Introduction to R

Claudia A Engel

Last updated: October 16, 2023

Contents

Prerequisites

- Geared specifically towards users who are **new** to R.

- Have R and RStudio installed (see setup instructions below).

Setup Instructions

R and **RStudio** are separate downloads and installations. R is the underlying statistical computing environment. It can be used on its own, 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 R before you launch RStudio.

Be aware that the most recent version of RStudio requires R version 3.3.0 or later.

macOS

If you already have R and RStudio installed

- Open RStudio, and click on "Help" > "Check for updates". If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check the version of R you are using, start RStudio and the first thing that appears on the terminal indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it.

If you don't have R and RStudio installed

- Download R from the CRAN website.
- Select the `.pkg` file for the latest R version. Make sure you choose the appropriate version for your hardware (Apple silicon or Intel)
- Double click on the downloaded file to install R
- Go to the RStudio download page

- Follow the link to download the latest version under “Install Rstudio”
- 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.

Windows

If you already have R and RStudio installed

- Open RStudio, and click on “Help” > “Check for updates”. If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check which version of R you are using, start RStudio and the first thing that appears in the console indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it. You can check here for more information on how to remove old versions from your system if you wish to do so.

If you don’t have R and RStudio installed

- Download R from the CRAN website.
- Run the `.exe` file that was just downloaded
- Go to the RStudio download page
- Follow the link to download the latest version under “Install Rstudio”
- 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.

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
- Follow the link to download the latest version under “Install Rstudio”
- Once it’s installed, open RStudio to make sure it works and you don’t get any error messages.

Acknowledgements

Part of the materials for this tutorial are adapted from http://datacarpentry.org and http://softwarecarpentry.org.

Chapter 1

R and Rstudio

Learning Objectives

- Be familiar with reasons to use R.
- Understand how R relates to RStudio.
- Be able to navigate the RStudio interface including the Script, Console, Environment, Help, Files, and Plots windows.
- Create an R Project in RStudio.
- Set a "working" directory.
- Send commands from the Script window to the Console in RStudio.
- Install additional packages with RStudio and using R commands.

1.1 What is R? What is RStudio?

The term "R" is used to refer to both the programming language to write scripts and the software ("environment") that interprets the scripts written in R. It is an alternative to statistical packages like SAS, SPSS, or Stata, which lets you perform a wide variety of data analysis, statistics, and visualization.

RStudio is currently a very popular way to not only write your R scripts but also to interact with the R software. To function correctly, RStudio needs R and therefore both need to be installed on your computer.

1.2 Why learn R?

1.2.1 R does not involve lots of pointing and clicking, and that's a good thing

The learning curve might be steeper than with other software, but with R, the results of your analysis does not rely on remembering a succession of pointing and clicking, but instead on a series of written commands, and that's a good thing! So, if you want to redo your analysis because you collected more data, you don't have to remember which button you clicked in which order to obtain your results, you just have to run your script again.

Working with scripts makes the steps you used in your analysis clear, and the code you write can be inspected by someone else who can give you feedback and spot mistakes.

Working with scripts forces you to have a deeper understanding of what you are doing, and facilitates your learning and comprehension of the methods you use.

1.2.2 R code is great for reproducibility

Reproducibility is when someone else (including your future self) can obtain the same results from the same dataset when using the same analysis.

R integrates with other tools to generate manuscripts from your code. If you collect more data, or fix a mistake in your dataset, the figures and the statistical tests in your manuscript are updated automatically.

An increasing number of journals and funding agencies expect analyses to be reproducible, so knowing R will give you an edge with these requirements.

1.2.3 R is multidisciplinary, extensible, and popular

With close to 20,000 packages on CRAN (The Comprehensive R Archive Network) that can be installed to extend its capabilities, R provides a framework that allows you to combine statistical approaches from many scientific disciplines to best suit the analytical framework you need to analyze your data. For instance, R has packages for image analysis, mapping, time series, text mining, and a lot more.

1.2.4 R works on data of all shapes and sizes

The skills you learn with R scale easily with the size of your dataset. Whether your dataset has hundreds or millions of lines, it won't make much difference to you.

R is designed for data analysis. It comes with special data structures and data types that make handling of missing data and statistical factors convenient.

R can connect to spreadsheets, databases, and many other data formats, on your computer or on the web.

1.2.5 R produces high-quality graphics

The plotting functionalities in R are endless, and allow you to adjust any aspect of your graph to convey most effectively the message from your data.

1.2.6 R has a large user community

Thousands of people use R daily. Many of them are willing to help you through mailing lists and websites such as Stack Overflow.

1.2.7 Not only is R free, but it is also open-source and cross-platform

Anyone can inspect the source code to see how R works. Because of this transparency, there is less chance for mistakes, and if you (or someone else) find some, you can report and fix bugs.

1.3 Knowing your way around RStudio

Let's start by learning about RStudio, which is an Integrated Development Environment (IDE) for working with R.

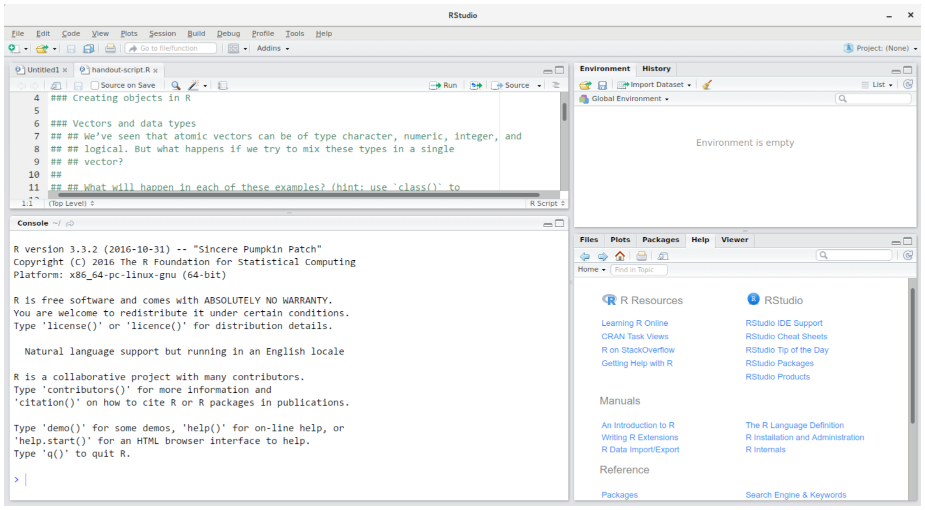
The RStudio IDE open-source product is free under the Affero General Public License (AGPL) v3. The RStudio IDE is also available with a commercial license and priority email support from RStudio, Inc.

We will use RStudio IDE to write code, navigate the files on our computer, inspect the variables we are going to create, and visualize the plots we will generate. RStudio can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that we will not cover during the workshop.

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.



```
### 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 1.1: The RStudio Interface

1.4 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 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.

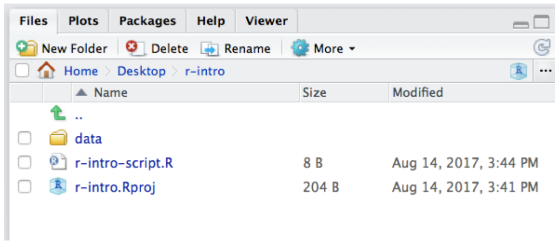
Whenever you create a project with RStudio it creates a working directory for you and remembers its location (allowing you to quickly navigate to it) and optionally preserves custom settings and open files to make it easier to resume work after a break. Below, we will go through the steps for creating 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 subdirecory 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 `Shift` + `Cmd` + `N`) to open a new file
- Save the empty script as `r-intro-script.R` in your `r-intro` directory.

Your `r-intro` directory should now look like in Figure 1.2.



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

1.4.1 The working directory

The working directory is an important concept to understand. It is the place where R will look for and save files. Whenever you start R/RStudio it must have a working directory and will automatically determine which it is. The determination is based on how exactly you start R/RStudio (open a project, open without loading a file, open with a file, etc.). It is important you know what your working directory is. When you write code for your project, your scripts **must refer to external files always in relation to your working directory** otherwise you will get error messages.

To check which working directory R thinks it is in:

```r
getwd()
```

To change to a different working directory in R go to the Console and type:

```r
setwd("Path/To/Desired/Workingdirectory")
```

You can also use the RStudio interface to set a different working directory, like seen in Figure 1.3.

Alternatively, you can use the shortcut `Ctrl` + `Shift` + `H` to set a working directory in RStudio.

1.4.2 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

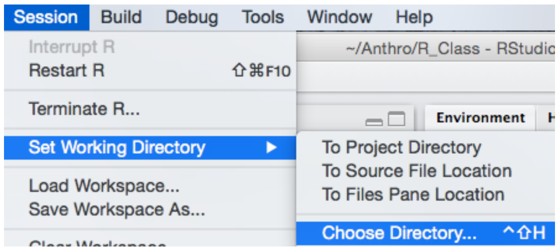


Figure 1.3: How to set a working directory with the RStudio interface

can be especially helpful when you have multiple projects. In general, you may create directories (folders) for **scripts**, **data**, and **documents**.

- `data/` Use this folder to store your raw data and intermediate datasets you may create for the need of a particular analysis. For the sake of transparency and provenance, you should *always* keep a copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically (i.e., with scripts, rather than manually) as possible. Separating raw data from processed data is also a good idea. For example, you could have subfolders in your `data` directory named `data/raw/` and `data/processed` that woudl contain the respective raw and processed files. I also like to log my data processing steps in a simple textfile that I keep there as well.
- `documents/` If you are wroking on a paper this would be a place to keep outlines, drafts, and other text.
- `scripts/` This would be the location to keep your R scripts. Again, depending on the complexity, you may want to add subfolders that contain, for example all the plotting scripts, or all the datas cleaning scripts.

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.

1.5 Interacting with R

The basis of programming is that we write down instructions for the computer to follow, and then we tell the computer to follow those instructions. We write, or *code*, instructions in R because it is a common language that both the computer and we can understand. We call the instructions *commands* and we tell the computer to follow the instructions by *executing* (also called *running*) those commands.

There are two main ways of interacting with R: by using the **console** or by using **script files** (plain text files that contain your code).

1.5.1 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.

> Challenge
>
> - Use R to determine what your working directory is.
> - Use R to change your working directory to some other place. What do you notice in the RStudio Files window?
> - Use RStudio to change back to your previous working directory (r-intro) What do you notice in the RStudio Console?

1.5.2 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.

```r
# 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 readibility.

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.

In addition to shortcuts RStudio also provides autocompletion. If you begin typing a command or the name of a variable or (under certain conditions) even filenames you have on your latop and thenb hit the `Tab` key, it will make suggestions and relieve you from typing. More on code completion in RStudio is here: https://support.rstudio.com/hc/en-us/articles/205273297-Code-Completion

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

Chapter 2

Getting Started with R

Learning Objectives

- Create R objects and and assign values to them.
- Use comments to inform script.
- Do simple arithmetic operations in R using values and objects.
- Call functions with arguments and change their default options.
- Inspect the content of vectors and manipulate their content.
- Subset and extract values from vectors.
- Correctly define and handle missing values in vectors.
- Use the built-in RStudio help interface
- Interpret the R help documentation
- Provide sufficient information for troubleshooting with the R user community.
- Download, install, and load R packages.

2.1 Creating objects in R

To do useful and interesting things in R, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
x <- 3
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as `3` **goes into** `x`. You can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing Alt + - (push Alt at the same time as the - key) will write `<-` in a single keystroke.

Here are a few rules as of how to name objects in R.

- Objects can be given any name such as `x`, `current_temperature`, or `subject_id`.
- You want your object names to be explicit and not too long.
- They **cannot** start with a number (`2x` is not valid, but `x2` is).
- R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`).
- There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it is allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). If in doubt, check the help to see if the name is already in use.
- It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it is best to avoid them.
- It is also recommended to use *nouns for variable names*, and *verbs for function names*.
- It's important to be consistent in the styling of your code (where you put spaces, how you name variables, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators.
  In R, three popular style guides are Google's, Jean Fan's and the tidyverse's. The tidyverse's is very comprehensive and may seem overwhelming at first. You can install the `lintr` to automatically check for issues in the styling of your code.

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```r
area_hectares <- 1.0    # doesn't print anything
area_hectares           # typing the name of the object prints the value of `area_hect
(area_hectares <- 1.0)  # and so does putting parenthesis around the call
```

Now that R has `area_hectares` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (area in acres is 2.47 times the area in hectares):

```r
area_hectares  * 2.47
```

We can also change a variable's value by assigning it a new value:

```r
area_hectares <- 2.5
area_hectares * 2.47
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the area in acres in a new variable, `area_acres`:

```r
area_acres <- area_hectares * 2.47
```

and then change `area_hectares` to 50.

```r
area_hectares <- 50
```

> Challenge
>
> What do you think is the current content of the object `area_acres`? 123.5 or 6.175?

2.1.1 Comments

The comment character in R is `#`, anything to the right of a `#` in a script will be ignored by R. It is useful to leave notes, and explanations in your scripts. RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard Ctrl + Shift + C. If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press Ctrl + Shift + C.

> Challenge
>
> Create two variables `r_length` and `r_width` and assign them values.
>
> Calculate the area based on the current values of `r_length` and `r_width` and assign the result to a new, third variable `r_area`.
>
> Demonstrate that changing the values of either `r_length` and `r_width` does not affect the value of `r_area`. Make ample use of comments in your code.

2.1.2 Functions and their arguments

Functions are "canned scripts" that automate a series of commands one might want to apply frequently.

Many functions are predefined, or can be made available by importing R *packages* (more on that later). A function usually gets one or more inputs called *arguments*. Functions often (but not always) return a *value*.

An example would be the function `sqrt()`, which calculates the square root. The input (=the *argument*) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```r
sqrt(64)
```

```r
# or provide the input to the function as a variable:
a <- 64
sqrt(a)

# we can also assign the output to a new variable:
b <- sqrt(a)
b
```

Here, we provide the number `64` as input to the `sqrt()` function, which calculates the square root of the input value, and returns the result. We can also assign the value `64` to a variable `a`, which is then given to the `sqrt()` function,we can also assign the output of the function to a variable, in this case a new variable `b`.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```r
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` or look at the help for this function using `?round`.

```r
args(round)
```

```
#> function (x, digits = 0)
#> NULL
```

```r
?round
```

We see that if we want a different number of digits, we can type `digits=2` or however many we want.

```r
round(3.14159, digits = 2)
```

```
#> [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```r
round(3.14159, 2)
```

```
#> [1] 3.14
```

And if you do name the arguments, you can switch their order:

```r
round(digits = 2, x = 3.14159)
```

```
#> [1] 3.14
```

Note:

- R evaluates function arguments in three steps: first, by *exact matching* on argument name, then by *partial matching* on argument name, and finally by *position*.

- you *do not have to* specify all of the arguments. If you don't, R will use default values if they are specified by the function. If no default value is specified, you will receive an error.

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

Functions usually return someting back to you as output. Whatever they return (a table, some informational text, a logical value, …) is by default written to the console, so you can see it right away.

Oftentimes, however, we want re-use the output of such a function. That is when you assign the output to an R object to be accessed later on.

2.1.3 Objects vs. variables

What are known as `objects` in `R` are known as `variables` in many other programming languages. Depending on the context, `object` and `variable` can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

2.2 Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of weights and assign it to a new object `area_hectares`:

```
area_hectares <- c(21, 34, 39, 54, 55)
area_hectares
```

```
#> [1] 21 34 39 54 55
```

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(area_hectares)
```

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

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(area_hectares)
```

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

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(area_hectares)
```

```
#>  num [1:5] 21 34 39 54 55
```

You can use the `c()` function to add other elements to your vector:

```
area_hectares <- c(area_hectares, 90) # add to the end of the vector
area_hectares <- c(30, area_hectares) # add to the beginning of the vector
area_hectares
```

```
#> [1] 30 21 34 39 54 55 90
```

In the first line, we take the original vector `area_hectares`, add the value `90` to the end of it, and save the result back into `area_hectares`. Then we add the value `30` to the beginning, again saving the result back into `area_hectares`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

A vector can also contain characters:

```r
animals <- c("mouse", "rat", "dog", "octopus")
class(animals)
```

The quotes around "mouse", "rat", etc. are essential here. Without the quotes R will assume there are objects called `mouse`, `rat` and `dog`. As these objects don't exist in R's memory, there will be an error message.

Lastly, we will introduce a vector with logical values (the boolean data type).

```r
has_tail <- c(TRUE, TRUE, TRUE, FALSE)
has_tail
```

We just saw 3 of the 6 main **atomic vector** types (or **data types**) that R uses: `"character"`, `"numeric"` and `"logical"`. These are the basic building blocks that all R objects are built from. The other 3 are:

- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1 + 4i`) and that's all we're going to say about them
- `"raw"` that we won't discuss further

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

> Challenge
>
> - 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 check the data type of your objects):
>
>   ```r
>   num_char <- c(1, 2, 3, 'a')
>   num_logical <- c(1, 2, 3, TRUE)
>   char_logical <- c('a', 'b', 'c', TRUE)
>   tricky <- c(1, 2, 3, '4')
>   ```
>
> - Why do you think it happens?
>
> - You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

2.3 Subsetting vectors

Subsetting (sometimes referred to as extracting or indexing) involves accessing out one or more values based on their numeric placement or "index" within a vector. If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals[2]
```

```
#> [1] "rat"
```

```
animals[c(3, 2)]
```

```
#> [1] "dog" "rat"
```

We can use the colon `:` to select a sequence of indices. `:` is a special function that creates numeric vectors of integers in increasing or decreasing order, for instance `1:10` or `10:1` for instance. We can take advantage of it like this:

```
animals[2:4]
```

```
#> [1] "rat"     "dog"     "octopus"
```

You can exclude elements of a vector using the "-" sign:

```
animals[-2]
```

```
#> [1] "mouse"   "dog"     "octopus"
```

```
animals[-c(1:3)]
```

```
#> [1] "octopus"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 4)]
more_animals
```

```
#> [1] "mouse"   "rat"     "dog"     "rat"     "mouse"   "octopus"
```

R indices start at 1. Programming languages like MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

2.3.1 Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not.

```
has_tail # this is a logical vector
```

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

```
animals[has_tail] # we use it here in the [ ] to subset
```

```
#> [1] "mouse" "rat"   "dog"
```

Typically, however, these logical vectors are not typed out by hand like we just did, but they are created as output of functions or of logical tests.

A typical example is to search for certain strings in a vector. One could use the "or" operator `|` to test for equality to multiple values like so:

```
animals[animals == "frog" | animals == "rat"]
```

```
#> [1] "rat"
```

But this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
animals %in% c("frog", "rat", "cat") # this creates the logical vector
```

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

```
animals[animals %in% c("frog", "rat", "cat")] # we use it here in the [ ] to subset
```

```
#> [1] "rat"
```

```
# The same, but here is how I would typically do it:
animals_to_find <- c("frog", "rat", "cat") # create a vector with the values you are looking for
animals[animals %in% animals_to_find] # apply it in the condition here
```

```
#> [1] "rat"
```

Equivalently, if you wanted to select only the areas above 50:

```
area_hectares > 50    # will return logicals with TRUE for the indices that meet the condition
```

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

```
## so we can use this to select only the values above 50
area_hectares[area_hectares > 50]
```

```
#> [1] 54 55 90
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
area_hectares[area_hectares < 30 | area_hectares > 50]
```

```
#> [1] 21 54 55 90
```

```
area_hectares[area_hectares > 40 & area_hectares < 90]
```

```
#> [1] 54 55
```

Here, `<` stands for "less than", `>` for "greater than", `>=` for "greater than or equal to", and `==` for "equal to". The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

> Challenge
>
> - Can you figure out why `"four" > "five"` returns `TRUE`?

2.4 Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as `NA`.

When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument `na.rm=TRUE` to calculate the result while ignoring the missing values.

```
age <- c(2, 4, 4, NA, 6, NA, 3)
max(age)
```

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

```
sum(age)
```

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

```
max(age, na.rm = TRUE)
```

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

```
sum(age, na.rm = TRUE)
```

```
#> [1] 19
```

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
# Extract elements which are not missing values.
## The ! character is also called the NOT operator
age[!is.na(age)]
```

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

```
# Returns the object with incomplete cases removed. The returned object is atomic.
na.omit(age)
```

```
#> [1] 2 4 4 6 3
#> attr(,"na.action")
#> [1] 4 6
#> attr(,"class")
#> [1] "omit"
```

```
## Count the number of missing values.
## The output of is.na() is a logical vector (TRUE/FALSE equivalent to 1/0) so the sum() functio
sum(is.na(age))
```

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

```
# Extract elements which are complete cases.
complete.cases(age) # this is a logical vector
```

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

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

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

Challenge

1. Using this vector of length measurements, create a new vector with the NAs removed.

   ```
   population <- c(10,24,NA,18,NA,20)
   ```

2. Use the function `sum()` to calculate the total of the `population`.

3. How many of the populations are of size 20 and over?

2.5 Common R Data Structures

Vectors are one of the many **data structures** that R uses. Other important ones are matrices (`matrix`), tables (`data.frame`), lists (`list`), and factors (`factor`).

2.5.1 Matrix

If we arrange data elements of a vector in a two-dimensional rectangular layout we have a matrix. To construct a matrix, we use a function conveniently called `matrix()`.

```
y <- matrix(1:20, nrow=5,ncol=4) # generates 5 x 4 numeric matrix
```

Subset a matrix with [row , column]:

```
y[,4]       # 4th column of matrix
y[3,]       # 3rd row of matrix
y[2:4,1:3]  # rows 2,3,4 of columns 1,2,3
```

2.5.2 List

Lists can have elements of any type. Here is how we construct lists. You may have guessed that to construct a list, we use the `list()` function:

```
myl <- list(id="ID_1", a_vector=animals, a_matrix=y, age=5.3) # example of a list with
myl[[2]] # 2nd component of the list
myl[["id"]] # component named id in list
```

2.5.3 Data frame

Data frames in R are a special case of lists, as they can have elements of any type, but they have to **all be of the same length**.

A data frame is the most common way of storing tabular data in R and something you will likely deal with a lot. As a first approximation, which holds true, probably in the most cases, you can really think of it as a table or a spreadsheet.

Here is how you could construct a data frame.

```
mydf <- data.frame(ID=c(1:4),
                   Color=c("red", "white", "red", NA),
                   Passed=c(TRUE,TRUE,TRUE,FALSE),
                   Weight=c(99, 54, 85, 70),
                   Height=c(1.78, 1.67, 1.82, 1.59))

mydf
```

We will go into more detail about data frames. For now, try the following:

Challenge

1. Create a data frame that holds the following information for yourself, your right and your left neighbor:

- first name
- last name
- lucky number

2. There are a few mistakes in this hand-crafted `data.frame`, can you spot and fix them? Don't hesitate to experiment!

```
animal_data <- data.frame(animal=c("dog", "cat", "sea cucumber", "sea urchin)
                          feel=c("furry", "squishy", "spiny"),
                          weight=c(45, 8 1.1, 0.8))
```

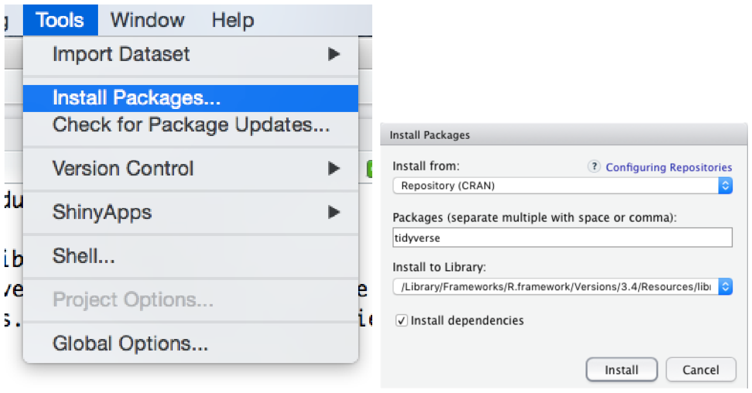
2.6 Extending R base functionality

R comes with a base system and some contributed core packages. This is what you just downloaded. The functionality of R can be significantly extended by using additional contributed packages. Those packages typically contain commands (functions) for more specialized tasks. They can also contain example datasets. We will make use of external packages later.

2.6.1 Installing additional packages

To install additional packages there are two main options:

1. You can use the RStudio interface like this:



Install Packages

Install from: Configuring Repositories
Repository (CRAN)

Packages (separate multiple with space or comma):
tidyverse

Install to Library:
/Library/Frameworks/R.framework/Versions/3.4/Resources/libi

✓ Install dependencies

Figure 2.1: How to install an R package with the RStudio interface

2. You can install from the R console like this:

```
# to install a package called "tidyverse", for example: (more on this later)
install.packages("tidyverse", dependencies = TRUE)
```

(We will talk about the `tidyverse` package collection shortly.)

2.6.2 Make use of the installed packages

In order to actually use commands from the installed packages you also will need to load the installed packages. This can be automated (whenever you launch R it will also load the libraries for you - see for example here) or otherwise you need to sumbit a command:

```
library(tidyverse)
```

or

```
require(tidyverse)
```

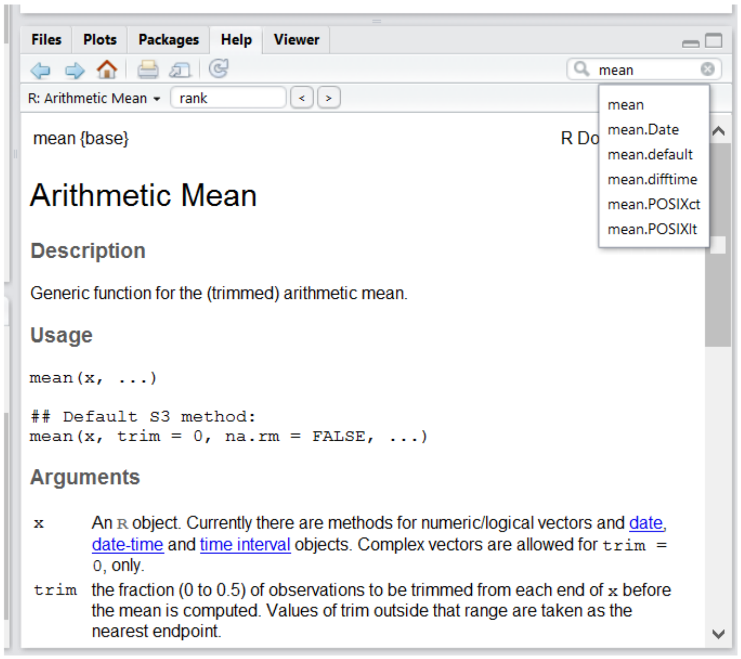
The difference between the two is that `library` will result in an error, if the library does not exist, whereas `require` will result in a warning.

Challenge

1. Google for an R package that might be of interest for your research.
2. Install and load it into R.

2.7 Seeking help

2.7.1 Use the built-in RStudio help interface



mean {base}

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.

trim the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

Figure 2.2: The RStudio help interface

One of the most immediate ways to get help, is to use the RStudio help interface (Figure 2.2. In the default conficuration this panel by default can be found at the lower right hand panel of RStudio. As seen in the screenshot, by typing the word “Mean”, RStudio tries to also give a number of suggestions that you might be interested in. The description is then shown in the display window.

2.7.2 I know the name of the function, but I’m not sure how to use it

If you need help with a specific function, let’s say `barplot()`, you can type:

```
?barplot
```

If you just need to remind yourself of the names of the arguments, you can use:

```r
args(lm)
```

2.7.3 There must be a function to do X but I don't know which one...

If you are looking for a function to do a particular task, you can use the `help.search()` function, which is called by the double question mark `??`. However, this only looks through the installed packages for help pages with a match to your search request

```r
??kruskal
```

If you can't find what you are looking for, you can use the rdocumentation.org website that searches through the help files across all packages available.

Finally, a generic Google or internet search "R <task>" will often either send you to the appropriate package documentation or a helpful forum where someone else has already asked your question.

2.7.4 I am stuck... I get an error message that I don't understand

Start by googling the error message. However, this doesn't always work very well because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. "subscript out of bounds"). If the message is very generic, you might also include the name of the function or package you're using in your query.

However, you should check Stack Overflow. Search using the `[r]` tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers: http://stackoverflow.com/questions/tagged/r

The Introduction to R can also be dense for people with little programming experience but it is a good place to understand the underpinnings of the R language.

The R FAQ is dense and technical but it is full of useful information.

2.7.5 How to ask for help

The key to receiving help from someone is for them to rapidly grasp your problem. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple *reproducible example*. If you can reproduce the problem using a very small data frame instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question. For instance instead of using a subset of your real dataset, create a small (3 columns, 5 rows) generic one. For more information on how to write a reproducible example see this article by Hadley Wickham.

To share an object with someone else, if it's relatively small, you can use the function `dput()`. It will output R code that can be used to recreate the exact same object as the one in memory:

```r
dput(head(iris)) # iris is an example data frame that comes with R and head() is a fun
```

```
#> structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
#>     Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c(1.4,
#>     1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
#>     0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
#>     1L), levels = c("setosa", "versicolor", "virginica"), class = "factor")), row.na
#> 6L), class = "data.frame")
```

If the object is larger, provide either the raw file (i.e., your CSV file) with your script up to the point of the error (and after removing everything that is not relevant to your issue). Alternatively, in particular if your question is not related to a data frame, you can save any R object to a file:

```r
saveRDS(iris, file="/tmp/iris.rds")
```

The content of this file is however not human readable and cannot be posted directly on Stack Overflow. Instead, it can be sent to someone by email who can read it with the `readRDS()` command (here it is assumed that the downloaded file is in a `Downloads` folder in the user's home directory):

```r
some_data <- readRDS(file="~/Downloads/iris.rds")
```

Last, but certainly not least, **always include the output of `sessionInfo()`** as it provides critical information about your platform, the versions of R and the packages that you are using, and other information that can be very helpful to understand your problem.

```r
sessionInfo()
```

```
#> R version 4.3.0 (2023-04-21)
#> Platform: aarch64-apple-darwin20 (64-bit)
#> Running under: macOS Ventura 13.5.1
#>
```

```
#> Matrix products: default
#> BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LA
#>
#> locale:
#> [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> time zone: America/Los_Angeles
#> tzcode source: internal
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> loaded via a namespace (and not attached):
#>  [1] compiler_4.3.0  fastmap_1.1.1   bookdown_0.35   cli_3.6.1
#>  [5] htmltools_0.5.5 tools_4.3.0     rstudioapi_0.14 yaml_2.3.7
#>  [9] rmarkdown_2.22  knitr_1.44      digest_0.6.31   xfun_0.40
#> [13] rlang_1.1.1     evaluate_0.22
```

2.7.6 Where to ask for help?

- The person sitting next to you during the workshop. Don't hesitate to talk to your neighbor during the workshop, compare your answers, and ask for help. You might also be interested in organizing regular meetings following the workshop to keep learning from each other.
- Your friendly colleagues: if you know someone with more experience than you, they might be able and willing to help you.
- Stack Overflow: if your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min. Remember to follow their guidelines on how to ask a good question.
- The R-help mailing list: it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than anywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using `packageDescription("name-of-package")`. You may also want to try to email the author of the package directly, or open an issue on the code repository (e.g., GitHub).
- There are also some topic-specific mailing lists (GIS, phylogenetics, etc…), the complete list is here.

2.7.7 Resources on getting help

- The Posting Guide for the R mailing lists.
- How to ask for R help useful guidelines
- This blog post by Jon Skeet has quite comprehensive advice on how to ask programming questions.
- The reprex package is very helpful to create reproducible examples when asking for help. The [rOpenSci community call “How to ask questions so they get answered”], Github link and video recording includes a presentation of the reprex package and of its philosophy.

Chapter 3

Working with tabular data in R

Learning Objectives

- Load external data from a .csv file into a data frame in R with `read_csv()`
- Find basic properties of a data frames including size, class or type of the columns, names of rows and columns by using `str()`, `nrow()`, `ncol()`, `dim()`, `length()` , `colnames()`, `rownames()`
- Use `head()` and `tail()` to inspect rows of a data frame.
- Generate summary statistics for a data frame
- Use indexing to select rows and columns
- Use logical conditions to select rows and columns
- Add columns and rows to a data frame
- Manipulate categorical data with `factors`, `levels()` and `as.character()`
- Change how character strings are handled in a data frame.
- Format dates in R and calculate time differences
- Use `df$new_col <- new_col` to add a new column to a data frame.
- Use `cbind()` to add a new column to a data frame.
- Use `rbind()` to add a new row to a data frame.
- Use `na.omit()` to remove rows from a data frame with `NA` values.

Data frames are the de facto data structure for tabular data in R, and what we use for data processing, statistics, and plotting.

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Data frames are analogous to the more familiar spreadsheet in programs such as Excel, with one key difference. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors). For example, here is a figure depicting a data frame comprising a numeric, a character, and a logical vector.

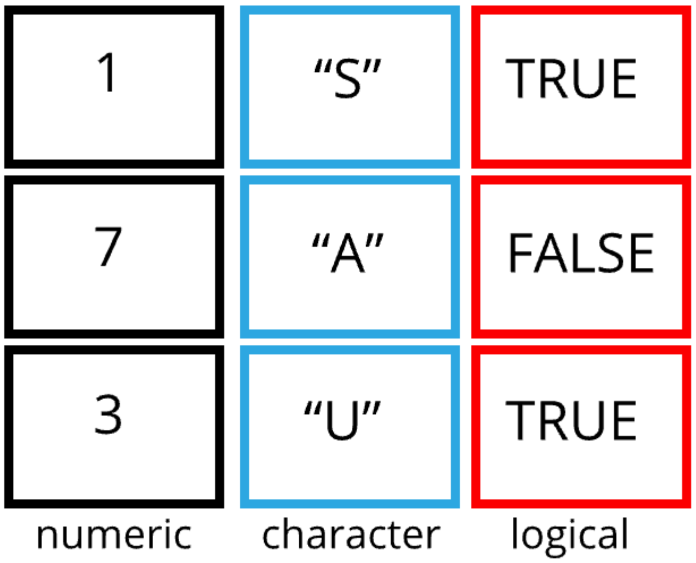


Figure 3.1: Structure of a data frame

As we have seen above, data frames can be created by hand, but most commonly they are generated by the functions like `read.csv()`, `read_csv()` or `read_table()` and others. These functions essentiallly import the tables or spreadsheets from your hard drive (or the web). We will now demonstrate how to import tabular data using `read_csv()`.

3.1 Importing tabular data

We will take a CSV file as example. What is a CSV file?

You may know about the Stanford Open Policing Project and we will be working with a sample dataset from their repository (https://openpolicing.stanford.edu/data/). The sample I extracted contains information about traffic stops for black and white drivers in the state of Mississippi during January 2013 to mid-July of 2016.

First, we are going to use the R function `download.file()` to download the CSV file that contains the traffic stop data, and we will use `read.csv()` to load into memory the content of the CSV file as an object of class `data.frame`.

To download the data into your local `data/` subdirectory, run the following:

```r
download.file("http://bit.ly/MS_trafficstops_bw", "data/MS_trafficstops_bw.csv")
```

You are going to load the data in R's memory using the function `read_csv()` from the `readr` package, which is part of the `tidyverse`; learn more about the tidyverse collection of packages (here)[https://www.tidyverse.org/]. `readr` gets installed as part as the `tidyverse` installation. When you load the `tidyverse` (`library(tidyverse)`), the core packages (the packages used in most data analyses) get loaded, including `readr`.

So lets make sure you have the `tidyverse` packages installed and loaded.

If you haven't done so already run the installation of `tidyverse` like this:

```r
install.packages("tidyverse" , dependencies = TRUE) # this is only necessary once
```

Then load `tidyverse` into memory like this:

```r
library(tidyverse) # do this whenever you need to access functions from the tidyverse packages
```

You may have noticed that when you loaded the tidyverse package that you received the following message:

```
  Conflicts                          tidyverse_conflicts()
dplyr::filter() masks stats::filter()       dplyr::lag()    masks
stats::lag()     Use the conflicted package to force all conflicts
to become errors
```

This presents a good opportunity to talk about conflicts. Certain packages we load can end up introducing function names that are already in use by pre-loaded R packages. For instance, when we load the `tidyverse` package below, we will introduce two conflicting functions: `filter()` and `lag()`. This happens because `filter` and `lag` are already functions used by the `stats` package (already pre-loaded in R). What will happen now is that if we, for example, call the `filter()` function, R will use the `dplyr::filter()` version and not the `stats::filter()` one. This happens because, if conflicted, **by default R uses the function from the most recently loaded package**. Conflicted functions may cause you some trouble in the future, so it is important that we are aware of them so that we can properly handle them, if we want.

To do so, we can use the following functions from the conflicted package:

- `conflicted::conflict_scout()`: Shows us any conflicted functions.
- `conflict_prefer("function", "package_prefered")`: Allows us to choose the default function we want from now on.

It is also important to know that we can, at any time, just call the function directly from the package we want, such as `stats::filter()`.

Ok. With that out of the way you are now ready to load the data.

```
stops <- read_csv("data/MS_trafficstops_bw.csv")
```

```
#> Rows: 211211 Columns: 11
#> -- Column specification --------------------------------------------------------
#> Delimiter: ","
#> chr  (8): id, state, county_name, police_department, driver_gender, driver_r...
#> dbl  (1): county_fips
#> date (2): stop_date, driver_birthdate
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

If you were to type in the code above, it is likely that the `read.csv()` (note the dot!) function would appear in the automatically populated list of functions. This function is different from the `read_csv()` (note the underscore!) function, as it is included in the "base" packages that come pre-installed with R. Overall, `read.csv()` behaves similar to `read_csv()`, with a few notable differences. First, `read.csv()` coerces column names with spaces and/or special characters to different names (e.g. interview date becomes interview.date). Second, `read.csv()` stores data as a **data.frame**, where `read_csv()` stores data as a **tibble**. We prefer tibbles because they have nice printing properties among other desirable qualities. Read more about tibbles here.

The second statement in the code above creates a data frame but doesn't output any data because, as you might recall, assignments (`<-`) don't display anything. (Note, however, that `read_csv` may show informational text about the data frame that is created.)

So let's check out the data! We can type the name of the object `stops`:

```
stops
```

```
#> # A tibble: 211,211 x 11
#>    id            state stop_date  county_name      county_fips police_department
#>    <chr>         <chr> <date>     <chr>                  <dbl> <chr>
#>  1 MS-2013-00001 MS    2013-01-01 Jones County           28067 Mississippi High~
#>  2 MS-2013-00002 MS    2013-01-01 Lauderdale Coun~       28075 Mississippi High~
#>  3 MS-2013-00003 MS    2013-01-01 Pike County            28113 Mississippi High~
#>  4 MS-2013-00004 MS    2013-01-01 Hancock County         28045 Mississippi High~
#>  5 MS-2013-00005 MS    2013-01-01 Holmes County          28051 Mississippi High~
#>  6 MS-2013-00006 MS    2013-01-01 Jackson County         28059 Mississippi High~
#>  7 MS-2013-00007 MS    2013-01-01 Jackson County         28059 Mississippi High~
#>  8 MS-2013-00008 MS    2013-01-01 Grenada County         28043 Mississippi High~
#>  9 MS-2013-00009 MS    2013-01-01 Holmes County          28051 Mississippi High~
#> 10 MS-2013-00010 MS    2013-01-01 Holmes County          28051 Mississippi High~
#> # i 211,201 more rows
#> # i 5 more variables: driver_gender <chr>, driver_birthdate <date>,
#> #   driver_race <chr>, violation_raw <chr>, officer_id <chr>
```

```
## Try also:
# head(stops)
# view(stops)
```

`read_csv()` assumes that fields are delimited by commas. For other delimiters (like semicolon or tab) check out the help: `?read_csv`

Note that `read_csv()` loads the data as a so called "tibble". A tibble is an extended form of R data frames (as an object of multiple classes `tbl_df`, `tbl`, and `data.frame`). This may sound confusing, but it is really not anything you typically need to deal with when working the data. In fact, it makes it a little more convenient.

As you may recall, a data frame in R is a special case of a list, and a representation of data where the columns are vectors that all have the same length. Because the columns are vectors, they all contain the same type of data (e.g., characters, integers, factors, etc.).

In this tibble you can see the type of data included in each column listed in an abbreviated fashion right below the column names. For instance, the `state` column is is of type character `<chr>`, the `stop_date` is in `<date>` format and `county_fips` are floating point numbers (abbreviated `<dbl>` for the word 'double').

3.2 Inspecting data frames

When calling a `tbl_df`object (like `stops` here), there is already a lot of information about our data frame being displayed such as the number of rows, the number of columns, the names of the columns, and as we just saw the class of data stored in each column. However, there are additional functions to extract this information from data frames. Here is a non-exhaustive list of some of these functions. Let's try them out!

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

(Note: most of these functions are "generic", they can be used on other types of objects besides data frames or tibbles.)

- Summary:
  - `str(stops)` - structure of the object and information about the class, length and content of each column
  - `summary(stops)` - summary statistics for each column
  - `glimpse(stops)` - returns the number of columns and rows of the tibble, the names and class of each column, and previews as many values will fit on the screen. Unlike the other inspecting functions

listed above, `glimpse()` is not a 'base R' function so you need to have the `dplyr` or `tibble` packages loaded to be able to execute it.

- Size:
  - `dim(stops)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the **dim**ensions of the object)
  - `nrow(stops)` - returns the number of rows
  - `ncol(stops)` - returns the number of columns
  - `length(stops)` - returns number of columns
- Content:
  - `head(stops)` - shows the first 6 rows
  - `tail(stops)` - shows the last 6 rows
- Names:
  - `names(stops)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
  - `rownames(stops)` - returns the row names

Challenge

Based on the output of `str(stops)`, can you answer the following questions?

- What is the class of the object `stops`?
- How many rows and how many columns are in this object?
- How many counties have been recorded in this dataset?

3.3 Indexing and subsetting data frames

Our stops data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the "coordinates" (i.e., indices) we want from it. Row numbers come first, followed by column numbers.

```
## first element in the first column of the tibble
stops[1, 1]
## first element in the 6th column of the tibble
stops[1, 6]
## first column of the tibble
stops[1]
## first column of the tibble (as a vector)
stops[[1]]

## the 3rd row of the tibble
stops[3, ]
## first three elements in the 7th column of the tibble
stops[1:3, 7]
## equivalent to head(stops)
stops[1:6, ]
```

```r
## Excludig with '-'
## The whole tibble, except the first column
stops[, -1]
## equivalent to head(stops)
stops[-c(7:nrow(stops)),]
```

Subsetting a `tibble` with `[` always results in a tibble. However, note that different ways of specifying these coordinates lead to results with different classes. Below are some example for `data.frame` objects.

```r
stops_df <- as.data.frame(stops)
stops_df[1, 1]   # first element in the first column of the data frame (as a vector)
stops_df[, 1]    # first column in the data frame (as a vector)
stops_df[1]      # first column in the data frame (as a data.frame)
```

An alternative to subsetting `tibbles` (and data frames) is to calling their column names directly.

```r
stops["violation_raw"]        # Result is a tibble
stops[, "violation_raw"]      # Result is a tibble
stops[["violation_raw"]]      # Result is a vector
stops$violation_raw           # Result is a vector
```

RStudio knows about the columns in your data frame, so you can take advantage of the autocompletion feature to get the full and correct column name.

> Challenge
>
> 1. Create a `tibble` (`stops_200`) containing only the observations from row 200 of the `stops` dataset.
>
> 2. Notice how `nrow()` gave you the number of rows in a `tibble`?
>
>   - Use that number to pull out just that last row in the data frame.
>   - Compare that with what you see as the last row using `tail()` to make sure it's meeting expectations.
>   - Pull out that last row using `nrow()` instead of the row number.
>   - Create a new data frame object (`stops_last`) from that last row.
>
> 3. Use `nrow()` to extract the row that is in the middle of the data frame. Store the content of this row in an object named `stops_middle`.
>
> 4. Combine `nrow()` with the `-` notation above to reproduce the behavior of `head(stops)` keeping just the first through 6th rows of the stops dataset.

3.4 Conditional subsetting

A very common need when working with tables is the need to extract a subset of a data frame based on certain conditions, depending on the actualcontent of the table. For example, we may want to look only at traffic stops in Webster County. In this case we can use logical conditions, exactly like we did above with vector subsetting. In base R this can be done like this:

```
# the condition:
# returns a logical vector of the length of the column
stops$county_name == "Webster County"

# use this vector to extract rows and all columns
# note the comma: we want *all* columns
stops[stops$county_name == "Webster County", ]

# assign extract to a new data frame
Webster_stops <- stops[stops$county_name == "Webster County", ]
```

This is also a possibility (but slower):

```
Webster_stops <- subset(stops, county_name == "Webster County")
nrow(Webster_stops) # 393 stops in Webster County!
```

```
#> [1] 156
```

```
# and if we wanted to see the breakdown by race:
table(Webster_stops$driver_race)
```

```
#>
#> Black White
#>    59    97
```

These commands are from the R base package. In the R Data Wrangling workshop we will discuss a different way of subsetting using functions from the `tidyverse` package.

> Challenge
>
> - Use subsetting to extract stops in Hancock, Harrison, and Jackson Counties into a separate data frame `coastal_counties`.
> - Using `coastal_counties`, count the total number of Black and White drivers in the coastal counties.
> - Bonus: Count the total number of Black and White drivers in the entire `stops` dataset. How does the ratio of Black to White stops in the three coastal counties compare to the same ratio for stops in the entire state of Mississippi?

3.5 Adding and removing rows and columns

To add a new column to the data frame we can use the `cbind()` function There also is a `bind_cols()` function from `dplyr` package (part of the `tidyverse`). An important difference with `bind_cols()` is that it displays an error message when you try to combine with vector that has fewer or more elements than the number of rows in the table. `cbind()` on the other hand, silently repeats values or rows, so you might introduce errors and be unaware of it.

```
id_column <- 1:nrow(stops) # create a unique ID number for each row
stops_with_id <- cbind(stops, id_column)
glimpse(stops_with_id)
```

```
#> Rows: 211,211
#> Columns: 12
#> $ id                <chr> "MS-2013-00001", "MS-2013-00002", "MS-2013-00003", "~
#> $ state             <chr> "MS", "MS", "MS", "MS", "MS", "MS", "MS", "MS", "MS"~
#> $ stop_date         <date> 2013-01-01, 2013-01-01, 2013-01-01, 2013-01-01, 201~
#> $ county_name       <chr> "Jones County", "Lauderdale County", "Pike County", ~
#> $ county_fips       <dbl> 28067, 28075, 28113, 28045, 28051, 28059, 28059, 280~
#> $ police_department <chr> "Mississippi Highway Patrol", "Mississippi Highway P~
#> $ driver_gender     <chr> "M", "M", "M", "M", "M", "F", "F", "F", "M", "M", "M~
#> $ driver_birthdate  <date> 1950-06-14, 1967-04-06, 1974-04-15, 1981-03-23, 199~
#> $ driver_race       <chr> "Black", "Black", "Black", "White", "White", "White"~
#> $ violation_raw     <chr> "Seat belt not used properly as required", "Careless~
#> $ officer_id        <chr> "J042", "B026", "M009", "K035", "D028", "K023", "K03~
#> $ id_column         <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1~
```

Alternatively, we can also add a new column adding the new column name after the `$` sign then assigning the value, like below. **Note that this will change the original data frame, which you may not always want to do.**

```
stops$row_numbers <- c(1:nrow(stops))
stops$all_false <- FALSE  # what do you think will happen here?
```

There is an equivalent function, `rbind()` to add a new row to a data frame. I use this far less frequently than the column equivalent. The one thing to keep in mind is that the row to be added to the data frame needs to match the order and type of columns in the data frame. Remember that R's way to store multiple different data types in one object is a `list`. So if we wanted to add a new row to `stops` we would say:

```
new_row <- data.frame(id="MS-2017-12345", state="MS", stop_date="2017-08-24",
                county_name="Tallahatchie County", county_fips=12345,
                police_department="MSHP", driver_gender="F", driver_birthdate="1999-06-14",
                driver_race="Hispanic", violation_raw="Speeding", officer_id="ABCD")

stops_withnewrow <- rbind(stops, new_row)
```

```
tail(stops_withnewrow)
```

```
#> # A tibble: 6 x 11
#>   id    state stop_date  county_name county_fips police_department driver_gender
#>   <chr> <chr> <date>     <chr>             <dbl> <chr>             <chr>
#> 1 MS-2~ MS    2016-07-09 George Cou~       28039 Mississippi High~ M
#> 2 MS-2~ MS    2016-07-10 Copiah Cou~       28029 Mississippi High~ M
#> 3 MS-2~ MS    2016-07-11 Grenada Co~       28043 Mississippi High~ M
#> 4 MS-2~ MS    2016-07-14 Copiah Cou~       28029 Mississippi High~ F
#> 5 MS-2~ MS    2016-07-14 Copiah Cou~       28029 Mississippi High~ M
#> 6 MS-2~ MS    2017-08-24 Tallahatch~       12345 MSHP              F
#> # i 4 more variables: driver_birthdate <date>, driver_race <chr>,
#> #   violation_raw <chr>, officer_id <chr>
```

Equivalently there is a `bind_rows` function available in `tidyverse`. One of the main reasons for using `bind_rows` over `rbind` is to combine two data frames having different number of columns. `rbind` throws an error in such a case whereas `bind_rows` assigns "NA" to those rows of columns missing in one of the data frames where the value is not provided by the data frames. Here is a systematic review of differences between the two.

There is also `add_row` fro the `tibble` package which allows you to specify where to insert the row. You can find out more with `?add_row`.

A convenient function to know about is `na.omit()`. It will remove all rows from a data frame that have at least one column with `NA` values. The function `drop_na()` from `tidyverse` works similarly and lets you name specific columns with NA.

> Challenge
>
> - Given the following data frame:
>
> ```
> dfr <- data.frame(col_1 = c(1:3),
>                   col_2 = c(NA, NA, "b"),
>                   col_3 = c(TRUE, NA, FALSE))
> ```
>
> What would you expect the following commands to return?
>
> ```
> nrow(dfr)
> nrow(na.omit(dfr))
> ```

3.6 Categorical data: Factors

Factors are very useful and are actually something that make R particularly well suited to working with data, so we're going to spend a little time introducing them.

Factors are used to represent categorical data. Factors can be ordered or unordered, and understanding them is necessary for statistical analysis and for plotting.

Factors are stored as integers, and have labels (text) associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts *levels* in alphabetical order. For instance, if you have a factor with 3 levels:

```
priority <- factor(c("low", "high", "medium", "low", "high"))
```

R will assign `1` to the level `"high"` and `2` to the level `"low"` and `3` to the level `low` (because it orders alphabetically, not according to position in the vector). You can check this by using the function `levels()`, and check the number of levels using `nlevels()`:

```
levels(priority)
```

```
#> [1] "high"   "low"    "medium"
```

```
nlevels(priority)
```

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

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "medium", "high"), it improves your visualization, or it is required by a particular type of analysis. Here, one way to reorder our levels in the `priority` vector would be:

```
priority # current order
```

```
#> [1] low    high   medium low    high
#> Levels: high low medium
```

```
priority <- factor(priority, levels = c("high", "medium", "low"))
priority # after re-ordering
```

```
#> [1] low    high   medium low    high
#> Levels: high medium low
```

In R's memory, these factors are represented by integers (1, 2, 3), but are more informative than integers because factors are self describing: `"high"`, `"medium"` and `"low"` is more descriptive than `1`, `2`, `3`. Which one is "low"? You wouldn't be able to tell just from the integer data. Factors, on the other hand, have this information built in.

3.6.1 Converting factors

If you need to convert a factor to a character vector, you use `as.character(x)`.

```r
as.character(priority)
```

```
#> [1] "low"    "high"   "medium" "low"    "high"
```

It is a little is a little trickier to convert factors where the levels appear as numbers, such or years, for example, to numbers.One method is to convert factors to characters and then numbers. Another method is to use the `levels()` function. Compare:

```r
y <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(y)                # wrong! and there is no warning...
```

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

```r
as.numeric(as.character(y)) # works...
```

```
#> [1] 1990 1983 1977 1998 1990
```

```r
as.numeric(levels(y))[y]    # The recommended way.
```

```
#> [1] 1990 1983 1977 1998 1990
```

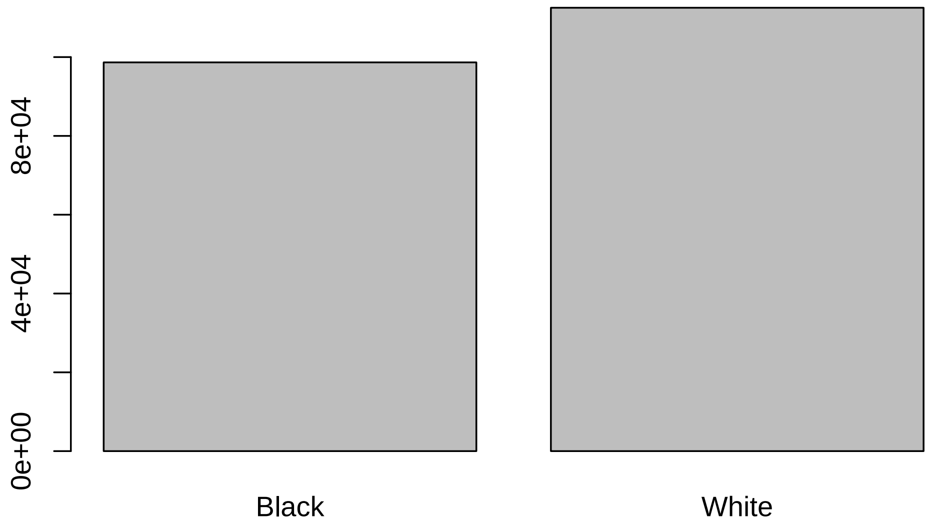
Notice that in the `levels()` approach, three important steps occur:

- We obtain all the factor levels using `levels(y)`
- We convert these levels to numeric values using `as.numeric(levels(y))`
- We then access these numeric values using the underlying integers of the vector `y` as indices inside the square brackets

3.6.2 Renaming factors

When your data is stored as a factor, you can use the `plot()` function to get a quick glance at the number of observations represented by each factor level. Let's look at the number of black and white drivers in the `stops` dataset:

```r
# We create a new variable with the column "driver_race" as a factor
race <- stops$driver_race
race <- factor(race)
plot(race)
```



This looks good, however, `plot` silently ignores NAs and we would like to know if there are any:

```
# We create a new variable with the column "driver_race" as a factor
sum(is.na(race))
```

```
#> [1] 28
```

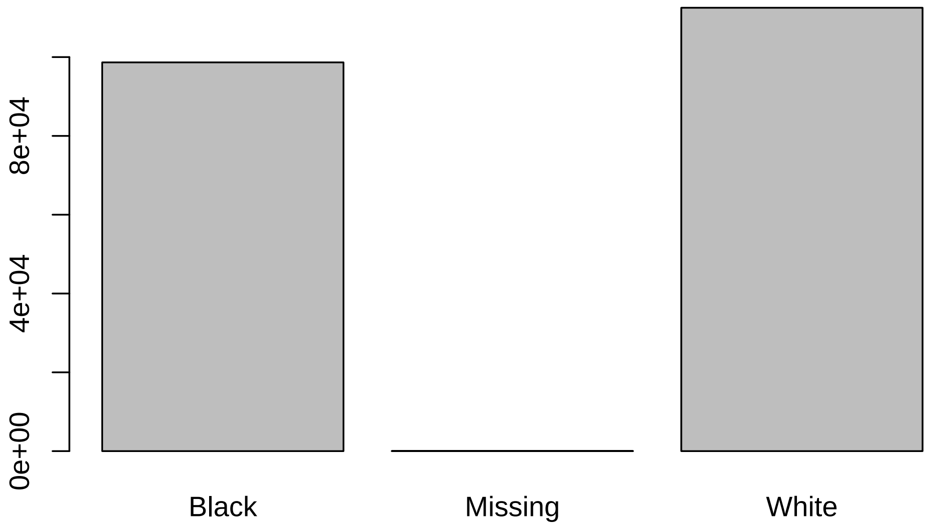
There seem to be a number of individuals for which the race information hasn't been recorded. Additionally, for these individuals, there is no label to indicate that the information is missing. Let's rename this label to something more meaningful:

```
## Let's recreate the vector from the data frame column driver_race
race <- stops$driver_race

## replace the missing data with "unknown"
race[is.na(race)] <- 'Missing'

## convert it into a factor
race <- as.factor(race)

## let's see what it looks like
plot(race)
```



Challenge

- Rename "Black" to "African American".
- Now that we have renamed the factor level to "Missing", can you recreate the barplot such that "Missing" is last (to the right)?

—>

3.7 Date Formats

One of the most common issues that new (and experienced!) R users have is converting date and time information into a variable that is appropriate and usable during analyses. If you have control over your data it might be useful to ensure that each component of your date is stored as a separate variable, i.e a separate column for day, month, and year. However, often we do not have control and the date is stored in one single column and with varying order and separating characters between its components.

Using `str()`, we can see that both dates in our data frame `stop_date` and `driver_birthdate` are each stored in one column.

```
str(stops)
```

```
#> spc_tbl_ [211,211 x 11] (S3: spc_tbl_df/tbl_df/tbl/data.frame)
#>  $ id               : chr [1:211211] "MS-2013-00001" "MS-2013-00002" "MS-2013-00003"
#>  $ state            : chr [1:211211] "MS" "MS" "MS" "MS" ...
#>  $ stop_date        : Date[1:211211], format: "2013-01-01" "2013-01-01" ...
#>  $ county_name      : chr [1:211211] "Jones County" "Lauderdale County" "Pike Count
#>  $ county_fips      : num [1:211211] 28067 28075 28113 28045 28051 ...
#>  $ police_department: chr [1:211211] "Mississippi Highway Patrol" "Mississippi Highway Patrol'
#>  $ driver_gender    : chr [1:211211] "M" "M" "M" "M" ...
#>  $ driver_birthdate : Date[1:211211], format: "1950-06-14" "1967-04-06" ...
#>  $ driver_race      : chr [1:211211] "Black" "Black" "Black" "White" ...
#>  $ violation_raw    : chr [1:211211] "Seat belt not used properly as required" "Careless drivi
#>  $ officer_id       : chr [1:211211] "J042" "B026" "M009" "K035" ...
#>  - attr(*, "spec")=
#>   .. cols(
#>   ..   id = col_character(),
#>   ..   state = col_character(),
#>   ..   stop_date = col_date(format = ""),
#>   ..   county_name = col_character(),
#>   ..   county_fips = col_double(),
#>   ..   police_department = col_character(),
#>   ..   driver_gender = col_character(),
#>   ..   driver_birthdate = col_date(format = ""),
#>   ..   driver_race = col_character(),
#>   ..   violation_raw = col_character(),
#>   ..   officer_id = col_character()
#>   .. )
#>  - attr(*, "problems")=<externalptr>
```

As an example for how to work with dates let us see if there are seasonal differences in the number of traffic stops.

Start by loading the required package:

```r
library(lubridate)
```

`read_csv()` has already recognized the Date format of the column when we read the table in earlier.

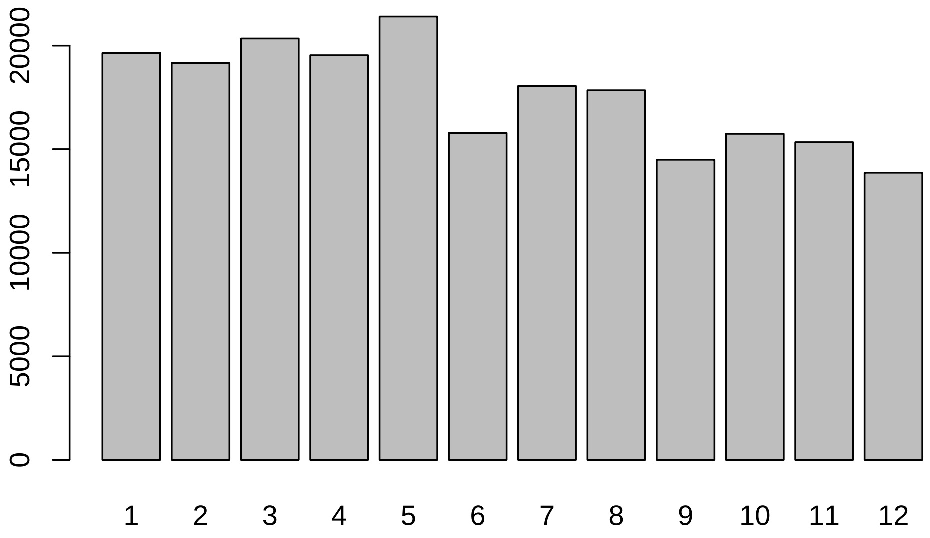
```r
stop_date <- stops$stop_date
str(stop_date) # notice the 'Date' class
```

```
#>  Date[1:211211], format: "2013-01-01" "2013-01-01" "2013-01-01" "2013-01-01" "2013-01-01" ...
```

We can now take advantage of different functions to extract year, month, and day of the month, and weekday: `year()`, `month()`, `day()`, `wday()` like so:

```r
stop_month <- month(stop_date) # extract the month

# convert year to factor to plot
plot(factor(stop_month))
```



If your dates are not in Date format, you can use the `ymd()` function from the package **`lubridate`**. This function is designed to take a vector representing year, month, and day and convert that information to a POSIXct vector. POSIXct is a class of data recognized by R as being a date or date and time. The argument that the function requires is relatively flexible, but, as a best practice, is a character vector formatted as “YYYY-MM-DD”.

Challenge

- Are there more stops in certain days of the week?
- Determine the age of the driver in years (approximate) at the time of the stop:
- Extract `driver_birthdate` into a vector `birth_date`
- Create a new vector `age` with the driver’s age at the time of the stop in years
- Coerce `age` to a factor and use the `plot` function to check your results. What do you find?