Basic R

Kálmán Abari

Last updated: 2021-03-31

Contents

Preface

As a researcher we need to know how to work with data. One of the best ways to do that is with R. R is a free and an open source language that was specifically developed for reading, manipulating, analysing data and publishing results. In this book, we'll take a look at how we can get started with R. This is an introductory book, so you don't need to have experience with R or with computer programming.

In order to start work with R, you need to install the *Base R* and *RStudio*.

Setup Instructions

The first step to working with R is to actually get *Basic R* on your computer. This is easy and it's free. The most common way by far to work with R is within a desktop application called *RStudio*. Like *Basic R*, this is free and it's open source and available for multiple platforms. *Basic R* is the underlying statistical computing environment, but using R alone is no fun. *RStudio* is a graphical integrated development environment (IDE) that makes using R much easier and more interactive. You need to install *Basic R* before you install *RStudio*.

Windows

- Download R from the CRAN website.
- Run the `.exe` file that was just downloaded
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Windows XP/Vista/7/8** (where x, y, and z represent version numbers)
- Double click the file to install it
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

macOS

- Download R from the CRAN website.

- Select the `.pkg` file for the latest R version
- Double click on the downloaded file to install R
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Mac OS X 10.6+ (64-bit)** (where x, y, and z represent version numbers)
- Double click the file to install RStudio
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

Linux

- Follow the instructions for your distribution from CRAN, they provide information to get the most recent version of R for common distributions. For most distributions, you could use your package manager (e.g., for Debian/Ubuntu run `sudo apt-get install r-base`, and for Fedora `sudo yum install R`), but we don't recommend this approach as the versions provided by this are usually out of date. In any case, make sure you have at least R 4.0.0.
- Go to the RStudio download page
- Under *Installers* select the version that matches your distribution, and install it with your preferred method (e.g., with Debian/Ubuntu `sudo dpkg -i   rstudio-x.yy.zzz-amd64.deb` at the terminal).
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

Chapter 1

How to use R

There are so many ways to analyse data in R. In my opinion the best way is the RStudio. Most R users use R via *RStudio*. We need both *Base R* and *RStudio*, as we did earlier, but if we only start the RStudio, so we can reach all functions of R. *RStudio* is an *integrated development environment (IDE)* that provides an interface by adding many convenient features and tools.

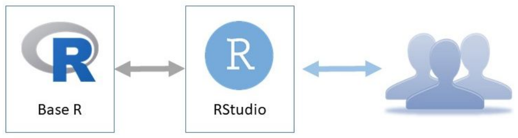


Figure 1.1: How to use R – the best way

Of course, we can use the *Base R* directly. Usually, this is the only option that is supported by mainframe environment. But, if you have the opportunity, always use *RStudio* instead of Basic R directly.

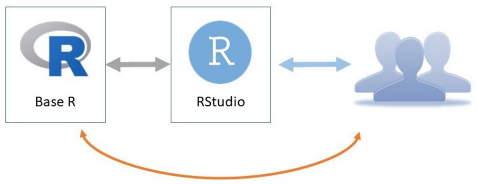
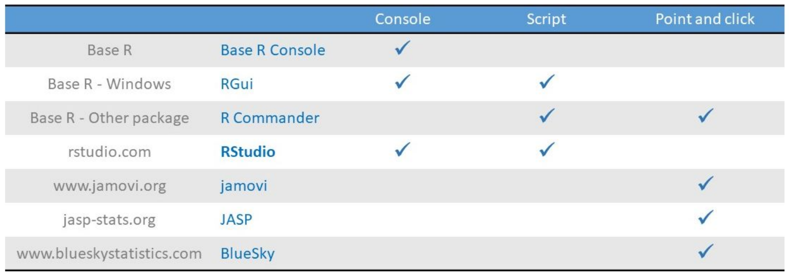


Figure 1.2: How to use R – the minimum way

Actually, there are several ways to use R, not only *Base R* and *RStudio*. The table below summarizes the interfaces in the columns and the tools in the rows.

There are three different types of interface: *Console*, *Script* and *Point and click*. Interfaces allow the user to interact with R.



		Console	Script	Point and click
Base R	Base R Console	✓		
Base R - Windows	RGui	✓	✓	
Base R - Other package	R Commander		✓	✓
rstudio.com	**RStudio**	✓	✓	
www.jamovi.org	jamovi			✓
jasp-stats.org	JASP			✓
www.blueskystatistics.com	BlueSky			✓

Figure 1.3: How to use R – the minimum way

Console provides a command-line *interface* that allows the user to interact with the computer by typing commands . The computer displays a prompt (>), the user types a command and presses Enter or Return, and gets the result. There are three tools, that provide console, the *Base R* Console (it's the only option in mainframe environment), *RGui* in *Base R* on Windows, and *RStudio*.

The second interface is the **script interface**. It gives you an editor window. You can type multiple lines of code into the source editor without having each line evaluated by R. Then, when you're ready, you can send the instructions to R - in other words, source the script -, and you get the result. You can reach this functionality in *RGui*, *R Commander* and *RStudio*. **Remember, the best way to use R is creating, editing and running scripts in *RStudio*. This is the best option.**

For beginners, the best option would be to use a **Point-and-click interface**. It has a menu, you can choose menupoints, menuitems, you can get dialog boxes and type in editfields, point on radio buttons and checkboxes. **But the knowledge of these systems have limits**. You can execute only methods, which you can reach from the menu. The descriptive measures, tables, plots and hypothesis tests, which you can point and click, are part of knowledge of R. The whole knowledge can be reached only from console or script interface. For example I only use *jamovi*, if I have a simple question and I want to get a quick answer. So I encourage you to install and try jamovi or JASP. These are free and user friendly ways to do statistics. By the way, the tools I listed in this table are all free, except the BlueSky. It is worth installing and trying them.

1.1 Base R

What components were installed with *Base R*? The *Base R* consists of three elements. The console for typing commands and getting results, the interpreter for evaluating the commands, and packages for extending R's knowledge. The interpreter is the heart of the R, all commands will be executed by the interpreter. For the users, for us, the console is the key. Apart from point and click interfaces, we will interact with the console directly or indirectly.

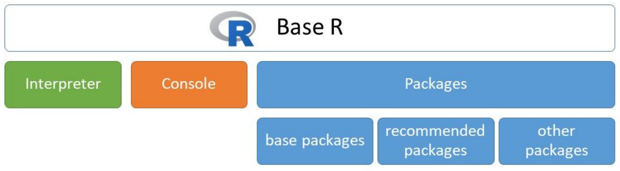
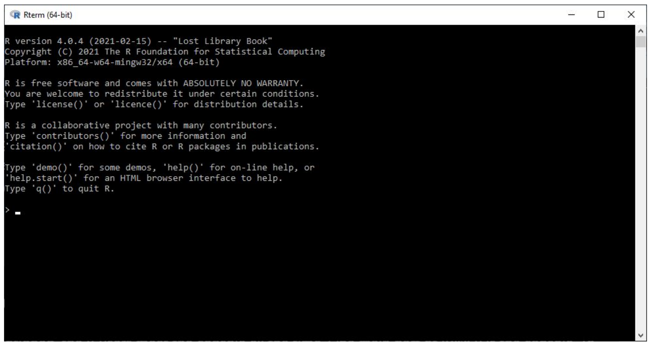


Figure 1.4: Components of Base R

1.1.1 Console of Base R

As we mentioned, the R users meet the console all the time. One main part of *Base R* is the console. To start the console, we should type R (the capital R letter) on all systems, or we can find and click on the R icon. If you are on Windows, you can launch the *R.exe* (e.g. `c:\Program Files\R\R-4.0.4\bin\x64\R.exe`).



```
R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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.

> 
```

Figure 1.5: Console in Base R

On the screen above, you can see some information about the R instance. At the bottom of the console window there is a prompt. It consists of a 'greater than' sign (symbol) and a space, and of course a cursor where you can type any character.

Let's type any character, delete characters with Delete and Backspace keys, move the cursor with Left arrow and Right arrow keys, insert any character in this position, and navigate the cursor the beginning of the line and the end of the line with the Home and End keys.

If we are ready, we can execute this line, the command, hitting Enter.

If the command is valid, R or more precisely its interpreter, will execute it, then it returns the result in the console. If the command in not valid, the interpreter returns an error message.

Let's start with numbers. Type 45 and hit Enter.

```
45
#> [1] 45
```

This is a valid command because there is no error message. But the result, the output, is meaningless.

Let's choose a more complicated expression:

```
45 + 5
#> [1] 50
```

Forty-five plus five sums fifty, so fifty is displayed in the output. The 1 in brackets at the beginning of the output means this is the first line of the output.

1.1.1.1 Console features

Every console has three features, that help us to execute commands.

History of commands We can use the Up arrow and Down arrow keys to browse the history of commands, which we typed earlier. When you press the Up arrow, you get the commands you typed earlier at the command line. Of course you can modify them as well. You can hit Enter at any time to run the command that is currently displayed.

Autocompletion Pressing TAB key completes the keyword or directory path we were currently typing. Type in `getw` hit TAB and you can see the whole function call, hitting Enter, you can get the working directory.

Continuation prompt Let's have a look at a small example. Tpye `45 -`(forty-five minus) and hit Enter. This is an invalid command, but we can not see any Error message. Instead, a new prompt has appeared, a continuation prompt indicated by a `+` (plus) followed by a space character. We can continue typing. The console allows us to complete the command. It's easy, type for example `5` (five), hit Enter. We can get the result. I'll show you another example. Type `getwd(` without closing parenthesis, hit Enter. We will get the continuation prompt, and typing closing parenthesis we get the working directory. It seems to help us. Continuation prompt seems to be a good thing. But, It is not. It is a really confusing feature. We can easily find ourselves in a never ending story. We can type `45 -`, Enter, `11 *`, Enter and so on, we keep getting the continuation prompt, and we don't really know how to complete it in a right way. So, It is very important to leave the continuation prompt as soon as possible. The key is Esc button. Lets' try this. Type in opening parenthesis and 6 (`(6`), hit Enter, and press Esc. We can get back the prompt 'greater than' (`>`), this is default prompt. When you see the plus prompt, continuation prompt, you must press the Esc key.

1.1.1.2 Working directory

In R, we answer the questions we face using functions. So, the expressions that we type into the command line, usually contains *function calls*. So, now, we can request the working directory. Let's type `getwd()` to get the interpreter to display our working directory.

```
getwd()
```

Working directory is the default directory that our command line reaches to access files if we don't specify a path ourselves. We can specify paths two ways either absolutely from the root directory or relatively starting from our working directory. Beside reaching our history in the command line we can also rely on the help of a built-in autocompletion feature, by pressing TAB key, which completes the keyword or directory path we were currently typing.

For example, let's type only `set` and press TAB and TAB again to list out all the commands that start with `set`. Press `w` and press TAB again, and as you can see the command line completes our command with a `d` to get an existing function name. All function call requires parentheses after the function name, which contains additional data for the function which we call arguments.

The `setwd()` function has only one required arguments, which is a path to a directory, which we want to set as our new working directory.

Let's try calling the `setwd()` function, start with the function name, then the opening parenthesis and inside quote marks we give the directory's path.

On Windows, after the first quote mark, type `c:/`, which refers the drive you want to use, and press TAB twice to see all subdirectories and files on the drive.

From here we can build our path directory by directory till we reach the directory which we want to set as our working directory. As you can see when jumping from directory into another we mark this jump with the slash character. Instead of writing out the path ourselves we can write the first few characters of directory's name and we can rely on TAB to autocomplet it for us. Of course with more common directory names we have to be more specific to get the desired autocompletions.

Finally, if we reached our desired new working directory, we can execute the command, by pressing enter, but make sure you have both your opening and closing quote marks and parenthesis. If you had, your command executed successfully thereby changing your working directory, but if you made a mistake either in the formality of the command (called syntactic error) or by giving a path to a directory which doesn't exist (called semantic error).

To sum up, to set a working directory in R type:

```
setwd("Path/To/Your/Workingdirectory")
```

If you need to check which working directory R thinks it is in:

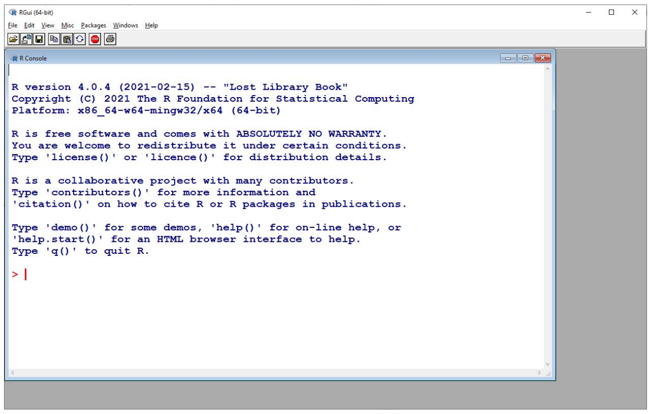
```
getwd()
```

1.1.1.3 Quit the console

In the end, let's quit the console, by typing and executing the `q()` command. Don't forget the parentheses. We don't need to save the workspace. Choose `No`.

1.1.2 RGui on Windows

On Windows operating system, *Base R* has another console which is more advanced. The *RGui* has a graphical user interface. To start it, find and click on the R icon. You should always use the latest version and the 64 bit version.



```
R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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.

> |
```

Figure 1.6: Console in RGui

Let's start up the aforementioned 64 bit version of it. Above you can see our console which in functionality is the same as the one we used in *Base R* recently. We can type any character and press Enter.

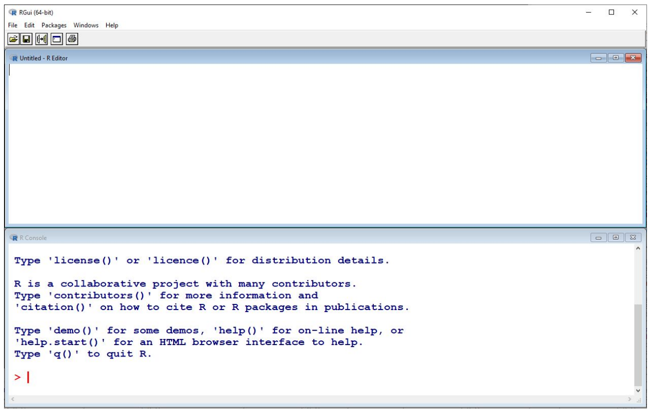
If you'd like to change the appearance, or the size of the console, you can do so in the `Edit > GUI preferences` menu in the upper menu bar. Let's choose this menu item, and increase the font size to 28 and set the style to bold. Close this dialog box with `OK` button, and you can see a more readable console window. But as you can see we also have menu and tool bar.

Let's try the same basic arithmetic command here. Type `45 + 5` and press Enter to execute it. And as we can see we get the same result here.

Let's try the history with the Up arrow and Down arrow. Navigate the cursor with Left arrow and Right arrow keys, use the Home and End buttons, insert or delete any character and press Enter.

1.1.2.1 Scripting in RGui

RGui has all the functions the Console of *Base R* had, and also a new one. We can create script files with which we can use to store commands in text files. Script files makes easy to store and organize commands. So let's click `File > New script` which will make us a new script window where you can edit your script file. We can arrange the Console and Script windows, click on the `Windows > Tile horizontally.` You can find the typical face of *RGui* below.



```
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.

> |
```

Figure 1.7: Console in RGui

Let's write the two commands in to this script file: `45 + 5` and `getwd()`.

Here we only type out command, to actually execute them we will need to transfer them to the console. This window is only a text editor through which we edit our script file. Here we can only use basic notepad like functionalities, so no autocompletion or history.

We can move in a line with the Home and End buttons and through lines with the Page Up and Page Down buttons. With Ctrl+Home we can get to the beginning of the script file and with Ctrl+End we can get to the bottom of it. Of course, this comes handy with much larger script files.

We can mark parts of the text with either holding the Shift key and using the Left-Right-Up-Down arrow keys or using the mouse. And we can use the clipboard as well, Ctrl+C, Ctrl+X and Ctrl+V.

It's important to know how to actually execute the commands we just wrote into our script file. With Ctrl+R we can execute the line that our cursor is currently at. The process consists of the command getting pulled into the console and then it executing it.

Let's try it. Move the cursor into the first line. Click in the first line anywhere.

Then press the Ctrl+R. Three things happened at he same time. The first line was pulled into the console, the line was executed, and the cursor jumped down a line. We can repress the Ctrl+R and repeat the whole process for the second line. And so on.

If you have any text selected in your editor prior, pressing Ctrl+R, then the selected text will be executed. Let's also try this. Select only `5 + 5` from the first line, and press Ctrl+R, and you get 10. Then, select the first two lines and execute them with Ctrl+R. The interpreter ran both lines. You can see the result in the console.

As you might have noticed, we have a colored console, the inputs, the commands colored by red, and the outputs, the results colored by blue.

This script files can also contain comments, which is useful to mark what's the command's intention. Which ones again may seem unimportant now, but is really useful when working with massive script files.

To mark something as a comment use the hash mark (`#`) which marks everything in a line after the hash mark as a comment. It's good practice to start your script file with 3 comment lines which contains the author of the file, the date and a name which gives some kind of information about what the script does.

```
# Kálmán Abari
# 2021-03-17
# First script file
```

Navigate the cursor to the first position of the file, for example pressing the Ctrl+Home. Then type a hash mark, and your name, Enter. Hash mark, date of today, Enter, hash mark `First script file`, Enter.

If we are ready, we are going to save the script file. It's important to save the script file with the `File > Save` menu. It's good practice to save your work every 15 minutes. It's important that when we save our files we give them names that doesn't contain any special characters or whitespaces, underscores are acceptable though. Choose a proper directory and type in `first_script.R` as the filename. Make sure that our file's extension should be `.R` which means that it contains an R script file.

With that we covered the basics of the *RGui*, so we can close it for now, we shouldn't worry about saving our workspace since we won't be needing it.

1.2 RStudio

The last tool we will get to know now and will be using for the rest of the book is *RStudio*. We can also launch it from the start menu. While we had multiple *Basic R* instances, we only have one *RStudio*, so it should be easy to find it.

It's the most advanced interface to use R from the aforementioned ones. And this will be the one that we will mainly use through out the course. Even though we will only use *RStudio*, it's important to mention that RStudio relies on *Base R* to work.

1.2.1 Customization

We can easily check which instance of *Base R* our *RStudio* is using. We can see this in the `Global options > Tools` menu option.

Let's check if it really is using the 64 bit version.

Here we can also do other customizations. While we're here we should also uncheck the `Restore .Rdata` option and set the `Save workspace to Rdata on exit` option to `Never`.

Another important one is the `Code` menu point from the left list. Here under the `Saving` option we have to set the `Default text encoding` to `UTF-8` which is a wildly used and accepted character-code standard.

You can customize to look of your editor under the `Appearance` option. Here you can change the theme of your editor, which is sets the color palette it uses. I recommend changing it to `Tomorrow night bright`. Let's close the settings, by pressing `OK`, to save the changes we made.

1.2.2 Using RStudio

A few words about RStudio. The main area consists of 3 or 4 different panes or windows which all are responsible for a different task. You have 3 panes by basic. The fourth panes you can add is a script file editor, which you can do by creating a new script file in `File > New file >-- R Script`.

You can easily resize the panes with clicking and dragging the vertical or horizontal line between the panes.

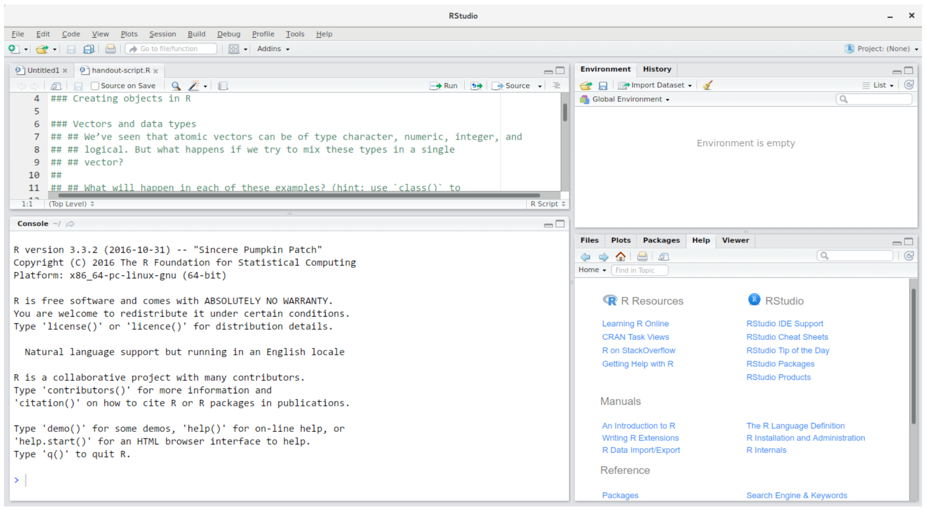
RStudio is divided into 4 "Panes":

- the **Source** for your scripts and documents (top-left, in the default layout),
- the R **Console** (bottom-left),
- your **Environment/History** (top-right), and
- your **Files/Plots/Packages/Help/Viewer** (bottom-right).

The placement of these panes and their content can be customized (see main Menu `Tools > Global Options > Pane Layout`. One of the advantages of using `RStudio` is that all the information you need to write code is available in a single window.

1.2.3 How to start an R project

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder. When working with R and RStudio you typically want that



```
### Creating objects in R

### Vectors and data types
## ## We’ve seen that atomic vectors can be of type character, numeric, integer, and
## ## logical. But what happens if we try to mix these types in a single
## ## vector?
##
## ## What will happen in each of these examples? (hint: use `class()` to
```

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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.

  Natural language support but running in an English locale

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.

>
```

Figure 1.8: The RStudio Interface

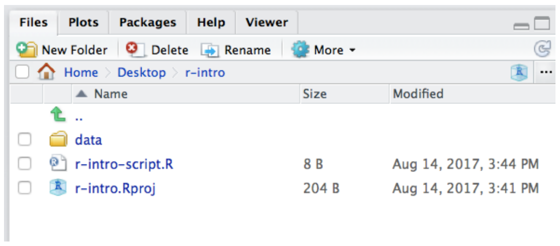
single top folder to be the folder you are working in. In order to tell R this, you will want to set that folder as your **working directory**. Whenever you refer to other scripts or data or directories contained within the working directory you can then use *relative paths* to files that indicate where inside the project a file is located. (That is opposed to absolute paths, which point to where a file is on a specific computer). Having everything contained in a single directory makes it a lot easier to move your project around on your computer and share it with others without worrying about whether or not the underlying scripts will still work.

Whenever you create a project with *RStudio* it creates a working directory for you and remembers its location (allowing you to quickly navigate to it) and optionally preserves custom settings and open files to make it easier to resume work after a break. Below, we will go through the steps for creating an "R Project" for this workshop.

- Start RStudio
- Under the `File` menu, click on `New project`, choose `New directory`, then `Empty project`
- As directory (or folder) name enter `r-intro` and create project as subdirectory of your desktop folder: `~/Desktop`
- Click on `Create project`
- Under the `Files` tab on the right of the screen, click on `New Folder` and create a folder named `data` within your newly created working directory (e.g., `~/r-intro/data`)
- On the main menu go to `Files` > `New File` > `R Script` (or use the shortcut Ctrl+Shift+N) to open a new file

- Save the empty script as `r-intro-script.R` in your working directory.

Your working directory should now look like in Figure 1.9.



Name	Size	Modified
..		
data		
r-intro-script.R	8 B	Aug 14, 2017, 3:44 PM
r-intro.Rproj	204 B	Aug 14, 2017, 3:41 PM

Figure 1.9: What it should look like at the beginning of this lesson

1.2.4 Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized, and will also make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you may create directories (folders) for **data**, **documents**, and **outputs**.

- **`data/`** Use this folder to store your raw input data.
- **`documents/`** If you are working on a paper this would be a place to keep outlines, drafts, and other text.
- **`output/`** Use this folder to store your intermediate or final datasets and images you may create for the need of a particular analysis. For the sake of transparency, you should *always* keep a copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically, You could have subfolders in your `output` directory named `output/data` that would contain the respective processed files. I also like to save my images in `output/image` directory.

You may want additional directories or subdirectories depending on your project needs, but this is a good template to form the backbone of your working directory.

1.2.5 RStudio Console and Command Prompt

The console pane in RStudio is the place where commands written in the R language can be typed and executed immediately by the computer. It is also where the results will be shown for commands that have been executed. You can type commands directly into the console and press Enter to execute those commands, but they will be forgotten when you close the session.

If R is ready to accept commands, the R console by default shows a `>` prompt. If it receives a command (by typing, copy-pasting or sent from the script editor using Ctrl Enter), R will try to execute it, and when ready, will show the results and come back with a new `>` prompt to wait for new commands.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation, i.e. you don't have the same number of left-parentheses as right-parentheses, or the same number of opening and closing quotation marks. When this happens, and you thought you finished typing your command, click inside the console window and press Esc; this will cancel the incomplete command and return you to the `>` prompt.

1.2.6 RStudio Script Editor

Because we want to keep our code and workflow, it is better to type the commands we want in the script editor, and save the script. This way, there is a complete record of what we did, and anyone (including our future selves!) can easily replicate the results on their computer.

Perhaps one of the most important aspects of making your code comprehensible for others and your future self is adding comments about why you did something. You can write comments directly in your script, and tell R not no execute those words simply by putting a hashtag (`#`) before you start typing the comment.

```
# this is a comment on its on line
getwd() # comments can also go here
```

One of the first things you will notice is in the R script editor that your code is colored (syntax coloring) which enhances readability.

Secondly, RStudio allows you to execute commands directly from the script editor by using the Ctrl + Enter shortcut (on Macs, Cmd + Enter will work, too). The command on the current line in the script (indicated by the cursor) or all of the commands in the currently selected text will be sent to the console and executed when you press Ctrl + Enter. You can find other keyboard shortcuts under `Tools` > `Keyboard Shortcuts Help` (or `Alt` + `Shift` + `K`)

At some point in your analysis you may want to check the content of a variable or the structure of an object, without necessarily keeping a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the Ctrl + 1 and `Ctrl` + `2` shortcuts allow you to jump between the script and the console panes.

All in all, RStudio is designed to make your coding easier and less error-prone.

1.3 RMarkdown

1.3.1 Introduction

RMarkdown allows you write reports that include both R codes and the output generated. Moreover, these reports are dynamic in the sense that changing the data and reprocessing the file will result in a new report with updated output. RMarkdown also lets you include Latex math, hyperlinks and images. These dynamic reports can be saved as

- PDF or PostScript documents
- Web pages
- Microsoft Word documents
- Open Document files
- and more like Beamer slides, etc.

When you render an RMarkdown file, it will appear, by default, as an HTML document in Viewer window of RStudio. If you want to create PDF documents, install a LaTeX compiler. Install MacTeX for Macs (`http://tug.org/mactex`), MiKTeX (www.miktex.org) for Windows, and TeX Live for Linux (www.tug.org/texlive). Alternatively, you can install TinyTeX from https://yihui.name/tinytex/.

1.3.2 Basic Structure of R Markdown

Let's start with a simple RMarkdown file and see what it looks like and the output that it produces when executed. Click `File > New File > R Markdown`, type in the title `Homework problems`, and the author. Click `OK`. Save the file with Ctrl+S, choose a name for example, `homework_1.Rmd`.

Rmarkdown files ends with the `.Rmd` extension. An `.Rmd` file contains three types of contents:

1. A YAML header :

```
---
title: "Homework problems"
author: "Abari Kálmán"
date: '2021 03 31 '
output: html_document
---
```

YAML stands for "yet another markup language" (https://en.wikipedia.org/wiki/YAML).

2. R code chuncks. For example:

~~~
```{r}
myDataFrame <- data.frame(names = LETTERS[1:3], variable_1 = runif(3))
~~~

```
myDataFrame
```
```

3. Text with formatting like bold text, mathematical expressions (), or headings # Heading, etc.

First let's see how we can execute an `.Rmd` file to produce the output as PDF, HTML, etc.

Now click `Knit` to produce a complete report containing all text, code, and results. Alternatively, pressing Ctrl+Shift+K renders the whole document. But in this case, all output formats that are specified in the YAML header will be produced. On the other hand, Knit allows you to specify the output format you want to produce. For example, `Knit > Knit to HTML` produces only HTML output, which is usually faster than producing PDF output.

You can also render the file programmatically with the following command:

```
rmarkdown::render("homework_1.Rmd")
```

This will display the report in the viewer pane, and create a self-contained HTML file.

Instead of running the whole document, you can run each individual code chunk by clicking the Run icon at the top right of the chunk or by pressing Ctrl+Shift+Enter. RStudio executes the code and displays the results inline with the code.

1.3.3 Text Formatting with R Markdown

This section demonstrates the syntax of common components of a document written in R Markdown. Inline text will be *italic* if surrounded by underscores or asterisks, e.g., `_text_` or `*text*`. **Bold** text is produced using a pair of double asterisks (`**text**`). A pair of tildes (`~`) turn text to a subscript (e.g., `H~3~PO~4~` renders H_3PO_4). A pair of carets (`^`) produce a superscript (e.g., `Cu^2+^` renders Cu^{2+}).

Hyperlinks are created using the syntax `[text](link)`, e.g., `[RStudio](https://www.rstudio.com)`. The syntax for images is similar: just add an exclamation mark, e.g., `![alt text or image title](path/to/image)`. Footnotes are put inside the square brackets after a caret `^[]`, e.g., `^[This is a footnote.]`.

Section headers can be written after a number of pound signs, e.g.,

```
# First-level header

## Second-level header

### Third-level header
```
```

If you do not want a certain heading to be numbered, you can add `{-}` or `{.unnumbered}` after the heading, e.g.,

```
# Preface {-}
```

Unordered list items start with `*`, `-`, or `+`, and you can nest one list within another list by indenting the sub-list, e.g.,

```
- one item
- one item
- one item
    - one more item
    - one more item
    - one more item
```

The output is:

- one item

- one item

- one item

    – one more item
    – one more item
    – one more item

Ordered list items start with numbers (you can also nest lists within lists), e.g.,

```
1. the first item
2. the second item
3. the third item
    - one unordered item
    - one unordered item
```

The output does not look too much different with the Markdown source:

1. the first item

2. the second item

3. the third item

    - one unordered item
    - one unordered item

Blockquotes are written after `>`, e.g.,

```
> "I thoroughly disapprove of duels. If a man should challenge me,
  I would take him kindly and forgivingly by the hand and lead him
  to a quiet place and kill him."
>
> --- Mark Twain
```

The actual output (we customized the style for blockquotes in this book):

> "I thoroughly disapprove of duels. If a man should challenge me, I would take him kindly and forgivingly by the hand and lead him to a quiet place and kill him."
>
> — Mark Twain

Plain code blocks can be written after three or more backticks, and you can also indent the blocks by four spaces, e.g.,

~~~
```
This text is displayed verbatim / preformatted
```

Or indent by four spaces:

    This text is displayed verbatim / preformatted
~~~

In general, you'd better leave at least one empty line between adjacent but different elements, e.g., a header and a paragraph. This is to avoid ambiguity to the Markdown renderer. For example, does "#" indicate a header below?

```
In R, the character
# indicates a comment.
```

And does "-" mean a bullet point below?

```
The result of 5
- 3 is 2.
```

Different flavors of Markdown may produce different results if there are no blank lines.

1.3.4 Math expressions

Inline LaTeX equations can be written in a pair of dollar signs using the LaTeX syntax, e.g., `$f(k) = {n \choose k} p^{k} (1-p)^{n-k}$` (actual output: $f(k) = {n \choose k} p^{k} (1-p)^{n-k}$); math expressions of the display style can be written in a pair of double dollar signs, e.g., `$$f(k) = {n \choose k} p^{k} (1-p)^{n-k}$$`, and the output looks like this:

$$f\left(k\right) = \binom{n}{k} p^{k} \left(1-p\right)^{n-k}$$

You can also use math environments inside `$ $` or `$$ $$`, e.g.,

```
$$\begin{array}{ccc}
x_{11} & x_{12} & x_{13}\\
```

```
x_{21} & x_{22} & x_{23}
\end{array}$$
```

$$\begin{array}{ccc}
x_{11} & x_{12} & x_{13}\\
x_{21} & x_{22} & x_{23}
\end{array}$$

```
$$X = \begin{bmatrix}1 & x_{1}\\
1 & x_{2}\\
1 & x_{3}
\end{bmatrix}$$
```

$$X = \begin{bmatrix}1 & x_{1}\\
1 & x_{2}\\
1 & x_{3}
\end{bmatrix}$$

```
$$\Theta = \begin{pmatrix}\alpha & \beta\\
\gamma & \delta
\end{pmatrix}$$
```

$$\Theta = \begin{pmatrix}\alpha & \beta\\
\gamma & \delta
\end{pmatrix}$$

```
$$\begin{vmatrix}a & b\\
c & d
\end{vmatrix}=ad-bc$$
```

$$\begin{vmatrix}a & b\\
c & d
\end{vmatrix}=ad-bc$$

Chapter 2

The R language

2.1 Basic data type

It this chapter, we'll focus on R language. First, we need to learn about data types. The R programming language has something called types, and there are four of them:

- character
- integer
- double
- logical.

Let's take a look at each one of these. Let's start with double.

2.1.1 Double

We can easily create numbers in R. For example:

```
45
#> [1] 45
5
#> [1] 5
0.5
#> [1] 0.5
-0.33
#> [1] -0.33
```

We can execute these lines, these are simple commands, more precisely numerical *constants*. These elements of R language have a fix value. We can't change the value of a constant. The `0.5` means 0.5. So executing of `0.5` five we get 0.5 in R console. Decimals omitting the leading zero are acceptable, we can write `.5`.

It means 0.5. So, we can get a tricky form of a number for example `-.5`, which means -0.5. You can check this out executing these lines.

```
-.5
#> [1] -0.5
```

We are going to move on to discuss the exponential format of numbers. This is the scientific notation, where the number after 'e' gives the powers of ten. For example `4e2` means 400, because 4 multiplied by 10 squared is 400 (4 multiplied by 10 the power of 2).

```
4e2
#> [1] 400
```

Generally, we use plus-minus sign before the power, for example `4e+3`, which value is 4000, `4.2e+3` means 4200, and `4.2e-3` means 0.0042. In this case, we have to divide by 10 cubed (10 to the power of 3), or multiplied by ten to the power of -3.

```
4e+3
#> [1] 4000
4.2e+3
#> [1] 4200
4.2e-3
#> [1] 0.0042
```

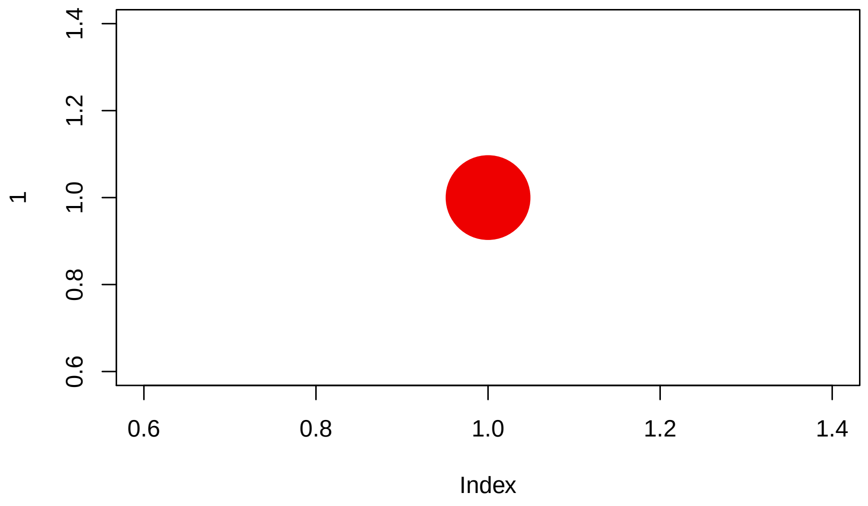
The last format of numbers is the hexadecimal. For example after '0x' prefix, we can type `0xfe3` which means 4067.

```
0xfe3
#> [1] 4067
```

The hexadecimal numbering system uses 16 as the base (as opposed to ten), so in this system we have 16 digits to represent numbers. The symbols "0"–"9" to represent values 0 to 9, and "A"-"F" (or alternatively "a"-"f") to represent values 10 to 15.

We use hexadecimal at most, when we specify a color. For example:

```
plot(1, col="#ee0000", pch=16, cex=8)
```



This command creates a plot (graph), with only one point coloured by red. A hexadecimal color is specified with a `#` and 2 digits for red, to digits for green and two digits for blue (#RRGGBB). RR (red), GG (green) and BB (blue) are hexadecimal integers between 00 and FF specifying the intensity of the colour. For example, `#0000FF` is displayed as blue, because the blue component is set to its highest value (FF) and the others are set to 00.

2.1.2 Integer

The next number type is the integer. Integer means the whole numbers, For example 4, 42 or -12. But in R we have to use the capital `L` suffix. `L` just indicates that this is a long, it's an internal storage type. It is a way to represent natural numbers like 1 and 2. Integers arise from counting, in most cases.

```
4L
#> [1] 4
42L
#> [1] 42
-12L
#> [1] -12
```

To sum it up, decimal values like `4.5` and whole numbers without `L` suffix are double in R. Whole numbers with `L` suffix are integers in R. Both double and integer are numerics. Let's try something. Type in 2 and 2L, and execute them. You don't see the difference between the double 2 and the integer 2 from the output.

```
2
#> [1] 2
2L
#> [1] 2
```

However, there are two functions that reveal the difference. The `typeof()` and `class()` functions return almost the same values, the types of the data. Notice, the `class()` function with double argument returns `"numeric"`.

```
typeof(2)
#> [1] "double"
typeof(2L)
#> [1] "integer"
class(2)
#> [1] "numeric"
class(2L)
#> [1] "integer"
```

Of course, we can try these functions with decimal values.

```
typeof(2.4)
#> [1] "double"
class(2.4)
#> [1] "numeric"
```

2.1.3 Characters

Text (or string) values are called characters in R. For example type in some text inside quote marks.

```
"some text"
#> [1] "some text"
'Dobó, István'
#> [1] "Dobó, István"
" sldjf odiuoiuoiu657676876987876875  32 23sdcsd)(/=(/%"
#> [1] " sldjf odiuoiuoiu657676876987876875  32 23sdcsd)(/=(/%"
```

Note how the quotation marks in the editor indicate that `"some text"` is a string. Syntax highlighting also helps you to identify string values. It may also be noted that autocompletion is also working. We typed in only one quote mark, the second one appeared automatically. We can use double quote mark (`"`) and single quote mark (`'`), but the opening and the closing quote marks need to match. If we start with single quotation mark, we have to finish with single one. If we start with double quotation mark, we have to finish with double one.

We can use any characters inside quotation marks, except surrounding quotation marks.

Let's check out the `typeof()` and `class()` function with character data. They return `"character"`.

```
typeof("Friday")
#> [1] "character"
class("Friday")
#> [1] "character"
```

2.1.4 Logical

The last data types is the logical. Boolean values (TRUE or FALSE) are called logical in R. Let's head over to the script window and start with `TRUE`, in capital letters. `TRUE` is a logical. Logical constants can be either `TRUE` or `FALSE`.

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

`TRUE` and `FALSE` can be abbreviated to `T` and `F` respectively. However, I want to strongly encourage you to use the full versions, `TRUE` and `FALSE`.

```
T
#> [1] TRUE
F
#> [1] FALSE
```

Finally, we can check out the type these logical values.

```
typeof(TRUE)
#> [1] "logical"
class(F)
#> [1] "logical"
```

Note, we did not use quotation marks in logical vales. If we use them, we will get character vales. For example:

```
typeof("TRUE")
#> [1] "character"
```

To sum it up, R works with numerous data types. Some of the most basic types are double, integer, character and logical. We learned how to write `constants` in R. There are two functions `typeof()` and `class()` with which we can check out constants' type.

2.2 Operators

2.2.1 Arithmetic operators

In its most basic form, R can be used as a simple calculator. We can use the following arithmetic operators:

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation.

Let's put a basic addition, subtraction, multiplication, division and an extra expression, an exponentiation into our editor window. We use plus (`+`), minus (`-`), asterisk (`*`), slash (`/`) and double asterisks (`**`) or hat symbols (`^`). Double asterisk `**` behaves exactly like `^` (hat, caret), these are to-the-power-of, exponent operators.

```
34.1 + 2e4 # Addition
#> [1] 20034.1
0xe4 - 23  # Subtraction
#> [1] 205
23 * 45000 # Multiplication
#> [1] 1035000
23/12      # Division
#> [1] 1.916667
23 ** 12   # Exponentiation
#> [1] 2.191462e+16
23 ^ 12    # Exponentiation
#> [1] 2.191462e+16
```

Additionally, the modulo (`%%`) returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

```
5 %% 3   # modulo: remainder of 5 divided by 3
#> [1] 2
```

The integer division `x %/% y` x divided by y but rounded down.

```
7 %/% 3   # integer division
#> [1] 2
```

2.2.2 Logical operators

R uses standard logical notation for OR and AND, and NOT. First of all, the exclamation mark (`!`) stands for NOT. So if I type in `!TRUE`, I'll get false. Or surprisingly if I type in `!FALSE` I'll get true. It just inverts the value.

The ampersand (`&`) is for AND. So I can type in `TRUE & TRUE`, which means if true and true then I get true. If I type in `TRUE & FALSE` then I'll receive false because both arguments have to be true in order for the result to be true.

Let's talk about the pipeline symbol (`|`). The pipeline symbol means OR. So in this case I can type in `TRUE | TRUE` and I'm going to get back true. If I type in `TRUE | FALSE`, I'll get back true because for OR it is enough for only one value to be true to evaluate it as true.

```
!TRUE
#> [1] FALSE
!FALSE
#> [1] TRUE
TRUE & TRUE
#> [1] TRUE
TRUE & FALSE
#> [1] FALSE
TRUE | TRUE
#> [1] TRUE
TRUE | FALSE
#> [1] TRUE
```

2.2.3 Relational operators

Relational operators are used to compare between values. Here is a list of relational operators available in R.

Operator	Description
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

```
2 < 2.3
#> [1] TRUE
2 <= 2.3
#> [1] TRUE
2 > 2.3
#> [1] FALSE
2 >= 2.3
#> [1] FALSE
2 == 2.3
#> [1] FALSE
```

```
2 != 2.3
#> [1] TRUE

"apple" == "Apple"
#> [1] FALSE
"apple" != "Apple"
#> [1] TRUE

TRUE == FALSE
#> [1] FALSE
TRUE != FALSE
#> [1] TRUE

TRUE == 1
#> [1] TRUE
TRUE != 1
#> [1] FALSE

(-6 * 14) == (17 - 101)
#> [1] TRUE
```

The result of comparison is a Boolean value (`TRUE` or `FALSE`).

2.2.4 Assignment operators

We will use one of them, the “left arrow” (`<-`) operator. The “left arrow” assignment operator is actually two symbols, a „less than” sign and a „minus”. Good to know, there is a shortcut for assignment operator, namely Alt+- in *RStudio*. What is the assignment operator for? It is for *objects*. Objects allow you to store a value in R. You can then later use this object’s name to easily access the value that is stored within this object.

Let’s create our first object. You can assign the value 4 to an object `my_object` with the command:

```
my_object <- 4
```

Then type in the name of the object my_object, and execute it. Notice that when you ask R to print my_object, the value 4 appears.

```
my_object
#> [1] 4
```

If we use `class()` or `typeof()` functions, we’ll get type of the object.

```
typeof(my_object)
#> [1] "double"
class(my_object)
```

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

2.2.5 Miscellaneous operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Colon operator (`:`) creates the series of numbers in sequence for a vector.

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

The `%in%` operator is used to identify if an element belongs to a vector. It returns a logical vector indicating if there is a match or not for all elements in the left operand in the right operand.

```
c(3, 4, 5, 7, 10) %in% c(2, 4, 6, 8, 10)
#> [1] FALSE  TRUE FALSE FALSE  TRUE
```

The double colon operator (`::`) is a binary operator to access functions or datasets from packages. As we mentioned, packages extend R's knowledge. Every R package contains functions and/or dataset. Every R package has a name. For example we have an installed packages called **MASS**. In **MASS** packages, there is dataset called `survey`. So, we can type in `MASS::survey`, to reach the survey dataset from **MASS** package.

```
str(MASS::survey)
#> 'data.frame':	237 obs. of  12 variables:
#>  $ Sex   : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
#>  $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
#>  $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
#>  $ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
#>  $ Fold  : Factor w/ 3 levels "L on R","Neither",..: 3 3 1 3 2 1 1 3 3 3 ...
#>  $ Pulse : int  92 104 87 NA 35 64 83 74 72 90 ...
#>  $ Clap  : Factor w/ 3 levels "Left","Neither",..: 1 1 2 2 3 3 3 3 3 3 ...
#>  $ Exer  : Factor w/ 3 levels "Freq","None",..: 3 2 2 2 3 3 1 1 3 3 ...
#>  $ Smoke : Factor w/ 4 levels "Heavy","Never",..: 2 4 3 2 2 2 2 2 2 2 ...
#>  $ Height: num  173 178 NA 160 165 ...
#>  $ M.I   : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ...
#>  $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
```

2.2.6 Operator Precedence in R

As we mentioned, R can be used as a powerful calculator. Simply type an arithmetic expression and press Ctrl+Enter.

```
4 + 8       # will return the result 12
#> [1] 12
```

```
4 + 5 + 3   # will return the result 12
#> [1] 12
```

But, there could be problems if you are not careful. R normally execute your arithmetic expression by evaluating each item from left to right. 4 plus 8 equals 12. 4 plus 5 plus 3 equals twelve. But good to know, operators have precedence in the order of evaluation.

Let's start with more complex expressions that can cause problems if you are not careful.

```
4 + 5 * 3    # will return the result 19
#> [1] 19
```

Notice that the expression was not evaluated strictly left to right. R actually evaluated 5 times 3 and then added that result to 4. The R operator precedence rules caused this result.

Multiplication and division have a higher precedence than the addition and subtraction operator so the multiplication is performed before the addition. We can arrange the operators in order from high precedence to low precedence. We can extend the list with exponentiation.

Operators with higher precedence (nearer top of the list) are performed before those with lower precedence (nearer to the bottom).

Operator	Description
`::`	access
`$`	component
`[ [[`	indexing
`^ **`	exponentiation
`- +`	unary minus, unary plus
`:`	sequence operator
`%any%` e.g. `%% %/% %in%`	special operators
`* /`	multiplication, division
`+ -`	addition, subtraction
`< > <= >= == !=`	comparisions
`!`	logical NOT
`&`	logical AND
`|`	logical OR
`<-`	assignment

The exponentiation operator has a higher precedence than the multiplication, so the the exponentiation is the first that performed. Three squared multiplied by two:

```
2 * 3 ** 2
#> [1] 18
```

Operator precedence can be overridden with explicit use of parentheses. In the case of this example, we could enter

```
(4 + 5) * 3
#> [1] 27
(2 * 3) ** 2
#> [1] 36
```

In practise, if you are at all unsure about the precedence of your operators, the simplest thing to do is to use parentheses to make the evaluation order explicit.

2.2.7 Binary and unary operators

Another thing about operators. There are two different type of operators. Binary and unary operators. The question is, how many operands they require to work properly.

A unary operator is an operator that operates on only one operand. Binary operators that we used earlier operates two operands. We've talked about binary operators so far. Addition, subtraction, multiplication, division, exponentiation are binary operators. They require two operands to work properly.

For example `5 -` (select these characters for executing) is wrong, we get a continuation prompt. We could complete the command. but we never do that. Click on console pane, and hit Esc.

Complete the line in the script editor, `5-2`, hit Ctrl+Enter equals 3. Subtraction is a binary operator, it has two operands. Five and two.

In R, there are a few unary operators, for example unary minus, and unary plus. They are the sign operators. They are used to indicate or change the sign of a value. The `+` and `-` signs indicate the sign of a value. The plus sign can be used to signal that we have a positive number. It can be omitted and it is mostly done so. We could type `5`, `+5`. The minus sign changes the sign of a value. To write negative five, we need type in `-5`. This is the unary minus operator.

```
5
#> [1] 5
+5
#> [1] 5
-5
#> [1] -5
```

To sum up, we really need another list in the script editor. R Terminology. We talked about constants, this is a language element which has fix value, we can not change. 5 means five, Friday in quotes means Friday. TRUE means logical TRUE. Operators perform mathematical or logical operations on values, on constants. Operators have precedence and operators can be unary or binary. Every constant has a basic type (double, integer, character, logical). And, we can build expressions with constants, operators and parentheses. We also talked about comment in R, that is everything after a # (a hashtag). It will have no effect if you run it in R.

2.3 Objects

We talked about object that is very important language element in R. We will be discussing everything that needs to be known about objects and data structures.

Lets' start with objects. An object allows you to store data in R for later use. Suppose the height of a rectangle is 2. Let's assign this value 2 to an object. Let's call it `height`.

```
height <- 2
```

This time, R does not print anything in the console, but we can not see error messages either. Command executing without error messages, even without any messages indicates everything is ok. The command evaluated successfully. Look at the top right pane. In the environment tab we have a new item in the list. `Height` and its value 2. All objects with name and value will appear in this list. We have only one object in this session, so this list has only one item.

If you now simply type and execute height in the script window, R returns 2.

```
height
#> [1] 2
```

We can do a similar thing for the width of our imaginary rectangle. We assign the value 4 to an object called `width`.

```
width <- 4
```

In the top right pane, we have two items in the list. Actually, this list in the environment tab shows the *workspace*. Workspace is a special location in your computer's memory that temporarily stores data we just created using R. Workspace is the place where R objects 'live'. You can list all objects with the `ls()` function.

```
ls()
#> [1] "height"    "my_object" "width"
```

This shows you a list of all the objects you have created up to now. There are two objects in your workspace at the moment, `height` and `width`. If we try to access object that's not in the workspace, `depth` for example, R throws an error.

```
depth  # error
```

Suppose you now want to find out the area of our imaginary rectangle, which is height multiplied by width. Height equals 2, and width equals 4, so the result is 8. We have two ways to calculate the area:

```
2 * 4
#> [1] 8
height * width
#> [1] 8
```

The second line with objects is more advanced than the first line with constants. Let's also assign this result to a new object, called area.

```
area <- height * width
area
#> [1] 8
```

We can print the value of object `area`, type and execute `area`. It's 8. Inspecting the workspace again with `ls()`, shows that the workspace contains three objects now: `area`, `height` and `width`.

Now, this is all great, but what if you want to recalculate the area of your imaginary rectangle when the height is 3 and the width is 6? You'd have to reassign the objects width and height in the script window, and then recalculate the area. The value of area will change, executing area will return 18.

```
height <- 3
width <- 6
area <- height * width
area
#> [1] 18
```

How to find the perimeter of this rectangle. Let's create a new object called perimeter.

```
perimeter <- 2*(width+height)
perimeter
#> [1] 18
```

Let's sum up the objects. The general form of creating or modifying an object is object name, assignment operator and an expression.

```
object_name <- expression
```

First, we have to choose a valid object name. Object name can contain any letters from English alphabet, underscore, dot, or digit. We need to start with letter in an object name. Expression can be a simply constant, or an object name, or constant and object names with operators and parentheses.

We can create logical object or double object, integer object and character object:

```
x.logical <- TRUE
y.double <- 12.3
z.integer <- 12L
k.character <- "Hello world!"
```

and we can print their type, with `typeof()` or `class()` function.

```
typeof(x.logical)
#> [1] "logical"
class(x.logical)
#> [1] "logical"

typeof(y.double)
#> [1] "double"
class(y.double)
#> [1] "numeric"

typeof(z.integer)
#> [1] "integer"
class(z.integer)
#> [1] "integer"

typeof(k.character)
#> [1] "character"
class(k.character)
#> [1] "character"
```

2.3.1 Testing the type

Instead of asking for the type or class of an object, you can also use the is-dot-functions to see whether objects are actually of a certain type. To see if an object is a double, we can use the `is.double()` function. It returns a logical value. `TRUE` or `FALSE`. To see if an object is integer, we can use `is.integer()`. There is `is.numeric()` function to see whether objects are numeric. The integer and double are numerics. Let's try the `is.logical()` and the `is.character()` functions.

```
# is.*() functions, test of types
is.double(x.logical)
#> [1] FALSE
is.double(y.double)
#> [1] TRUE
is.double(z.integer)
#> [1] FALSE
```

```
is.double(k.character)
#> [1] FALSE

is.integer(x.logical)
#> [1] FALSE
is.integer(y.double)
#> [1] FALSE
is.integer(z.integer)
#> [1] TRUE
is.integer(k.character)
#> [1] FALSE

is.numeric(x.logical)
#> [1] FALSE
is.numeric(y.double)
#> [1] TRUE
is.numeric(z.integer)
#> [1] TRUE
is.numeric(k.character)
#> [1] FALSE

is.logical(x.logical)
#> [1] TRUE
is.logical(y.double)
#> [1] FALSE
is.logical(z.integer)
#> [1] FALSE
is.logical(k.character)
#> [1] FALSE

is.character(x.logical)
#> [1] FALSE
is.character(y.double)
#> [1] FALSE
is.character(z.integer)
#> [1] FALSE
is.character(k.character)
#> [1] TRUE
```

2.3.2 Coercion

There are cases in which you want to change the type of an object to another one. How would that work? This is where coercion comes into play! By using the as-dot-functions one can coerce the type of a variable to another type. Many ways of transformation between types are possible. Have a look at these examples.

```
# as.*() functions, coercion
as.logical(y.double)
#> [1] TRUE
as.logical(z.integer)
#> [1] TRUE
as.logical(k.character)
#> [1] NA
```

The first three commands here coerce three different objects to a logical. Every number except zero coerced to TRUE, so the first two commands return TRUE. The third command outputs an NA, a missing value. R doesn't understand how to transform "Hello world" into a logical, and decides to return a Not Available instead. We can try to convert zero to logical.

```
as.logical(0)
#> [1] FALSE
```

The result is `FALSE`. We can easily coerce logical, integer and double to character.

```
as.character(y.double)
#> [1] "12.3"
as.character(z.integer)
#> [1] "12"
as.character(x.logical)
#> [1] "TRUE"
```

Let's try to find out how to convert logical and character to number? What functions we have in R with which we can achieve this? Yes, `as.double()`, `as.integer()` and `as.numeric()`. `as.numeric()` is identical to `as.double()`.

```
as.double(x.logical)
#> [1] 1
as.double(z.integer)
#> [1] 12
as.double(k.character)
#> [1] NA

as.integer(y.double)
#> [1] 12
as.integer(x.logical)
#> [1] 1
as.integer(k.character)
#> [1] NA

as.numeric(x.logical)
#> [1] 1
as.numeric(y.double)
```

```
#> [1] 12.3
as.numeric(z.integer)
#> [1] 12
as.numeric(k.character)
#> [1] NA
```

Logical TRUE coerces to the numeric one (1). FALSE, however, coerces to the numeric zero (0). Valid number in a string coerces to number, invalid number in a string, for example "hello", coerces missing value. R doesn't understand how to transform "hello" into a numeric, and decides to return a Not Available (NA) instead.

```
as.double(TRUE)
#> [1] 1
as.numeric(TRUE)
#> [1] 1
as.integer(TRUE)
#> [1] 1

as.double(FALSE)
#> [1] 0
as.numeric(FALSE)
#> [1] 0
as.integer(FALSE)
#> [1] 0

as.double("12.3")
#> [1] 12.3
as.numeric("12.3")
#> [1] 12.3
as.integer("12.3")
#> [1] 12

as.double("hello")
#> [1] NA
as.numeric("hello")
#> [1] NA
as.integer("hello")
#> [1] NA
```

2.4 Data structures

2.4.1 Vectors

In R, we use data sets all the time. Data sets are a collection or group of values, double, integer, character or logical values. They are the result of a scientific measurements, a surveys or other data collection methods. For example, you may record the ages of each member of your family. In R, we have to use the `c()` function for this, which allows you to combine values into a vector. This is a four member family, 2 children, mother, father. We could combine the ages of each member of family. Execute this command.

```
c(18, 20, 47, 49)
#> [1] 18 20 47 49
```

As you can see in the output, it is a vector. A vector is nothing more than a sequence of data elements of the same basic data type. This is a double vector. We can check it with `typeof()` or `is.double()` functions.

```
typeof(c(18, 20, 47, 49))
#> [1] "double"
is.double(c(18, 20, 47, 49))
#> [1] TRUE
```

Of course we could also assign this double vector to a new object, `age` for example.

```
age <- c(18, 20, 47, 49)
```

We can assert that it is a vector, by typing

```
is.vector(age)
#> [1] TRUE
```

We can also check the top right pane, the workspace. Age is listed, and we can see, it is a double vector indicated by num with four elements. “Num” means double.

We can print the value of this vector.

```
age
#> [1] 18 20 47 49
```

Please execute the `age` command. We can see, `age` contains four elements. The firs element is 18, the second 20, the third is 47, and the last element is 49. Every vector is a sequence of data elements. We can check the length of this vector with `length()` function.

```
length(age)
#> [1] 4
```

It tells us, `age` vector holds four elements. The length of this vector is 4.

Good to know, the vector is the simplest data structure in R. Objects we've created in the previous topic, are also vectors. They're all just vectors of length 1. They contain a single number (for example object `height`) or a single character, object `k.character`. We can check this with `is.vector()` function.

```
is.vector(height)
#> [1] TRUE
is.vector(k.character)
#> [1] TRUE
```

So, to sum up, a vector is a sequence of data elements, so a vectors is a one-dimensional data structure. The last important thing is that in R, a vector can only hold elements of the same type. This means that you cannot have a vector that contains both logicals and numerics, for example. If you do try to build such a vector, R automatically performs coercion to make sure that you end up with a vector that contains elements of the same type. Let's see how that works with an example.

```
c(12, TRUE)
#> [1] 12  1
c(12, "Hello")
#> [1] "12"    "Hello"
c("Hello", TRUE)
#> [1] "Hello" "TRUE"
c("Hello", TRUE, 23.1)
#> [1] "Hello" "TRUE"  "23.1"
```

If you now inspect these vectors, you'll see that logical value coerced to numeric in the first command. and the numeric or logical values coerced to characters otherwise. So, to sum up, vector is a one-dimensional and homogeneous data structure. Let's practise creating vector. Store the gender of family members.

```
gender <- c("male", "male", "female", "male")
gender
#> [1] "male"   "male"   "female" "male"
```

Gender is a character vector that has length 4. You can check with the `length()` function, and the top right pane.

```
length(gender)
#> [1] 4
```

2.4.2 Factor

Factor is about categorical variables. Unlike numerical variables, categorical variables can only take on a limited number of different values. A categorical variable can only belong to a limited number of categories. If you want to store categorical data in R, you have to use factors. This is the only way that the statistical modelling techniques handle such data correctly. If we meet categorical variables, we need the factor data structure in R.

A good example of a categorical variable is a person's gender. It can be male or female. Gender is a categorical variable in statistics. We've created the gender object as a character vector. But, as we mentioned, we are not ready. We need to convert this vector to factor. You can use the `factor()` function.

```
gender.fact <- factor(gender)
gender.fact
#> [1] male   male   female male
#> Levels: female male
```

The printout looks somewhat different than the original one: there are no double quotes anymore and also the factor levels, corresponding to the different categories, are printed.

R basically does two things when you call the factor function on a character vector: first of all, it scans through the vector to see the different categories that are in there. In this case, that's "female" and "male". Notice here that R sorts the levels alphabetically. Next, it converts the character vector, `gender` in this example, to a vector of integer values. These integers correspond to a set of character values to use when the factor is displayed. These character values are called labels or levels. Inspecting the structure reveals this. We can use the `unclass()` to uncover the factor. You can see the underlying integer vector and the character vector of levels. We're dealing with a factor with 2 levels. The "female"'s are encoded as 1, because it's the first level, "male" is encoded as 2, because it's the second level.

```
unclass(gender.fact)
#> [1] 2 2 1 2
#> attr(,"levels")
#> [1] "female" "male"
```

Why this conversion? Well, it can be that your categories are very long character strings. Each time repeating this string per observation can take up a lot of memory. By using this simple encoding, much less space is necessary. Just remember that factors are actually integer vectors, where each integer corresponds to a category, or a level.

We can also use the `str()` function to display the internal structure of an R object, of a factor in this case.

```
str(gender.fact)
#>  Factor w/ 2 levels "female","male": 2 2 1 2
```

Finally, we can check the type and the class of this factor with `typeof()` and `class()` functions. We can ask whether the factor is a vector or a factor.

```
typeof(gender.fact)
#> [1] "integer"
class(gender.fact)
#> [1] "factor"
is.vector(gender.fact)
#> [1] FALSE
is.factor(gender.fact)
#> [1] TRUE
```

To sum up the factors. Factor is one-dimensional and homogueonus as a vector. In fact, factor is stored as an integer vectors where each integer has a label. Factor elements can take on one of a specific set of values. Factor `gender.fact` will take on only the values "male" or "female". The set of values that elements of a factor can take are called its level.

2.4.3 Matrix

A matrix is similar to a vector. Where a vector is a sequence of data elements, which is one-dimensional, a matrix is a similar collection of data elements, but this time arranged into a fixed number of rows and columns. Since you are only working with rows and columns, a matrix is called two-dimensional. As with the vector, the matrix can contain only one type.

To build a matrix, you use the `matrix()` function. Most importantly, it needs a vector, containing the values you want to place in the matrix, and at least one matrix dimension: rows and/or columns.

Have a look at the following example, that creates a 2-by-3 matrix containing the values 1 to 6, by specifying the vector and setting the `nrow=` argument to 2:

```
matrix(1:6, nrow = 2)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

R sees that the input vector has length 6 and that there have to be two rows. It then infers that you'll probably want 3 columns, such that the number of matrix elements matches the number of input vector elements. You could just as well specify `ncol=` instead of `nrow=`; in this case, R infers the number of rows automatically.

```
matrix(1:6, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

In both these examples, R takes the vector containing the values 1 to 6, and fills it up, column by column. If you prefer to fill up the matrix in a row-wise fashion, such that the 1, 2 and 3 are in the first row, you can set the `byrow=` argument of matrix to `TRUE`

```
matrix(1:6, nrow = 2, byrow = T)
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
```

Suppose you pass a vector containing the values 1 to 3 to the matrix function, and explicitly say you want a matrix with 2 rows and 3 columns:

```
matrix(1:3, nrow = 2, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    2
#> [2,]    2    1    3
```

R fills up the matrix column by column and simply repeats the vector. If you try to fill up the matrix with a vector whose multiple does not nicely fit in the matrix, for example when you want to put a 4-element vector in a 6-element matrix, R generates a warning message.

```
matrix(1:4, nrow = 2, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    1
#> [2,]    2    4    2
```

Actually, apart from the `matrix()` function, there's yet another easy way to create matrices that is more intuitive in some cases. You can paste vectors together using the `cbind()` and `rbind()` functions. Have a look at these calls:

```
cbind(1:3, 1:3)
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    2
#> [3,]    3    3
rbind(1:3, 1:3)
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    1    2    3
```

`cbind()`, short for column bind, takes the vectors you pass it, and sticks them together as if they were columns of a matrix. The `rbind()` function, short for row bind, does the same thing but takes the input as rows and makes a matrix out of them. These functions can come in pretty handy, because they're often more easy to use than the `matrix()` function.

The bind functions I just introduced can also handle matrices actually, so you can easily use them to paste another row or another column to an already existing matrix. Suppose you have a matrix `m`, containing the elements 1 to 6:

```
m <- matrix(1:6, byrow = T, nrow=2)
m
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
```

If you want to add another row to it, containing the values 7, 8, 9, you could simply run this command:

```
rbind(m, c(7, 8, 9))
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
#> [3,]    7    8    9
```

You can do a similar thing with `cbind()`:

```
cbind(m, c(1,2))
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    1
#> [2,]    4    5    6    2
```

Next up is naming the matrix. You could assign names to both columns and rows. That's why R came up with the `rownames()` and `colnames()` functions. Their use is pretty straightforward. Retaking the matrix `m` from before,

```
rownames(m) <- c("row.1", "row.2")
m
#>       [,1] [,2] [,3]
#> row.1    1    2    3
#> row.2    4    5    6
colnames(m) <- c("col.1", "col.2", "col.3")
m
#>       col.1 col.2 col.3
#> row.1     1     2     3
#> row.2     4     5     6
```

Printing m shows that it worked.

Just as with vectors, there are also one-liner ways of naming matrices while you're building it. You use the `dimnames=` argument of the matrix function for this. Check this out.

```
matrix(1:6, byrow = T, nrow=2,
       dimnames = list(
         rows=c("row.1", "row.2"),
         cols=c("col.1", "col.2", "col.3")))
#>        cols
#> rows    col.1 col.2 col.3
#>   row.1     1     2     3
#>   row.2     4     5     6
```

You can create logical or character matrices as well.

```
matrix(c(T, F), nrow=3, ncol=4)
#>       [,1]  [,2]  [,3]  [,4]
#> [1,]  TRUE FALSE  TRUE FALSE
#> [2,] FALSE  TRUE FALSE  TRUE
#> [3,]  TRUE FALSE  TRUE FALSE
matrix(c("Jane", "Mark"), nrow=3, ncol=4)
#>      [,1]   [,2]   [,3]   [,4]
#> [1,] "Jane" "Mark" "Jane" "Mark"
#> [2,] "Mark" "Jane" "Mark" "Jane"
#> [3,] "Jane" "Mark" "Jane" "Mark"
```

2.4.4 Array

In R an array is a vector two or more dimensions. A matrix is actually a two dimensional array. Array is like a stacked matrix. So let's build some data that we can use to demonstrate that. First of all, let's build up a character vector.

```
vector.chr <- c("twas","brillig","and","the","slithey","toves","did","gyre","and","gim
```

Now let's create an array out of that with `array()` functions.

```
array.chr <- array(data = vector.chr, dim = c(2, 3, 2))
array.chr
#> , , 1
#>
#>      [,1]      [,2]  [,3]
#> [1,] "twas"    "and" "slithey"
#> [2,] "brillig" "the" "toves"
#>
#> , , 2
#>
#>      [,1]   [,2]     [,3]
#> [1,] "did"  "and"    "in"
#> [2,] "gyre" "gimble" "wabe"
```

With `dim=` argument we can give it some dimensions. And in this case, we are going to concatenate three values, two rows, three columns, and two levels. We have an array, and you can see that there are three dimensions. So you can see that I have two tables. Actually looks like two matrices. And then there's a second level. And again it has two rows and three columns. It's the second half.

Of course, we can create logical or numeric arrays.

```
array(1:30, dim = c(2, 3, 5))
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12
#>
#> , , 3
#>
#>      [,1] [,2] [,3]
#> [1,]   13   15   17
#> [2,]   14   16   18
#>
#> , , 4
#>
#>      [,1] [,2] [,3]
#> [1,]   19   21   23
#> [2,]   20   22   24
#>
#> , , 5
#>
#>      [,1] [,2] [,3]
#> [1,]   25   27   29
#> [2,]   26   28   30
array(c(T, F, T), dim = c(2, 3, 5))
#> , , 1
#>
#>       [,1] [,2]  [,3]
#> [1,]  TRUE TRUE FALSE
#> [2,] FALSE TRUE  TRUE
#>
#> , , 2
#>
#>       [,1] [,2]  [,3]
#> [1,]  TRUE TRUE FALSE
#> [2,] FALSE TRUE  TRUE
#>
#> , , 3
#>
```

```
#>       [,1] [,2]  [,3]
#> [1,]  TRUE TRUE FALSE
#> [2,] FALSE TRUE  TRUE
#>
#> , , 4
#>
#>       [,1] [,2]  [,3]
#> [1,]  TRUE TRUE FALSE
#> [2,] FALSE TRUE  TRUE
#>
#> , , 5
#>
#>       [,1] [,2]  [,3]
#> [1,]  TRUE TRUE FALSE
#> [2,] FALSE TRUE  TRUE
```

2.4.5 List

List is a one-dimensional and heterogeneous data structure. A list can contain all kinds of R objects, such as vectors and matrices, but also other R objects, such as data frames, factors and even an other list. Let's build a lists. We will store information about a family. We can create a list with `list()` function, we want to store the address, how many cars does the family have, name and age of the family members.

```
my.family <- list(address="10 Downing Street", cars=5, age=c(12, 15), name=c("Hermione"
my.family
#> $address
#> [1] "10 Downing Street"
#>
#> $cars
#> [1] 5
#>
#> $age
#> [1] 12 15
#>
#> $name
#> [1] "Hermione" "Harry"
```

2.4.6 Data frame

The data frame is the most important data structure in R. R is a statistical programming language, and in statistics we are working with data sets. Vectors and factors are good examples for a minimal data sets. Data sets are typically comprised of observations (cases, instances), and all these observations have some variables associated with them. We can have for example, a data set of 4 people. Each person is an instance, and the properties about these people, such as for example their age, and their gender. How could you store such information in R? We have done it, in a numeric vector and a factor, called `age` and `gender.fact`. One-dimensional structures is not really useful to work with. We have to keep together the observations. We need a two-dimensional structure. We need a data frame.

Let's create a data frame. We need to use `data.frame()` function.

```
age <- c(18, 20, 47, 49)
gender <- c("male", "male", "female", "male")
gender.fact <- factor(gender)
df <- data.frame(gender.fact, age)
```

We'used the `age` vector and `gender.fact` factor to create the new data frame object. Executing `df`, we can see the value of the data frame.

```
df
#>   gender.fact age
#> 1        male  18
#> 2        male  20
#> 3      female  47
#> 4        male  49
```

This a two-dimensional structure. It has rows and columns. The rows correspond to the observations, the people in our example, while the columns correspond to the variables, or the properties of each of these people.

We can see that a data frame can contain elements of different types. The first column contains factor labels, and the second one is numerics. We can see the names of the coloumns: gender.fact, age, that come from the function call, from the vector's name. We can specify the names explicitly, for example

```
df <- data.frame(gender=gender.fact, age)
df
#>   gender age
#> 1   male  18
#> 2   male  20
#> 3 female  47
#> 4   male  49
```

If we print the value of this data frame we can see the new name of the first column. We can also see the names of the rows, which are simply number from 1 to 4. There still is a restriction on the data frame data types. Elements in the same column should be of the same type. That's not really a problem, because in one column, the `age` column for example, you'll always want a numeric, because an `age` is always a number, regardless of the observation.

Data frame is a two-dimensional and heterogeneous data structure to store small or big data sets. Typically a data frame contains numeric vectors or factors with the same length. Rows correspond to observations to the four member of a family, columns correspond to variables, the properties of the members of the family.

We can check the type and a class of the data frame.

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.vector(df)
#> [1] FALSE
is.factor(df)
#> [1] FALSE
is.data.frame(df)
#> [1] TRUE
```

Finally, practise creating data frame. We also know the heights of the members of the family. How could we store this data in a data frame.

```
height <- c(172, 180, 167, 183)
df.2 <- data.frame(gender = gender.fact, age, height)
df.2
#>   gender age height
#> 1   male  18    172
#> 2   male  20    180
#> 3 female  47    167
#> 4   male  49    183
str(df.2)
#> 'data.frame':    4 obs. of  3 variables:
#>  $ gender: Factor w/ 2 levels "female","male": 2 2 1 2
#>  $ age   : num  18 20 47 49
#>  $ height: num  172 180 167 183
```

To sum up, you can find the data structures corresponding types and classes below:

Data structure	`typeof()`	`class()`
double vector	`"double"`	`"numeric"`
integer vector	`"integer"`	`"integer"`
logical vector	`"logical"`	`"logical"`
character vector	`"character"`	`"character"`
double matrix	`"double"`	`"matrix"`
integer matrix	`"integer"`	`"matrix"`
logical matrix	`"logical"`	`"matrix"`

Data structure	typeof()	class()
character matrix	"character"	"matrix"
double array	"double"	"matrix" "array"
integer array	"integer"	"matrix" "array"
logical array	"logical"	"matrix" "array"
character array	"character"	"matrix" "array"
factor	"integer"	"factor"
list	"list"	"list"
data frame	"list"	"data.frame"

Table above is based on following code:

```
# double vector ----
x <- c(1, 2)
typeof(x)
#> [1] "double"
class(x)
#> [1] "numeric"

# integer vector ----
x <- c(1L, 2L)
typeof(x)
#> [1] "integer"
class(x)
#> [1] "integer"

# logical vector ----
x <- c(TRUE, FALSE)
typeof(x)
#> [1] "logical"
class(x)
#> [1] "logical"

# character vector ----
x <- c("Paul", "Jane")
typeof(x)
#> [1] "character"
class(x)
#> [1] "character"

# double matrix ----
x <- matrix(c(1, 2), nrow=2, ncol=2)
typeof(x)
#> [1] "double"
class(x)
```

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

# integer matrix ----
x <- matrix(c(1L, 2L), nrow=2, ncol=2)
typeof(x)
#> [1] "integer"
class(x)
#> [1] "matrix" "array"

# logical matrix ----
x <- matrix(c(TRUE, FALSE), nrow=2, ncol=2)
typeof(x)
#> [1] "logical"
class(x)
#> [1] "matrix" "array"

# character matrix ----
x <- matrix(c("Paul", "Jane"), nrow=2, ncol=2)
typeof(x)
#> [1] "character"
class(x)
#> [1] "matrix" "array"

# double array ----
x <- array(c(1,2), dim = c(2,3,2))
typeof(x)
#> [1] "double"
class(x)
#> [1] "array"

# integer array ----
x <- array(c(1L,2L), dim = c(2,3,2))
typeof(x)
#> [1] "integer"
class(x)
#> [1] "array"

# logical array ----
x <- array(c(T,F), dim = c(2,3,2))
typeof(x)
#> [1] "logical"
class(x)
#> [1] "array"

# character array ----
```

```
x <- array(c("Paul", "Jane"), dim = c(2,3,2))
typeof(x)
#> [1] "character"
class(x)
#> [1] "array"

# factor ----
x <- factor(c("Paul", "Jane"))
typeof(x)
#> [1] "integer"
class(x)
#> [1] "factor"

# list ----
x <- list(c("Paul", "Jane"), c(2,3,2))
typeof(x)
#> [1] "list"
class(x)
#> [1] "list"

# data frame ----
x <- data.frame(name=c("Paul", "Jane"), score=c(2,3))
typeof(x)
#> [1] "list"
class(x)
#> [1] "data.frame"
```

2.5 Functions

You have already used a number of functions in the previous chapters, including `c()`, `str()`, `matrix()`, `length()`, and `factor()`. However, before we look many more useful functions, it is handy to know how to work with functions in R. When you call a function in R, you use the function name with a number of arguments, which you give inside parentheses to pass information to that function about how it should run and what data it should use. So how do you know what the arguments to a function are? You can either look in the help file—using `?functionName` or `help("functionName")` or you can use a function called `args()`, which will print the arguments to a function in the console. As an example of using a function, we will look at `sample()`. This function allows us to randomly sample a number of values from a vector of given values (this is the R way of selecting balls from an urn). So let's take a look at the arguments to this function:

```
args(sample)
#> function (x, size, replace = FALSE, prob = NULL)
#> NULL
```

You can see that we have four arguments to this function. You will notice that the first two are simply given as `x=` and `size=`, whereas the second two are followed by `=` value. This indicates that they have a default value, so we don't need to supply an alternative. Because `x=` and `size=` do not have a default, we have to tell R what value we want them to take. To know the purpose of the arguments, you will need to take a look at the help files, which will tell you more.

```
?sample
```

In this case, `x=` is the vector that we want to sample from and `size=` is the number of samples we want to take, whereas replace allows us to put values back and we can set the probability of each value with prob. When it comes to calling the function, we can supply the arguments in a number of ways.

To start with, we can name all the arguments in full:

```
sample(x = c("red", "yellow", "green", "blue"), size = 2, replace = FALSE, prob = NULL)
#> [1] "blue"   "yellow"
```

Because `replace=` and `prob=` have default values, this is the same as the following:

```
sample(x = c("red", "yellow", "green", "blue"), size = 2)
#> [1] "yellow" "red"
```

Using this form of complete naming of arguments, we can actually supply them in any order we like. Therefore, the preceding would do the same as this:

```
sample(size = 2, x = c("red", "yellow", "green", "blue"))
#> [1] "blue"   "yellow"
```

It's worth remembering that when you actually run each of these lines, you will most likely get a different result because the function is randomly sampling from the vector `x`. If you provide all the arguments in the same order as the `args()` function gives them, you do not actually need to give the names of the arguments. Therefore, we can also say this:

```
sample(c("red", "yellow", "green", "blue"), 2)
#> [1] "red"  "blue"
```

In reality, you will often see, and use, a combination of naming and ordering of arguments because you will tend to remember what should come first but not the order of other arguments. Therefore, you might see something like the following:

```
sample(c("red", "yellow", "green", "blue"), size = 2, replace = TRUE)
#> [1] "red" "red"
```

2.6 Vectorized operations

Vectorized operations, is one of the features of the R language that make it, that makes it easy to use. It makes very, kind of, nice to write code, without having to do lots of looping, and things like that.

As we've seen earlier, we can add two numeric constants:

```
2 + 3
#> [1] 5
x <- 2
y <- 3
x + y
#> [1] 5
```

The idea with vectorized operations is that things can happen in parallel. For example, suppose we got two vectors here `x` and `y`. `x` is the sequence 1 through 4 and `y` is the sequence 11 through 14.

```
x <- 1:4
y <- 11:14
x
#> [1] 1 2 3 4
y
#> [1] 11 12 13 14
x + y
#> [1] 12 14 16 18
```

And we want to add the two vectors together. Now, when we say we want to add them, what we mean is we want to add the first element of `x` to the first element of `y`, the second element of `x` to the second element of `y`, etc., the third element to the third element. It adds 1 to 11, 2 to 12, 3 to 13, and 4 to 14, so you get the vector 12, 14, 16, 18.

Similarly, you can use the greater than (`>`), or less than symbols (`<`) to, give you logical vectors.

```
x > y
#> [1] FALSE FALSE FALSE FALSE
x < y
#> [1] TRUE TRUE TRUE TRUE
```

Suppose, we have a new `y` vector with only two elements, and we want to add them together.

```
x <- 1:4
y <- 11:12
x
#> [1] 1 2 3 4
y
```

```
#> [1] 11 12
x + y
#> [1] 12 14 14 16
```

There is an important rule in R, *recycling rule*: if two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, our two vectors `x` and `y` have different lengths, and their sum is computed by recycling values of the shorter vector `y`. In this case, when we say we want to add them, what we mean is we want to add 1 to 11, 2 to 12, 3 to 11, and 4 to 12, so you get the vector 12, 14, 14, 16.

Another example is `x > 2`. So well `x` is actually a vector of 4 numbers. So, which number are you comparing to 2? According to the recycling rule, the vectorized operation compares all the numbers to 2, and it gives you a vector of falses and trues depending on which numbers happen to be bigger than 2.

```
x > 2
#> [1] FALSE FALSE  TRUE  TRUE
```

Finally, suppose, we have a `y` vector with three elements.

```
x <- 1:4
y <- 11:13
x
#> [1] 1 2 3 4
y
#> [1] 11 12 13
x + y
#> [1] 12 14 16 15
```

As you can see, we get a warning message: the length of vector `x` is not multiple of length of `y`. In this case, when we say we want to add them, what we mean is we want to add 1 to 11, 2 to 12, 3 to 13, and 4 to 11, so you get the vector 12, 14, 16, 15.

2.7 Creating date sequences

2.7.1 Creating a sequence of numeric values

As we have seen earlier, the colon (`:`) operator in syntax `from:to` generates a sequence from `from=` to `to=` in steps of 1 or -1.

```
1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
11:-2
#>  [1] 11 10  9  8  7  6  5  4  3  2  1  0 -1 -2
1.2:10
#> [1] 1.2 2.2 3.2 4.2 5.2 6.2 7.2 8.2 9.2
```

A more general way of performing the same operation is with the `seq()` function. The first two arguments to `seq()` are the starting and ending values, and the default gap is one. Therefore, the following lines are equivalent:

```
1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
seq(from = 1, to = 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

The advantage of using the `seq()` function is that it has an additional argument, `by=`, that allows you to specify the gap between consecutive sequence values, as shown in the following examples:

```
seq(from = 1, to = 10, by = 0.5) # Sequence from 1 to 10 by 0.5
#>  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5
#> [13]  7.0  7.5  8.0  8.5  9.0  9.5 10.0
seq(from = 2, to = 20, by = 2) # Sequence from 2 to 20 by 2
#>  [1]  2  4  6  8 10 12 14 16 18 20
seq(from = 5, to = -5, by = -2) # Sequence from 5 to -5 by -2
#> [1]  5  3  1 -1 -3 -5
```

These examples illustrate some simple sequences of values. However, let's consider the following examples, where we create a sequence of values from 1.3 to 8.4 by 0.3:

```
seq(from = 1.3, to = 8.4, by = 0.3) # Sequence from 1.3 to 8.4 by 0.3
#>  [1] 1.3 1.6 1.9 2.2 2.5 2.8 3.1 3.4 3.7 4.0 4.3 4.6 4.9 5.2 5.5
#> [16] 5.8 6.1 6.4 6.7 7.0 7.3 7.6 7.9 8.2
```

In this example, note that the last value in the vector is 8.2, whereas we requested a sequence from 1.3 to 8.4. Of course, the reason that the last value is not precisely 8.4 is that the difference between the start and end of the sequence is not divisible by 0.3 (the specified "gap").

If instead we wanted to create a sequence of values from a start point to a particular end point, we could specify a length of the output vector instead of the gap in consecutive sequence values:

```
seq(from = 1.3, to = 8.4, length.out = 10) # Sequence of 10 values from 1.3 to 8.4
#>  [1] 1.300000 2.088889 2.877778 3.666667 4.455556 5.244444
#>  [7] 6.033333 6.822222 7.611111 8.400000
```

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

2.7.2 Creating a Sequence of Repeated Values

We can use the `rep()` function in R to create a vector containing repeated values. The first two arguments to the `rep()` function are the value(s) to repeat and the number of times to repeat the value(s), as shown here:

```
rep(x = "Hello", times = 5) # Repeat "Hello" 5 times
#> [1] "Hello" "Hello" "Hello" "Hello" "Hello"
```

In the last example, we are repeating a single value, but the first argument to `rep()` could be a vector of values.

```
x <- c(1, 2, 3)
rep(x, times = 5) # Repeat the x vector 5 times
#>  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

We can further simplify this example as follows:

```
rep(1:3, times = 5) # Repeat the x vector 5 times
#>  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

In these examples, we repeat a series of values a specific number of times. Alternatively, we can repeat each of the values a specified number of times by supplying a vector value for the second argument the same length as that in the first argument:

```
rep(x = c("A", "B", "C"), times = c(4, 1, 3))
#> [1] "A" "A" "A" "A" "B" "C" "C" "C"
```

In this example, we repeat “A” four times, “B” once, and “C” three times. Using this same approach, we can replace each value of a vector a specific number of times, as shown here:

```
rep(x = c("A", "B", "C"), times = c(3, 3, 3))
#> [1] "A" "A" "A" "B" "B" "B" "C" "C" "C"
```

Alternatively, because the second input is a repeated set of values, this could be written as follows: Click here to view code image

```
rep(x = c("A", "B", "C"), each = 3)
#> [1] "A" "A" "A" "B" "B" "B" "C" "C" "C"
```

As you can see, the `rep()` function can be used to create a variety of vectors with repeated sequences. Let’s quickly recap the three ways of using rep, as illustrated in this section:

```
rep(x = c("A", "B", "C"), times = 3)          # Repeat the vector 3 times
#> [1] "A" "B" "C" "A" "B" "C" "A" "B" "C"
rep(x = c("A", "B", "C"), times = c(4, 1, 3)) # Repeat each value a specific number
#> [1] "A" "A" "A" "A" "B" "C" "C" "C"
rep(x = c("A", "B", "C"), each = 3)           # Repeat each value 3 times
```

```
#> [1] "A" "A" "A" "B" "B" "B" "C" "C" "C"
```

2.7.3 Sequential names

Finally, you can create sequential names from a series of strings or numeric values using the `paste()` function. Let's say we have 10 survey questions, or items, and we want the names of the items to be sequential so they reflect their order in which the respondents were exposed to them.

We can create the prefix of the names and a sequence of values to be the suffix:

```
prefix <- "survey.item"
suffix <- 1:10
```

We can create a vector which takes the prefix and attaches the suffix as a character string. Note; there are two examples below. The first contains no separator (`sep = ""`) between the prefix and suffix; the second example contains a period as the separator (`sep = "."`).

```
paste(prefix, suffix, sep="")
#>  [1] "survey.item1"  "survey.item2"  "survey.item3"
#>  [4] "survey.item4"  "survey.item5"  "survey.item6"
#>  [7] "survey.item7"  "survey.item8"  "survey.item9"
#> [10] "survey.item10"
paste(prefix, suffix, sep=".")
#>  [1] "survey.item.1"  "survey.item.2"  "survey.item.3"
#>  [4] "survey.item.4"  "survey.item.5"  "survey.item.6"
#>  [7] "survey.item.7"  "survey.item.8"  "survey.item.9"
#> [10] "survey.item.10"
```

We can simplify this example as follows:

```
paste("survey.item", 1:10, sep=".")
#>  [1] "survey.item.1"  "survey.item.2"  "survey.item.3"
#>  [4] "survey.item.4"  "survey.item.5"  "survey.item.6"
#>  [7] "survey.item.7"  "survey.item.8"  "survey.item.9"
#> [10] "survey.item.10"
```

We can concatenate two or more vectors:

```
paste(5:1, "cell", 1:5 , sep=".")
#> [1] "5.cell.1" "4.cell.2" "3.cell.3" "2.cell.4" "1.cell.5"
```

2.8 Subsetting

In this section, we look at the ways in which to extract subsets of data from an object. We can achieve this using square brackets (`[ ]`), double square brackets (`[[ ]]`) and dollar sign (`$`).

There are three operators that can be used to extract subsets of R objects.

Operator	Description
`[`	Always returns an object of the same class as the original. It can be used to select multiple elements of an object.
`[[`	Extracts elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
`$`	Extract elements of a list or data frame by literal name.

Different data structures cab be used vary index operators and index vectors, as shown below:

```r
# subsetting with []
obj.vector[index.vector]
obj.factor[index.vector]
obj.list[index.vector]
obj.matrix[index.vector.1, index.vector.2]
obj.array.3D[index.vector.1, index.vector.2, index.vector.3]
obj.data.frame[index.vector]
obj.data.frame[index.vector.1, index.vector.2]

# subsetting with [[]]
obj.vector[[single.index]]
obj.factor[[single.index]]
obj.list[[single.index]]
obj.data.frame[[single.index]]

# subsetting with $
obj.list$element.name
obj.data.frame$element.name
```

As with index vectors, you can put one of five input types in the square brackets (`[ ]`), as shown below:

Index vectors	Effect
Blank	All values are returned
A vector of positive integers	Used as an index to return
A vector of negative integers	Used as an index to omit
A vector of logical values	Only corresponding `TRUE` elements are returned
A vector of character values	Refers to the names of element to return

Single index can be an integer or a string. To illustrate the subsetting of objects, we will discuss the vector and data frame subsetting.

2.8.1 Subsetting vector

We can index any object in R. We start the vector, then we are moving on to data frame. Subsetting basically comes down to selecting parts of your vector to end up with a new vector, which is a subset of the original vector.

We have a `name` vector with names of observed people.

```
name <- c("Paul", "Jane", "Mark", "Ann")
```

After Ctrl+Enter, we can get the whole vector.

Suppose you want to select the first element from this vector, corresponding to the first person's name. You can use square brackets `[]` for this.

```
name[1]
#> [1] "Paul"
```

The number one inside the square brackets indicates that you want to get the first element from the `name` vector. The result is again a vector, because a single string is actually a vector of length 1. This new vector contains the string `"Paul"`. If you instead want to select the third element, corresponding to third person's name, you could code remain followed by 3 in square brackets.

```
name[3]
#> [1] "Mark"
```

Suppose now you want to select the elements in the vector that give the first three people's names. Instead of using a single number inside the square brackets, you can use a vector to specify which indices you want to select. You use vector containing 1, 2 and 3 inside the square brackets.

```
name[c(1, 2, 3)]  # or name[1:3]
#> [1] "Paul" "Jane" "Mark"
```

How the resulting vector is ordered depends on the order of the indices inside the selection vector. If you change `c(1, 2, 3)` to `c(2, 3, 1)`, you will get a vector where the second person comes first.

```
name[c(2, 3, 1)]
#> [1] "Jane" "Mark" "Paul"
```

As we mentioned, we can create regular sequences. For example the colon (`:`) operator can create the `c(1, 2, 3)` with `1:3`. Or construction `3:1` may be used to generate a sequence backwards. So, we can use these operation inside square brackets.

```
name[1:3]
#> [1] "Paul" "Jane" "Mark"
name[3:1]
#> [1] "Mark" "Jane" "Paul"
```

2.8.2 Subsetting data frames

In data frames we can use single brackets with two indices inside, because data frame is two dimensional. First, print the whole data frame.

```
# creating inline data frame
d <- data.frame(name=c("Paul", "Jane", "Mark", "Ann"),
                gender=c("male", "female", "male", "female"),
                height=c(184, 167, 111, 172),
                age=c(32L, 19L, 13L, 78L),
                child=c(T, F, F, F),
                cars=c(0, 2, 1, 2))
d
#>   name gender height age child cars
#> 1 Paul   male    184  32  TRUE    0
#> 2 Jane female    167  19 FALSE    2
#> 3 Mark   male    111  13 FALSE    1
#> 4  Ann female    172  78 FALSE    2
```

To select the height of Jane, who is on row 2 in the data frame, you can use the single brackets with two indices inside. The row, index 2, comes first, and the column, index 3, comes second. They will be separated by comma.

```
d[2, 3]
#> [1] 167
```

Indeed, Jane is 167 cm tall. You can also use the column names to refer to the columns of data frame.

```
d[2, "height"]
#> [1] 167
```

Of course we select the height and age information on Jane and Mark.

```
d[c(2,3), "height"]
#> [1] 167 111
```

And Of course we do the same on Paul and Ann. Additionally, we can add the name of the people to the end.

```
d[c(2,3), c("height", "name")]
#>   height name
#> 2    167 Jane
#> 3    111 Mark
```

We can also choose to omit one of the two indices, to end up with an entire row or an entire column. If you want to have all information on Jane, you can use this command:

```
d[2, ]
#>   name gender height age child cars
#> 2 Jane female    167  19 FALSE    2
```

The result is a data frame with a single observation, because there has to be a way to store the different types.

On the other hand, to get the entire age column, you could use this command:

```
d[, 4]
#> [1] 32 19 13 78
```

Here, the result is a vector, because columns contain elements of the same type. We can prevent drop dimensions with `drop=F`.

```
d[, 4, drop=F]
#>   age
#> 1  32
#> 2  19
#> 3  13
#> 4  78
```

Another way to select only one columns is the `$` (dollar sing) operator.

```
d$cars
#> [1] 0 2 1 2
```

2.9 String data

We can often find ourselves having to perform string manipulation tasks in R, including creation of character strings and searching for patterns in character strings. In this section, we look at some of the functions in the *Base R* installation.

2.9.1 Simple Character Manipulation

Some of the basic manipulations you'll want to perform are counting characters, extracting substrings, and combining elements to create or update a string. Let's start with counting characters. You do this using the `nchar()` function, simply providing the string that you are interested in:

```
fruits <- "apples oranges pears"
nchar(fruits)
#> [1] 20
```

Notice that all characters are counted, including the spaces. To extract substrings, you use the `substring()` function. Here, you need to give the string along with the start and end points for the substring. You can extract multiple substrings by giving the vectors of the start and end points.

```
substring(text = fruits, first = 1, last = 6)
#> [1] "apples"
fruits.2 <- substring(text = fruits, first = c(1, 8, 16), last = c(6, 14, 20))
fruits.2
#> [1] "apples"  "oranges" "pears"
```

Finally, you can create a character string from a series of strings or numeric values using the `paste()` function. You can provide as many strings and objects as you wish to the paste function and they will all be converted to character data and pasted together. Like with many R functions, you can pass vectors to the paste function. Here's an example:

```
paste(5, "apples")
#> [1] "5 apples"
nfruits <- c(5, 9, 2)
paste(nfruits, fruits.2)
#> [1] "5 apples"  "9 oranges" "2 pears"
```

You can use the argument `sep=` to change the separator between the pasted strings, which as you can see in the preceding example is a space by default, like so:

```
paste(fruits.2, nfruits, sep = " = ")
#> [1] "apples = 5"  "oranges = 9" "pears = 2"
```

2.9.2 Searching and Replacing

Two of the most useful functions for working with character data are the functions `grep()` and `gsub()`. These functions allow you to search elements of a vector for a particular pattern (`grep()`) and replace a particular pattern with a given string (`gsub()`). You search for patterns using regular expressions (that is, a pattern that describes the character string). Much more information on regular expressions can be found in the R help pages for the function `regex()`. If you are familiar with Perl expressions, you can use these along with the argument `perl = TRUE`. Let's start by looking at the function `grep()`. The first argument that we are going to give is the pattern to search for, which can be as simple as the string "red". The second argument will be the vector to search.

```
colourStrings <- c("green", "blue", "orange", "red", "yellow", "lightblue", "navyblue"
grep(pattern = "red", x = colourStrings, value = TRUE)
#> [1] "red"       "indianred"
```

In this example, we have used an additional argument, `value=`. This allows us to return the actual values of the vector that include the pattern rather than simply the index of their position in the vector. Alternatives to `grep(values=TRUE)`:

```
grep(pattern = "red", x = colourStrings)  # returns a vector of the indices of the elements of x
#> [1] 4 8
grepl(pattern = "red", x = colourStrings) # returns a logical vector (match or not for each eleme
#> [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

Some more examples of using the `grep()` function, with a variety of regular expressions, are shown below:

```
colourStrings <- c("green", "blue", "orange", "red", "yellow", "lightblue", "navyblue", "indianre
grep("^red", colourStrings, value = TRUE)
#> [1] "red"
grep("red$", colourStrings, value = TRUE)
#> [1] "red"       "indianred"
grep("r+", colourStrings, value = TRUE)
#> [1] "green"     "orange"    "red"       "indianred"
grep("e{2}", colourStrings, value = TRUE)
#> [1] "green"
```

You can see how the symbols `^` and `$` have been used to mark the start and end of the string. In the example in line 2, we are specifying that immediately following the start of the string is the pattern `"red"`, whereas in line 3 the string ends straight after the pattern `"red"`. The examples in lines 4 and 5 show how to specify that something must appear a given number of times. In line 4, the `+` indicates that the letter `r` should appear at least once in the string. In line 5, the `{2}` following the e indicates that there should be two occurrences of the letter.

The `gsub()` function, which allows you to substitute a pattern for a value, is very similar, because you also use regular expressions to search for the pattern. The only additional information you need to give is what to substitute in its place. Here is an example:

```
gsub(pattern = "red", replacement = "brown", x = colourStrings)
#> [1] "green"       "blue"        "orange"      "brown"
#> [5] "yellow"      "lightblue"   "navyblue"    "indianbrown"
```

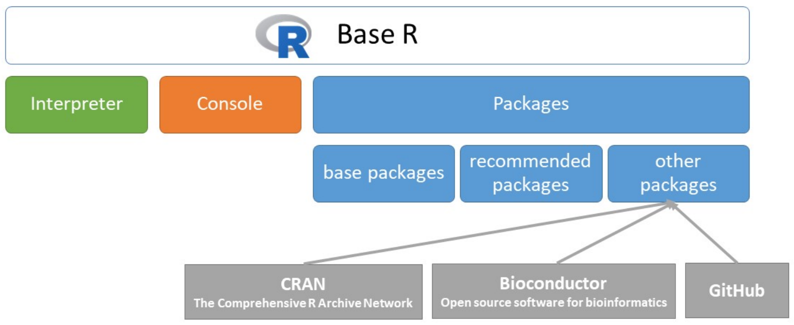
As with grep, you can use any regular expression to match the pattern you wish to replace.

2.10 Packages in R

R's functionality is distributed among many *packages*. Each has a certain focus; for example, the **stats** package contains functions that apply common statistical methods, and the **graphics** package has functions concerning plotting. When you download R, you automatically get a set of *base* and *recommended* packages, which can be seen in the "library" subdirectories of the R installation.

```
.libPaths() # paths to packages
#> [1] "C:/Users/RStudio/Documents/R/win-library/4.0"
#> [2] "C:/Program Files/R/R-4.0.4/library"
```

File paths of `.libPaths()` are used for getting or setting the library trees that R knows about (and hence uses when looking for packages). These core R packages represent a small subset of all the packages you can use with R. In fact, at the time of writing, there are more than 17000. These other packages we call *other* packages, because you have to add them to R, from CRAN, Bioconductor or GitHub yourself.



```
pkg <- installed.packages()
table(pkg[,"Priority"], useNA = "ifany") # number of installed packages
#>
#>        base recommended        <NA>
#>          14          15        1684
```

As we can see, there are 14 base packages, 15 recommended packages in R, and I have 1684 other packages installed before.

We can print the name of base and recommended packages:

```
rownames(pkg)[pkg[,"Priority"] %in% "base"]         # base packages
#>  [1] "base"      "compiler"  "datasets"  "graphics"  "grDevices"
#>  [6] "grid"      "methods"   "parallel"  "splines"   "stats"
#> [11] "stats4"    "tcltk"     "tools"     "utils"
rownames(pkg)[pkg[,"Priority"] %in% "recommended"]  # recommended packages
#>  [1] "boot"       "class"       "cluster"     "codetools"
#>  [5] "foreign"    "KernSmooth" "lattice"    "MASS"
#>  [9] "Matrix"     "mgcv"       "nlme"       "nnet"
#> [13] "rpart"      "spatial"    "survival"
```

Only a small subset of the installed packages is actually loaded when you start an R session. This helps reduce the start-up time and avoid a behavior known as masking. The `search()` function shows you which packages are loaded on your machine.

```
search()  # loaded packages (with "package:" prefix)
#>  [1] ".GlobalEnv"        "package:dplyr"     "package:MASS"
#>  [4] "package:stats"     "package:graphics"  "package:grDevices"
#>  [7] "package:utils"     "package:datasets"  "package:methods"
#> [10] "Autoloads"         "package:base"
```

During starting up the R, for examle the **base**, **methods**, **datasets**, and **utils** packages are loaded automatically.

2.10.1 Load packages

To load any of installed packages, call the `library()` function. If R cannot find the specified package library, it will produce an error. For example **MASS** is a pre-installed package, part of the recommended packages. We can load it successfully.

```
library(MASS)    # load MASS package
```

We can check the loaded packages, the return value of `search()` contains the `"package:MASS"` string.

```
search()
#>  [1] ".GlobalEnv"        "package:dplyr"     "package:MASS"
#>  [4] "package:stats"     "package:graphics"  "package:grDevices"
#>  [7] "package:utils"     "package:datasets"  "package:methods"
#> [10] "Autoloads"         "package:base"
```

But, **psych** or **DescTools** packages are part of *other* packages, the `library()` function calls may cause error message (in that case we did not install them before).

```
library(psych)      # load psych package
library(DescTools)  # load DescTools package
```

2.10.2 Install packages

To load these packages successfully, we need to install them.

These packages are on CRAN, so we type in:

```
install.packages("psych")          # installing from CRAN
install.packages("DescTools")
```

To install packages from Bioconductor, first type the following:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install()
```

Install specific packages, e.g., **GenomicFeatures** and **AnnotationDbi**, with

```
BiocManager::install(c("GenomicFeatures", "AnnotationDbi"))
```

The third case is installing from GitHub. You can install **emo** from GitHub with:

```
# install.packages("devtools")
devtools::install_github("hadley/emo")
```

So we can insert: .

It is worth to see pacman: A package management tools for R or remotes if you find an elegant way to handle packages.

Finally, we can check repository of our installed packages:

```
inst.pkg <- installed.packages()[,1]  # all installed packages
cran.pkg <- available.packages(
  contrib.url(repos = "https://cran.rstudio.com/",
              type = "both"))         # all CRAN packages
bioc.pkg <- BiocManager::available()  # all CRAN & Bioconductor packages

library(dplyr)
repos <- case_when(
  inst.pkg %in% cran.pkg ~ "CRAN",
  !(inst.pkg %in% cran.pkg) & (inst.pkg %in% bioc.pkg) ~ "Bioconductor",
  TRUE ~ "GitHub?"
)
df.pkg <- data.frame(inst.pkg, repos)
table(df.pkg$repos)
#>
#> Bioconductor         CRAN      GitHub?
#>           42         1646           25
```

2.10.3 Masking

Masking occurs when two or more “environments” on the search path contain one or more objects with the same name. Whenever we refer to an object by typing its name, R looks in each of the loaded environments on the search path for that object in turn, starting with the *Global Environment*. If R finds an object with the name it is looking for, it stops searching. Any objects it doesn't find have been hidden, or "masked."

To avoid any potential masking issues, it is possible to reference an object within a package directly by using the `packageName::objectName` syntax, for example,

```
base::pi
#> [1] 3.141593
str(MASS::survey)
#> 'data.frame':    237 obs. of  12 variables:
#>  $ Sex   : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
#>  $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
#>  $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
#>  $ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
#>  $ Fold  : Factor w/ 3 levels "L on R","Neither",..: 3 3 1 3 2 1 1 3 3 3 ...
#>  $ Pulse : int  92 104 87 NA 35 64 83 74 72 90 ...
#>  $ Clap  : Factor w/ 3 levels "Left","Neither",..: 1 1 2 2 3 3 3 3 3 3 ...
#>  $ Exer  : Factor w/ 3 levels "Freq","None",..: 3 2 2 2 3 3 1 1 3 3 ...
#>  $ Smoke : Factor w/ 4 levels "Heavy","Never",..: 2 4 3 2 2 2 2 2 2 2 ...
#>  $ Height: num  173 178 NA 160 165 ...
#>  $ M.I   : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ...
#>  $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
```

2.11 Internal help

The `help()` function can be used to display help on a function or indeed any R object. If you know the name of the object you require help with, you can use a function `help()` or its shorthand, `?`.

```
help(mean)
?mean
```

A general search of all help files can be achieved using either the `help.search()` function or the shorthand version, `??`.

```
help.search("test")
??test
```

You can also read about any packages with `help()`function.

```
help(package="MASS")
```

Chapter 3

Getting started with a data analysis

3.1 Terminology

Before we begin data management tasks, we need a few vocabulary terms would be useful to discuss. Data scientists were usually interested in the characteristics and behaviors of humans and organizations. To understand these things, scientists often measured and recorded information about people or organizations.

Dataset 1 - Marijuana legalization

For example, a data scientist working on social science might be interested in understanding whether age is related to votes for marijuana legalization. To get this information, in several years, including 2016, the GSS survey included a question asking the survey participants whether they support marijuana legalization.

The GSS question was worded as follows:

- Do you think the use of marijuana should be legal or not?

Below the question, the different response options were listed:

- legal,
- not legal,
- don't know (`DK`),
- no answer (`NA`),
- not applicable (`IAP`).

The GSS Data Explorer (https://gssdataexplorer.norc.org) allows people to create a free account and browse the data that have been collected in the surveys.

We used the Data Explorer to select the marijuana legalization question and a question about age. The age is important, since marijuana legalization had been primarily up to voters so far, the success of ballot initiatives in the future will depend on the support of people of voting age. If younger people are more supportive, this suggests that over time, the electorate will become more supportive as the old electorate decreases.

We saved age and vote data from the GSS and made the data file with the file name `legal_weed_age_GSS2016_ch1.csv`. You can see the first 6 rows from this data file:

Table 3.1: Data set for marijuana legalization

grass	age
IAP	47
LEGAL	61
NOT LEGAL	72
IAP	43
LEGAL	55
LEGAL	53

As you can see, each person is an *observation*, and there are two *variables*, voting behavior (`grass`) and `age`. In a typical *dataset*, observations are the rows and variables are the columns.

Example 2 - Student survey

Another example a data frame contains the responses of 237 Statistics I. students at the University of Adelaide to a number of questions. It contains a lot of variables:

- `Sex` - The sex of the student. (Factor with levels “Male” and “Female”.)
- `Wr.Hnd` - span (distance from tip of thumb to tip of little finger of spread hand) of writing hand, in centimetres.
- `NW.Hnd` - span of non-writing hand.
- `W.Hnd` - writing hand of student. (Factor, with levels “Left” and “Right”.)
- `Fold` - “Fold your arms! Which is on top” (Factor, with levels “R on L”, “L on R”, “Neither”.)
- `Pulse` - pulse rate of student (beats per minute).
- `Clap` - ‘Clap your hands! Which hand is on top?’ (Factor, with levels “Right”, “Left”, “Neither”.)
- `Exer` - how often the student exercises. (Factor, with levels “Freq” (frequently), “Some”, “None”.)
- `Smoke` - how much the student smokes. (Factor, levels “Heavy”, “Regul” (regularly), “Occas” (occasionally), “Never”.)
- `Height` - height of the student in centimetres.

- `M.I` - whether the student expressed height in imperial (feet/inches) or metric (centimetres/metres) units. (Factor, levels "Metric", "Imperial".)
- `Age` - age of the student in years.

Table 3.2: Student survey

Sex	Wr.Hnd	NW.Hnd	W.Hnd	Fold	Pulse	Clap	Exer	Smoke	Height	M.I	Age
Female	18.5	18.0	Right	R on L	92	Left	Some	Never	173.00	Metric	18.250
Male	19.5	20.5	Left	R on L	104	Left	None	Regul	177.80	Imperial	17.583
Male	18.0	13.3	Right	L on R	87	Neither	None	Occas	NA	NA	16.917
Male	18.8	18.9	Right	R on L	NA	Neither	None	Never	160.00	Metric	20.333
Male	20.0	20.0	Right	Neither	35	Right	Some	Never	165.00	Metric	23.667
Female	18.0	17.7	Right	L on R	64	Right	Some	Never	172.72	Imperial	21.000

In statistics data are organized in what we call a *data matrix* or *dataset*, where each row represents an observation or a case and each column represents a variable. If you ever use spreadsheets, for example an Excel spreadsheet, this representation should be familiar to you as well. There are two types of variables, numerical and categorical. Numerical, in other words, quantitative variables, take on numerical values. It is sensible to add, subtract, take averages, etc., with these values. Categorical, or qualitative variables, take on a limited number of distinct categories. These categories can be identified with numbers or labels, but it wouldn't be sensible to do arithmetic operations with these values.

Numerical variables can further be categorized as continuous or discrete. Continuous numerical variables are usually measured, such as height, and they can take on any numerical value. While we tend to round our height when we record it, it's actually measured on a continuous scale.

Discrete numerical variables are generally counted, such as the number of cars a house. These can only be whole, non-negative numbers.

Categorical variables that have ordered levels are called ordinal. Think about a survey question where you're asked how satisfied you are with the customer service you received, and the options are "very unsatisfied", "unsatisfied", "neutral", "satisfied", or "very satisfied". These levels have an inherent ordering, and hence the variable would be called ordinal. If the levels of a categorical variable do not have an inherent ordering to them, then the variable is simply called nominal.

These terms in statistics have a pair in R. Dataset corresponds to data frame. Variable corresponds to columns of data frame. Discrete variables must be an integer or double vector in R. Continuous variables must be an integer or double vector in R, as well. Nominal or ordinal variables must be factor in R.

To sum it up, study the list below.

Terms in statistics - **terms in R** - *example* :

- Data matrix, dataset - **data frame** - *marijuana legalization dataset and survey dataset*
- Variable - **columns of data frame** - *each column of two dataset: grass, age, Sex, Wr.Hnd, etc.*
  - numerical / quantitative
    - discrete - **integer or double vector** - *Pulse*
    - continuous - **integer or double vector** - *age, Wr.Hnd, NW.Hnd, Height, Age*
  - categorical / qualitative
    - ordinal - **factor** - *Exer, Smoke*
    - nominal - **factor** - *grass, Sex, W.Hnd, Fold, Clap, M.I*

3.2 Read and write data

R has an extensive range of functions to import many types of data files. For example, R can import data from from text files, from Microsoft Excel, from popular statistical packages, and from web sites.

3.2.1 Importing data from a delimited text file

You can import data from delimited text files using `read.table()`, a function that reads a file in table format and saves it as a data frame. Each row of the table appears as one line in the file. The syntax is

```
mydataframe <- read.table(file, options)
```

where file is a delimited file and the options are parameters controlling how data is processed. The most common options are listed below:

- `header=` - A logical value indicating whether the file contains the variable names in the first line.
- `sep=` - The delimiter separating data values. The default is `sep=""`, which denotes one or more spaces, tabs, new lines, or carriage returns. Use `sep=","` to read comma-delimited files, `sep="\t"` to read tab-delimited files, and `sep=";"` to read semicolon-delimited files
- `dec=` - The character `","` or `"."` used in the file for decimal points
- `quote=` - Character(s) used to delimit strings that contain special characters. By default this is either double (`"`) or single (`'`) quotes.
- `comment.char=` - A character vector of length one containing a single character or an empty string. Use ”” to turn off the interpretation of comments altogether.

- `fileEncoding=` - Character string for encoding name, e.g. `"UTF-8"`, `"UTF-8-BOM"` or `"latin2"`.

Consider a text file named `legal_weed_age_GSS2016_ch1.csv` containing voters' response for marijuana legalization question and age. Each line of the file represents a student. The first line contains the variable names, separated with commas. Each subsequent line contains a voter's information, also separated with commas. The first few lines of the file are as follows:

```
grass,age
IAP,47
LEGAL,61
NOT LEGAL,72
IAP,43
LEGAL,55
LEGAL,53
IAP,50
NOT LEGAL,23
```

The file can be imported into a data frame using the following code:

```
# read the GSS 2016 data
gss.2016 <- read.table(file = "data/legal_weed_age_GSS2016_ch1.csv",
                       header = T, sep = ",", fileEncoding = "UTF-8-BOM")
```

The results are as follows:

```
head(gss.2016) # first 6 rows
#>       grass age
#> 1       IAP  47
#> 2     LEGAL  61
#> 3 NOT LEGAL  72
#> 4       IAP  43
#> 5     LEGAL  55
#> 6     LEGAL  53
str(gss.2016)
#> 'data.frame':    2867 obs. of  2 variables:
#>  $ grass: chr  "IAP" "LEGAL" "NOT LEGAL" "IAP" ...
#>  $ age  : chr  "47" "61" "72" "43" ...
```

There are several interesting things to note about how the data is imported. By default, `read.table()` do not convert character variables to factors. You can suppress this behaviour in a number of ways. Including the option `stringsAsFactors=TRUE` turns off this behaviour for all character variables. The variable `age` is a character vector, which is not desirable. Age is a continuous variable, so it must be a numeric in R. We will discuss this issue in detail later.

3.2.2 Importing data from Excel

The best way to read an Excel file is to import Excel worksheets directly using the **rio** package. Be sure to download and install it before you first use it. Alternatively, export it to a comma-delimited file from Excel and import it into R using the method described earlier.

The **rio** package can be used to read, write, many file formats. The `import()` function imports a worksheet into a data frame. The simplest format is

```
import(file)
```

where `file=` is the path to an Excel workbook.

Let's import the student survey: imports the first worksheet from the workbook `survey.xlsx` stored on project `data` directory and saves it as the data frame `survey`.

```
library(rio)
survey <- import(file = "data/survey.xlsx")
```

The results are as follows:

```
head(survey) # first 6 rows
#>      Sex Wr.Hnd NW.Hnd W.Hnd    Fold Pulse    Clap Exer Smoke
#> 1 Female   18.5   18.0 Right  R on L    92    Left Some Never
#> 2   Male   19.5   20.5  Left  R on L   104    Left None Regul
#> 3   Male   18.0   13.3 Right  L on R    87 Neither None Occas
#> 4   Male   18.8   18.9 Right  R on L    NA Neither None Never
#> 5   Male   20.0   20.0 Right Neither    35   Right Some Never
#> 6 Female   18.0   17.7 Right  L on R    64   Right Some Never
#>   Height      M.I    Age
#> 1 173.00   Metric 18.250
#> 2 177.80 Imperial 17.583
#> 3     NA     <NA> 16.917
#> 4 160.00   Metric 20.333
#> 5 165.00   Metric 23.667
#> 6 172.72 Imperial 21.000
str(survey)
#> 'data.frame':    237 obs. of  12 variables:
#>  $ Sex   : chr  "Female" "Male" "Male" "Male" ...
#>  $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
#>  $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
#>  $ W.Hnd : chr  "Right" "Left" "Right" "Right" ...
#>  $ Fold  : chr  "R on L" "R on L" "L on R" "R on L" ...
#>  $ Pulse : num  92 104 87 NA 35 64 83 74 72 90 ...
#>  $ Clap  : chr  "Left" "Left" "Neither" "Neither" ...
#>  $ Exer  : chr  "Some" "None" "None" "None" ...
#>  $ Smoke : chr  "Never" "Regul" "Occas" "Never" ...
#>  $ Height: num  173 178 NA 160 165 ...
#>  $ M.I   : chr  "Metric" "Imperial" NA "Metric" ...
#>  $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
```

As you can see, the variable `Sex`, `W.Hnd`, etc. are a character vectors, which is not desirable. They are a categorical variables, so they must be factors in R.

3.2.3 Exporting data from R

So far, we reviewed a wide range of methods for importing data into R. But sometimes you'll want to go the other way - exporting data from R - so that data can be archived or imported into external applications. Now, you'll learn how to output an R object to a delimited text file, an Excel spreadsheet, or a statistical application (such as SPSS, SAS, or Stata).

You can use the `write.table()` function to output an R object to a delimited text file. The format is

```
write.table(x, outfile, sep=delimiter, quote=TRUE, na="NA")
```

where `x` is the object and `outfile` is the target file. For example, the statement

```
write.table(x = survey, file = "output/data/survey.txt", sep = "\t",
            dec = ",", row.names = FALSE, quote = FALSE)
```

saves the dataset `survey` to a tab-delimited file named `survey.txt` in the project `output/data` directory. Replacing `sep="\t"` with `sep=";"` saves the data in a semicolon-delimited file. By default, strings are enclosed in quotes (`""`) and missing values are written as `NA`. We will not print row names (`row.names = FALSE`) and quote (`quote = FALSE`) in the output text file.

The `export()` function in the **rio** package can be used to save an R data frame to an Excel workbook. For example, the statements

```
library(rio)
export(x = gss.2016, file = "output/data/gss.xlsx")
```

export the data frame `gss.2016` to a worksheet (Sheet 1 by default) in an Excel workbook named `gss.xlsx` in the project `output/data` directory. By default, the variable names in the dataset are used to create column headings in the spreadsheet, and row names are placed in the first column of the spreadsheet. If `gss.xlsx` already exists, it's overwritten.

The `export()` function in the **rio** package can be used to export a data frame to an external statistical application. For example, the code

```
library(rio)
export(x = survey, file = "output/data/survey.sav")
```

exports the data frame `survey` into an SPSS data file named `survey.sav`.

Please study carefully the following codes and outputs:

```
# Export (tab or semicolon) delimited text files with and withot encoding
write.table(x = survey, file = "output/data/survey.txt", sep = "\t",
            dec = ",", row.names = FALSE, quote = FALSE)
write.table(x = survey, file = "output/data/survey.csv", sep = ";",
            dec = ",", row.names = FALSE, quote = FALSE)
write.table(x = survey, file = "output/data/survey_utf-8.txt", sep = "\t",
            dec = ",", row.names = FALSE, quote = FALSE, fileEncoding = "UTF-8")
write.table(x = survey, file = "output/data/survey_latin2.txt", sep = "\t",
            dec = ",", row.names = FALSE, quote = FALSE, fileEncoding = "latin2")
write.table(x = survey, file = "output/data/survey_utf-8.csv", sep = ";",
            dec = ",", row.names = FALSE, quote = FALSE, fileEncoding = "UTF-8")
write.table(x = survey, file = "output/data/survey_latin2.csv", sep = ";",
            dec = ",", row.names = FALSE, quote = FALSE, fileEncoding = "latin2")

# Export Excel and SPSS files
library(rio)
export(x = survey, file = "output/data/survey.xlsx")
export(x = survey, file = "output/data/survey.sav")
export(x = gss.2016, file = "output/data/gss.xlsx")
export(x = gss.2016, file = "output/data/gss.sav")
```

3.3 Data manipulation

In the previous chapter, we covered a variety of methods for importing data into R. Unfortunately, getting your data in the rectangular arrangement of a matrix or data frame is only the first step in preparing it for analysis. In this early stage we try to get as much information as we can.

3.3.1 Get information

When working with (large) data frames, you must first develop a clear understanding of the structure and main elements of the data set. Therefore, it can often be useful to show only a small part of the entire data set. To do this in R, you can use the functions `head()` or `tail()`. The `head()` function shows the first part of the data frame. The `tail()` function shows the last part. Both functions print a top line called the *header* which contains the names of the different variables in the data set.

```
head(gss.2016)
#>       grass age
#> 1       IAP  47
#> 2     LEGAL  61
#> 3 NOT LEGAL  72
#> 4       IAP  43
#> 5     LEGAL  55
#> 6     LEGAL  53
tail(gss.2016, n = 3)
#>          grass age
#> 2865     LEGAL  87
#> 2866       IAP  55
#> 2867 NOT LEGAL  72
```

Another method to get a rapid overview of the data is the `str()` function. The `str()` function shows the structure of the data set.

```
str(gss.2016)
#> 'data.frame':    2867 obs. of  2 variables:
#>  $ grass: chr  "IAP" "LEGAL" "NOT LEGAL" "IAP" ...
#>  $ age  : chr  "47" "61" "72" "43" ...
```

For a data frame it gives the following information:

- The total number of observations (e.g. 2867 voters)
- The total number of variables (e.g. 2 variables)
- A full list of the variables names (`grass`, `age`)
- The data type of each variable (`chr`)
- The first observations

When you receive a new data frame, applying the `str()` function is often the first step. It is a great way to get more insight into the data set before deeper analysis.

Please study carefully the following codes and outputs:

```
str(gss.2016)        # Structure of an Arbitrary R Object
#> 'data.frame':    2867 obs. of  2 variables:
#>  $ grass: chr  "IAP" "LEGAL" "NOT LEGAL" "IAP" ...
#>  $ age  : chr  "47" "61" "72" "43" ...
head(gss.2016)       # Return the First Parts of an Object
#>       grass age
#> 1       IAP  47
#> 2     LEGAL  61
#> 3 NOT LEGAL  72
#> 4       IAP  43
#> 5     LEGAL  55
#> 6     LEGAL  53
dim(gss.2016)        # Dimensions of an Object
#> [1] 2867    2
ncol(gss.2016)       # The Number of Rows of a data frame
#> [1] 2
nrow(gss.2016)       # The Number of Columns of a data frame
#> [1] 2867
```

```
names(gss.2016)       # The Column Names of an Object
#> [1] "grass" "age"
typeof(gss.2016)      # Type of an Object
#> [1] "list"
class(gss.2016)       # Class of an Object
#> [1] "data.frame"
str(survey)
#> 'data.frame':    237 obs. of  12 variables:
#>  $ Sex   : chr  "Female" "Male" "Male" "Male" ...
#>  $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
#>  $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
#>  $ W.Hnd : chr  "Right" "Left" "Right" "Right" ...
#>  $ Fold  : chr  "R on L" "R on L" "L on R" "R on L" ...
#>  $ Pulse : num  92 104 87 NA 35 64 83 74 72 90 ...
#>  $ Clap  : chr  "Left" "Left" "Neither" "Neither" ...
#>  $ Exer  : chr  "Some" "None" "None" "None" ...
#>  $ Smoke : chr  "Never" "Regul" "Occas" "Never" ...
#>  $ Height: num  173 178 NA 160 165 ...
#>  $ M.I   : chr  "Metric" "Imperial" NA "Metric" ...
#>  $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
head(survey)
#>      Sex Wr.Hnd NW.Hnd W.Hnd    Fold Pulse    Clap Exer Smoke
#> 1 Female   18.5   18.0 Right  R on L    92    Left Some Never
#> 2   Male   19.5   20.5  Left  R on L   104    Left None Regul
#> 3   Male   18.0   13.3 Right  L on R    87 Neither None Occas
#> 4   Male   18.8   18.9 Right  R on L    NA Neither None Never
#> 5   Male   20.0   20.0 Right Neither    35   Right Some Never
#> 6 Female   18.0   17.7 Right  L on R    64   Right Some Never
#>   Height      M.I    Age
#> 1 173.00   Metric 18.250
#> 2 177.80 Imperial 17.583
#> 3     NA     <NA> 16.917
#> 4 160.00   Metric 20.333
#> 5 165.00   Metric 23.667
#> 6 172.72 Imperial 21.000
dim(survey)
#> [1] 237  12
ncol(survey)
#> [1] 12
nrow(survey)
#> [1] 237
names(survey)
#>  [1] "Sex"    "Wr.Hnd" "NW.Hnd" "W.Hnd"  "Fold"   "Pulse"
#>  [7] "Clap"   "Exer"   "Smoke"  "Height" "M.I"    "Age"
typeof(survey)
```

```
#> [1] "list"
class(survey)
#> [1] "data.frame"
```

3.3.2 Data type conversions

As you have known, in R, you use numeric vectors to represent quantitative variables, and you use factors to represent categorical variables. In the data frame `gss.2016`, the variable `grass` is character vector, but it should be a factor. R provides a set of functions to identify an object's data type and convert it to a different data type. You can use the function `factor()` to convert from character or numeric to factor.

```
str(gss.2016)                              # grass is character
#> 'data.frame':    2867 obs. of  2 variables:
#>  $ grass: chr  "IAP" "LEGAL" "NOT LEGAL" "IAP" ...
#>  $ age  : chr  "47" "61" "72" "43" ...
gss.2016$grass <- factor(gss.2016$grass)  # convert
str(gss.2016)                              # grass is factor
#> 'data.frame':    2867 obs. of  2 variables:
#>  $ grass: Factor w/ 4 levels "DK","IAP","LEGAL",..: 2 3 4 2 3 3 2 4 2 4 ...
#>  $ age  : chr  "47" "61" "72" "43" ...
```

The continuous variable `age` is also character, but it should be a numeric. What is the problem with age variable? Use the `unique()` and `table()` functions.

```
unique(gss.2016$age)
#>  [1] "47"          "61"          "72"          "43"
#>  [5] "55"          "53"          "50"          "23"
#>  [9] "45"          "71"          "33"          "86"
#> [13] "32"          "60"          "76"          "56"
#> [17] "62"          "31"          "58"          "37"
#> [21] "25"          "22"          "74"          "75"
#> [25] "68"          "46"          "35"          "59"
#> [29] "79"          "40"          "44"          "36"
#> [33] "70"          "28"          "20"          "41"
#> [37] "42"          "57"          "26"          "51"
#> [41] "39"          "27"          "30"          "29"
#> [45] "80"          "49"          "78"          "52"
#> [49] "66"          "89 OR OLDER" "54"          "48"
#> [53] "81"          "69"          "21"          "64"
#> [57] "38"          "65"          "67"          "84"
#> [61] "34"          "77"          "19"          NA
#> [65] "83"          "73"          "63"          "24"
#> [69] "82"          "85"          "87"          "18"
#> [73] "88"
```

```
table(gss.2016$age, useNA = "ifany")
#> 
#>          18          19          20          21          22 
#>           7          33          26          33          44 
#>          23          24          25          26          27 
#>          49          35          56          42          58 
#>          28          29          30          31          32 
#>          42          56          54          57          42 
#>          33          34          35          36          37 
#>          54          49          56          52          58 
#>          38          39          40          41          42 
#>          44          42          46          36          50 
#>          43          44          45          46          47 
#>          45          52          27          45          55 
#>          48          49          50          51          52 
#>          46          41          48          49          65 
#>          53          54          55          56          57 
#>          60          53          48          48          70 
#>          58          59          60          61          62 
#>          67          58          53          56          56 
#>          63          64          65          66          67 
#>          43          34          44          47          49 
#>          68          69          70          71          72 
#>          43          42          32          27          26 
#>          73          74          75          76          77 
#>          22          24          19          25          23 
#>          78          79          80          81          82 
#>          26          21          25          21          11 
#>          83          84          85          86          87 
#>          22          11          11          12           9 
#>          88 89 OR OLDER        <NA> 
#>           3          22          10
```

`unique(x)` returns an object of the same type of `x`, but with only one copy of each duplicated element. `table(x)` returns the same, plus the number of times a particular value of `x` occurs.

Age appears to be measured in years up to age 88, and then `"89 OR OLDER"` represents people who are 89 years old or older. Since `"89 OR OLDER"` can not be a number, trying to force the age variable with `"89 OR OLDER"` in it into a numeric variable would result in an error. Before converting `age` into a numeric variable, you should first recode anyone who has a value of `"89 OR OLDER"` to instead have a value 89.

```
gss.2016$age[gss.2016$age %in% "89 OR OLDER"] <- "89"  # recoding
gss.2016$age <- as.numeric(gss.2016$age)               # data type conversion
```

```
str(gss.2016)
#> 'data.frame':    2867 obs. of  2 variables:
#>  $ grass: Factor w/ 4 levels "DK","IAP","LEGAL",..: 2 3 4 2 3 3 2 4 2 4 ...
#>  $ age  : num  47 61 72 43 55 53 50 23 45 71 ...
```

By now, the data frame `gss.2016` is in the desired structure.

What about the data frame `survey`? As we mentioned, there are a few variables, namely `Sex`, `W.Hnd`, `Fold`, `Clap`, `Exer`,`Smoke`, and`M.I`, that are categorical, so you need to convert to factor. We can use the `factor()` function:

```
survey$Sex <- factor(survey$Sex)
survey$W.Hnd <- factor(survey$W.Hnd)
survey$Fold <- factor(survey$Fold)
survey$Clap <- factor(survey$Clap)
survey$Exer <- factor(survey$Exer)
survey$Smoke <- factor(survey$Smoke)
survey$M.I <- factor(survey$M.I)
str(survey)
#> 'data.frame':    237 obs. of  12 variables:
#>  $ Sex   : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
#>  $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
#>  $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
#>  $ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
#>  $ Fold  : Factor w/ 3 levels "L on R","Neither",..: 3 3 1 3 2 1 1 3 3 3 ...
#>  $ Pulse : num  92 104 87 NA 35 64 83 74 72 90 ...
#>  $ Clap  : Factor w/ 3 levels "Left","Neither",..: 1 1 2 2 3 3 3 3 3 3 ...
#>  $ Exer  : Factor w/ 3 levels "Freq","None",..: 3 2 2 2 3 3 1 1 3 3 ...
#>  $ Smoke : Factor w/ 4 levels "Heavy","Never",..: 2 4 3 2 2 2 2 2 2 2 ...
#>  $ Height: num  173 178 NA 160 165 ...
#>  $ M.I   : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ...
#>  $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
```

Our dataset `survey` contains only numeric and factor variables. But two variables (`Exer` an`Smoke`) are ordinal categorical variable, so you need to check the levels.

Sometimes it's useful to know the number of levels of a factor. The convenience function `nlevels()` extracts the number of levels from a factor:

```
nlevels(survey$Exer)
#> [1] 3
```

To look at the levels of a factor, you use the `levels()` function. For example, to extract the factor levels of `Exer`, use the following:

```
levels(survey$Exer)
#> [1] "Freq" "None" "Some"
```

As you can see, each student has a status of exercise (None, Some, Freq), how often the student exercises. Notice, in the output above the levels are ordered alphabetically. However, we need to sort in the order None, Some, Freq:

```
survey$Exer <- factor(survey$Exer, levels=c("None", "Some", "Freq"))
levels(survey$Exer)
#> [1] "None" "Some" "Freq"
```

In R, there is a really big practical advantage to order factor's level. A great many R functions recognize and treat ordered factors differently by printing results in the order that you expect. For example,

```
table(survey$Exer, useNA = "ifany")
#>
#> None Some Freq
#>   24   98  115
```

We need to order the levels in `Smoke` variable.

```
levels(survey$Smoke)
#> [1] "Heavy" "Never" "Occas" "Regul"
table(survey$Smoke, useNA = "ifany")
#>
#> Heavy Never Occas Regul  <NA>
#>    11   189    19    17     1
survey$Smoke <- factor(survey$Smoke, levels=c("Never", "Occas", "Regul","Heavy"))
levels(survey$Smoke)
#> [1] "Never" "Occas" "Regul" "Heavy"
table(survey$Smoke, useNA = "ifany")
#>
#> Never Occas Regul Heavy  <NA>
#>   189    19    17    11     1
```

3.3.3 Transformation

3.3.4 Identifying and treating missing values

In addition to making sure the variables used are an appropriate type, it was also important to make sure that missing values were treated appropriately by R. In R, missing values are recorded as `NA`, which stands for not available. Researchers code missing values in many different ways when collecting and storing data. Some of the more common ways to denote missing values are the following:

- blank
- 777, -777, 888, -888, 999, -999, or something similar
- a single period
- -1

- NULL.

Other responses, such as “Don’t know” or “Inapplicable,” may sometimes be treated as missing or as response categories depending on what is most appropriate given the characteristics of the data and the analysis goals.

In the summary of the `gss.2016` data,

```
summary(gss.2016)
#>        grass          age       
#>  DK       : 110   Min.   :18.00  
#>  IAP      : 911   1st Qu.:34.00  
#>  LEGAL    :1126   Median :49.00  
#>  NOT LEGAL: 717   Mean   :49.16  
#>  NA's     :   3   3rd Qu.:62.00  
#>                   Max.   :89.00  
#>                   NA's   :10
```

the `grass` variable has five possible values: `DK` (don’t know), `IAP` (inapplicable), `LEGAL`, `NOT LEGAL`, and `NA` (not available). The `DK`, `IAP`, and `NA` could all be considered missing values. However, R treats only `NA` as missing. Before conducting any analyses, the `DK` and `IAP` values could be converted to `NA` to be treated as missing in any analyses. That is, the `grass` variable could be recoded so that these values are all `NA`. Note that `NA` is a reserved “word” in R. In order to use `NA`, both letters must be uppercase (`Na` or `na` does not work), and there can be no quotation marks (R will treat `"NA"` as a character rather than a true missing value). There are many ways to recode variables in R. For example,

```
table(gss.2016$grass, useNA = "ifany")   # before recoding
#> 
#>        DK       IAP     LEGAL NOT LEGAL      <NA> 
#>       110       911      1126       717         3
library(car)
gss.2016$grass <- car::recode(var = gss.2016$grass, recodes = 'c("DK", "IAP")=NA')
table(gss.2016$grass, useNA = "ifany")   # after recoding
#> 
#>     LEGAL NOT LEGAL      <NA> 
#>      1126       717      1024
```

3.3.5 Numeric to factor

In addition to solving the `age` and `grass` recoding, the final plan to create the age categories shown below. The `age` variable currently holds the age in years rather than age categories. The age can be in four categories:

- 18-29
- 30-59
- 60-74

- 75+

The function `cut()` can be used to divide a continuous variable into categories by cutting it into pieces and adding a label to each piece.

```
gss.2016$age.f <- cut(x = gss.2016$age, breaks = c(-Inf, 29, 59, 74, Inf),
    labels = c("<30", "30-59", "60-74", "75+" ))
table(gss.2016$age.f, useNA = "ifany")
#>
#>   <30 30-59 60-74   75+  <NA>
#>   481  1517   598   261    10
```

`cut()` takes a variable like `age` as the first argument. The second thing to add after the variable name is a vector made up of the breaks. Breaks specify the lower and upper limit of each category of values. The first entry is the lowest value of the first category, the second entry is the highest value of the first category, the third entry is the highest value of the second category, and so on. The first and last values in the vector are `-Inf` and `Inf`. These are negative infinity and positive infinity. This was for convenience rather than looking up the smallest and largest values of variable `age`. It also makes the code more flexible in case there is a new data point with a smaller or larger value. The final thing to add is a vector made up of the labels for the categories, with each label inside quote marks.

3.4 Descriptive statistics

R has built in functions for a large number of summary statistics. To illustrate the main R functions we will use the `survey` and `gss.2016` datasets. R has tons of packages to explore our dataset, but we focus on built-in possibilities, **psych** and **DescTools** packages.

Let us first see what kind of objects are included in `survey` and `gss.2016` by using `summary()` function.

```
summary(gss.2016)
#>         grass           age          age.f
#>  LEGAL    :1126   Min.   :18.00   <30  : 481
#>  NOT LEGAL: 717   1st Qu.:34.00   30-59:1517
#>  NA's     :1024   Median :49.00   60-74: 598
#>                   Mean   :49.16   75+  : 261
#>                   3rd Qu.:62.00   NA's :  10
#>                   Max.   :89.00
#>                   NA's   :10
summary(survey)
#>      Sex          Wr.Hnd          NW.Hnd         W.Hnd
#>  Female:118   Min.   :13.00   Min.   :12.50   Left : 18
#>  Male  :118   1st Qu.:17.50   1st Qu.:17.50   Right:218
#>  NA's  :  1   Median :18.50   Median :18.50   NA's :  1
#>               Mean   :18.67   Mean   :18.58
#>               3rd Qu.:19.80   3rd Qu.:19.73
#>               Max.   :23.20   Max.   :23.50
#>               NA's   :1       NA's   :1
#>       Fold          Pulse             Clap       Exer
#>  L on R : 99   Min.   : 35.00   Left   : 39   None: 24
#>  Neither: 18   1st Qu.: 66.00   Neither: 50   Some: 98
#>  R on L :120   Median : 72.50   Right  :147   Freq:115
#>                Mean   : 74.15   NA's   :  1
#>                3rd Qu.: 80.00
#>                Max.   :104.00
#>                NA's   :45
#>    Smoke          Height             M.I           Age
#>  Never:189   Min.   :150.0   Imperial: 68   Min.   :16.75
#>  Occas: 19   1st Qu.:165.0   Metric  :141   1st Qu.:17.67
#>  Regul: 17   Median :171.0   NA's    : 28   Median :18.58
#>  Heavy: 11   Mean   :172.4                  Mean   :20.37
#>  NA's :  1   3rd Qu.:180.0                  3rd Qu.:20.17
#>              Max.   :200.0                  Max.   :73.00
#>              NA's   :28
```

The type of the descriptive statistics we use depends on whether data is numeric (continuous) or categorical and so we will look at each case separately next.

3.4.1 Measurements

Recall that for numeric variables, we are usually interested in measuring center tendency and spread to get a sense of data. Suppose that we are interested in `Height` column, in which students' height is measured. From the `summary(survey)` table above we know that this variable is indeed a numeric data, and therefore we can measure central tendency and spread of this variable as we do in the following codes, respectively:

```
# Central Tendency
mean(survey$Height, na.rm = T)     # Mean
#> [1] 172.3809
median(survey$Height, na.rm = T)   # Median
#> [1] 171

# Spread
min(survey$Height, na.rm = T)      # Minimum
#> [1] 150
max(survey$Height, na.rm = T)      # Maximum
#> [1] 200
range(survey$Height, na.rm = T)    # Range
```

```
#> [1] 150 200
IQR(survey$Height, na.rm = T)       # IQR
#> [1] 15
var(survey$Height, na.rm = T)       # Variance
#> [1] 96.9738
sd(survey$Height, na.rm = T)        # Standard Deviation
#> [1] 9.847528
```

All of these functions have optional arguments to address various complications that your data might have. For example, if your data includes some NAs, then instead of using `mean(survey$Height)` you should use `mean(survey$Height, na.rm = T)`, which tells R to ignore NAs in the data.

Please study carefully the following codes and outputs:

```
#install.packages("psych")
library(psych)
describe(gss.2016)
#>        vars    n  mean    sd median trimmed   mad min max range
#> grass*    1 1843  1.39  0.49      1    1.36  0.00   1   2     1
#> age       2 2857 49.16 17.69     49   48.62 20.76  18  89    71
#> age.f*    3 2857  2.22  0.83      2    2.17  0.00   1   4     3
#>        skew kurtosis   se
#> grass* 0.45    -1.79 0.01
#> age    0.17    -0.90 0.33
#> age.f* 0.51    -0.16 0.02
describe(survey)
#>        vars   n   mean    sd median trimmed   mad    min   max
#> Sex*      1 236   1.50  0.50   1.50    1.50  0.74   1.00   2.0
#> Wr.Hnd    2 236  18.67  1.88  18.50   18.61  1.48  13.00  23.2
#> NW.Hnd    3 236  18.58  1.97  18.50   18.55  1.63  12.50  23.5
#> W.Hnd*    4 236   1.92  0.27   2.00    2.00  0.00   1.00   2.0
#> Fold*     5 237   2.09  0.96   3.00    2.11  0.00   1.00   3.0
#> Pulse     6 192  74.15 11.69  72.50   74.02 11.12  35.00 104.0
#> Clap*     7 236   2.46  0.76   3.00    2.57  0.00   1.00   3.0
#> Exer*     8 237   2.38  0.66   2.00    2.48  1.48   1.00   3.0
#> Smoke*    9 236   1.36  0.81   1.00    1.15  0.00   1.00   4.0
#> Height   10 209 172.38  9.85 171.00  172.19 10.08 150.00 200.0
#> M.I*     11 209   1.67  0.47   2.00    1.72  0.00   1.00   2.0
#> Age      12 237  20.37  6.47  18.58   18.99  1.61  16.75  73.0
#>        range  skew kurtosis   se
#> Sex*     1.00  0.00    -2.01 0.03
#> Wr.Hnd 10.20  0.18     0.30 0.12
#> NW.Hnd 11.00  0.02     0.44 0.13
#> W.Hnd*   1.00 -3.17     8.10 0.02
#> Fold*    2.00 -0.18    -1.89 0.06
#> Pulse  69.00 -0.02     0.33 0.84
#> Clap*   2.00 -0.98    -0.60 0.05
#> Exer*   2.00 -0.61    -0.68 0.04
#> Smoke*  3.00  2.15     3.45 0.05
#> Height 50.00  0.22    -0.44 0.68
#> M.I*    1.00 -0.74    -1.46 0.03
#> Age    56.25  5.16    33.47 0.42

#install.packages("DescTools")
library(DescTools)
Desc(gss.2016, plot=F)
#> ------------------------------------------------------------------
#> Describe gss.2016 (data.frame):
#>
#> data frame:  2867 obs. of  3 variables
#>      1836 complete cases (64.0%)
#>
#>   Nr  ColName  Class    NAs           Levels
#>   1   grass    factor   1024 (35.7%)  (2): 1-LEGAL, 2-NOT
#>                                       LEGAL
#>   2   age      numeric    10 (0.3%)
#>   3   age.f    factor     10 (0.3%)   (4): 1-<30, 2-30-59,
#>                                       3-60-74, 4-75+
#>
#>
#> ------------------------------------------------------------------
#> 1 - grass (factor - dichotomous)
#>
#>   length      n    NAs unique
#>    2'867  1'843  1'024      2
#>           64.3%  35.7%
#>
#>             freq   perc  lci.95  uci.95'
#> LEGAL      1'126  61.1%   58.8%   63.3%
#> NOT LEGAL    717  38.9%   36.7%   41.2%
#>
#> ' 95%-CI (Wilson)
#>
#> ------------------------------------------------------------------
#> 2 - age (numeric)
#>
#>   length      n    NAs  unique     0s   mean  meanCI'
#>    2'867  2'857     10      72      0  49.16   48.51
#>           99.7%   0.3%           0.0%          49.80
#>
```

```
#>      .05    .10    .25  median    .75    .90     .95
#>    22.80  26.00  34.00   49.00  62.00  73.00   80.00
#>
#>    range     sd  vcoef     mad    IQR   skew    kurt
#>    71.00  17.69   0.36   20.76  28.00   0.17   -0.90
#>
#> lowest : 18.0 (7), 19.0 (33), 20.0 (26), 21.0 (33), 22.0 (44)
#> highest: 85.0 (11), 86.0 (12), 87.0 (9), 88.0 (3), 89.0 (22)
#>
#> ' 95%-CI (classic)
#>
#> ------------------------------------------------------------------
#> 3 - age.f (factor)
#>
#>   length      n    NAs unique levels  dupes
#>    2'867  2'857     10      4      4      y
#>           99.7%   0.3%
#>
#>    level   freq   perc  cumfreq  cumperc
#> 1  30-59  1'517  53.1%    1'517    53.1%
#> 2  60-74    598  20.9%    2'115    74.0%
#> 3    <30    481  16.8%    2'596    90.9%
#> 4    75+    261   9.1%    2'857   100.0%
Desc(survey, plot=F)
#> ------------------------------------------------------------------
#> Describe survey (data.frame):
#>
#> data frame:  237 obs. of  12 variables
#>      168 complete cases (70.9%)
#>
#>   Nr  ColName  Class    NAs          Levels
#>   1   Sex      factor    1 (0.4%)    (2): 1-Female, 2-Male
#>   2   Wr.Hnd   numeric   1 (0.4%)
#>   3   NW.Hnd   numeric   1 (0.4%)
#>   4   W.Hnd    factor    1 (0.4%)    (2): 1-Left, 2-Right
#>   5   Fold     factor    .           (3): 1-L on R,
#>                                      2-Neither, 3-R on L
#>   6   Pulse    numeric  45 (19.0%)
#>   7   Clap     factor    1 (0.4%)    (3): 1-Left, 2-Neither,
#>                                      3-Right
#>   8   Exer     factor    .           (3): 1-None, 2-Some,
#>                                      3-Freq
#>   9   Smoke    factor    1 (0.4%)    (4): 1-Never, 2-Occas,
#>                                      3-Regul, 4-Heavy
#>   10  Height   numeric  28 (11.8%)
#>   11  M.I       factor   28 (11.8%)  (2): 1-Imperial,
#>                                      2-Metric
#>   12  Age       numeric   .
#> 
#> 
#> ----------------------------------------------------------------
#> 1 - Sex (factor - dichotomous)
#> 
#>   length      n    NAs unique
#>      237    236      1      2
#>           99.6%   0.4%
#> 
#>          freq   perc  lci.95  uci.95'
#> Female   118  50.0%   43.7%   56.3%
#> Male     118  50.0%   43.7%   56.3%
#> 
#> ' 95%-CI (Wilson)
#> 
#> ----------------------------------------------------------------
#> 2 - Wr.Hnd (numeric)
#> 
#>   length      n    NAs  unique     0s   mean  meanCI'
#>      237    236      1      60      0  18.67   18.43
#>           99.6%   0.4%           0.0%          18.91
#> 
#>      .05    .10    .25  median    .75    .90     .95
#>    16.00  16.50  17.50   18.50  19.80  21.15   22.05
#> 
#>    range     sd  vcoef     mad    IQR   skew    kurt
#>    10.20   1.88   0.10    1.48   2.30   0.18    0.30
#> 
#> lowest : 13.0 (2), 14.0 (2), 15.0, 15.4, 15.5 (2)
#> highest: 22.5 (4), 22.8, 23.0 (2), 23.1, 23.2 (3)
#> 
#> heap(?): remarkable frequency (9.7%) for the mode(s) (= 17.5)
#> 
#> ' 95%-CI (classic)
#> 
#> ----------------------------------------------------------------
#> 3 - NW.Hnd (numeric)
#> 
#>   length       n     NAs  unique      0s    mean  meanCI'
#>      237     236       1      68       0  18.583  18.330
#>            99.6%    0.4%            0.0%          18.835
#> 
```

```
#>       .05      .10      .25   median      .75      .90      .95
#>    15.500   16.300   17.500   18.500   19.725   21.000   22.225
#>
#>     range       sd    vcoef      mad      IQR     skew     kurt
#>    11.000    1.967    0.106    1.631    2.225    0.024    0.441
#>
#> lowest : 12.5, 13.0 (2), 13.3, 13.5, 15.0
#> highest: 22.7, 23.0, 23.2 (2), 23.3, 23.5
#>
#> heap(?): remarkable frequency (8.9%) for the mode(s) (= 18)
#>
#> ' 95%-CI (classic)
#>
#> ------------------------------------------------------------------
#> 4 - W.Hnd (factor - dichotomous)
#>
#>   length      n    NAs unique
#>      237    236      1      2
#>           99.6%   0.4%
#>
#>         freq   perc  lci.95  uci.95'
#> Left      18   7.6%    4.9%   11.7%
#> Right    218  92.4%   88.3%   95.1%
#>
#> ' 95%-CI (Wilson)
#>
#> ------------------------------------------------------------------
#> 5 - Fold (factor)
#>
#>   length      n    NAs unique levels  dupes
#>      237    237      0      3      3      y
#>          100.0%   0.0%
#>
#>      level  freq   perc  cumfreq  cumperc
#> 1   R on L   120  50.6%      120    50.6%
#> 2   L on R    99  41.8%      219    92.4%
#> 3  Neither    18   7.6%      237   100.0%
#>
#> ------------------------------------------------------------------
#> 6 - Pulse (numeric)
#>
#>   length      n    NAs  unique     0s   mean  meanCI'
#>      237    192     45      43      0  74.15   72.49
#>           81.0%  19.0%           0.0%          75.81
#>
```

```
#>       .05    .10    .25  median    .75    .90     .95
#>     59.55  60.00  66.00   72.50  80.00  90.00   92.00
#> 
#>     range     sd  vcoef     mad    IQR   skew    kurt
#>     69.00  11.69   0.16   11.12  14.00  -0.02    0.33
#> 
#> lowest : 35.0, 40.0, 48.0 (2), 50.0 (2), 54.0
#> highest: 96.0 (3), 97.0, 98.0, 100.0 (2), 104.0 (2)
#> 
#> heap(?): remarkable frequency (9.4%) for the mode(s) (= 80)
#> 
#> ' 95%-CI (classic)
#> 
#> ---------------------------------------------------------------
#> 7 - Clap (factor)
#> 
#>   length      n    NAs unique levels  dupes
#>      237    236      1      3      3      y
#>           99.6%   0.4%
#> 
#>      level  freq   perc  cumfreq  cumperc
#> 1    Right   147  62.3%      147    62.3%
#> 2  Neither    50  21.2%      197    83.5%
#> 3     Left    39  16.5%      236   100.0%
#> 
#> ---------------------------------------------------------------
#> 8 - Exer (factor)
#> 
#>   length      n    NAs unique levels  dupes
#>      237    237      0      3      3      y
#>          100.0%   0.0%
#> 
#>    level  freq   perc  cumfreq  cumperc
#> 1   Freq   115  48.5%      115    48.5%
#> 2   Some    98  41.4%      213    89.9%
#> 3   None    24  10.1%      237   100.0%
#> 
#> ---------------------------------------------------------------
#> 9 - Smoke (factor)
#> 
#>   length      n    NAs unique levels  dupes
#>      237    236      1      4      4      y
#>           99.6%   0.4%
#> 
#>    level  freq   perc  cumfreq  cumperc
#> 1  Never   189  80.1%      189    80.1%
#> 2  Occas    19   8.1%      208    88.1%
#> 3  Regul    17   7.2%      225    95.3%
#> 4  Heavy    11   4.7%      236   100.0%
#>
#> ------------------------------------------------------------------
#> 10 - Height (numeric)
#>
#>   length        n      NAs  unique       0s     mean  meanCI'
#>      237      209       28      67        0  172.38  171.04
#>           88.2%   11.8%             0.0%          173.72
#>
#>      .05      .10      .25  median      .75      .90      .95
#>   157.00   160.00   165.00  171.00   180.00   185.42   189.60
#>
#>    range       sd    vcoef      mad      IQR     skew     kurt
#>    50.00     9.85     0.06    10.08    15.00     0.22    -0.44
#>
#> lowest : 150.0, 152.0, 152.4, 153.5, 154.94 (2)
#> highest: 191.8, 193.04, 195.0, 196.0, 200.0
#>
#> ' 95%-CI (classic)
#>
#> ------------------------------------------------------------------
#> 11 - M.I (factor - dichotomous)
#>
#>   length       n     NAs unique
#>      237     209      28      2
#>           88.2%   11.8%
#>
#>           freq   perc  lci.95  uci.95'
#> Imperial    68  32.5%   26.5%   39.2%
#> Metric     141  67.5%   60.8%   73.5%
#>
#> ' 95%-CI (Wilson)
#>
#> ------------------------------------------------------------------
#> 12 - Age (numeric)
#>
#>    length         n       NAs    unique        0s      mean   meanCI'
#>       237       237         0        88         0   20.3745  19.5460
#>              100.0%      0.0%                0.0%            21.2030
#>
#>       .05       .10       .25    median       .75       .90       .95
#>   17.0830   17.2168   17.6670   18.5830   20.1670   23.5830   30.6836
```

```
#>
#>      range        sd     vcoef       mad       IQR      skew      kurt
#>    56.2500    6.4743    0.3178    1.6057    2.5000    5.1630   33.4720
#>
#> lowest : 16.75, 16.917 (3), 17.0 (2), 17.083 (7), 17.167 (11)
#> highest: 41.583, 43.833, 44.25, 70.417, 73.0
#>
#> ' 95%-CI (classic)

# measurements for groups
describeBy(x = survey$Wr.Hnd, group = survey$Sex, mat=T)
#>     item group1 vars   n     mean       sd median  trimmed
#> X11    1 Female    1 118 17.59576 1.314768   17.5 17.64479
#> X12    2   Male    1 117 19.74188 1.750775   19.5 19.72737
#>         mad min  max range        skew   kurtosis        se
#> X11 1.18608  13 20.8   7.8 -0.65369868 1.59655733 0.1210342
#> X12 1.48260  14 23.2   9.2 -0.05094141 0.01581485 0.1618592
Desc(Wr.Hnd~Sex, data=survey, plot=F)
#> ------------------------------------------------------------------
#> Wr.Hnd ~ Sex (survey)
#>
#> Summary:
#> n pairs: 237, valid: 235 (99.2%), missings: 2 (0.8%), groups: 2
#>
#>
#>          Female     Male
#> mean     17.596   19.742
#> median   17.500   19.500
#> sd        1.315    1.751
#> IQR       1.500    2.500
#> n           118      117
#> np      50.213%  49.787%
#> NAs           0        1
#> 0s            0        0
#>
#> Kruskal-Wallis rank sum test:
#>   Kruskal-Wallis chi-squared = 83.878, df = 1, p-value < 2.2e-16
#>
#>
#> Warning:
#>   Grouping variable contains 1 NAs (0.422%).
```

3.4.2 Tables

For categorical variables, counts and percentages can be used to summarize data:

```
table(gss.2016$grass, useNA = "ifany")
#>
#>     LEGAL NOT LEGAL      <NA>
#>      1126       717      1024
table(gss.2016$age.f, useNA = "ifany")
#>
#>   <30 30-59 60-74   75+  <NA>
#>   481  1517   598   261    10
table(survey$Sex, useNA = "ifany")
#>
#> Female   Male   <NA>
#>    118    118      1

prop.table(table(gss.2016$grass, useNA = "ifany"))
#>
#>     LEGAL NOT LEGAL      <NA>
#> 0.3927450 0.2500872 0.3571678
prop.table(table(gss.2016$age.f, useNA = "ifany"))
#>
#>         <30       30-59       60-74         75+        <NA>
#> 0.167771189 0.529124520 0.208580398 0.091035926 0.003487967
prop.table(table(survey$Sex, useNA = "ifany"))
#>
#>      Female        Male        <NA>
#> 0.497890295 0.497890295 0.004219409

Desc(gss.2016$grass, plot=F)
#> ---------------------------------------------------------------
#> gss.2016$grass (factor - dichotomous)
#>
#>   length      n    NAs unique
#>    2'867  1'843  1'024      2
#>           64.3%  35.7%
#>
#>              freq   perc  lci.95  uci.95'
#> LEGAL       1'126  61.1%   58.8%   63.3%
#> NOT LEGAL     717  38.9%   36.7%   41.2%
#>
#> ' 95%-CI (Wilson)

# 2D tables
```

```
table(gss.2016$grass, gss.2016$age.f, useNA = "ifany")
#>
#>             <30 30-59 60-74 75+ <NA>
#>   LEGAL     237   625   213  48    3
#>   NOT LEGAL  95   364   151 103    4
#>   <NA>      149   528   234 110    3
Desc(grass~age.f, data=gss.2016, plot=F)
#> ---------------------------------------------------------------
#> grass ~ age.f (gss.2016)
#>
#> Summary:
#> n: 1'836, rows: 2, columns: 4
#>
#> Pearson's Chi-squared test:
#>   X-squared = 72.253, df = 3, p-value = 1.405e-15
#> Log likelihood ratio (G-test) test of independence:
#>   G = 71.218, X-squared df = 3, p-value = 2.331e-15
#> Mantel-Haenszel Chi-squared:
#>   X-squared = 59.423, df = 1, p-value = 1.271e-14
#>
#> Phi-Coefficient        0.198
#> Contingency Coeff.     0.195
#> Cramer's V             0.198
#>
#>
#>             age.f    <30   30-59   60-74    75+    Sum
#> grass
#>
#> LEGAL       freq     237     625     213     48  1'123
#>             perc   12.9%   34.0%   11.6%   2.6%  61.2%
#>             p.row  21.1%   55.7%   19.0%   4.3%      .
#>             p.col  71.4%   63.2%   58.5%  31.8%      .
#>
#> NOT LEGAL   freq      95     364     151    103    713
#>             perc    5.2%   19.8%    8.2%   5.6%  38.8%
#>             p.row  13.3%   51.1%   21.2%  14.4%      .
#>             p.col  28.6%   36.8%   41.5%  68.2%      .
#>
#> Sum         freq     332     989     364    151  1'836
#>             perc   18.1%   53.9%   19.8%   8.2% 100.0%
#>             p.row      .       .       .      .      .
#>             p.col      .       .       .      .      .
#>
```

3.5 Plots

In statistics and other sciences, being able to plot your results in the form of a graphic is often useful. An effective and accurate visualization can make your data come to life and convey your message in a powerful way. R has very powerful graphics capabilities that can help you visualize your data. In this chapter, we give you a look at *traditional graphics* and *ggplot2* graphics.

We will look at five methods of visualizing data:

- Scatterplot
- Bar plot
- Box plot
- Histogram
- One-dimensional strip plot

3.5.1 Traditional graphics

With traditional graphics, you can create many different types of plots, such as scatterplots and bar charts. Here are just a few of the different types of plots you can create:

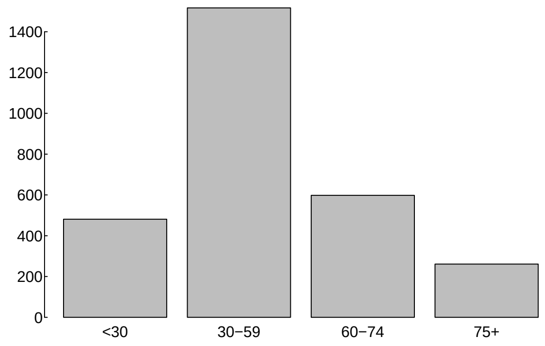
- Scatterplot: `plot()`, `stripchart()`
- Bar plot: `barplot()`
- Box plot: `boxplot()`
- Histogram: `hist()`
- One-dimensional strip plot: `stripchart()`

For a complete list of the different types of plots, see the Help at `?graphics`.

3.5.1.1 Bar plot

A bar plot displays the distribution (frequency) of a categorical variable through vertical or horizontal bars. In its simplest form, the format of the barplot() function is

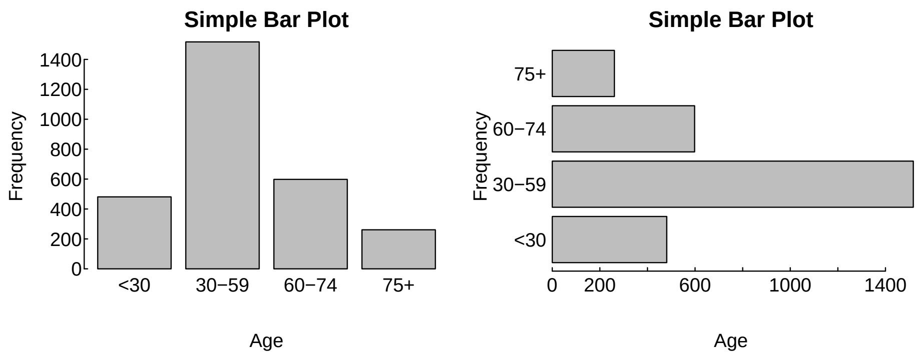
```
par(mar = c(2, 2, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
barplot(table(gss.2016$age.f))
```



where `gss.2016$age.f` is a factor.

The values `table(gss.2016$age.f)` determine the heights of the bars in the plot, and a vertical bar plot is produced. Including the option `horiz=TRUE` produces a horizontal bar chart instead. You can also add annotating options. The main option adds a plot title, whereas the xlab and ylab options add x-axis and y-axis labels, respectively.

```
par(mar = c(4, 4, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
counts <- table(gss.2016$age.f)
barplot(counts,
        main="Simple Bar Plot",
        xlab="Age", ylab="Frequency")
barplot(counts,
        main="Simple Bar Plot",
        xlab="Age", ylab="Frequency",
        horiz=T)
```



You can customize many features of a graph (fonts, colors, axes, and labels) through options called graphical parameters. One way is to specify these options through the `par()` function. Values set in this manner will be in effect for the rest of the session or until they're changed. The format is `par(optionname=value, optionname=value, ...)`. Specifying `par()` without parameters produces a list of the current graphical settings.

The relevant parameters are shown below

Parameter	Description
`mar`	A numerical vector of the form `c(bottom, left, top, right)` which gives the number of lines of margin to be specified on the four sides of the plot. The default is `c(5, 4, 4, 2) + 0.1`.
`las`	Specifies that labels are parallel (= 0) or perpendicular (= 2) to the axis.
`tck`	Length of each tick mark as a fraction of the plotting region (a negative number is outside the graph, a positive number is inside, 0 suppresses ticks, and 1 creates gridlines). The default is -0.01.
`mgp`	The margin line for the axis title, axis labels and axis line. Note that `mgp[1]` affects title whereas `mgp[2:3]` affect axis. The default is `c(3, 1, 0)`.

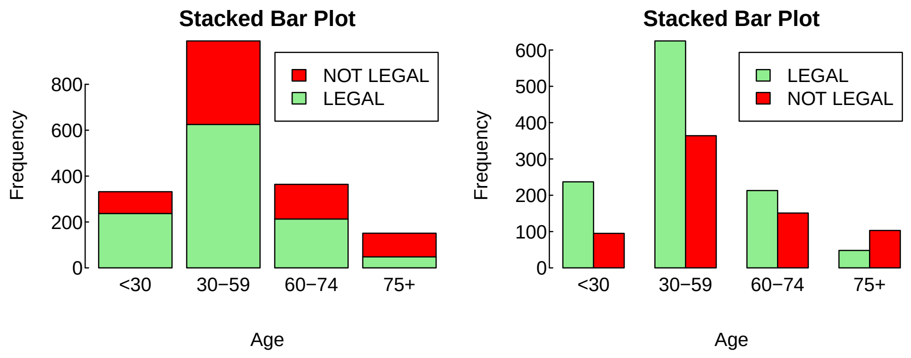
If the argument of `barplot()` is a matrix rather than a vector, the resulting graph will be a stacked or grouped bar plot. If `beside=FALSE` (the default), then each column of the matrix produces a bar in the plot, with the values in the column giving the heights of stacked "sub-bars." If `beside=TRUE`, each column of the matrix represents a group, and the values in each column are juxtaposed rather than stacked.

Consider the cross-tabulation of age and votes:

```r
counts <- table(gss.2016$grass, gss.2016$age.f)
counts
#>
#>              <30 30-59 60-74 75+
#>   LEGAL      237   625   213  48
#>   NOT LEGAL   95   364   151 103
```

You can graph the results as either a stacked or a grouped bar plot. The resulting graphs are displayed below

```r
par(mar = c(4, 4, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
barplot(counts,
        main="Stacked Bar Plot",
        xlab="Age", ylab="Frequency",
        col=c("lightgreen", "red"),
        legend=T)
barplot(counts,
        main="Stacked Bar Plot",
        xlab="Age", ylab="Frequency",
        col=c("lightgreen", "red"),
        legend=T, beside = T)
```



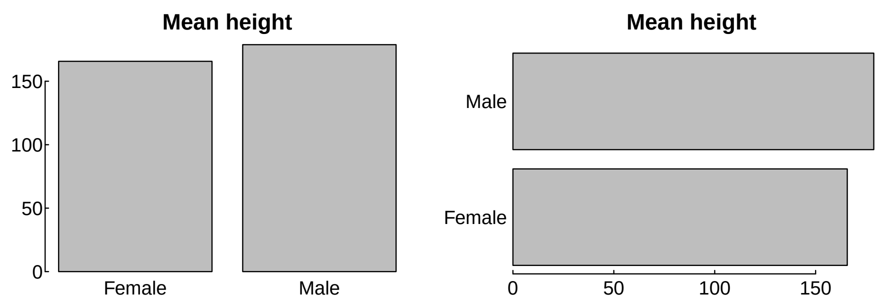
Bar plots needn't be based on counts or frequencies. You can create bar plots that represent means, medians, standard deviations, and so forth by using the aggregate function and passing the results to the `barplot()` function. The following listing shows an example, which is displayed below.

```
par(mar = c(4, 4, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
means <- aggregate(survey$Height,
                   survey[,"Sex",drop=F], mean, na.rm=T)
means
#>      Sex        x
#> 1 Female 165.6867
#> 2   Male 178.8260
barplot(means$x, names.arg = means$Sex,
        main="Mean height")
barplot(means$x, names.arg = means$Sex,
        main="Mean height", horiz = T)
```

`means$x` is the vector containing the heights of the bars, and the option `names.arg=means$Sex` is added to provide labels.

Please study carefully the following codes and outputs:

```
par(mar = c(4, 4, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
barplot(table(gss.2016$age.f))
barplot(table(gss.2016$grass))
```

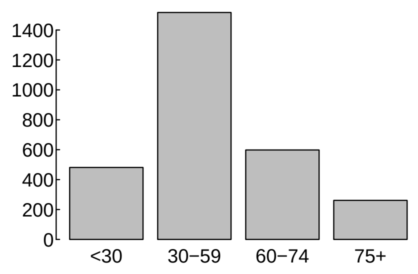


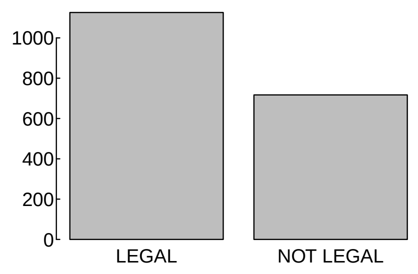
```
barplot(table(gss.2016$grass), col=c("green", "purple"))
barplot(table(gss.2016$grass), col=c("#78A678", "#7463AC"))
barplot(table(gss.2016$grass), col=c("#78A678", "#7463AC"),
xlab="Should marijuana be legal?", ylab="Number of responses")

par(las=1, mgp=c(2,0.2,0), tcl=0.1, mar=c(2,3,1,1))
barplot(table(gss.2016$grass), col=c("#78A678", "#7463AC"),
xlab="Should marijuana be legal?", ylab="Number of responses")

# to save plot to PNG
par(las=1, mgp=c(2,0.2,0), tcl=0.1, mar=c(2,3,1,1))
barplot(table(gss.2016$grass, gss.2016$age.f), beside = T, legend=T)

png(filename = "output/image/barplot_1.png", width = 400, height = 300)
par(las=1, mgp=c(2,0.2,0), tcl=0.1, mar=c(2,3,1,1))
barplot(table(gss.2016$grass, gss.2016$age.f), beside = T, legend=T)
dev.off()
#> pdf
#>   2
```

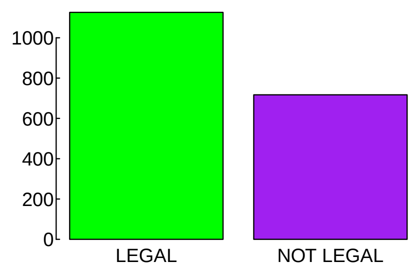
3.5.1.2 Histogram

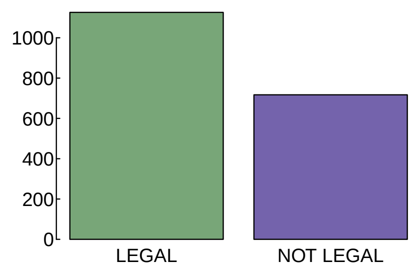
Histograms display the distribution of a continuous variable by dividing the range of scores into a specified number of bins on the x-axis and displaying the frequency of scores in each bin on the y-axis. You can create histograms with the function

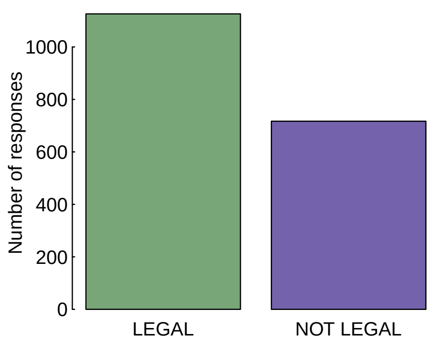
```
par(mar = c(2, 2, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
hist(survey$Wr.Hnd)
```

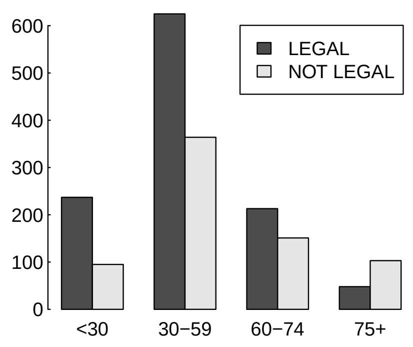

1400
1200
1000
800
600
400
200
0
<30
30−59
60−74
75+

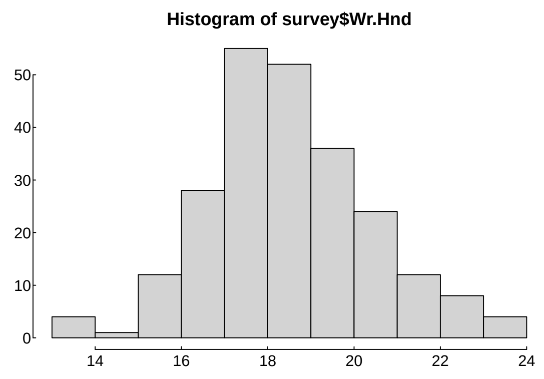

1000
800
600
400
200
0
LEGAL
NOT LEGAL


1000
800
600
400
200
0
LEGAL
NOT LEGAL


1000
800
600
400
200
0
LEGAL
NOT LEGAL


Number of responses
1000
800
600
400
200
0
LEGAL
NOT LEGAL


600
500
400
300
200
100
0
■ LEGAL
□ NOT LEGAL
<30
30−59
60−74
75+

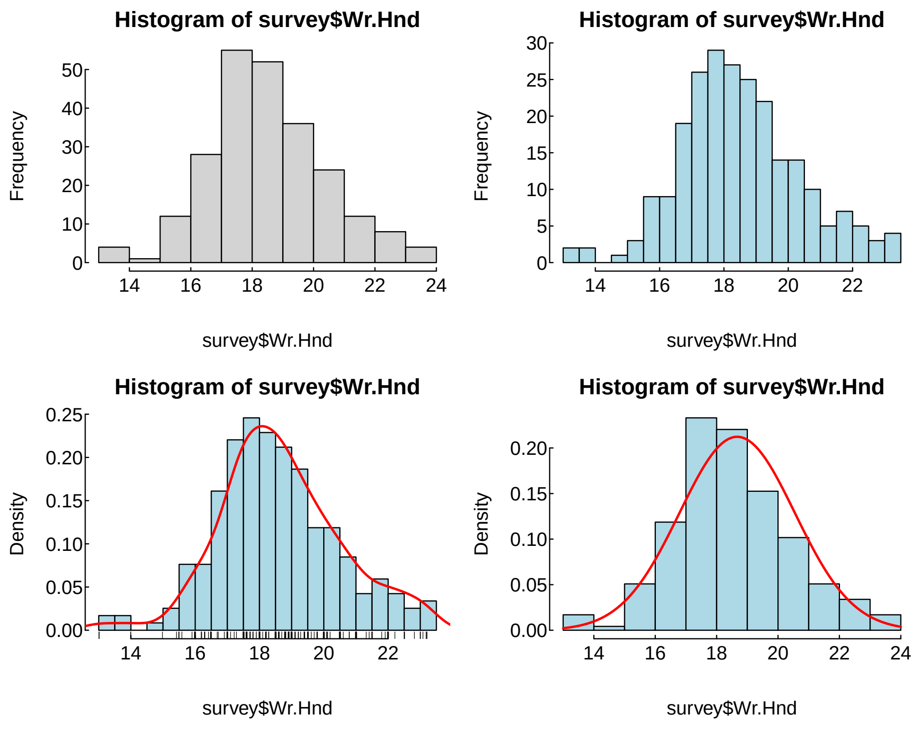


where `survey$Wr.Hnd` is a numeric vector of values. The option `freq=FALSE` creates a plot based on probability densities rather than frequencies. The `breaks=` option controls the number of bins. The default produces equally spaced breaks when defining the cells of the histogram. The following listing provides the code for four variations of a histogram; the results are plotted in figure 6.8.

```
par(mar = c(4, 4, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
hist(survey$Wr.Hnd)
hist(survey$Wr.Hnd,
     breaks=20, col = "lightblue")
hist(survey$Wr.Hnd,
     breaks=20, col = "lightblue",
     freq = F)
rug(jitter(survey$Wr.Hnd))
lines(density(survey$Wr.Hnd, na.rm = TRUE),
      col="red", lwd=2)
range(survey$Wr.Hnd, na.rm = T)
#> [1] 13.0 23.2
hist(survey$Wr.Hnd,
     breaks=seq(from=13, to=24, by=1),
     col = "lightblue", freq = F)
curve(dnorm(x, mean=mean(survey$Wr.Hnd, na.rm = T),
            sd=sd(survey$Wr.Hnd, na.rm = T)),
      from=13, to=24, add=T, col="red", lwd=2)
```

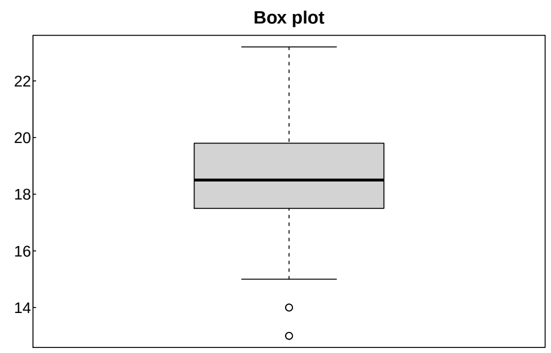
3.5.1.3 Box plot

A box-and-whiskers plot describes the distribution of a continuous variable by plotting its five-number summary: the minimum, lower quartile (25th percentile), median (50th percentile), upper quartile (75th percentile), and maximum. It can also display observations that may be outliers (values outside the range of $\pm 1.5 * IQR$, where IQR is the interquartile range defined as the upper quartile minus the lower quartile). For example, this statement produces the


Histogram of survey$Wr.Hnd
Frequency
survey$Wr.Hnd
Histogram of survey$Wr.Hnd
Frequency
survey$Wr.Hnd
Histogram of survey$Wr.Hnd
Density
survey$Wr.Hnd
Histogram of survey$Wr.Hnd
Density
survey$Wr.Hnd

plot shown below:

```
par(mar = c(2, 2, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
boxplot(survey$Wr.Hnd, main="Box plot", ylab="cm")
```



Box plots can be created for individual variables or for variables by group. The format is

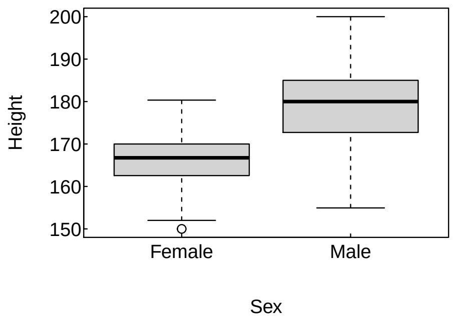
```
boxplot(formula, data=dataframe)
```

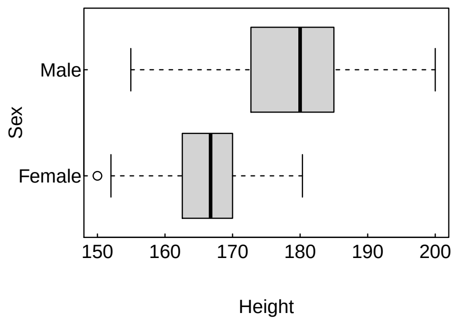
where `formula` is a formula and `dataframe` denotes the data frame (or list) providing the data. An example of a formula is `y ~ A`, where a separate box plot for numeric variable `y` is generated for each value of categorical variable `A`. The formula `y ~ A*B` would produce a box plot of numeric variable `y`, for each combination of levels in categorical variables `A` and `B`.

Adding the option `horizontal=TRUE` to reverse the axis orientation.

The following code revisits the impact of sex on height with parallel box plots.

```
par(mar = c(4, 4, 2, 0.1), las=1, mgp=c(2.5,0.1, 0), tcl=0.15)
boxplot(Height ~ Sex, data=survey)
boxplot(Height ~ Sex, data=survey, horizontal=TRUE)
```



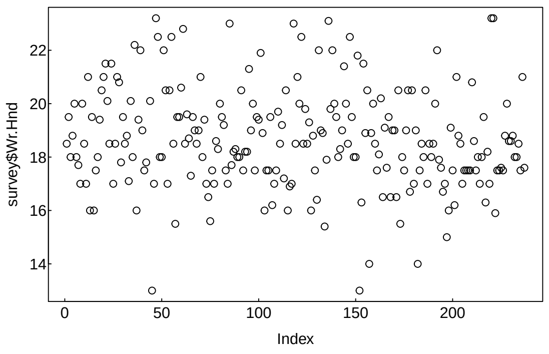


3.5.1.4 Scatterplot

To create a scatterplot, you use the `plot()` function. A scatterplot creates points (or sometimes bubbles or other symbols) on your chart. Each point corresponds to an observation in your data. You've probably seen or created this type of graphic a million times, so you already know that scatterplots use the Cartesian coordinate system, where one variable is mapped to the x-axis and a second variable is mapped to the y-axis.

The most common high level function used to produce plots in R is the `plot()` function.

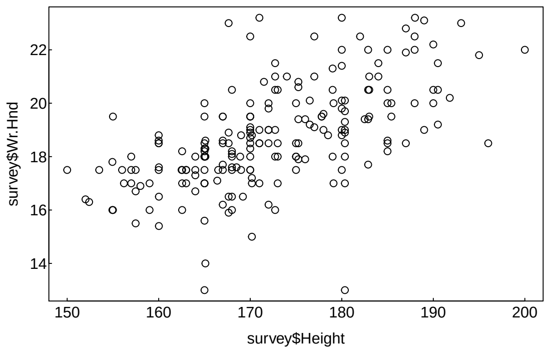
```
par(mar = c(3, 3, 2, 0.1), las=1, mgp=c(1.5,0.1, 0), tcl=0.15)
plot(survey$Wr.Hnd)
```



R has plotted the values of `Wr.Hnd` (on the y axis) against an index since we are only plotting one variable to plot. The index is just the order of the `Wr.Hnd` values in the data frame (1 first in the data frame and 237 last). The `Wr.Hnd` variable name has been automatically included as a y axis label and the axes scales have been automatically set.

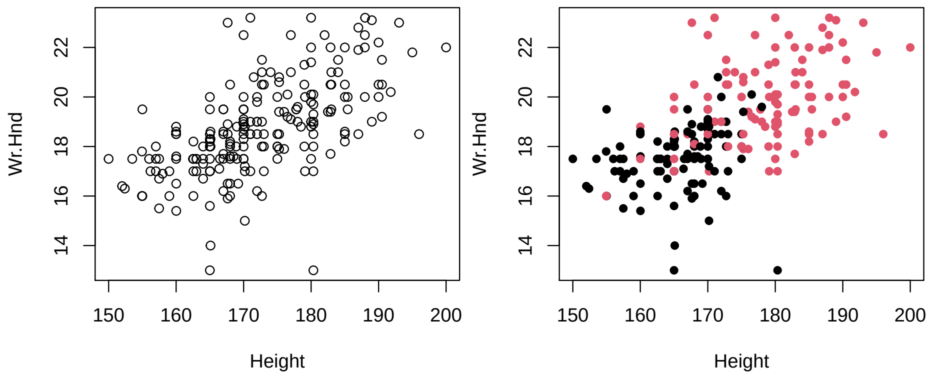
To plot a scatterplot of one numeric variable against another numeric variable we just need to include both variables as arguments when using the `plot()` function. For example to plot `Wr.Hnd` on the y axis and `Height` of the x axis.

```
par(mar = c(3, 3, 2, 0.1), las=1, mgp=c(1.5,0.1, 0), tcl=0.15)
plot(x = survey$Height, y = survey$Wr.Hnd)
```



There is an equivalent approach for these types of plots which often causes some confusion at first. You can also use the formula notation when using the `plot()` function. However, in contrast to the previous method the formula method requires you to specify the y axis variable first, then a ~ and then our x axis variable.

```r
par(mar = c(4, 4, 0.1, 0.1))
plot(Wr.Hnd~Height, data=survey)
plot(Wr.Hnd~Height, data=survey, col=survey$Sex, pch=16)
```



3.5.2 ggplot2 graphics

ggplot2 graphics is based on **ggplot2** package. Because **ggplot2** isn't part of the standard distribution of R, you have to download the package from CRAN and install it. To install the **ggplot2** package, use:

```r
install.packages("ggplot2")
```

And then to load the package, use:

```
library("ggplot2")
```

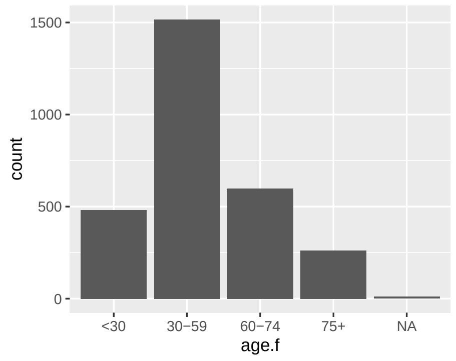
The basic concept of a ggplot2 graphic is that you combine different plot elements into layers. Each layer of a ggplot2 graphic contains information about the following:

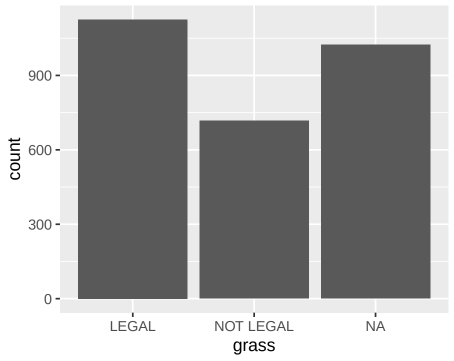
- The data that you want to plot: for `ggplot()`, this must be a data frame.
- A mapping from the data to your plot: this usually is as simple as telling `ggplot()` what goes on the x-axis and what goes on the y-axis.
- A geometric object, or geom in ggplot terminology: the geom defines the overall look of the layer (for example, whether the plot is made up of bars, points, or lines).

3.5.2.1 Bar plot

Please study carefully the following codes and outputs:

```
library(ggplot2)
ggplot(data = gss.2016, mapping = aes(x=age.f)) + geom_bar()
ggplot(data = gss.2016, mapping = aes(x=grass)) + geom_bar()
```

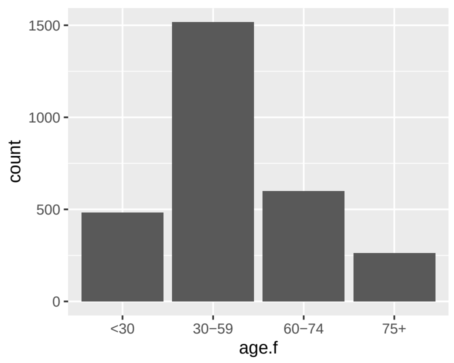


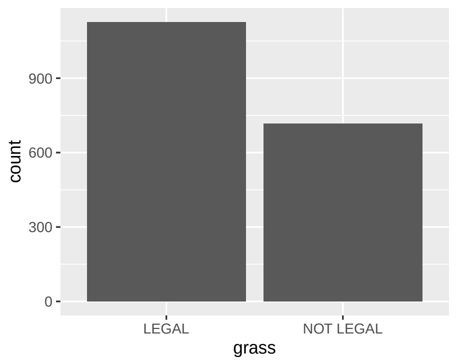


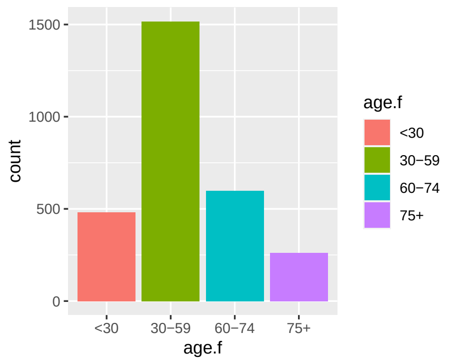
```
ggplot(data = gss.2016, mapping = aes(x=age.f)) + geom_bar() +
  scale_x_discrete(na.translate = F)
ggplot(data = gss.2016, mapping = aes(x=grass)) + geom_bar() +
  scale_x_discrete(na.translate = F)
```

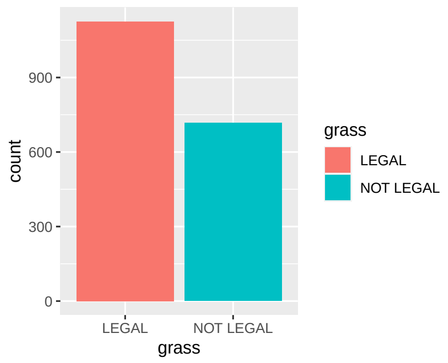
```
ggplot(data = gss.2016, mapping = aes(x=age.f, fill=age.f)) + geom_bar() +
  scale_x_discrete(na.translate = F)
ggplot(data = gss.2016, mapping = aes(x=grass, fill=grass)) + geom_bar() +
  scale_x_discrete(na.translate = F)
```

```
ggplot(data = gss.2016, mapping = aes(x=grass, fill=grass)) + geom_bar() +
  scale_x_discrete(na.translate = F) +
  scale_fill_manual(values = c("#78A678", "#7463AC"), guide=F)
```

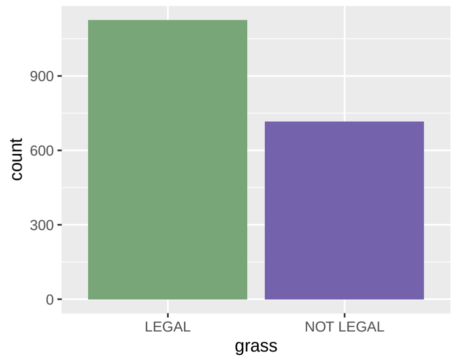

1500
1000
500
0
count
<30
30–59
60–74
75+
age.f

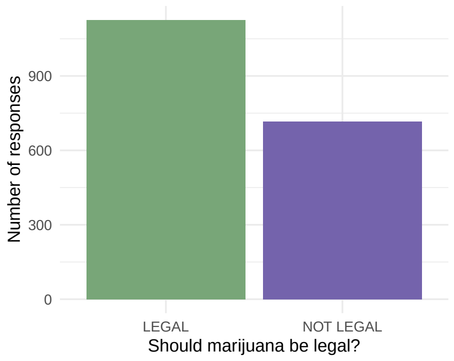

900
600
300
0
count
LEGAL
NOT LEGAL
grass


1500
1000
500
0
count
<30
30–59
60–74
75+
age.f
age.f
<30
30–59
60–74
75+

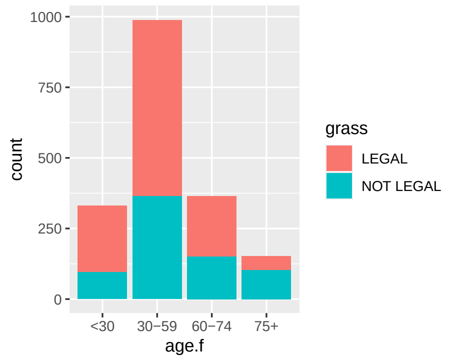

900
600
300
0
count
LEGAL
NOT LEGAL
grass
grass
LEGAL
NOT LEGAL

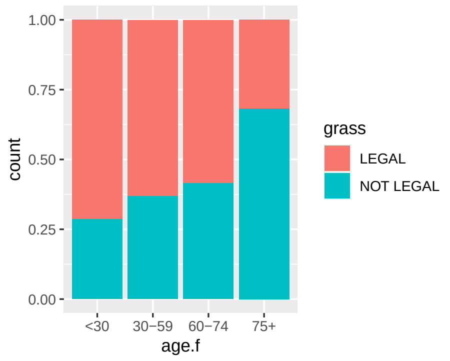
```
ggplot(data = gss.2016, mapping = aes(x=grass, fill=grass)) + geom_bar() +
  scale_x_discrete(na.translate = F) +
  scale_fill_manual(values = c("#78A678", "#7463AC"), guide=F) +
  theme_minimal() +
  labs(x="Should marijuana be legal?", y="Number of responses")
```



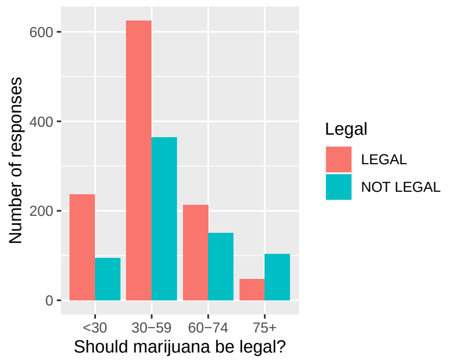


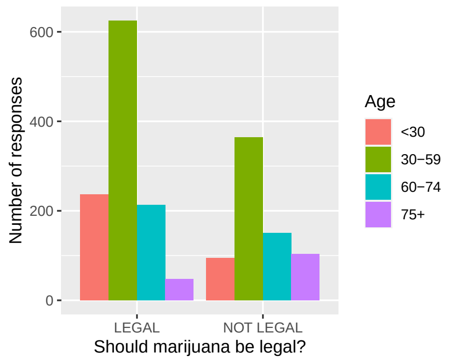
```
ggplot(data = gss.2016[!is.na(gss.2016$grass),],
       mapping = aes(x=age.f, fill=grass)) + geom_bar() +
  scale_x_discrete(na.translate = F)

ggplot(data = gss.2016[!is.na(gss.2016$grass),],
       mapping = aes(x=age.f, fill=grass)) + geom_bar(position = "fill") +
  scale_x_discrete(na.translate = F)
```





```
ggplot(data = gss.2016[!is.na(gss.2016$grass),],
       mapping = aes(x=age.f, fill=grass)) + geom_bar(position = "dodge") +
  scale_x_discrete(na.translate = F) +
    labs(x="Should marijuana be legal?", y="Number of responses", fill="Legal")
ggplot(data = gss.2016[!is.na(gss.2016$age.f),],
       mapping = aes(x=grass, fill=age.f)) + geom_bar(position = "dodge") +
  scale_x_discrete(na.translate = F) +
    labs(x="Should marijuana be legal?", y="Number of responses", fill="Age")
```

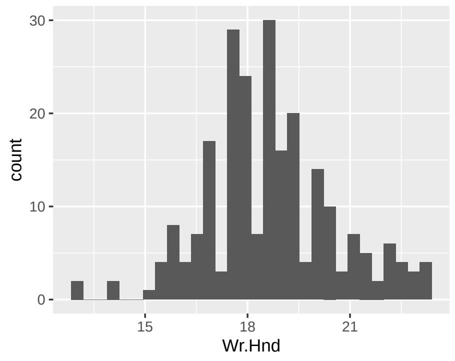


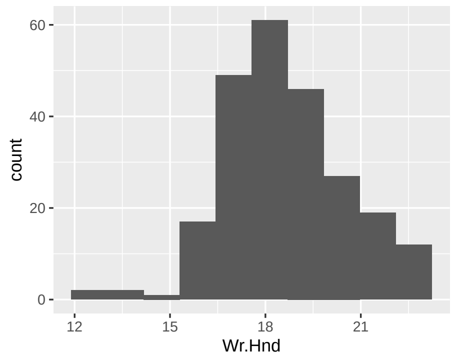


3.5.2.2 Histogram

Please study carefully the following codes and outputs:

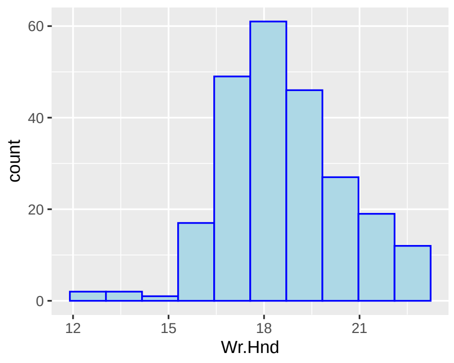
```
ggplot(data = survey, mapping = aes(x=Wr.Hnd)) + geom_histogram()
ggplot(data = survey, mapping = aes(x=Wr.Hnd)) + geom_histogram(bins = 10)
```

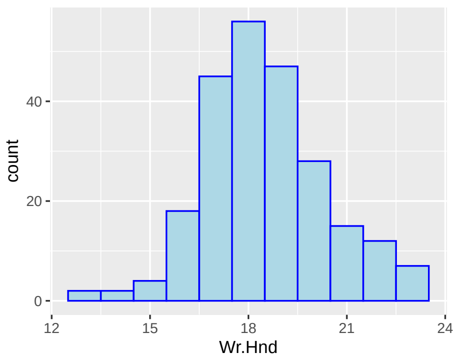


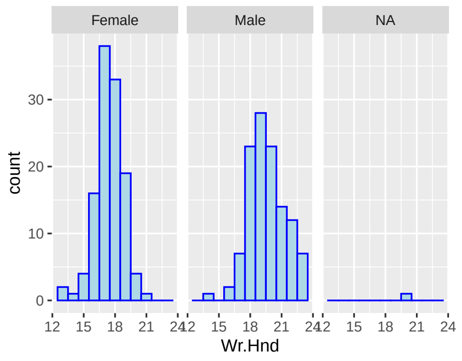


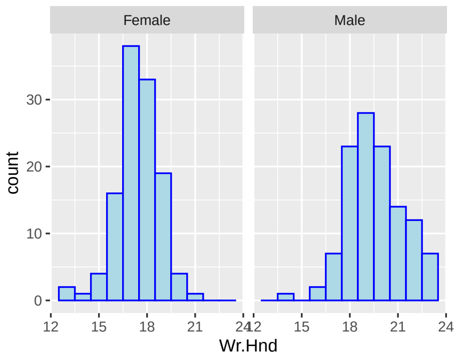
```
ggplot(data = survey, mapping = aes(x=Wr.Hnd)) +
  geom_histogram(bins = 10, fill="lightblue", colour="blue")
ggplot(data = survey, mapping = aes(x=Wr.Hnd)) +
  geom_histogram(binwidth = 1, fill="lightblue", colour="blue")
```

```
ggplot(data = survey, mapping = aes(x=Wr.Hnd)) +
  geom_histogram(binwidth = 1, fill="lightblue", colour="blue") +
  facet_wrap(~Sex)
ggplot(data = survey[!is.na(survey$Sex),], mapping = aes(x=Wr.Hnd)) +
  geom_histogram(binwidth = 1, fill="lightblue", colour="blue") +
  facet_wrap(~Sex)
```







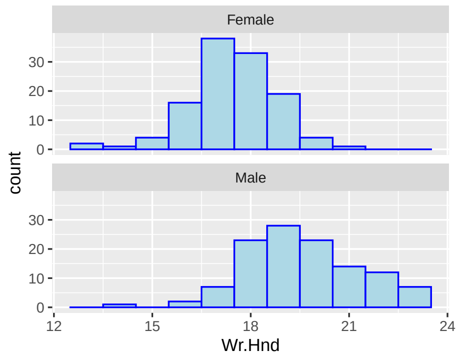


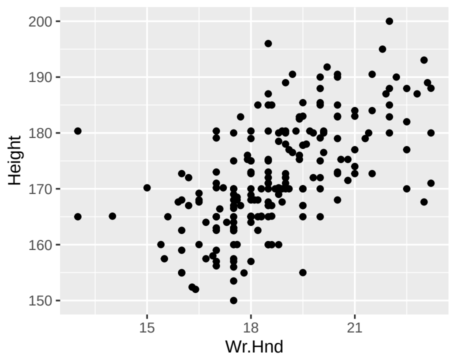
```
ggplot(data = survey[!is.na(survey$Sex),], mapping = aes(x=Wr.Hnd)) +
  geom_histogram(binwidth = 1, fill="lightblue", colour="blue") +
  facet_wrap(~Sex, ncol=1)
```

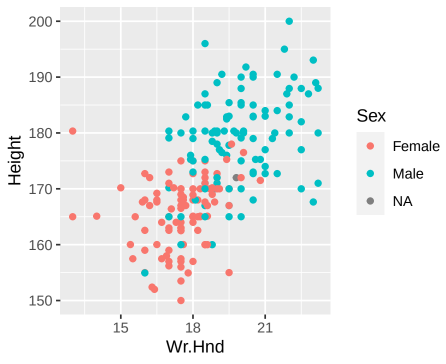
3.5.2.3 Scatterplot

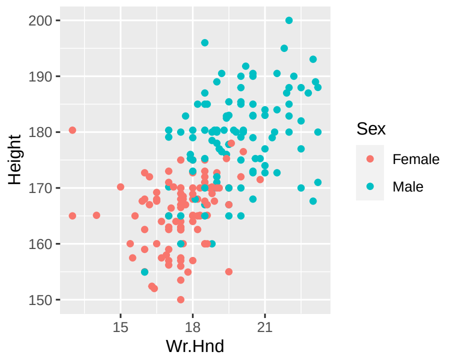
```
ggplot(data = survey, mapping = aes(x=Wr.Hnd, y=Height)) +
  geom_point()
ggplot(data = survey, mapping = aes(x=Wr.Hnd, y=Height, color=Sex)) +
  geom_point()
```

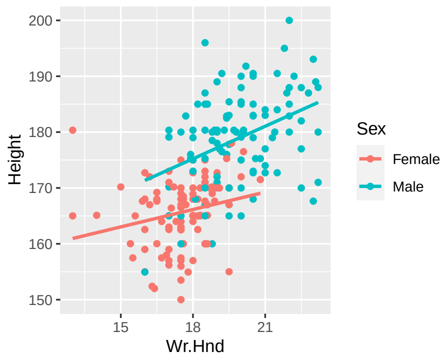
```
ggplot(data = survey, mapping = aes(x=Wr.Hnd, y=Height, color=Sex)) +
  geom_point() + scale_color_discrete(na.translate=F)
ggplot(data = survey, mapping = aes(x=Wr.Hnd, y=Height, color=Sex)) +
  geom_point() + scale_color_discrete(na.translate=F) +
  geom_smooth(se = F, method = lm)
```


Female
Male
count
0
10
20
30
12
15
18
21
24
Wr.Hnd


Height
150
160
170
180
190
200
15
18
21
Wr.Hnd

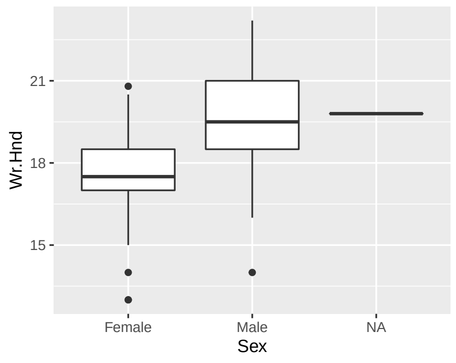

Height
Wr.Hnd
Sex
Female
Male
NA

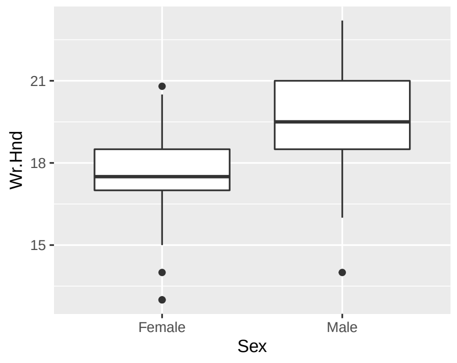

Height
Wr.Hnd
Sex
Female
Male


Height
Wr.Hnd
Sex
Female
Male

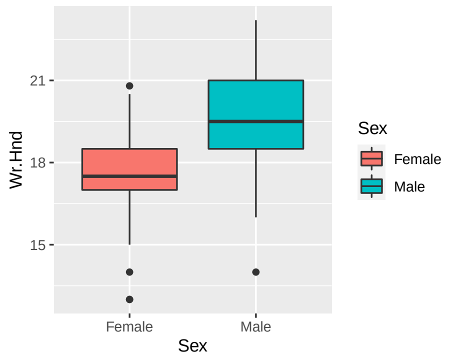
3.5.2.4 Box plot

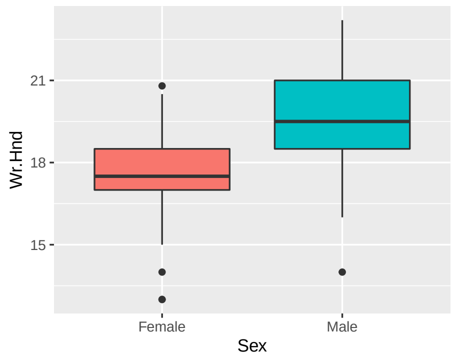
```
ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd)) +
  geom_boxplot()
ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd)) +
  geom_boxplot() + scale_x_discrete(na.translate=F)
```





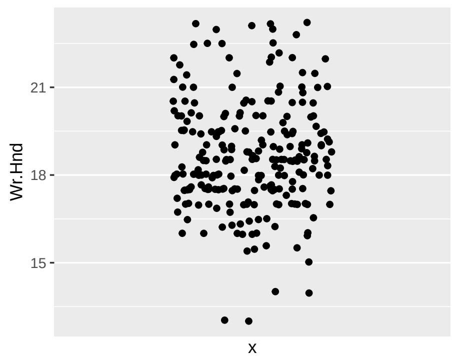
```
ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd, fill=Sex)) +
  geom_boxplot() + scale_x_discrete(na.translate=F)

ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd, fill=Sex)) +
  geom_boxplot() + scale_x_discrete(na.translate=F) +
  scale_fill_discrete(guide=F)
```

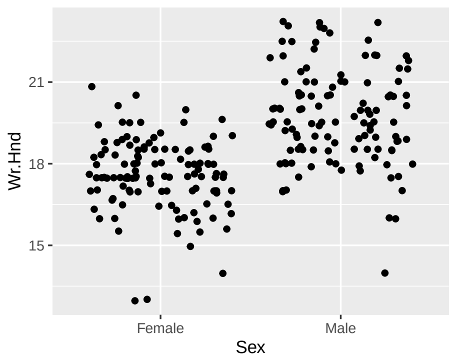




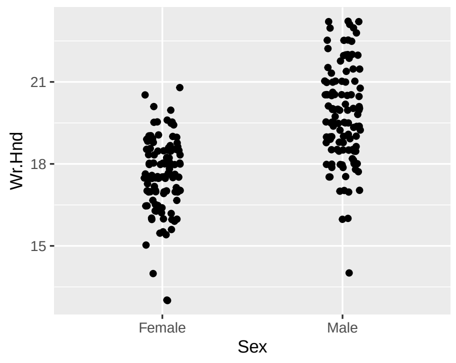
3.5.2.5 Stripchart

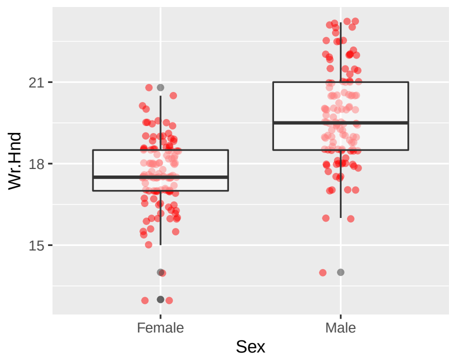
```
ggplot(data = survey, mapping = aes(x=1, y=Wr.Hnd)) +
  geom_jitter() + scale_x_discrete(na.translate=F)
ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd)) +
  geom_jitter() + scale_x_discrete(na.translate=F)
```





```
ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd)) +
  geom_jitter(width = 0.1) + scale_x_discrete(na.translate=F)
ggplot(data = survey, mapping = aes(x=Sex, y=Wr.Hnd)) +
  geom_jitter(width = 0.1, alpha=0.5, color="red") +
  scale_x_discrete(na.translate=F) +
  geom_boxplot(alpha=0.5)
```





Appendix A

Recaps in 1 minutes or less

A.1 Possibilities of using R

- Console: type a command and hit Enter
  - *Base R* Console
  - *RGui* Console on Windows
  - *RStudio* Console
- Script: edit a text files and hit Ctrl+R or Ctrl+Enter
  - *RGui* Script Window (Ctrl+R)
  - *RStudio* Source Pane (Ctrl+Enter)
- Point and Click
  - *R Commander*
  - *jamovi*, *JASP* etc.

A.2 Console features

- history of command: Up/Down arrows
- autocompletion: TAB
- continuation prompt: Esc

A.3 Advantages of Script editor in RStudio

- multi-line editor
- full-featured text editor: e.g. row numbering, syntax highlighting
- autocompletion of filenames, function names, arguments and objects
- cross-platform interface to R

- surrounded by integrated graphical environment (workspace, files, plots, help, etc.)

A.4 Useful keyboard shortcuts in RStudio

- Ctrl+Enter: Run commands
- Clipboard Operations (Cut, Copy, Paste Operations): Ctrl+X, Ctrl+C, Ctrl+V
- Ctrl++, Ctrl+-: Zoom in/out
- Ctrl+Shift+C: Comment lines/uncomment lines
- Ctrl+F: Find and replace text within script editor
- Ctrl+S: Save the script file
- Alt+-: Write assignment operator
- Ctrl+Shift+F10: Restart R session

A.5 Base types in R

- character (or string): `"apple juice"`
- integer (whole numbers): `12L`
- double (real numbers, decimal numbers): `12`, `12.4`
- logical (true false type things): `TRUE`, `FALSE`

A.6 Data structures

- *Vector*: one-dimensional, homogeneous
- *Matrix*: two-dimensional, homogeneous
- *Array*: two or more dimensional, homogeneous
- *List*: one-dimensional, heterogeneous
- *Factor*: integer vector with levels, which is a character vector
- *Data frame*: two-dimensional, heterogeneous

Table A.1: Data structures

Dimension	Homogenous	Heterogeneous
1D	Vector, Factor	List
2D	Matrix	Data frame
nD	Array	

A.7 Operators

Table A.2: R operators in order of precedence from highest to lowest

Operator	Description	Example
`::`	access	`MASS::survey`
`$`	component	`my.s$Sex`
`[ [[`	indexing	`my.s$Height[c(2, 45)]`
`^ **`	exponentiation	`2^3`
`- +`	unary minus, unary plus	`-2`
`:`	sequence operator	`1:10`
`%any%` e.g. `%%` `%/%` `%in%`	special operators	`12%%3`
`* /`	multiplication, division	`12*3`
`+ -`	addition, subtraction	`2.3 + 2`
`< > <= >= == !=`	comparisions	`2<=3`
`!`	logical NOT	`!TRUE`
`&`	logical AND	`TRUE & FALSE`
`|`	logical OR	`TRUE | FALSE`
`<-`	assignment	`col <- 12`

R language provides following types of operators:

- Arithmetic Operators: `^` `**`, `-` (unary), `+` (unary), `%%`, `%/%`, `*`, `/`, `-` (binary), `+` (binary)
- Relational Operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, `%in%`
- Logical Operators: `!`, `&`, `|`
- Assignment Operators: `<-`
- Miscellaneous Operators: `::`, `$`, `[`, `[[`, `:`, `?`

A.8 Maths functions

Table A.3: Mathematical functions

Function	Description
abs(x)	Takes the absolute value of x
sign(x)	The signs of `x`
sqrt(x)	Returns the square root of x
exp(x)	Returns the exponential of x
log(x,base=exp(1))	Takes the logarithm of x with base y; if base is not specified, returns the natural logarithm
log10(x);log2(x)	Takes the logarithm of x with base 10 or 2
cos(x);sin(x);tan(x)	Trigonometric functions
round(x,digits=0)	Rounds a numeric input to a specified number of decimal places
floor(x)	Rounds a numeric input down to the next lower integer
ceiling(x)	Rounds a numeric input up to the next higher integer
trunc(x)	Truncates (i.e. cuts off) the decimal places of a numeric input

A.9 String functions

Function	Description	Example
`paste();paste0(sep="")`	Concatenate strings	`paste('a','b',sep='=')`
`nchar(x)`	Count the number of characters	`nchar('alma')`
`substr(x,start,stop)`	Substrings of a character vector	`substr('alma', 3, 5)`
`tolower(x)`	Convert to lower-case	`tolower('Kiss Géza')`
`toupper(x)`	Convert to upper-case	`toupper('Kiss Géza')`
`chartr(old,new,x)`	Translates characters	`chartr('it','ál','titik')`
`cat(sep=" ")`	Concatenate and print	`cat('alma','fa\n',sep='')`
`grep();grepl();regexpr()`	Pattern matching	`grepl(pattern='lm',x='alma')`
`sub();gsub()`	Pattern matching and replacement	`gsub('lm',repl='nyj',x='alma')`

A.10 Base R Statistical Functions

Table A.5: Base R Statistical Functions

Function	Description	Example	Value of Example
max(x)	The largest value of `x`	max(1:10)	10
min(x)	The smallest value of `x`	min(11:20)	11
sum(x)	The sum of all the values of `x`	sum(1:5)	15
prod(x)	The product of all the values of `x`	prod(1:5)	120
mean(x)	Mean of `x`	mean(1:10)	5.5
median(x)	Median of `x`	median(1:10)	5.5
range(x)	The minimum and the maximum	range(1:10)	1 10
sd(x)	Standard deviation of x	sd(1:10)	3.03
var(x)	Variance of `x`	var(1:10)	9.17
cor(x,y)	Correlation between `x` and `y`	cor(1:10,11:20)	1

A.11 Regular sequences

Function	Description	Example	Value of Example
`from:to`	generates a sequence from `from=` to `to=` in steps of 1 or -1	`1:5`	`1 2 3 4 5`

Function	Description	Example	Value of Example
`seq(from, to, by, length.out)`	generate regular sequences	`seq(from=2, to=10, by=2)`	`2 4 6 8 10`
`rep(x, times, each)`	replicate elements of vectors	`rep(x=0, times=3)`	`0 0 0`
`paste(sep, collapse)`	concatenate vectors	`paste("No", 1:3, sep="_")`	`"No_1" "No_2" "No_3"`

A.12 Subsetting

Data structure	Example
• Vector • Factor • List • Data frame	• `x[3]` • `x[1:3]` • `x[c(2, 3, 1)]` • `x[-2]` • `x[-c(1, 2)]` • `x["Jane"]` • `x[c("Jane", "Mark")]` • `x[c(T, F, T, T)]` • `x[[2]]` • `x[["Jane"]]`
• Matrix • Data frame	• `x[1, 2]` • `x[, 2]` • `x[2:4, ]` • `x[c(2, 3, 1), c("name", "sport")]` • `x[c("Jane", "Mark"), c(T, F, T)]`
• Array (3D)	• `x[1:3, c(2,1), 2:3]`
• List, Data frame	• `x$name`

A.13 Packages

Operation	Example
Install a package from CRAN	`install.packages("package_name")`
Load a package	`library(package_name)`

A.14 Reading/Writing data files

Operation	Example
Import text files	`read.table(file, sep, dec, header, fileEncoding)`
Import Excel or SPSS files	`rio::import(file)`
Export text files	`write.table(x, file, sep, dec, row.names, quote, fileEncoding)`
Export Excel or SPSS files	`rio::export(x, file)`

A.15 Filter

Data structure	Example
• Vector	• `x[x < 2]` • `x[x == "Jane"]` • `x[x == "Jane" | x == "Mark"]`
• Data frame	• `x[x$v1 < 2, ]` • `x[x$v2 == "Jane", ]` • `x[x$v2 == "Jane" \| x$v2 == "Mark", ]`

A.16 Sort

Data structure	Example
• Vector	• `sort(x)` • `sort(x, decreasing=T)`
• Factor	• `sort(table(x))` • `sort(table(x), decreasing=T)`

Data structure	Example
• Data frame	• `x[order(x$name), ]` • `x[order(x$name, decreasing=T), ]`

A.17 Data type conversion

Conversion	Example
• Numeric vector to factor • Character vector to factor	• `factor(x)`
• Character to numeric	• `as.numeric(x)`
• Factor to character	• `as.character(x)`
• Factor to numeric	• `as.numeric(as.character(x))`

A.18 Transformation

Transformation	Example
• Numeric to factor	• `cut(x, breaks, labels)`
• Factor to factor	• `car::recode(x, '')`

• Numeric to numeric	• mathematical functions
	  – `round()`, `log()`
	  – `exp()`, `sin()`, etc.
	• operators
	  – `+`,`-`, `/`, `*`, etc.

A.19 Descriptive statistics

Descriptive statistics	Example
• Measurements	• `psych::describe()` • `psych::describeBy()` • `DescTools::Desc()`
• Tables	• `table(useNA="ifany")` • `DescTools::Desc()`
• Plots	• Traditional graphics – `hist()` – `boxplot()` – `stripchart()` – `plot()` – `barplot()` • ggplot2 graphics `ggplot() +` – `geom_histogram()` – `geom_boxplot()` – `geom_jitter()` – `geom_point()` – `geom_bar()`

Appendix B

Terminology

B.1 Terms in Statistics

Bar chart A graph used to display summary statistics such as the *mean* (in the case of a *scale variable*) or the *frequency* (in the case of a *nominal variable*).

Boxplot a visual representation of data that shows central tendency (usually the median) and spread (usually the interquartile range) of a numeric variable for one or more groups; boplots are often used to compare the distribution of a continuous variable across several groups

Case / Observation A case is the unit of analysis; one person or other entity. In psychology, this is normally the data deriving from a single participant. In some research, the cases will not be people. For example, we may be interested in the average academic attainment for pupils from different schools. Here, the cases would be the schools. In R, a single row of data in a data frame represents a case.

Categorical variable variable measured in categories; there are two types of categorical variables: ordinal variables have categories with a logical order (e.g., Liker scales), while nominal variables have categories with no logical order (e.g., religious affiliation)

Data A set of values. A data set is typically made up of a number of *variables*. In quantitative research, data are numeric.

Descriptive statistics Procedures that allow you to describe data by summarising, displaying or illustrating them. Often used as a general term for summary descriptive statistics: *measures of central tendency* and *measures of dispersion*. Graphs are descriptive statistics used to illustrate the data.

Frequency/ies The number of times a particular value of a variable occurs.

Histogram a visual display of data used to examine the distribution of a numeric variable

Line graph a visual display of data often used to examine the relationship between two continuous variables or for something measured over time

Missing values A data set may be incomplete, for example, if some observations or measurements failed or if participants didn't respond to some questions. It is important to distinguish these missing data points from valid data. Missing values are the values R has reserved for each variable to indicate that a data point is missing. These missing values can either be specified by the user (user missing) or automatically set by R (`NA`).

Nominal data Data collected at a level of measurement that yields nominal data (nominal just means 'named'), also referred to as 'categorical data', where the value does not imply anything other than a label; for example, 1 = male and 2 = female.

Observation / Case An observation is the unit of analysis; one person or other entity. In psychology, this is normally the data deriving from a single participant. In some research, the cases will not be people. For example, we may be interested in the average academic attainment for pupils from different schools. Here, the observation would be the schools. In R, a single row of data in a data frame represents an observation.

Participant People who take part in an experiment or research study. Previously, the word 'subject' was used, and still is in many statistics books.

Population The total set of all possible scores for a particular variable.

Quantitative data Is used to describe numeric data measured on any of the four levels of measurement. Sometimes though, the term 'qualitative data' is then used to describe data measured with nominal scales.

Sample A subset of observations from some *population* that is often analyzed to learn about the poplulation sampled.

Scatterplot a graph that shows one dot for each observation in the data set

Summary statistics used to provide an overview of the characteristics of a sample; this typically includes measures central tendency and spread for numeric variables and the frequencies and percentages of categorical variables

Statistics A general term for procedures for summarising or displaying data (*descriptive statistics*) and for analysing data (*inferential statistical tests*).

Variable a measured characteristic of some entity (e.g., income, years of education, sex, height, blood pressure, smoking status, etc.); A variable in R is represented by a column in data frame.

B.2 Terms in R

Argument information input into a function that controls how the function behaves

Assigning assigning a value to an object is done by using a left-arrow (`<-`), with the arrow separating the name of the object on the left from the expression itself on the right: `object_name <- expression`

Character a basic data type in R that comprises things that cannot be used in mathematical operations; often, character variables are names, addresses, zip codes, or other similar values

Comment Statements included in code but not analyzed; in R, comment is denoted by hashtag (`#`) and is often used to clarify the codes

Constants Constants, as the name suggests, are entities whose value cannot be altered. Basic types of constant are double constants, integer constants, logical constants and character constants.

csv a file extension indicating that the file contains comma separated values or semicolon separated values

Data frame an object type in R that holds data with values in rows and columns with rows treated as observations and columns treated as variables

Data management the procedures used to prepare the data for analysis; data management often includes recoding variables, ensuring that missing values are treated properly, checking and fixing data types, and other data-cleaning procedures

Data types in R, these include numeric (double, integer), character, logical; the data type suggests how a variable was measured and recorded or re-coded, and different analytic strategies are used to manage and analyze different variable types

Expression An expression is an instruction to perform a particular task. An expression is any sequence of R constants, object's names, operators, function calls, and parentheses. An expression has a type as well as a value.

Factor A categorical variable and its value labels. Value labels may be nothing more than "1," "2,"..., if not assigned explicitly. More formally, a type of object that represents a categorical variable. It stores its labels in its levels attribute.

Function a set of machine-readable instructions to perform a task in R; often, the task is to conduct some sort of data management or analysis, but there are also functions that exist just for fun.

Index The order number of a variable in a data set or the subscript of a value in a object. The number of the component in a list or data frame, or of an element in a vector.

Integer a similar data type to numeric, but containing only whole numbers

Length The number of observations/cases in a variable, including missing values, or the number of variables in a data set. For vectors, it is the number of its elements (including NAs). For lists or data frames, it is the number of its components.

Levels The values that a categorical variable can have. Actually stored as a part of the factor itself in what appears to be a very short character variable (even when the values themselves are numbers).

List A set of objects of any class. Can contain vectors, data frames, matrices and even other lists.

Matrix A data set that must contain only one type of variable, e.g. all numeric or character. More formally, a two-dimensional array; that is, a vector with a dim attribute of length 2. Information, or data elements, stored in a rectangular format with rows and columns.

NA the R placeholder for missing values, often translated as "not available."

NaN A missing value. Stands for Not a Number. Something that is undefined mathematically such as zero divided by zero.

NULL An object you can use to drop variables or values. E.g. `mydata$x <- NULL` drops the variable `x` from the data set `mydata`. Assigning it to an object deletes it.

Numeric A variable that contains only numbers. This can be double and integer.

Object information stored in R; data analysis and data management are then performed on these stored objects. Includes data frames, vectors, factors, matrices, arrays, lists and functions.

Operators An operator is a symbol that tells the compiler to perform specific mathematical, logical, or other manipulations. R language is rich in built-in operators and provides following types of operators: Arithmetic Operators, Relational Operators, Logical Operators, Assignment Operators, Miscellaneous Operators.

Package a collection of functions and datasets for use in R that usually has a specific purpose, such as conducting partial correlation anaylyses (**ppcor** package)

Precedence of operations the order in which mathematical operations should be performed when solving an equation: parentheses, exponents, multiplication, division, addition, and subtraction (PEMDAS)

Recycling rules If one tries to add two structures with a different number of elements, then the shortest is recycled to length of longest. That is, if for instance you add `c(1, 2, 3)` to a six-element vector then you will really add `c(1, 2, 3, 1, 2, 3)`. If the length of the longer vector is not a multiple of the shorter one, a warning is given.

RMarkdown file RMarkdown provides an authoring framework for data science. You can use a single R Markdown file to both 1) save and execute code; 2) generate high quality reports that can be shared with an audience.

sav the file extension for a data file saved in a format for the Statistical Package for Social Sciences (SPSS) statistical software

Script file a text file in R similar to something written in the Notepad text editor on a Windows computer or the TextEdit text editor on a Mac computer; it is saved with a `.R` file extension

Vector Vectors are one-dimensional and homogenous data structures. It can exist on its own in memory or it can be part of a data frame. More formally, a set of values that have the same base type. A vector can be a vector of characters, logical, integers or double.

Working directory R uses a working directory, where R will look, by default, for files you ask it to load. It also where, by default, any files you write to disk will go.

Workspace A temporary work area in which all R computation happens. Data that exists there will vanish if not saved to your hard drive before quitting R. More formally, the area of your computer's main memory where R does all its work. Data must be loaded into it from files, and packages must be loaded into it from the library, before you can use either.

B.3 Terms in Statistics and R

Table B.1: Terms in statistics and R

Terms in Statistics	Terms in R
• dataset • sample	• data frame
• observation	• rows in a data frame
• variable	• columns in a data frame
• categorical variable • qualitative variable – nominal variable – ordinal variable	• factor

Terms in Statistics	Terms in R

• numeric variable
• quantitative variable
   – continuous variable
   – discrete variable
	• numeric vector
	   – double vector
	   – integer vector

Appendix C

Miscellaneous

C.1 Rules of using R

- use *RStudio*
- use *RStudio* in project-oriented environment
- use *RMarkdown* files in *RStudio*
- use as many comments as possible

C.2 Good to know

- R is case sensitive: `Apple` and `apple` are different objects.
- Use a semicolon to put two or more commands on a single line: `a <- 2+2; a`
- Force R to print the value of expression by using parentheses: `(a <- 2+2)`

C.3 Why is my code broken?

- Are all your parentheses in the right places?
- Do you have commas where you should?
- How's your capitalization?
- What about continuation prompt?
- Did you load the package you're trying to use?
- If none of these fix your problem, try googling the error message R gives you. There's usually a good StackOverflow question on whatever you're trying to accomplish.

C.4 Other resources

- R Cheatsheets contain information-dense infographics for many of the packages we've used in this course, and plenty other useful tools you may need in your own work.
- My collection (in Hungarian)