qu.:1.000   Class :character
##  Median :1.000   Mode  :character
##  Mean   :1.391
##  3rd Qu.:2.000
##  Max.   :4.000
```

5.1 How to execute a regression analysis

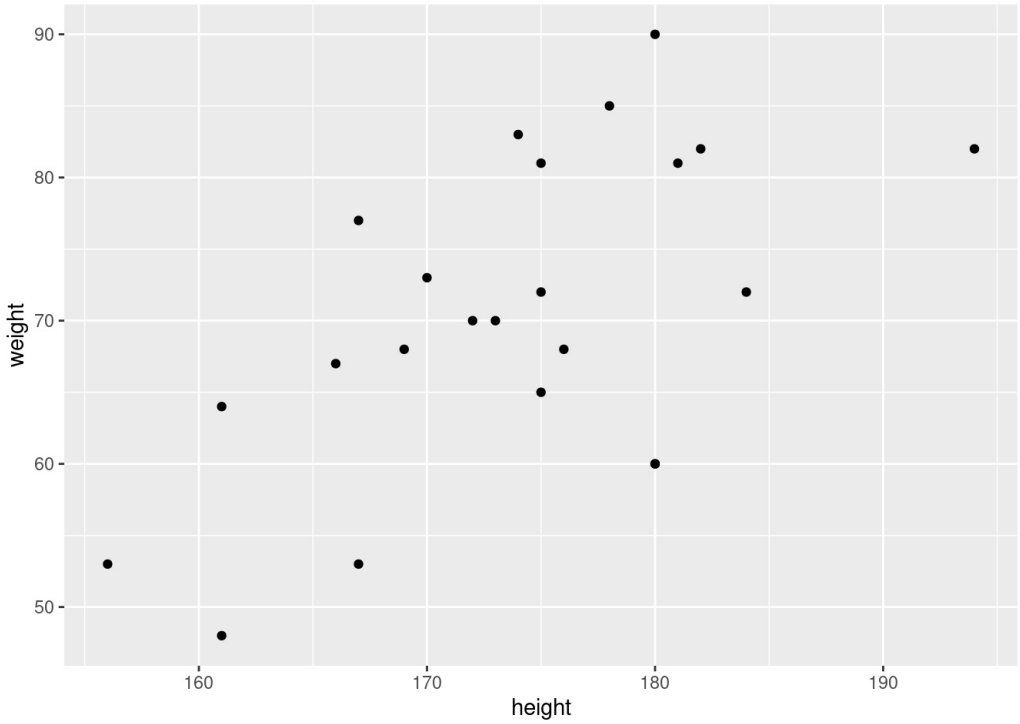
1. get known to the data
2. build a theory on how the variables may be related
3. derive a estimated equation from your theory
4. estimate
5. evaluate your empirics
6. go back to 2. and improve your theory
7. interpret your results

A more elaborated flow diagram for regression analysis can be found below. Source: http://medrescon.tripod.com/regression_explained.pdf (http://medrescon.tripod.com/regression_explained.pdf)



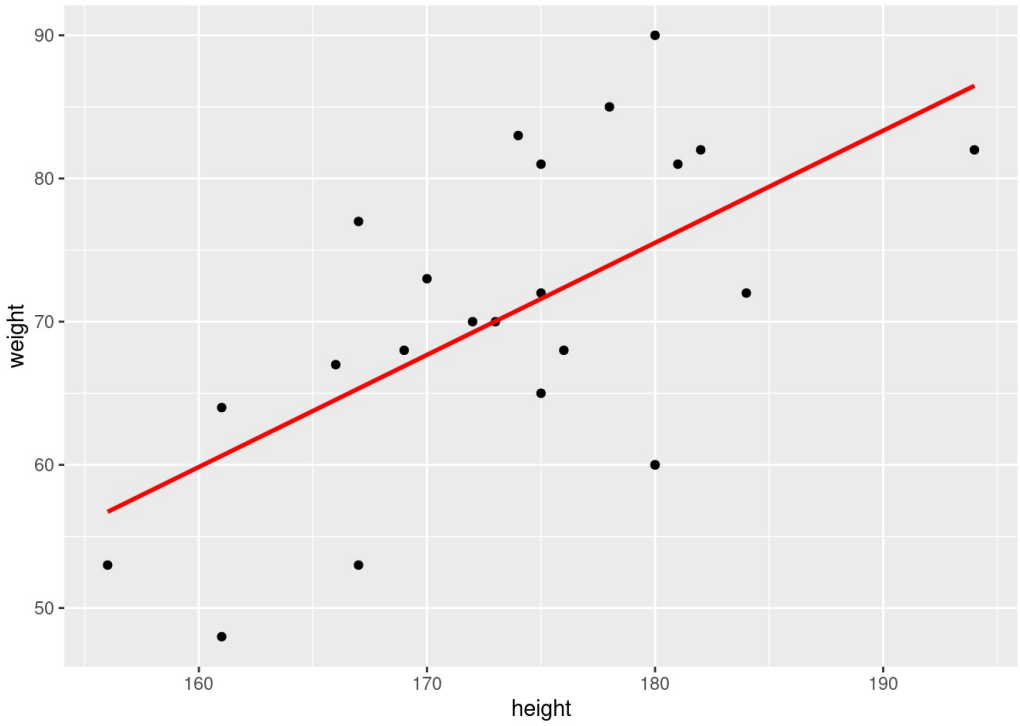
5.2 First look at data

```
library("ggplot2")
ggplot(classdata, aes(x=height, y=weight)) + geom_point()
```



include a regression line:

```
ggplot(classdata, aes(x=height, y=weight)) +
  geom_point() +
  stat_smooth(formula=y~x, method="lm", se=FALSE, colour="red", linetype=1)
```



distinguish male/female by including a seperate constant:

```
## baseline regression  model
model  <- lm(weight ~ height + sex , data = classdata )
show(model)
```

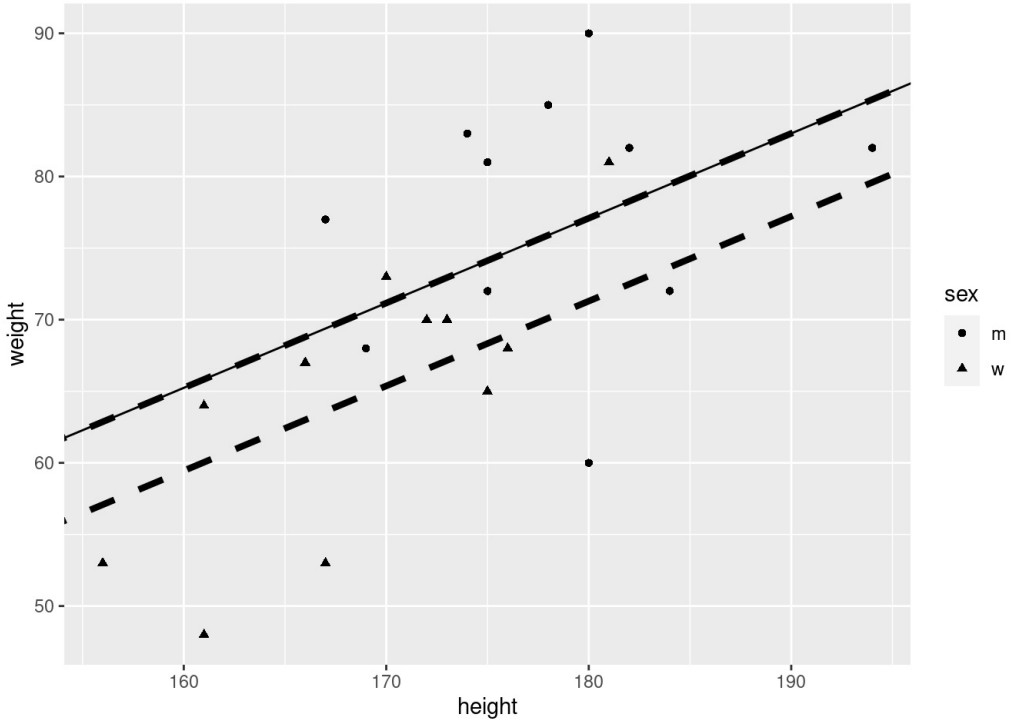
```
##
## Call:
## lm(formula = weight ~ height + sex, data = classdata)
##
## Coefficients:
## (Intercept)       height         sexw
##    -29.5297       0.5923      -5.7894
```

```
interm <- model$coefficients[1]
slope  <- model$coefficients[2]
interw <- model$coefficients[1]+model$coefficients[3]
```

```
summary(model)
```

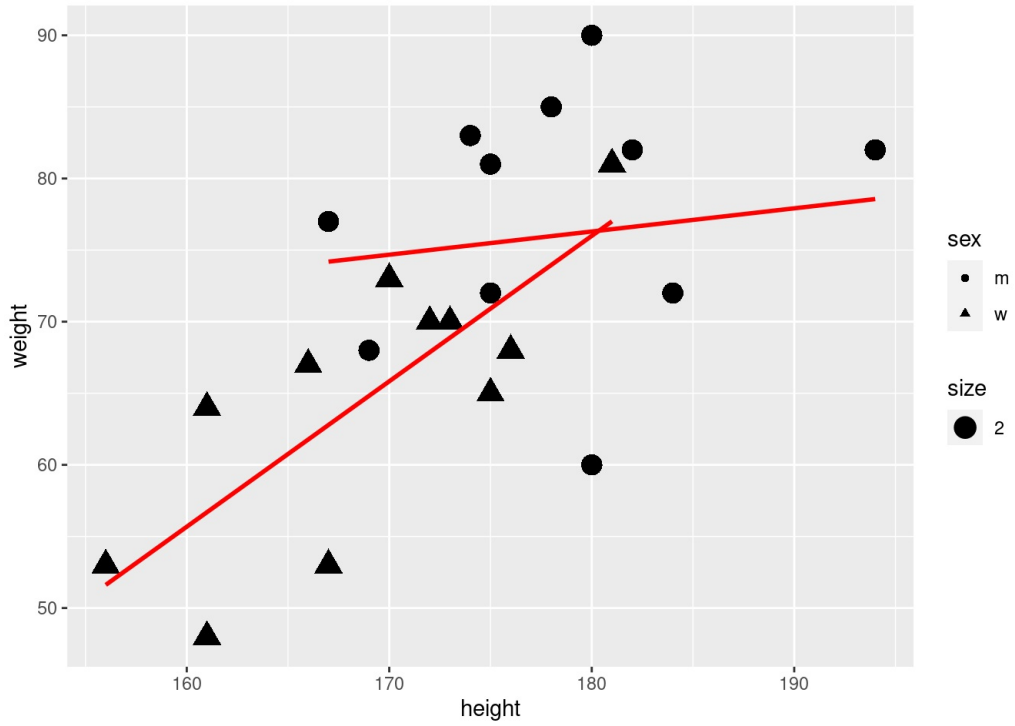
```
##
## Call:
## lm(formula = weight ~ height + sex, data = classdata)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.086  -3.730   2.850   7.245  12.914
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -29.5297    47.6606  -0.620   0.5425
## height        0.5923     0.2671   2.217   0.0383 *
## sexw         -5.7894     4.4773  -1.293   0.2107
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.942 on 20 degrees of freedom
## Multiple R-squared:  0.4124, Adjusted R-squared:  0.3537
## F-statistic: 7.019 on 2 and 20 DF,  p-value: 0.004904
```

```
ggplot(classdata, aes(x=height, y=weight, shape = sex)) +
  geom_point() +
  geom_abline(slope = slope, intercept = interw, linetype = 2, size=1.5)+
  geom_abline(slope = slope, intercept = interm, linetype = 2, size=1.5) +
  geom_abline(slope = coef(model)[[2]], intercept = coef(model)[[1]])
```



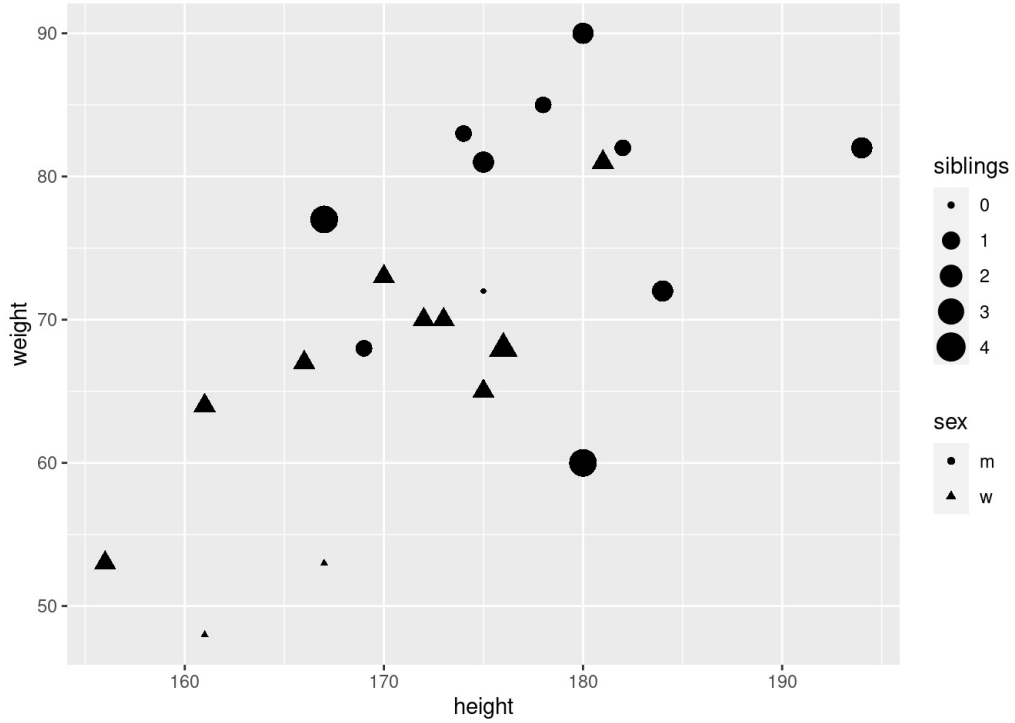
does not look to good, maybe we should introduce also different slopes for m/w

```
ggplot(classdata, aes(x=height, y=weight, shape = sex)) +
  geom_point( aes(size = 2)) +
  stat_smooth(formula = y ~ x,  method = "lm",
              se = FALSE, colour = "red", linetype = 1)
```



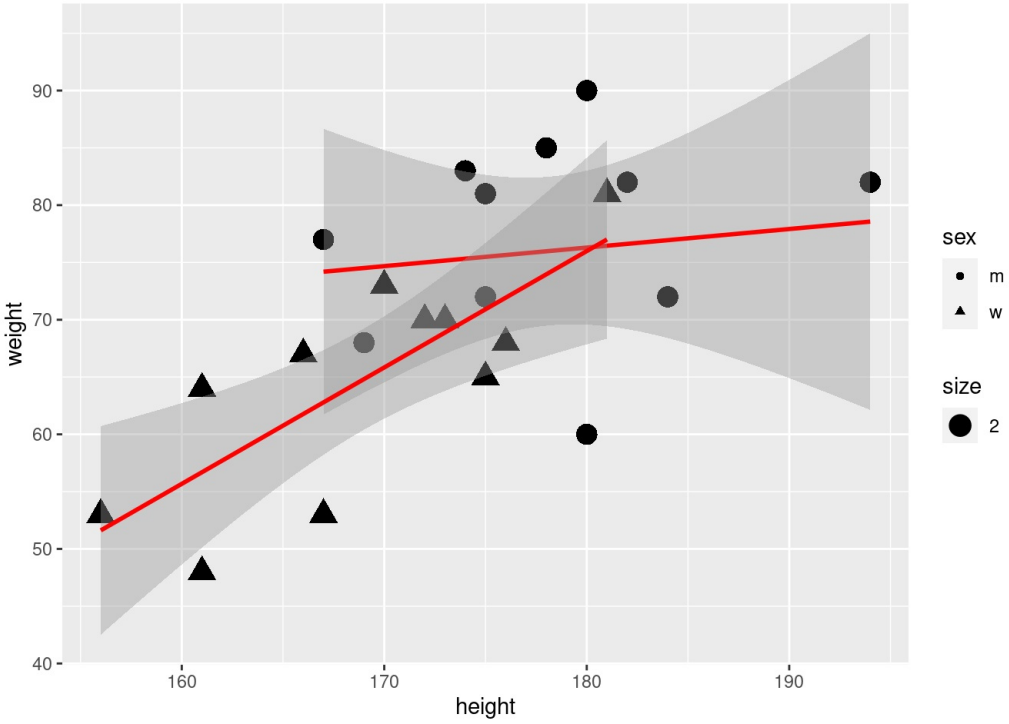
Can we use other available variables: siblings?

```
ggplot(classdata, aes(x=height, y=weight, shape = sex)) +
  geom_point( aes(size = siblings))
```




```
## baseline model
model  <- lm(weight ~ height + sex , data = classdata )

ggplot(classdata, aes(x=height, y=weight, shape = sex)) +
  geom_point( aes(size = 2)) +
  stat_smooth(formula = y ~ x,
              method = "lm",
              se = T,
              colour = "red",
              linetype = 1)
```



Let us look at regression output:

```
m1 <- lm(weight ~ height , data = classdata )
m2 <- lm(weight ~ height + sex , data = classdata )
m3 <- lm(weight ~ height + sex + height * sex , data = classdata )
m4 <- lm(weight ~ height + sex + height * sex + siblings , data = classdata )
m5 <- lm(weight ~ height + sex + height * sex , data = subset(classdata, siblings < 4 ))

library(sjPlot)
```

```
## Registered S3 method overwritten by 'parameters':
##   method                          from
##   format.parameters_distribution datawizard
```

```
tab_model(m1, m2, m3, m4, m5,
          p.style = "stars",
          p.threshold = c(0.2, 0.1, 0.05),
          show.ci = FALSE,
          show.se = FALSE)
```

	weight	**weight**	**weight**	**weight**
Predictors	*Estimates*	*Estimates*	*Estimates*	*Estimates*
(Intercept)	-65.44 *	-29.53	47.14	50.27
height	0.78 ***	0.59 ***	0.16	0.16
sex [w]		-5.79	-153.96 **	-161.92 **
height * sex [w]			0.85 *	0.89 *
siblings				-1.16
Observations	23	23	23	23
R^2 / R^2 adjusted	0.363 / 0.333	0.412 / 0.354	0.487 / 0.407	0.496 / 0.385

- *p<0.2 ** p<0.1 *** p<0.05*

excluding outliers with four siblings:

	weight	**weight**
Predictors	*Estimates*	*Estimates*

(Intercept)	47.14	27.69
height	0.16	0.28
sex [w]	-153.96 **	-134.51 *
height * sex [w]	0.85 *	0.74 *
Observations	23	21
R^2 / R^2 adjusted	0.487 / 0.407	0.572 / 0.497

- *p<0.2 ** p<0.1 *** p<0.05*

5.3 Interpretation of the results

- We can make predictions about the impact of height on male and female
- As both, the intercept and the slope differs for male and female we should interpret the regressions seperately:
- One centimeter more for **MEN** is *on average* and *ceteris paribus* related with 0.16 kg more weight.
- One centimeter more for **WOMEN** is *on average* and *ceteris paribus* related with 1.01 kg more weight.

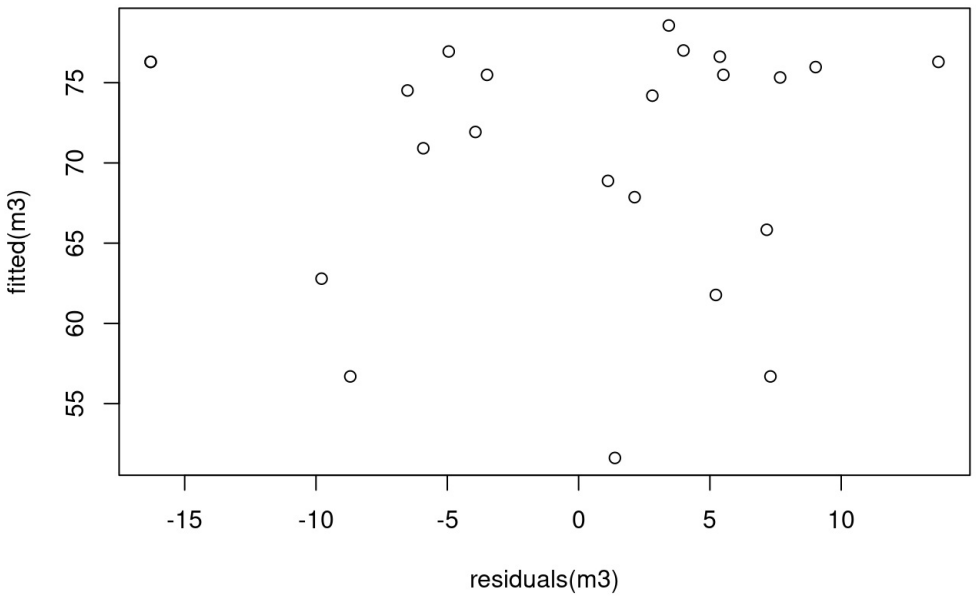
5.4 Regression Diagnostics

Linear Regression makes several assumptions about the data, the model assumes that:

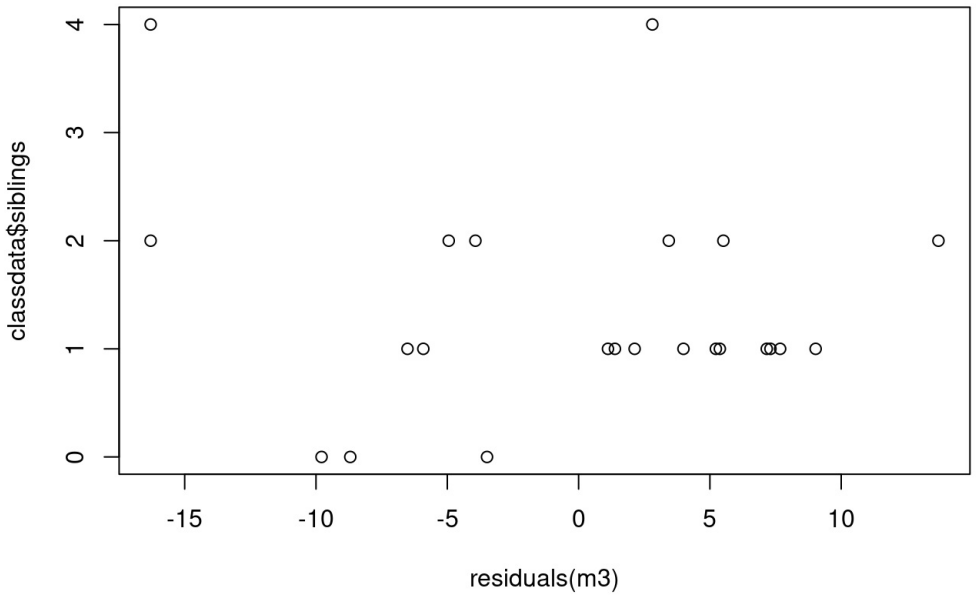
- The relationship between the predictor (x) and the dependent variable (y) has linear relationship.
- The residuals are assumed to have a constant variance.
- The residual errors are assumed to be normally distributed.
- Error terms are independent and have zero mean.

More on regression Diagnostics can be found Applied Statistics with R: 13 Model Diagnostics (https://daviddalpiaz.github.io/appliedstats/model-diagnostics.html#r-markdown-6)

```
plot(residuals(m3), fitted(m3))
```

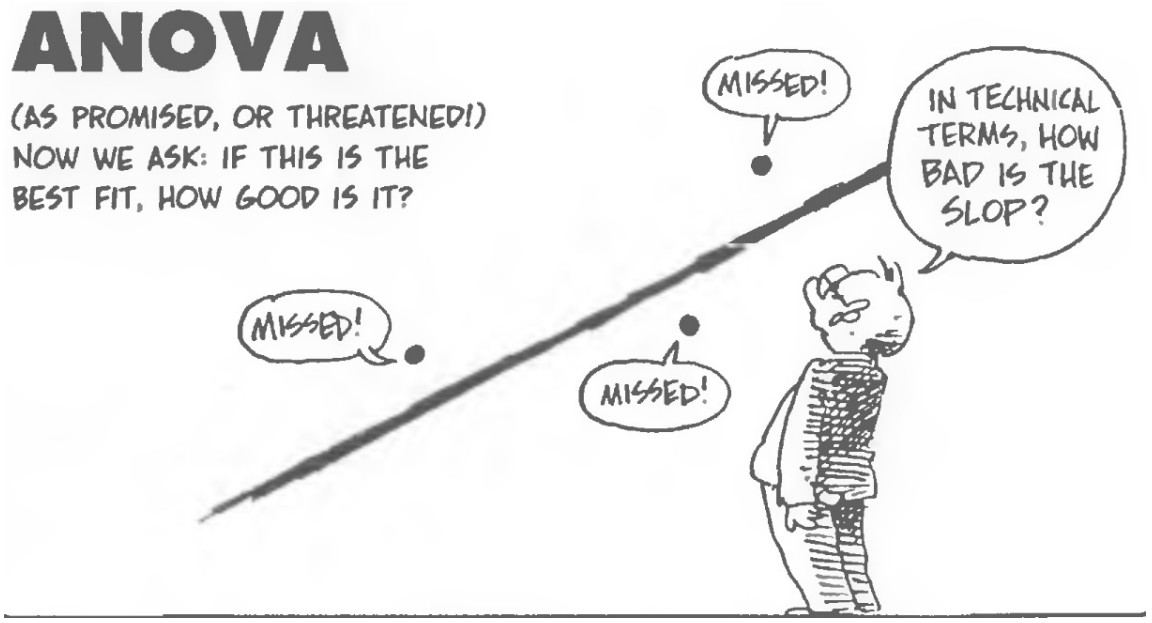



```
plot(residuals(m3), classdata$siblings)
```



5.5 Measures of fit

5.5.1 R squared

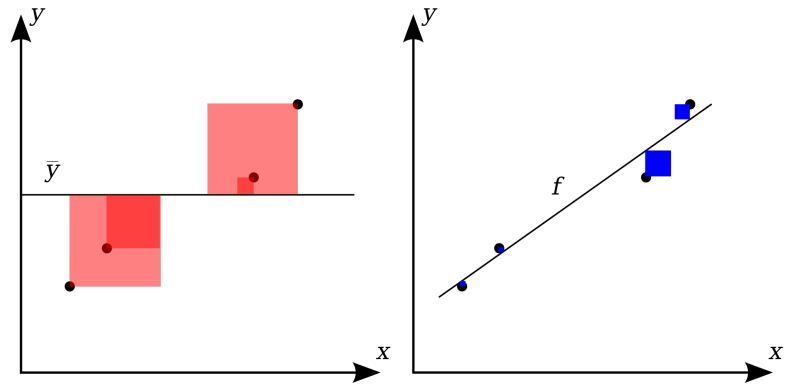


R^2 is the fraction of the sample variance of Y_i that is explained by X_i. It can be written as the ratio of the explained sum of squares (ESS) to the total sum of squares (TSS): $$ESS = \sum_{i = 1}^n \left(\hat{Y_i} - \bar{Y} \right)^2,$$ $$TSS = \sum_{i = 1}^n \left(Y_i - \bar{Y} \right)^2,$$ $$R^2 = \frac{ESS}{TSS}.$$

Since $TSS = ESS + SSR$ we can also write

$R^2 = 1- \frac{SSR}{TSS}$

with $SSR=\sum_{i=1} \epsilon_i^2$



R^2 lies between 0 and 1. It is easy to see that a perfect fit, i.e., no errors made when fitting the regression line, implies $R2=1$ since then we have $SSR=0$. On the contrary, if our estimated regression line does not explain any variation in the Y_i, we have $ESS=0$ and consequently $R^2=0$.

5.5.2 Adjusted R-squared

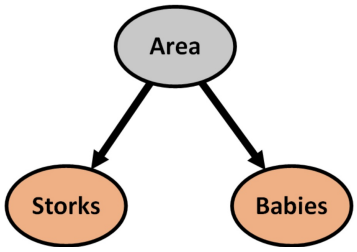
Including more independent variables into an estimated model must decrease SSR. Thus, the R-squared never decreases, when adding new variables even when it just a chance correlation between variables. Having more coefficients to estimate the precision at which we can estimate the effects decrease. Thus, we need to deal with the trade-off. The adjusted R^2 can help here:

$$\displaystyle {\bar {R}}^{2}=1-(1-R^{2}){n-1 \over n-p-1}}$$

Always use adjusted R^2 when you compare specifications with different number of coefficients.

5.6 The miracle of CONTROL VARIABLES in multiple regressions

Control variables are usually variables that you are not particularly interested in, but that are related to the dependent variable. You want to remove their effects from the equation. A control variable enters a regression in the same way as an independent variable – the method is the same.

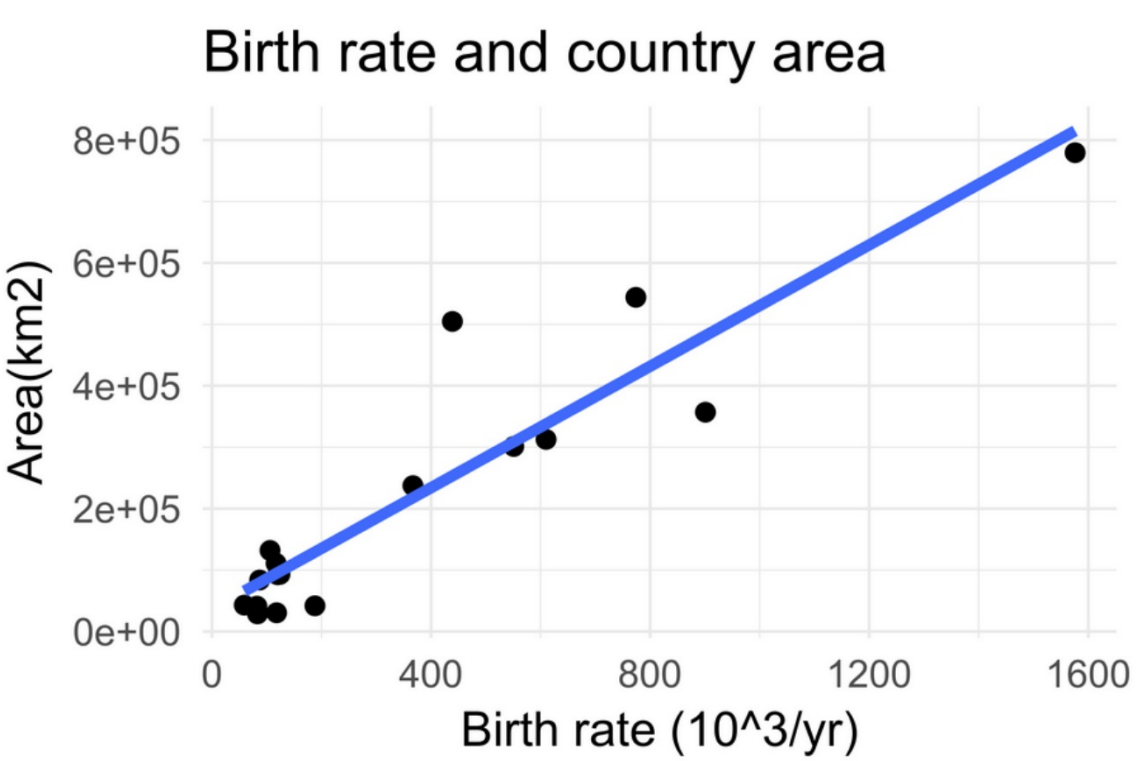


5.7 When do we need (more) control variables

From the Gauss-Markov theorem (https://www.econometrics-with-r.org/5-5-the-gauss-markov-theorem.html) we know that if the OLS assumptions (https://www.econometrics-with-r.org/6-4-ols-assumptions-in-multiple-regression.html) are fullfiled, the OLS estimator is (in the sense of smallest variance) the **best linear conditionally unbiased estimator (BLUE)**.

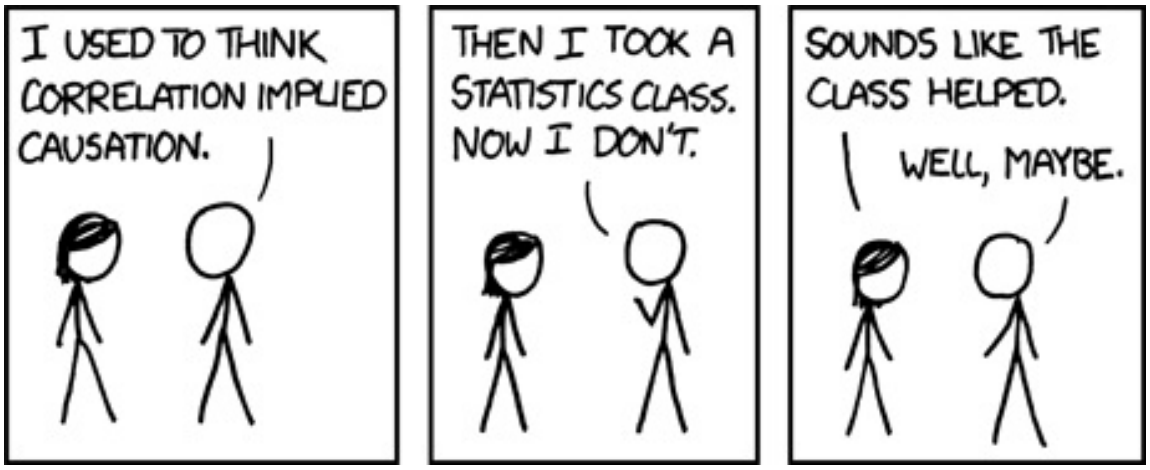
However, OLS estimates can suffer from **omitted variable bias** when the regressor, X, is correlated with an omitted variable. For omitted variable bias to occur, two conditions must be fulfilled:

```
1. X is correlated with the omitted variable.
2. The omitted variable is a determinant of the dependent variable Y.
```



6 Take away messages

- Regressions rule the world and may kill alternative facts.
- Correlation does not imply causation.
- It is hard to find the true *data generating process*.




```
rmarkdown::render("regress_lecture.Rmd", "all")
```

```
wkhtmltopdf regress_lecture.html regress_lecture.pdf
```

```
## Loading page (1/2)
## [>                                                            ] 0%
[======>                                                       ] 10%
[=====================================================>        ] 90%
[=============================================================] 100%
Printing pages (2/2)
## [>                                                            ]
Done
```

A.3 Transcript of swirl learning moduls

> The following stuff, i.e, the transcript of the swirl learning modules, can contain errors and I do not give warranty that the content matches to the learning modules. In particular, things like figures, R-scripts and output will not be shown here.

A.3.1 Swirl learning module *tidyverse*

I wrote a short swirl module which you can find on my github page. To install it type

```
library("swirl)
install_course_github("hubchev", "swirl-it")
swirl()
```

Then you should choose the course "swirl-it" and the learning module "huber-tidyverse".[1]

Here is a transcript of the swirl learning module:

This lesson should help you to understand why the 'tidyverse' world is interesting. Working with R is easier with tidyverse!

The tidyverse is an collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Developed by RStudio's chief scientist Hadley Wickham, tidyverse provides an efficient, fast, and well-documented workflow for general data modeling, wrangling, and visualization tasks.

Open the tidyverse website and see what all is involved. www.tidyverse.org/packages/

The tidyverse introduces a set of useful data analysis packages to help streamline your work in R. In particular, the Tidyverse was designed to address the top three common issues that arise when dealing with data analysis in R functions.

Results obtained from a base R function often depend on the type of data being used.

When R expressions are used in a non-standard way, they can confuse beginners.

Hidden arguments often have various default operations that beginners are unaware of.

I would like to briefly introduce you to the most important packages

dplyr is actually not only a pliers, but a whole toolbox of functions to manipulate the dataset the way we need it. No, this is not about p-hacking or faking results! It's about data manipulation and accesses, calculating new variables, looking only at certain subsets of data... so the basics before we can start with a meaningful data analysis.

tidyr helps us to get a 'clean' dataset. That means a well structured dataset where we can find everything well. Also it helps us to create two different types of data formats (wide and long). Some functions in R like to be fed with a wide data set, others with the long format.

Hadley Wickham is the main developer of the whole tidyverse and the chief researcher of RStudio. His book 'R for Data Science' is great. Please give chapter 3 'Data visualisation' a reading after you have finished this swirl lesson (https://r4ds.had.co.nz/data-visualisation.html)

Another package is tibble. It is the name of the package, but also of the data structure that can be created with it. Basically, a tibble is a data frame. However, a tibble doesn't do weird conversions and gives more error messages when something goes wrong. That means less weird things happen automatically and there is more information about where problems come from.

Read chapter 4 'Workflow basics' of the book 'R for Data Science' now. Here you can review what you already know, but also learn a few new aspects. (It's not very long). (https://r4ds.had.co.nz/workflow-basics.html)

I am serious. Read it now!

And don't forget the exercises of chapter 4. Try them now!

If you struggle with some exercises in the book 'R for Data Science' you can look at the book 'R for Data Science. Exercise Solutions' by Jeffrey B. Arnold. This nice guy solved everything. (https://jrnold.github.io/r4ds-exercise-solutions/)

[1] If the course has failed to install, you can try to download the file `swirl-it.swc` from https://github.com/hubchev/swirl-it and install the course with `install_course()`.

An important tool is the so called 'pipe'. It comes from the package 'magrittr' and is automatically loaded by packages from the tidyvers. So it is always automatically available to you as well.

At the beginning of this lesson, tidyverse was already loaded. (If it was not already installed, Swirl asked you if it should be installed). You can also load tidyverse manually with 'library(tidyverse)'.

Now what is this 'pipe'? The pipe looks like this `%>%`. The key combination CTRL + SHIFT + M (CMD + SHIFT + M for OSX) creates the pipe for you. There are also special pipes, but we are not interested in them for now.

How does this help us? Pipe allows us to write our R code in a form that is much easier to read and understand.

Let's look at an example. Let us load the data set 'dataset'. This data should already be loaded in your environment. I am now, for some reason, interested in the median income. However, only from people who have job 1 and have more than two friends. Feel free to look at the dataset again so you know what it's about. Do you remember how to do this?

Let's print first 10 rows of the data set 'dataset' by typing `head(dataset, n=10)`

```
head(dataset, n=10)
```

```
##      name birth_date job friends alcohol income neurotic
## 1     Ben   7/3/1977   1       5      10  20000       10
## 2  Martin  5/24/1969   1       2      15  40000       17
## 3    Andy  6/21/1973   1       0      20  35000       14
## 4    Paul  7/16/1970   1       4       5  22000       13
## 5  Graham 10/10/1949   1       1      30  50000       21
## 6  Carina  11/5/1983   2      10      25   5000        7
## 7  Karina  10/8/1987   2      12      20    100       13
## 8    Doug  1/23/1989   2      15      16   3000        9
## 9    Mark  5/20/1973   2      12      17  10000       14
## 10    Zoe 11/12/1984   2      17      18     10       13
```

One possibility would be this code `mean(dataset[dataset$job==1 & dataset$friends >2, c('income')])`. Don't be scared! We'll go through everything step by step. Copy this code into the console and run it.

```
mean(dataset[dataset$job==1 & dataset$friends >2, c('income')])
```

```
## [1] 21000
```

We have a nesting of functions here. We have to work our way from the inside to the outside. So let us look at this part `dataset[dataset$job==1 & dataset$friends >2, c('income')]`

We access the dataset and make a data access (i.e. we use square brackets [ ]).

Do you know how the access works? It works like this `data [rows, columns]`. Each row contains the data for one person. We want only the rows, or only the people who have job 1. So the column job in the dataset should have the value 1 `dataset$job==1`.Furthermore the column friends should be greater than two and `dataset$friends > 2`.

If we would now type `dataset[dataset$job == 1 & dataset$friends > 2,]` R will give us all rows where this is true. In addition, R outputs all columns to us. We have not told it anything else.

However, we are only interested in the values of the income column. Therefore we type `dataset[dataset$job == 1 & dataset$friends > 2, c('income')]`

Note, we would not necessarily need to write c() around 'income'. R only accepts one specification for the columns to be displayed. Here we want only one column and have no problem. However, if we want income and neuroticism to be displayed, we can't(!) just write `dataset[... , 'income', 'neuroticism']`. So we would have to use `dataset[... , c('income', 'neuroticism')]`. c() creates a vector with the two elements. Thus, it is only one element for R and R knows what to do.

Let us illustrate this again with another example. We will now use the function mean(). Type ?mean to get information about the function. (We use the function from the 'base' package, which is always available to us).

```
?mean
```

You should now see the help for the mean function at the bottom right. We need to pass an object x to the function. We can use comma to give the function more arguments 'trim' and 'na.rm'

Now type 'mean(10,20,30)'

```
mean(10,20,30)
```

```
## [1] 10
```

R interprets 10 as the object x. R interprets 20 and 30 as further arguments of the function that do something. But we wanted to have the mean value of 10,20,30.

Now type 'mean(c(10,20,30))'. We now give the function mean a vector with the numbers 10, 20 and 30. The whole vector is now understood as the object x of the function, with the contents 10, 20 and 30. Now we also get the correct result - yay! Try it out.

```
mean(c(10,20,30))
```

```
## [1] 20
```

The important point here is that R can only know what to do if we 'feed' it correctly. If a command does not work, we can troubleshoot here.

But now back to the pipe... We used `dataset[dataset$job == 1 & dataset$friends > 2, c('income')]` to select the income of all people who have job 1 and have more than two friends. The results are output to us in a vector.

Around this result we now build the function mean(). Now we have completely understood the code from before. But hand on heart, it was quite a way to get there...

Now we will do exactly the same only with the pipe and functions from dplyr.

Run this code `dataset %>% filter(job==1, friends >2) %>% summarise(mean(income))`

```
dataset \%>\% filter(job==1, friends >2) \%>\% summarise(mean(income))
```

```
##   mean(income)
## 1        21000
```

As you can see, the result is the same as before. Look at the code `dataset %>% filter(job==1, friends >2) %>% summarise(mean(income))`. Can you imagine what the code does? Do you find it easier to read?

First, we tell R what data we are talking about 'dataset'. Now we 'move the pipe'. We pass the dataset to the next step. With the function filter() we now pick out only the cases where the job is 1 and there are more than two friends. We pass the result to the next function. With summarise() we can request a summary. i.e., we would like to have the mean value of the income.

So with the pipe we can reach our goal step by step. The intermediate steps are automatically passed on to the next step.

So the pipe can help us. But it is also not a universal remedy. When we create very long concatenations, it becomes harder and harder to find errors. Sometimes it is also useful to save or at least view the intermediate results.

I hope that you did not get confused, but realized that the Pipe can be very useful. If you got a small overview here and an idea of what all this is useful for you, then you have already learned more than enough in this lesson. But you can also read chapter 18 of the book 'R for data science'. Time for a break!

A.3.2 Swirl learning modules *huber-intro-1* and *huber-intro-2*

huber-intro-1

Welcome to this swirl course. If you find any errors or if you have suggestions for improvement, please let me know via stephan.huber@hs-fresenius.de .

The RStudio interface consists of several windows. You can change the size of the windows by dragging the grey bars between the windows. We'll go through the most important windows now.

Bottom left is the Console window (also called command window/line). Here you can type commands after the > prompt and R will then execute your command. This is the most important window, because this is where R actually does stuff.

Top left is the Editor window (also called script window). Here collections of commands (scripts) can be edited and saved. When you do not get this window, you can open it with 'File' > 'New' > 'R script'.

Just typing a command in the editor window is not enough, it has to be send to the Console before R executes the command. If you want to run a line from the script window (or the whole script), you can click 'Run' or press 'CTRL+ENTER' to send it to the command window.

The shortcut to run send the current line to the console is _____.

1. CTRL+SHIFT
2. CTRL+ENTER
3. CTRL+SPACE
4. SHIFT+ENTER

CTRL+ENTER

If you are a Mac user, your shortcut is 'Cmd+Return' instead of 'SHIFT+ENTER'.

Top right is the environment window (a.k.a workspace). Here you can see which data R has in its memory. You can view and edit the values by clicking on them.

Bottom right is the plots / packages / help window. Here you can view plots, install and load packages or use the help function.

The first thing you should do whenever you start Rstudio is to check if you are happy with your working directory. That directory is the folder on your computer in which you are currently working. That means, when you ask R to open a certain file, it will look in the working directory for this file, and when you tell R to save a data file or figure, it will save it in the working directory.

You can check your working directory with the function getwd(). So let's do that. Type in the command window 'getwd()' .

```
getwd()
```

```
## [1] "/home/sthu/Dropbox/hsf"
```

Are you happy with that place? if not, you should set your working directory to where all your data and script files are (or will be). Within RStudio you can go to 'Session' > 'Set working directory' > 'Choose directory'. Please do this now.

Instead of clicking, you can use the function setwd("/YOURPATH"). For example, setwd("/Users/MYNAME/MYFOLDER") or setwd("C:/Users/jenny/myrstuff"). Make sure that the slashes are forward slashes and that you do not forget the apostrophes. R is case sensitive, so make sure you write capitals where necessary.

Whenever you want R to do something you need to use a function. It is like a command. All functions of R are organized in so-called packages or libraries. With the standard installation many packages are already installed. However, many more exist and some of them are really cool. For example, with installed.packages() all installed packages are listed. Or, with swirl(), you started swirl.

Of course, you can also go to the Packages window at the bottom right. If the box in front of the package name is ticked, the package is loaded (activated) and can be used. To see via Console which packages are loaded type in the console '(.packages())'

```
(.packages())
```

```
## [1] "swirl"     "stats"     "graphics"  "grDevices" "utils"     "datasets"
## [7] "methods"   "base"
```

There are many more packages available on the R website. If you want to install and use a package (for example, the package called 'geometry') you should first install the package. Type install.packages("geometry") in the console. Don't be afraid about the many messages. Depending on your PC and your internet connection this may take some time.

```
install.packages("geometry")
```

```
## Installing package into '/home/sthu/R/x86_64-pc-linux-gnu-library/4.1'
## (as 'lib' is unspecified)
```

After having installed a package, you need to load the package. That is a bit anoying but essential. Type in library(geometry) in the Console. You also did this for the swirl package (otherwise you couldn't have been doing these exercises).

```
library(geometry)
```

Check if the package is loaded typing '(.packages())'

```
(.packages())
```

```
## [1] "geometry"  "swirl"     "stats"     "graphics"  "grDevices" "utils"
## [7] "datasets"  "methods"   "base"
```

Now, let's get started with the real programming.

R can be used as a calculator. You can just type your equation in the command window after the >. Type 10^2 + 36.

```
10^2 + 36
```

```
## [1] 136
```

And R gave the answer directly. By the way, spaces do not matter.

If you use brackets and forget to add the closing bracket, the > on the command line changes into a +. The + can also mean that R is still busy with some heavy computation. If you want R to quit what it was doing and give back the >, press ESC.

You can also give numbers a name. By doing so, they become so-called variables which can be used later. For example, you can type in the command window A <- 4 .

```
A <- 4
```

The '<-' is the so-called assignment operator. It allows you to assign data to a named object in order to store the data.

Don't be confussed about the term object. All sorts of data are stored in so-called objects in R. All objects of a session are shown in the Environment window. In the second part of this course, I will introduce different data types.

You can see that A appeared in the environment window in the top right corner, which means that R now remembers what A is.

You can also ask R what A is. Just type A in the command window.

```
A
```

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

You can also do calculations with A. Type A * 5 .

```
A*5
```

```
## [1] 20
```

If you specify A again, it will forget what value it had before. You can also assign a new value to A using the old one. Type A <- A + 10 .

```
A <- A + 10
```

You can see that the value in the environment window changed.

To remove all variables from R's memory, type rm(list=ls()) .

```
rm(list=ls())
```

You see that the environment window is now empty. You can also click the broom icon (`clear all`) in the environment window. You can see that RStudio then empties the environment window. If you only want to remove the variable A, you can type rm(A).

Like in many other programs, R organizes numbers in scalars (a single number, 0-dimensional), vectors (a row of numbers, also called arrays, 1-dimensional) and matrices (like a table, 2-dimensional).

The A you defined before was a scalar. To define a vector with the numbers 3, 4 and 5, you need the function c, which is short for concatenate (paste together). Type B=c(3,4,5) .

```
B=c(3,4,5)
```

If you would like to compute the mean of all the elements in the vector B from the example above, you could type (3+4+5)/3. Try this

```
(3+4+5)/3
```

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

But when the vector is very long, this is very boring and time-consuming work. This is why things you do often are automated in so-called functions. For example, type mean(x=B) and guess what this function mean() can do for you.

```
mean(x=B)
```

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

Within the brackets you specify the arguments. Arguments give extra information to the function. In this case, the argument x says of which set of numbers (vector) the mean should be computed (namely of B). Sometimes, the name of the argument is not necessary; mean(B) works as well. Try it.

```
mean(B)
```

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

Compute the sum of 4, 5, 8 and 11 by first combining them into a vector and then using the function sum. Use the function c inside the function sum.

```
sum(c(4,5,8,11))
```

```
## [1] 28
```

The function rnorm, as another example, is a standard R function which creates random samples from a normal distribution. Type rnorm(10) and you will see 10 random numbers

```
rnorm(10)
```

```
##  [1]  0.76977837 -0.14622141  0.53924840 -0.88086971  1.92077037 -0.28015385
##  [7]  0.01594662  1.64208469 -0.66669651  1.08656820
```

Here rnorm is the function and the 10 is an argument specifying how many random numbers you want - in this case 10 numbers (typing n=10 instead of just 10 would also work). The result is 10 random numbers organised in a vector with length 10.

If you want 10 random numbers out of normal distribution with mean 1.2 and standard deviation 3.4 you can type rnorm(10, mean=1.2, sd=3.4). Try this.

```
rnorm(10, mean=1.2, sd=3.4)
```

```
##  [1]  3.56916212 -0.50465437  0.95331590  0.07725623 -5.56529257  2.53201285
##  [7]  0.15829096  6.05524550 -3.35290698 -0.03229734
```

This shows that the same function (rnorm) may have different interfaces and that R has so called named arguments (in this case mean and sd).

Comparing this example to the previous one also shows that for the function rnorm only the first argument (the number 10) is compulsory, and that R gives default values to the other so-called optional arguments. Use the help function to see which values are used as default by typing ?rnorm.

```
?rnorm
```

You see the help page for this function in the help window on the right. RStudio has a nice features such as autocompletion and snapshots of the R documentation. For example, when you type rnorm(in the command window and press TAB, RStudio will show the possible arguments.

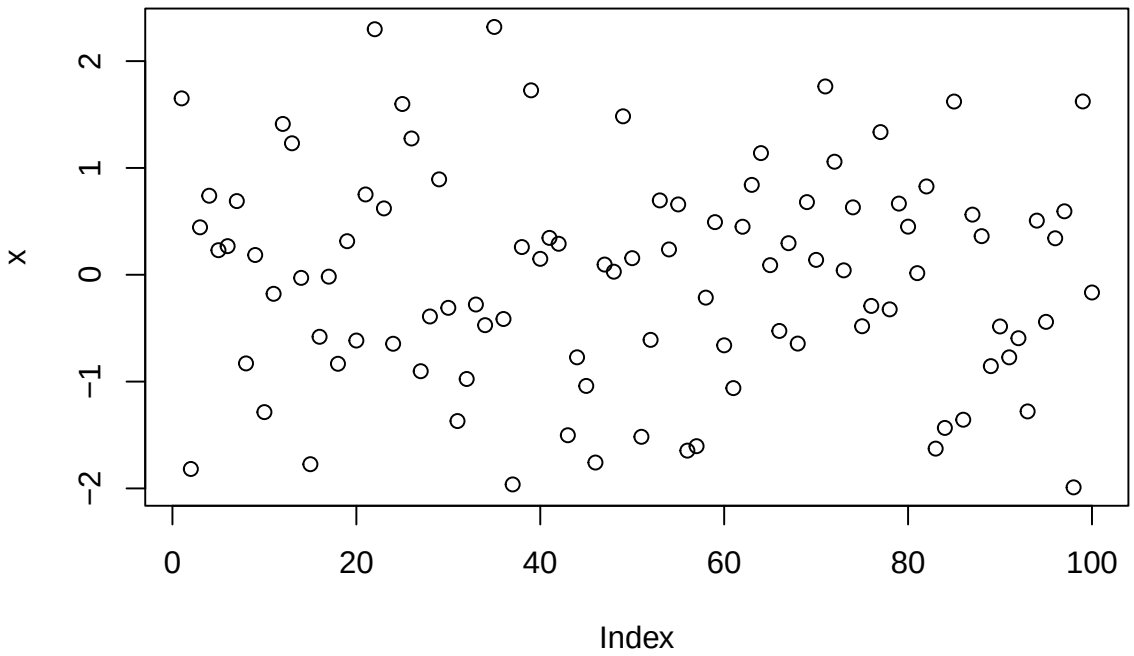
You can also store the output of the function in a variable. Type x=rnorm(100).

```
x=rnorm(100)
```

Now 100 random numbers are assigned to the variable x, which becomes a vector by this operation. You can see it appears in the Environment window.

R can also make graphs. Type plot(x) for a very simple example.

```
plot(x)
```



The 100 random numbers are now plotted in the plots window on the right.

You now are more familiar to RStudio and you know some basic R stuff. In particular, you know...

...that everything in R is said with functions,

...that functions can but don't have to have arguments,

...that you can install packages which contain functions,

...that you must load the installed packages every time you start a session in RStudio, and

...that this is just the beginning. Thus, please continue with the second module of this introduction.

huber-intro-2

Welcome to the second module. Again, if you find any errors or if you have suggestions for improvement, please let me know via stephan.huber@hs-fresenius.de .

Before you start working, you should set your working directory to where all your data and script files are or should be stored. Within RStudio you can go to 'Session'> 'Set working directory', or you can type in setwd(YOURPATH). Please do this now.

```
setwd(getwd())
```

R is an interpreter that uses a command line based environment. This means that you have to type commands, rather than use the mouse and menus. This has many advantages. Foremost, it is easy to get a full transcript of everything you did and you can replicate your work easy.

As already mentioned, all commands in R are functions where arguments come (or do not come) in round brackets after the function name.

You can store your workflow in files, the so-called scripts. These scripts have typically file names with the extension, e.g., foo.R .

You can open an editor window to edit these files by clicking 'File' and 'New'. Try this. Under 'File' you also find the options 'Open file...', 'Save' and 'Save as'. Alternatively, just type CTRL+SHIFT+N.

You can run (send to the Console window) part of the code by selecting lines and pressing CTRL+ENTER or click 'Run' in the editor window. If you do not select anything, R will run the line your cursor is on.

You can always run the whole script with the console command source, so e.g. for the script in the file foo.R you type source('foo.R'). You can also click 'Run all' in the editor window or type CTRL+SHIFT+S to run the whole script at once.

Make a script called firstscript.R. Therefore, open the editor window with 'File' > 'New'. Type plot(rnorm(100)) in the script, save it as firstscript.R in the working directory. Then type source("firstscript.R") on the command line.

```
#source("firstscript.R")
```

Run your script again with source("firstscript.R"). The plot will change because new numbers are generated.

```
#source("firstscript.R")
```

Vectors were already introduced, but they can do more. Make a vector with numbers 1, 4, 6, 8, 10 and call it vec1.

```
vec1 <- c(1,4,6,8,10)
```

Elements in vectors can be addressed by standard [i] indexing. Select the 5th element of this vector by typing vec1[5].

```
vec1[5]
```

```
## [1] 10
```

Replace the 3rd element with a new number by typing vec1[3]=12.

```
vec1[3] <- 12
```

Ask R what the new version is of vec1.

```
vec1
```

```
## [1]  1  4 12  8 10
```

You can also see the numbers of vec1 in the environment window. Make a new vector vec2 using the seq() (sequence) function by typing seq(from=0, to=1, by=0.25) and check its values in the environment window.

```
vec2 <-seq(from=0, to=1, by=0.25)
```

Type sum(vec1).

```
sum(vec1)
```

```
## [1] 35
```

The function sum sums up the elements within a vector, leading to one number (a scalar). Now use + to add the two vectors.

```
vec1+vec2
```

```
## [1]  1.00  4.25 12.50  8.75 11.00
```

If you add two vectors of the same length, the first elements of both vectors are summed, and the second elements, etc., leading to a new vector of length 5 (just like in regular vector calculus).

Matrices are nothing more than 2-dimensional vectors. To define a matrix, use the function matrix. Make a matrix with matrix(data=c(9,2,3,4,5,6),ncol=3) and call it mat.

```
mat<-matrix(data=c(9,2,3,4,5,6),ncol=3)
```

The third type of data structure treated here is the data frame. Time series are often ordered in data frames. A data frame is a matrix with names above the columns. This is nice, because you can call and use one of the columns without knowing in which position it is. Make a data frame with t = data.frame(x = c(11,12,14), y = c(19,20,21), z = c(10,9,7)).

```
t <- data.frame(x = c(11,12,14), y = c(19,20,21), z = c(10,9,7))
```

Ask R what t is.

```
t
```

```
##    x  y  z
## 1 11 19 10
## 2 12 20  9
## 3 14 21  7
```

The data frame is called t and the columns have the names x, y and z. You can select one column by typing t$z. Try this.

```
t$z
```

```
## [1] 10  9  7
```

Another option is to type t[[“z”]]. Try this as well.

```
t[["z"]]
```

```
## [1] 10  9  7
```

Compute the mean of column z in data frame t.

```
mean(t$z)
```

```
## [1] 8.666667
```

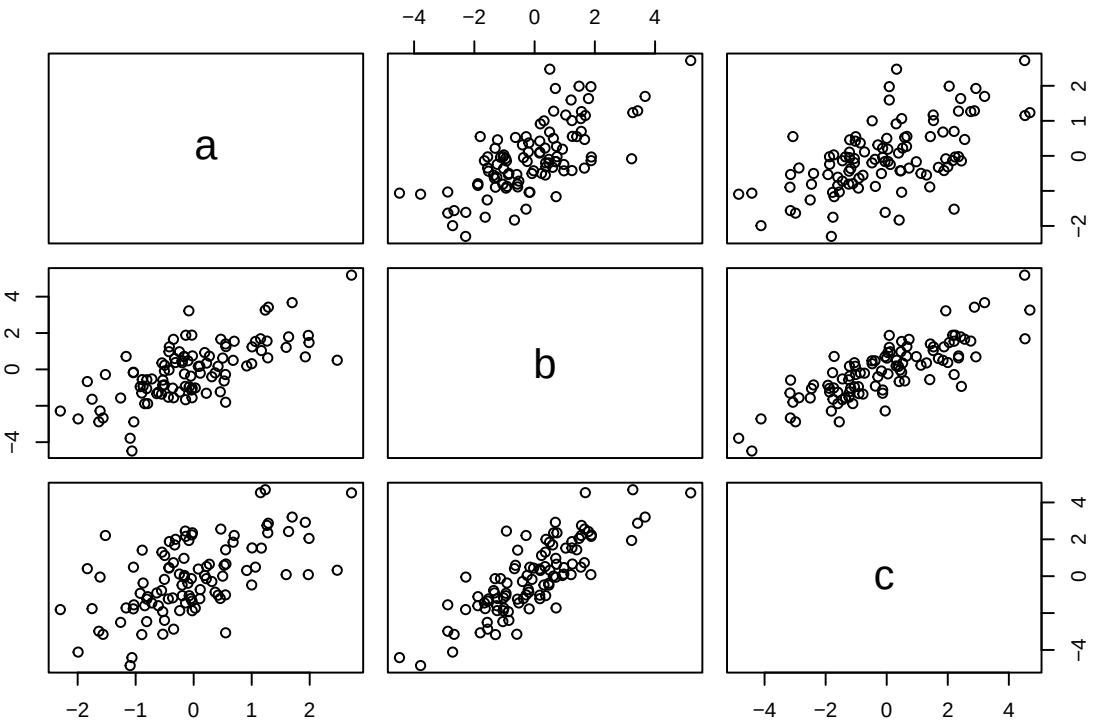
In the following question you will be asked to modify a script that will appear as soon as you move on from this question. When you have finished modifying the script, save your changes to the script and type submit() and the script will be evaluated. There will be some comments in the script that opens up. Be sure to read them!

Make a script file which constructs three random normal vectors of length 100. Call these vectors x1, x2 and x3. Make a data frame called t with three columns (called a, b and c) containing respectively x1, x1+x2 and x1+x2+x3. Call plot(t) for this data frame. Then, save it and type submit() on the command line.

```
# Text behind the #-sign is not evaluated as code by R.
# This is useful, because it allows you to add comments explaining what the script does.

# In this script, replace the ... with the appropriate commands.

x1 = rnorm(100)
x2 = rnorm(100)
x3 = rnorm(100)
t = data.frame(a=x1, b=x1+x2, c=x1+x2+x3)
plot(t)
```



Do you understand the results?

Another basic structure in R is a list. The main advantage of lists is that the `columns` (they are not really ordered in columns any more, but are more a collection of vectors) don't have to be of the same length, unlike matrices and data frames. Make this list L <- list(one=1, two=c(1,2), five=seq(0, 1, length=5)).

```
L <- list(one=1, two=c(1,2), five=seq(0, 1, length=5))
```

The list L has names and values. You can type L to see the contents.

```
L
```

```
## $one
## [1] 1
##
## $two
## [1] 1 2
##
## $five
## [1] 0.00 0.25 0.50 0.75 1.00
```

L also appeared in the environment window. To find out what's in the list, type names(L).

```
names(L)
```

```
## [1] "one"  "two"  "five"
```

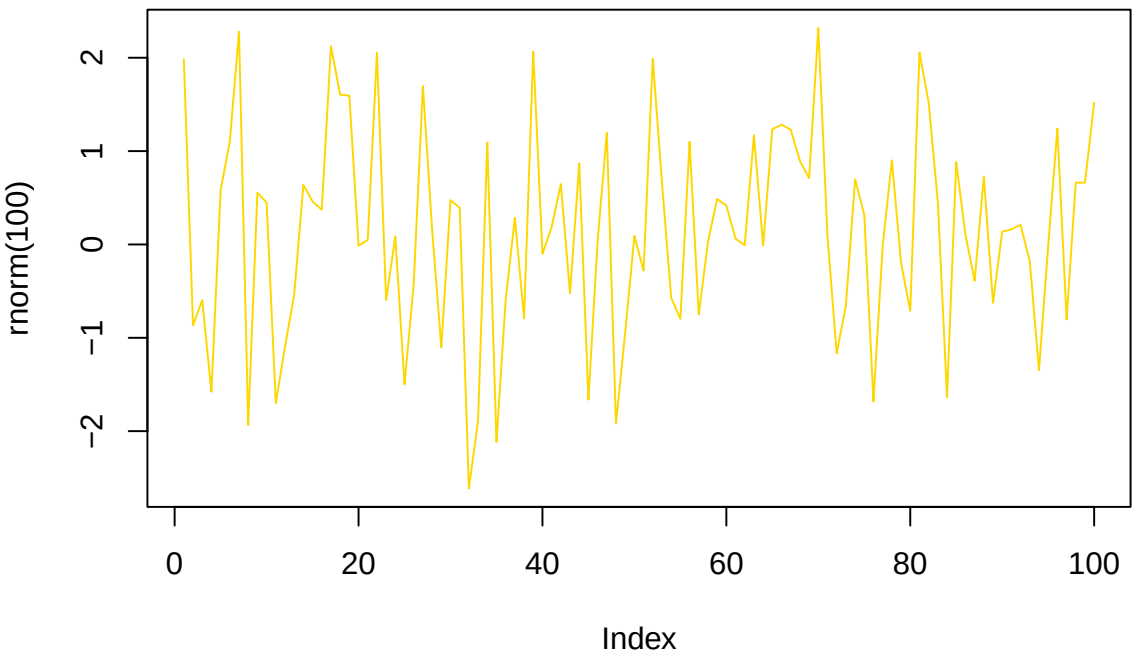
Add 10 to the column called five.

```
L$five + 10
```

```
## [1] 10.00 10.25 10.50 10.75 11.00
```

Plotting is an important statistical activity. So it should not come as a surprise that R has many plotting facilities. Type plot(rnorm(100), type="l", col="gold").

```
plot(rnorm(100), type="l", col="gold")
```



Hundred random numbers are plotted by connecting the points by lines in a gold color.

Another very simple example is the classical statistical histogram plot, generated by the simple command hist. Make a histogram of 100 random numbers.

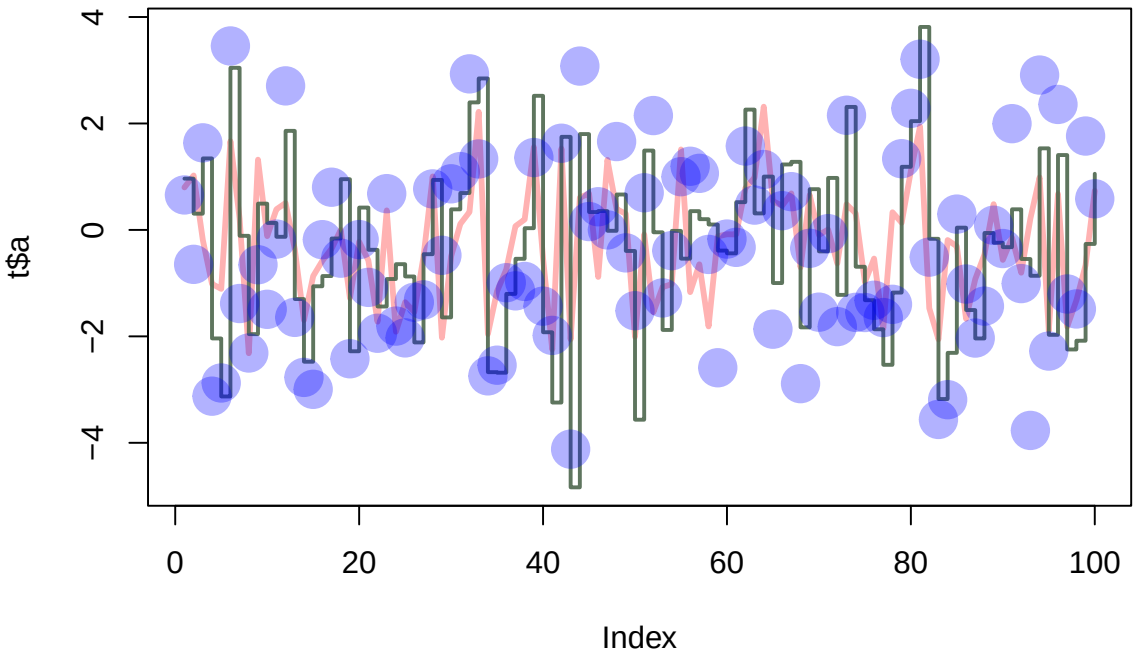
```
hist(rnorm(100))
```

The script that opens up is the same as the script you made before, but with more plotting commands. Type submit() on the command line to run it (you don't have to change anything yet).

```
# Text behind the #-sign is not evaluated as code by R.
# This is useful, because it allows you to add comments explaining what the script does.

# Make data frame
x1 = rnorm(100)
x2 = rnorm(100)
x3 = rnorm(100)
t = data.frame(a=x1, b=x1+x2, c=x1+x2+x3)

# Plot data frame
plot(t$a, type='l', ylim=range(t), lwd=3, col=rgb(1,0,0,0.3))
lines(t$b, type='s', lwd=2, col=rgb(0.3,0.4,0.3,0.9))
points(t$c, pch=20, cex=4, col=rgb(0,0,1,0.3))
```




```
# Note that with plot you get a new plot window while points and lines add to the previous plot.
```

Try to find out by experimenting what the meaning is of rgb, the last argument of rgb, lwd, pch, cex. Type play() on the command line to experiment. Modify lines 11, 12 and 13 of the script by putting your cursor there and pressing CTRL+ENTER. When you are finished, type nxt() and then ?par.

```
?par
```

You searched for par in the R help. This is a useful page to learn more about formatting plots. Google 'R color chart' for a pdf file with a wealth of color options.

To copy your plot to a document, go to the plots window, click the 'Export' button, choose the nicest width and height and click 'Copy' or 'Save'.

After having almost completed the second learning module, you are getting closer to become a nerd as you know...

...that everything in R is stored in objects (values, vectors, matrices, lists, or data frames),

...that you should always work in scripts and send code from scripts to the Console,

...that you can do it if you don't give up.

Please continue choosing another swril learning module.

A.3.3 Swirl learning modules *huber-data-1*, *huber-data-2*, and *huber-data-3*

> The following stuff can contain errors and I do not give warranty that the content matches to the learning modules. In particular, things like figures, R-scripts and output will not be shown here.

The learning modules are also available here: https://github.com/hubchev/swirl-it

huber-data-1

In this course, I'll be teaching you the basics of data analysis. It probably makes sense to start by defining the word DATA.

According to Wikipedia, "Data are values of qualitative or quantitative variables, belonging to a set of items."

Often the "set of items" that we are interested in studying is referred to as the POPULATION. Data analysis usually involves studying a subset, or SAMPLE, of an entire population.

Here is a diagram showing the relationship between a population and a sample.

Data analysis should always start with a specific question of interest. For example, we might ask "What percentage of people living in the United States are over six feet tall?"

Here, our population of interest is everyone living in the US. Since it's impractical to measure the heights of over 300 million people, we could instead choose 100 people at random and measure their heights. Our hope would be that this sample of 100 people is REPRESENTATIVE of the entire US population.

Lets quickly test your understanding of the term REPRESENTATIVE. If you were interested in studying the health of men living in the US, ages 18-25, which sample would be more representative of the target population– a sample of 50 men who live in a nearby retirement home, or a sample of 50 men who are students at a local university?

1. Men living at the retirement home
2. College students

College students

Would you like to watch a video on these topics now?

The purpose of analyzing a sample is to draw conclusions about the population from which the sample was selected. This is called INFERENCE and is the primary goal of INFERENTIAL STATISTICS.

In order to make any inferences about the population, we first need to describe the sample. This is the primary goal of DESCRIPTIVE STATISTICS.

If we want to describe our sample using just one number, how would we best do it? A good start is to find the center, the middle, or the most common element of our data. Statisticians call this the CENTRAL TENDENCY.

There are three different methods for finding such a number and the applicability of each method depends on the situation. Those three methods are called the MEAN, MEDIAN, and MODE.

Mean, median, and mode are all measures of _____.

1. variation
2. significance
3. deviation
4. central tendency

central tendency

Which of the following terms are of most importance when describing the central tendency of a data set?

1. median, mode, range
2. statistics, population, mode
3. population, sample, representative
4. mode, median, mean

mode, median, mean

To illustrate these concepts, we will now look at a real dataset from the 'openintro' R package, which has already been loaded for you. Type 'cars93' and press Enter to see the dataset we'll be working with.

```
cars93
```

```
## # A tibble: 54 × 6
##    type    price mpg_city drive_train passengers weight
##    <fct>   <dbl>    <int> <fct>            <int>  <int>
##  1 small    15.9       25 front                5   2705
##  2 midsize  33.9       18 front                5   3560
##  3 midsize  37.7       19 front                6   3405
##  4 midsize  30         22 rear                 4   3640
##  5 midsize  15.7       22 front                6   2880
##  6 large    20.8       19 front                6   3470
##  7 large    23.7       16 rear                 6   4105
##  8 midsize  26.3       19 front                5   3495
##  9 large    34.7       16 front                6   3620
## 10 midsize  40.1       16 front                5   3935
## # … with 44 more rows
```

You will notice the rows are numbered 1 through 54, each representing exactly one car in the dataset. For each car, the following VARIABLES, or characteristics, are reported: 'type' (small, midsize, large), 'price' (USD), 'mpg_city' (city miles per gallon), 'driveTrain' (4WD, front, rear), 'passengers' (total capacity), and 'weight' (lbs).

We will be focusing on the 'mpg_city' variable in this lesson. For simplicity, let's extract it from our dataset and store it in a new variable.

Access the 'mpg_city' variable from the 'cars93' dataset using the 'dataset$variable' notation.

```
cars93$mpg_city
```

```
##  [1] 25 18 19 22 22 19 16 19 16 16 21 17 20 20 29 23 21 29 20 31 23 21 18 46 42
29 22 20
## [29] 17 18 18 17 18 29 28 19 19 29 18 29 21 23 19 31 19 19 28 33 25 39 32 22 25
20
```

Now store the contents of the 'cars93$mpg_city' in a new variable called 'myMPG'.

```
myMPG <- cars93$mpg_city
```

The ARITHMETIC MEAN, or simply the MEAN or AVERAGE, is the most common measurement of central tendency. To calculate the mean of a dataset, you first sum all of the values and then divide that sum by the total number of values in the dataset.

However, when there are many values of interest, it becomes tedious to do this calculation by hand.

Luckily, R has a built-in function for computing the mean. The syntax for doing so is 'mean(variable)'.

Compute the mean value for the 'myMPG' variable now.

```
mean(myMPG)
```

```
## [1] 23.31481
```

Extreme values in our dataset can have a significant influence on the mean. For instance, if there was a car in our dataset that got 200 miles per gallon, this would inflate the mean upwards. This could be misleading since none of the other cars93 get anywhere near this gas mileage.

An alternative to the mean, which is not influenced at all by extreme values, is the MEDIAN. The median is computed by sorting all values from least to greatest and then selecting the middle value. If there is an even number of values, then there are actually 2 middle values. In this case, the MEDIAN is equal to the MEAN of the 2 middle values. Don't worry if this is a little confusing. It will become more clear with practice.

R also has a function for computing the median of a dataset and this is done by typing 'median(variable)'. Find the median value of your 'myMPG' variable now.

```
median(myMPG)
```

```
## [1] 21
```

Finally, we may be most interested in finding the value that shows up the most in our dataset. In other words, what is the most common value in our dataset? This is called the MODE and it is found by counting the number of times that each value appears in the dataset and selecting the most frequent value.

Use the 'table' function to see how many times each value appears for your 'myMPG' variable. The syntax for this function is the same as for the others you've seen.

```
table(myMPG)
```

```
## myMPG
## 16 17 18 19 20 21 22 23 25 28 29 31 32 33 39 42 46
##  3  3  6  8  5  4  4  3  3  2  6  2  1  1  1  1  1
```

Look at your table for the 'myMPG' variable that you created above. The first row gives you the value of your variable and the second row gives you the number of times it appears in your dataset. Since the mode is the value of our variable that appears most frequently, what is the mode of your 'myMPG' variable?

Congratulations! You've made it through your first lesson. We introduced basic concepts related to data and data analysis. Specifically, you learned three important measures of central tendency: mean, median, and mode. You also know how to compute these using R.

huber-data-2

Now that you have learned the basics techniques and statistical calculations used to describe a data set, the next step is figuring out how to effectively illustrate and visualize your data.

It is useful to first visualize the data before a statistician engages in a thorough analysis of the data set. In this lesson, we are going to learn useful techniques for visualizing numerical variables.

By organizing the data into a PLOT or GRAPH, a statistician is able to explore and summarize some basic properties of the data set. The discipline of quantitatively describing the main properties of a data set is known as DESCRIPTIVE STATISTICS.

The simplest type of plot is the DOT PLOT, which is used to visually convey the values of one variable. In a dot plot, there is only a horizontal x-axis, and the data points are represented as dots above this axis.

Here is a dot plot created using the variable 'price' from our 'cars93' data set (which is part of the openintro package). As you may notice, the price is reported along the x-axis in $1000s, and each point above the axis represents the price of one of the 54 cars in our data set.

When looking at this dot plot, around what price (in $1000s) does there appear to be the highest density of data points?

Since dot plots effectively display the specific numerical value of one variable for each individual in the data set, they are particularly useful when analyzing smaller data sets.

A HISTOGRAM is similar to a dot plot, but instead of showing every specific value, it partitions the values of your data into several bins, providing a more condensed representation of the data.

Here I have created a histogram using the miles per gallon data for all of our cars. As you may notice, the values of the MPG along the x-axis are partitioned into bins with a range of 5. The second bin, for example, groups together all of the cars that get 21-25 MPG in the city, and so forth. Note that the bin to the left of this contains those cars with 20 MPG since this value cannot be counted in both bins. The frequency of values in each bin, or the number of cars in each of the intervals, is reported along the y-axis.

Taller bars signify the range of values in which the majority of the data is located, whereas shorter bars represent a range of values in which only a little bit of the data is located. In other words, histograms provide a view of the DATA DENSITY.

By simply looking at this histogram, can you tell me which MPG interval has the highest frequency of values? For example, the lowest frequencies of values occur in the intervals 36-40, 41-45, and 46-50.

1. 16-20
2. 21-25
3. 26-30
4. 31-35
5. 36-40
6. 41-45
7. 46-50

16-20

How many cars get 16-20 MPG in the city?

A red line has been drawn on our histogram illustrating the previous answer.

Histograms are particularly useful in viewing and describing the shape of the distribution of the data. A distribution of data may have a left skew, a right skew, or no skew at all. SKEWNESS is a measure of the extent to which the distribution of the data 'leans' to one side or the other.

A distribution that has a left skew is one in which the left TAIL of the plot is longer. In other words, on a histogram the majority of the distribution is located to the right of the mean.

When a distribution is left-skewed, the value of the MEAN is less than that of the MEDIAN, and thus the MEAN is located further to the left of the distribution. In this plot, the green line represents the median and the blue line represents the mean.

On the other hand, a distribution that has a right skew is one in which the right tail is longer, such that the majority of the data falls to the left of the mean, when viewed on the histogram.

When a distribution is right-skewed, the value of the MEAN is greater than that of the MEDIAN, and thus the MEAN is located further to the right of the distribution. In this plot, the green line represents the median and the blue line represents the mean.

A plot that has no skew is one in which the tails on both sides of the mean balance out, and is referred to as symmetric. When a distribution is symmetric, the MEAN and MEDIAN are approximately equal in value.

In this plot, the green line represents the median and the blue line represents the mean. The green line is the only one visible since the mean and median are close to the same value.

Now, let us take a look back at the histogram we made earlier, which represents the distribution of the values for city MPG for each of the 54 cars from our 'cars93' data set.

How would you classify the shape of the distribution represented by this histogram?

1. Symmetric
2. Right-skewed
3. Left-skewed

Right-skewed

Referring to the histogram above, and keeping in mind the real shape of the distribution, would you expect the MEDIAN to be greater than, less than, or equal to the MEAN?

1. Greater than
2. Less than
3. Equal to

Less than

A special type of histogram is known as a STEM-AND-LEAF PLOT. This plot organizes numerical data in order of decimal place value. The left-hand column of the plot contains the STEMS, or the numerical values of the tens digit for each of the data points, organized vertically in increasing order. The LEAVES are located in the right-hand column of the plot and are the values of the ones digit for each data point of the corresponding stem, organized horizontally in increasing order.

In a stem-and-leaf plot, the number of leaves is equal to the number of items in the data set. The easiest way to understand a stem-and-leaf plot is to see one!

I have created a stem-and-leaf plot above using the same values for the 'mpgCity' variable as we just used for our histogram. As you can see, a stem-and-leaf plot is a useful type of histogram if you want to see the frequencies of specific values of the data. Often, there will only be one bin per tens digit, but in this case, R gives us the same bins as we saw in our histogram.

Demonstrated on this stem-and-leaf plot, how many occurences of the value '22' are there in this particular data set?

The final plot that can be used for discrete or continuous variables is known as the BOX PLOT, also called a BOX-AND-WHISKER PLOT. As you previously learned, this plot is used to summarize the main descriptive statistics of a particular data set and help illustrates the concept of variability. I have created a box-and-whisker plot so that you can be reminded of what it looks like.

A box plot is used to visually represent the MINIMUM, FIRST QUARTILE (Q1), MEDIAN, THIRD QUARTILE (Q3), and MAXIMUM of a data set. The R-command 'summary(cars93$price)' returns values for these main descriptive statistics. Try this now.

```
summary(cars93$price)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    7.40   10.95   17.25   19.99   26.25   61.90
```

huber-data-3

Welcome to Lesson 3! In this lesson, we will learn what DISPERSION is and what statistical values are needed in order to best describe the spread of data. Further, you will learn all about a box-and-whisker plot which is a plot commonly used by statisticians when determining variability.

While measures of central tendency are used to estimate the middle values of a dataset, measures of dispersion are important for describing the spread of the data.

The term dispersion refers to degree to which the data values are scattered around an average value. Dispersion is synonymous with other words such as variability and spread.

Why is it important to analyze the spread of a particular set of data? Two different samples may have the same mean or median, but different levels of variability or vice versa.

Therefore, it is important to describe both the _____ and _____ of a data set.

1. median, variability
2. central tendency, dispersion
3. middle, mean
4. spread, variability

central tendency, dispersion

In this lesson, we will discuss the three statistical values most commonly used to describe the dispersion or variability of a data set. Variability is a fancy term used to classify how variable or spread out the data is.

The first descriptive statistic that can describe the variability of a data set is known as the RANGE. The range is the difference between the maximum and minimum values of the data set.

To demonstrate how you can use R to determine the range of a data set we will refer back to the cars93 data set from the previous lesson.

Type in the R-command 'range(cars93$price)' to determine the exact values for the minimum and maximum prices of cars in the data set.

```
range(cars93$price)
```

```
## [1]  7.4 61.9
```

The second important measure of variability is known as VARIANCE. Mathematically, VARIANCE is the average of the squared differences from the mean. More simply, variance represents the total distance of the data from the mean.

In R, you can use the command 'var(data)' to easily calculate the variance of a particular set of data. Try calculating the variance for the data 'cars93$price'.

```
var(cars93$price)
```

```
## [1] 132.3984
```

The values for variance and standard deviation are very closely related. The standard deviation can be calculated by taking the square root of the variance where as the variance can be calculated by

squaring the standard deviation.

To statisticians, the standard deviation is a more conventional measure of variability because it is expressed in the same units as the original data values.

Similar to variance, you can use the R-command 'sd(data)' to calculate the standard deviation of a particular set of data. Try calculating the standard deviation for the data 'cars93$price'.

```
sd(cars93$price)
```

```
## [1] 11.50645
```

The standard deviation is very important when analyzing our data set. A small standard deviation indicates that the data points tend to be located near the mean value, while a large standard deviation indicates that the data points are spread further from the mean.

Three important measures of variability are which of the following:

1. mean, median, range
2. spread, mean, central tendency
3. variance, dispersion, spread
4. range, variance, standard deviation

range, variance, standard deviation

A BOX PLOT, also called a BOX-AND-WHISKER PLOT, is used to summarize the main descriptive statistics of a particular data set and this type of plot helps illustrate the concept of variability. A box plot is used to visually represent the MINIMUM, FIRST QUARTILE (Q1), MEDIAN, THIRD QUARTILE (Q3), and MAXIMUM of a data set.

Here I have created a box plot to represent the price data for each of the three car types: large, midsize, and small. You'll notice that each of the 3 figures is composed of a closed 4-sided "box" with "whiskers" on the top and bottom, hence the name box-and-whisker plot.

The height of each box is referred to as the INTERQUARTILE RANGE (IQR). The more variability within the data, the larger the IQR. On the other hand, less variability within the data means a smaller IQR. The bottom of the box in the box plot corresponds to the value of the first quartile (Q1), and the top of the box corresponds to the value of the third quartile (Q3). To calculate the value of the IQR, simply subtract the value of Q1 from that of Q3.

The whiskers, or lines, that extend above and below each box represent roughly the upper 25% and lower 25% of data points, respectively. The only exception is when there are outliers, which I'll explain shortly.

Let's take a closer look at how quartiles are calculated. We start by sorting the data from least to greatest, just like when calculating the median. The first quartile (Q1), also known as the 25th PERCENTILE (since 25% of the data points fall at or below this value), is simply the median of the first half of the sorted data. Likewise, the third quartile (Q3), also known as the 75th percentile, is the median of the second half of the sorted data.

As shown in this plot, the blue horizontal line illustrates how to find the value for the first quartile. The green horizontal line illustrates how to find the value for the third quartile. The interquartile range is the range of data values that is contained in between these two lines.

Look again at our box plot of price vs. car type. You may be thinking to yourself, 'What is that circle above the box plots for the midsize cars, and why is there no circle above the box plot for the large

cars?' Those circles represent OUTLIERS in the data set.

An OUTLIER is an observation that is unusual or extreme relative to the other values in the data set. Outliers are useful in identifying a heavy skew in a distribution, and may signify a data collection or data entry error to a scientist. There are many different conventions for identifying outliers within a data set.

When looking at the box plot, which car types appear to have outliers?

1. small, midsize
2. midsize, large
3. small, large
4. small, midsize, large

small, midsize

As you can see in the box plot, the data for prices of 'midsize' cars vary from around 15 to 62, encompassing a range of approximately 50. This is a great deal larger than the variation for 'small' cars which range from around 5 to 15, encompassing a range of approximately 10. Therefore, since the range is much greater for 'midsize' cars, prices of 'midsize' cars have much higher variability in comparison to the prices of 'small' cars.

Now it is your turn! Is the variability of prices of cars of type 'large' higher or lower than that of cars of type 'small'?

1. higher
2. lower
3. the same

higher

You have officially mastered the concept of dispersion and have fully learned how to read and interpret a box-and-whisker plot. These are two very valuable tools used everyday in descriptive statistics. Congratulations!